

EXERCISES QUE TRABAJAREMOS EN EL CUE:

- EXERCISE 1: CAPTURANDO EVENTO DENTRO DE UN MÉTODO.
- EXERCISE 2: MÉTODOS CON ARGUMENTOS.
- EXERCISE 3: VALIDANDO FORMULARIO.
- EXERCISE 4: PROPIEDADES COMPUTADAS (I).
- EXERCISE 5: PROPIEDADES COMPUTADAS (II).

EXERCISE 1: CAPTURANDO EVENTO DENTRO DE UN MÉTODO.

Iniciaremos creando un proyecto con vue/cli. Para ello, ingresamos al terminal, buscamos un directorio donde deseemos trabajar, en este caso se ha elegido escritorio (Desktop), y escribimos `vue create nombre_proyecto`.



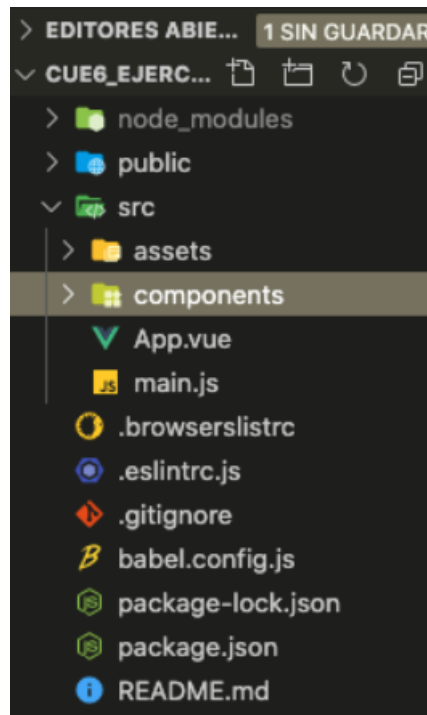
```
~/Desktop vue create cueé
```

DESVINCULAR COMPONENTE HELLOWORLD.VUE

Al crear un proyecto con vue/cli, por defecto, éste nos agrega un componente de ejemplo llamado HelloWorld.vue, el cual se encuentra dentro de la carpeta "components".

1. Eliminar HelloWorld.vue, desde la carpeta components.
2. Borrar referencias en archivo App.vue, del componente recién eliminado.

Así, la estructura de nuestro proyecto debería quedar como la siguiente imagen:



En el archivo App.vue, es donde debemos borrar la referencia al componente HelloWorld.vue, quedando de esta forma:

```

1 <template>
2   <div id="app">
3
4   </div>
5 </template>
6
7 <script>
8
9 export default {
10   name: 'App',
11   components: {
12
13   }
14 }
15 </script>
16 <style>
17 /* #app {
18   font-family: Avenir, Helvetica, Arial, sans-serif;
19   -webkit-font-smoothing: antialiased;

```

```
20 -moz-osx-font-smoothing: grayscale;
21 text-align: center;
22 color: #2c3e50;
23 margin-top: 60px;
24 } */
25 </style>
```

CAPTURANDO EVENTO NATIVO JS.

En algunas ocasiones, es necesario obtener el evento nativo que está generando algún elemento HTML. Para ello, debemos saber que el evento lo podemos capturar para generar alguna lógica que necesitemos. Al ejemplificar ésto, vamos a crear un nuevo componente llamado Form.vue.

```
1 <template>
2   <form>
3     <label for="">Titulo</label><br>
4     <input type="text" v-model="title"><br>
5     <label for="">Comentario</label><br>
6     <textarea cols="30" rows="10" v-model="body"></textarea>
7     <input type="submit" @click="addComment">
8   </form>
9 </template>
```

En este segmento de código, hemos agregado un método llamado "addComment", el cual nos permitirá capturar el evento.

CREANDO SCRIPT:

```
1 <script>
2 export default {
3   name: "Form-component",
4   data: function() {
5     return{
6       title:"",
7       body:"",
8     }
9   },
10  methods:{
11    addComment(event) {
12      console.log(event.target);
13      event.preventDefault();
14    }
15  }
16 }
```

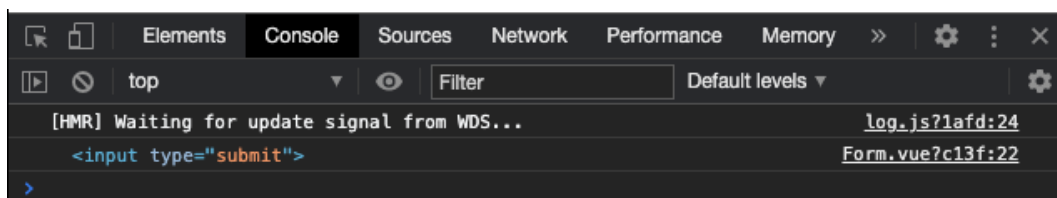
```
15 }  
16 }  
17 </script>
```

El método, aunque no esté recibiendo ningún argumento, puede generar cualquier lógica que sea necesaria. En el caso del ejemplo, está previniendo el comportamiento de recargar página, luego de presionar un input submit.

Título

Comentario

Enviar



EXERCISE 2: MÉTODOS CON ARGUMENTOS.

Continuaremos utilizando la misma estructura creada en el ejercicio 1, en la cual teníamos un componente Form.vue, que era invocado en el componente principal App.vue.

En algunos casos, es necesario pasar ciertos datos a algún método, y para que éste se ejecute de manera correcta, los argumentos a pasar dependerán de lo que se necesite realizar.

En el siguiente ejemplo, vamos a utilizar un método llamado `message`, el cual necesita un dato de entrada, que será un string, para que éste muestre una alerta con el mensaje.

TEMPLATE:

Para realizar el ejemplo, se ha tomado el mismo componente form.vue, y se la ha agregado un `<div>` raíz, y 2 botones que contienen el método `message`.

```
1 <template>
2   <div>
3     <form>
4       <label for="">Titulo</label><br>
5       <input type="text" v-model="title"><br>
6       <label for="">Comentario</label><br>
7       <textarea cols="30" rows="10" v-model="body"></textarea>
8       <input type="submit" @click="addComment">
9     </form>
10
11     <button @click="message('Hola bienvenido')">Mensaje
12 Saludo</button>
13
14     <button @click="message('Adios')">Mensaje Despida</button>
15   </div>
16
17 </template>
```

Si guardamos, se debería visualizar lo siguiente:

Título

Comentario

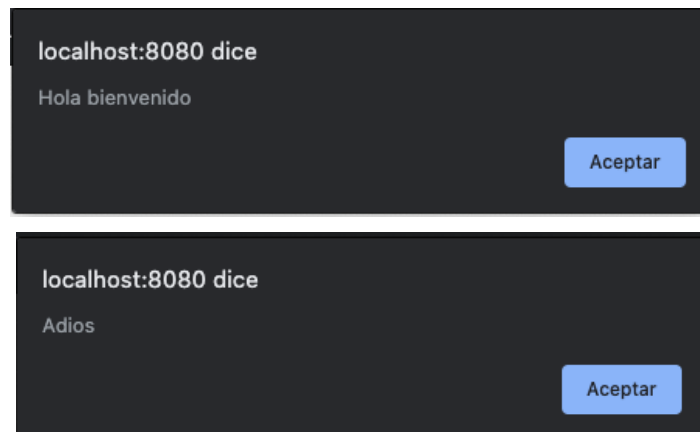
Enviar

Mensaje Saludo **Mensaje Despida**

Script:

```
1 <script>
2 export default {
3   name: "Form-component",
4   data: function() {
5     return{
6       title:"",
7       body:"",
8     }
9   },
10  methods:{
11    addComment(event) {
12      console.log(event.target);
13      event.preventDefault();
14    },
15    message(msg) {
16      alert(msg);
17    }
18  }
19 }
20 </script>
```

La función `message`, solo está enviando un mensaje. Es importante destacar, que podemos pasar los mismos tipos de datos, como lo hacemos con Javascript.



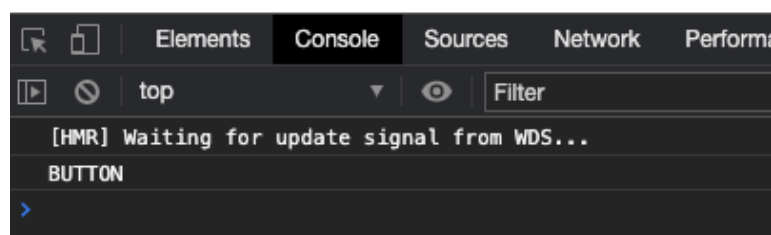
CAPTURANDO EVENTO DE UNA FUNCIÓN CON ARGUMENTOS:

Como vimos en el ejercicio 1, podemos capturar el evento de método de forma simple, mientras que éste no tenga argumentos. Pero ¿qué pasa en el caso de necesitar obtener el evento que está gatillando al método?, aquí tendremos que usar el argumento `$event`.

```
1 <button @click="message('Hola bienvenido',$event)">Mensaje
2 Saludo</button>
```

```
1 message(msg, event) {
2   console.log(event.target.tagName);
3   alert(msg);
4 }
```

En el ejemplo anterior, obtenemos el evento y posteriormente, mostramos por consola el tag HTML que genera el evento.



EXERCISE 3: VALIDANDO FORMULARIO.

Para seguir aprendiendo cómo usar los métodos en Vue, usaremos como ejemplo el ejercicio 1, el cual contenía un formulario y un método.

```
1 <template>
2   <div>
3     <form>
4       <label for="">Titulo</label><br>
5       <input type="text" v-model="title"><br>
6       <label for="">Comentario</label><br>
7       <textarea cols="30" rows="10" v-model="body"></textarea>
8       <input type="submit" @click="addComment">
9     </form>
10  </div>
11 </template>
12
13 <script>
14 export default {
15   name: "Form-component",
16   data: function() {
17     return{
18       title:"",
19       body:"",
20     }
21   },
22   methods:{
23     addComment(event) {
24       console.log(event.target);
25       event.preventDefault();
26     },
27   }
28 }
29 </script>
30
31 <style>
32
33 </style>
```


Cabe destacar, que este archivo ha sido llamado Form.vue, y es referenciado en el componente principal App.vue.

```
1 <template>
2   <div id="app">
3     <FormComponent></FormComponent>
4   </div>
5 </template>
6
7 <script>
8 import FormComponent from "@components/Form.vue"
9 export default {
10   name: 'App',
11   components: {
12     FormComponent,
13   }
14 }
15 </script>
16
17 <style>
18
19 </style>
```

CREAR INPUT EMAIL:

Para comenzar, vamos a modificar el archivo Form.vue, específicamente en el template, agregaremos un label y un input llamado email.

```
1 <label for="">Correo</label><br>
2 <input ref="email" type="text" v-model="email"><br>
```

En el input, hemos agregado una ref y un v-model.

```
1 <form>
2   <label for="">Titulo</label><br>
3   <input type="text" v-model="title"><br>
4   <label for="">Correo</label><br>
5   <input ref="email" type="text" v-model="email"><br>
6   <label for="">Comentario</label><br>
7   <textarea cols="30" rows="10" v-model="body"></textarea>
8   <input type="submit" @click="addComment">
9 </form>
```

Si lo guardamos, deberíamos verlo de la siguiente forma:

Título

Correo

Comentario

Enviar

REF EN INPUTS:

Como lo hicimos en el input de correo, debemos agregar una ref para el título, y una para el comentario. Esto nos servirá para posteriormente darle focus.

```
1 <input ref="title" type="text" v-model="title"><br>
```

```
1 <textarea ref="body" cols="30" rows="10" v-model="body"></textarea>
```

OBJETO DATA:

Aquí debemos agregar el dato email, que fue enlazado en el formulario.

```
1 data: function() {  
2     return {  
3         title: "",  
4         body: "",  
5         email: "",  
6     }  
7 },
```

Si nos fijamos, los 3 input del formulario partirán vacíos al iniciar el componente.

REGLAS DE VALIDACIÓN:

Para poder validar nuestro formulario, necesitamos crear cierta lógica, es por ello que creamos métodos que validarán a cada input. Partiremos creando con la regla del input título.

Dentro del objeto methods, crearemos esta función.

REGLA INPUT TITULO:

```
1 titleRules() {  
2     if(this.title == "") {  
3         alert('Titulo es requerido');  
4         this.$refs.title.focus();  
5         return false;  
6     }  
7     else {  
8         return true;  
9     }  
10 },
```

Si analizamos la función, lo que está haciendo es verificar que el campo no esté vacío; en caso de estarlo, lanzará una alerta al usuario, dará el focus en el input título, y retornará falso, para indicar que hubo un error en esa validación.

REGLA INPUT EMAIL:

```
1 emailRules() {  
2     let regex = /^\\w+@[a-zA-Z_]+?\\. [a-zA-Z]{2,3}$/;  
3     if(regex.test(this.email)) {  
4         return true  
5     }  
6     else{  
7         alert('Correo no valido')  
8         this.$refs.email.focus();  
9         return false;  
10    }  
11 },
```

En el caso del email, haremos una validación a través de una expresión regular. Si este dato cumple el patrón, retornará true. Si no se cumple, lanzaremos un alert al usuario, le daremos focus al input, y retornaremos falso.

REGLA INPUT COMENTARIO:

```
1 commentRules() {  
2     if(this.body == "") {  
3         alert('Comentario es requerido');  
4         this.$refs.body.focus();  
5         return false;  
6     }  
7     else{  
8         return true;  
9     }  
10 },
```

Para validar el comentario, utilizaremos la misma lógica de verificar que el campo no esté vacío para validar el input.

METODO VALIDATE ():

Ahora que ya tenemos las reglas por separado, podemos unir las en un método en común. Para ello, crearemos **validate()**, el cual llamará a los 3 métodos anteriores. Si todos estos retornan "true", el formulario está validado; y en caso de que cualquier validación falle, el formulario no lo estará.

```
1 validate() {  
2     if (this.titleRules() && this.emailRules() &&  
3 this.commentRules()) {  
4         return true;  
5     } else {  
6         return false;  
7     }  
8 },
```

METODO ADDCOMMENT:

Será el encargado de llamar a **validate()**, ya que está ligado al evento clic del input submit.

```
1 addComment(event) {  
2     event.preventDefault();  
3     if (this.validate()) {  
4         alert("Podemos enviar el formulario");  
5     }  
6 },
```

PROBANDO FORMULARIO:

Titulo

Correo

Comentario

Enviar

Ahora, si hacemos clic en enviar, se visualizará así.

localhost:8080 dice

Titulo es requerido

Aceptar

Titulo

Correo

Comentario

Enviar

Si rellenamos el campo título y presionamos enviar, nos saldrá la alerta para el input correo.

Título

Correo

Comentario

Enviar

localhost:8080 dice

Correo no valido

Aceptar

En el caso del input email, no basta con rellenar el input, pues éste también debe cumplir el patrón de un correo.

Título

Correo

Comentario

Enviar

localhost:8080 dice

Correo no valido

Aceptar

Si rellenamos correctamente el correo, pasará a verificar la regla 3.

Título

Correo

Comentario

Si hacemos clic, nos saldrá el mensaje de la validación del comentario.

localhost:8080 dice

Comentario es requerido

Rellenamos el comentario, y el formulario ya estaría en condiciones de ser enviado.

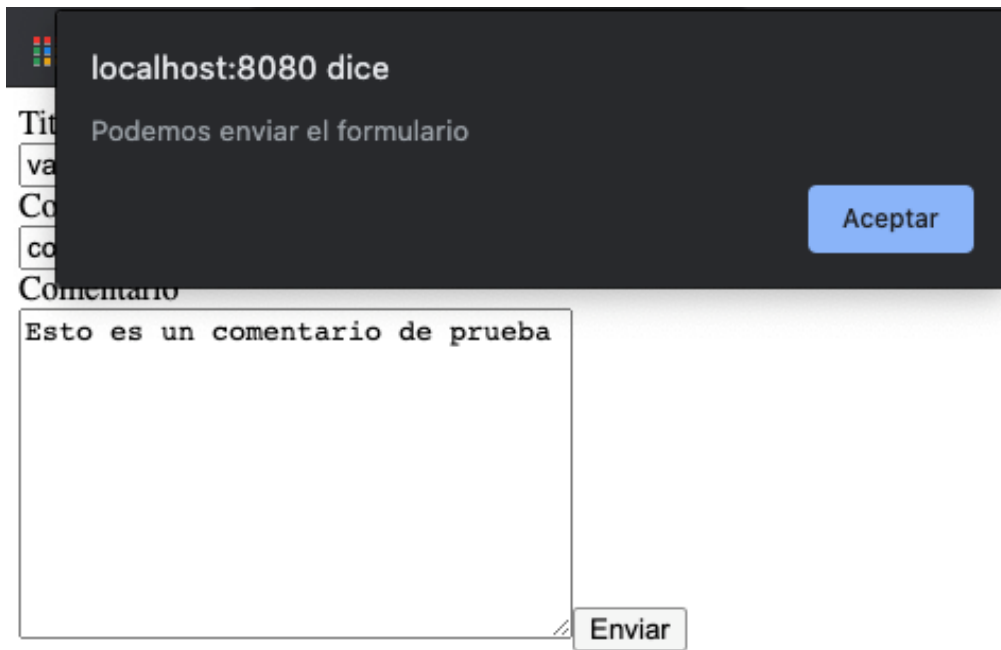
Título

Correo

Comentario

Esto es un comentario de prueba

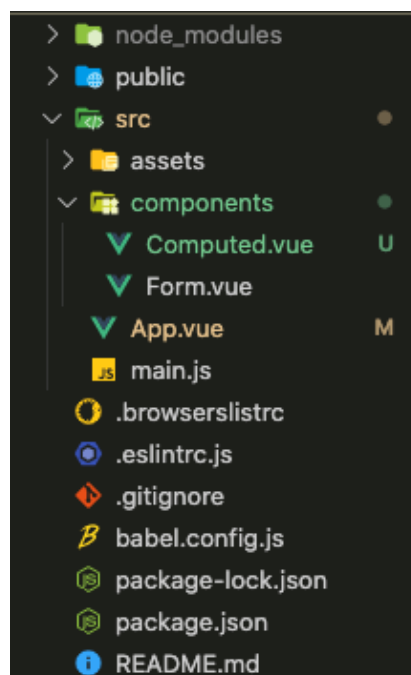
Si presionamos enviar, debería salir el mensaje, y ahí ya podemos enviar el formulario.



The screenshot shows a web application interface. A dark modal box is overlaid on the page, displaying the text "localhost:8080 dice" and "Podemos enviar el formulario". A blue button labeled "Aceptar" is located in the bottom right corner of the modal. Below the modal, a form is visible with a label "Comentario" and a text input field containing the text "Esto es un comentario de prueba". A button labeled "Enviar" is positioned at the bottom right of the form.

EXERCISE 4: PROPIEDADES COMPUTADAS (I).

Para empezar a entender cómo funcionan las propiedades computadas, vamos a continuar trabajando en el proyecto en el que estudiamos los métodos. En primer lugar, debemos crear un nuevo componente dentro de la carpeta components, y le llamaremos **Computed.vue** (el nombre es solo para ejemplificar).



MODIFICAR APP.VUE

Si seguiste el ejercicio anterior, creamos un component Form.vue. En este caso, hemos comentado dicho componente, y agregado Computed.vue.

```

1 <template>
2   <div id="app">
3     <!-- <FormComponent></FormComponent> -->
4     <ComputedComponent></ComputedComponent>
5   </div>
6 </template>
7
8 <script>
```

```
9 // import FormComponent from "@components/Form.vue"
10 import ComputedComponent from '@components/Computed.vue'
11 export default {
12   name: 'App',
13   components: {
14     //FormComponent,
15     ComputedComponent
16   }
17 }
18 </script>
19 <style></style>
```

ESTRUCTURA COMPONENTE:

```
1 <template>
2
3 </template>
4
5 <script>
6 export default {
7
8 }
9 </script>
10
11 <style>
12
13 </style>
```

AGREGANDO DATOS EN SCRIPT:

Comenzaremos añadiendo el objeto data, para incluir datos:

```
1 <script>
2 export default {
3   data: function() {
4     return {
5       nombre: "Alejandro",
6       apellido: "Perez",
7     }
8   },
9 }
10 </script>
```

AGREGANDO COMPUTED PROPERTY:

Para poder trabajar con propiedades computadas, necesitamos crear el objeto computed dentro del objeto principal, y quedaría así:

```
1 <script>
2 export default {
3   data: function() {
4     return {
5       nombre: "Alejandro",
6       apellido: "Perez",
7     }
8   },
9   computed: {
10
11   },
12 }
13 </script>
```

AGREGANDO UNA FUNCIÓN AL OBJETO COMPUTED:

Las propiedades computadas son funciones que toman los datos desde el objeto data, y permite crear nueva lógica. Éstas se ejecutan cada vez que los datos involucrados cambien. Ahora, vamos a crear la primera propiedad computada llamada **getFullName**, la cual tomará el dato "nombre y apellido", los cambiará a mayúsculas, y los concatenará.

```
1 <script>
2 export default {
3   data: function() {
4     return {
5       nombre: "Alejandro",
6       apellido: "Perez",
7     }
8   },
9   computed: {
10     getFullName() {
11       return this.nombre.toUpperCase() + " " +
12 this.apellido.toUpperCase()
13     }
14   },
15 }
16 </script>
```

TEMPLATE

Para poder llamar a una propiedad computa dentro del template, lo hacemos igual que con un dato incluido en el objeto data.

```
1 <template>
2   <div>
3     <h1>{{nombre}} {{apellido}}</h1>
4     <h1>{{getFullName}}</h1>
5   </div>
6 </template>
```

Si guardamos y vemos el navegador, deberíamos obtener lo siguiente:

Alejandro Perez

ALEJANDRO PEREZ

EXERCISE 5: PROPIEDADES COMPUTADAS (II).

Para continuar estudiando las propiedades computadas, seguiremos en el mismo archivo Computed.vue. En este caso, agregaremos un Arreglo con productos, que nos servirán para calcular distintos valores a partir de él.

AGREGANDO PRODUCTOS:

Dentro del objeto data, vamos a agregar el Array de objetos productos.

```
1 data: function() {  
2   return {  
3     nombre: "Alejandro",  
4     apellido: "Perez",  
5     productos: [  
6       {id: 1, name: "CocaCola", price: 1000},  
7       {id: 2, name: "PepsiCola", price: 1000},  
8       {id: 3, name: "Sprite", price: 1000},  
9       {id: 4, name: "Fanta", price: 1100}  
10    ]  
11  }  
12 },
```

COMPUTED PARA LA CANTIDAD DE PRODUCTOS:

Vamos a crear la primera propiedad computada, para así obtener la cantidad de elementos (en este caso, productos) que tenemos.

```
1 getQuantity() {  
2   return this.productos.length;  
3 },
```

COMPUTED PARA EL PRECIO TOTAL DE PRODUCTOS:

En este caso, vamos a usar la propiedad de Array “reduce”, la cual nos permite obtener un valor, a partir de valores dentro de un Array.

```
1 getTotal() {  
2   let total = this.productos.reduce((total, prod) => {  
3     return total + prod.price  
4   }, 0)  
5   return total  
6 }
```

TEMPLATE:

```
1 <template>  
2   <div>  
3     <h1>Productos {{getQuantity}}</h1>  
4     <ul>  
5       <li v-for="producto in productos" :key="producto.id">  
6         {{producto.name}} {{producto.price}}  
7       </li>  
8     </ul>  
9     <h2>Total: {{getTotal}}</h2>  
10  </div>  
11 </template>
```

Si lo notamos, tanto getQuantity, como getTotal, son llamados en el template como si fueran datos.

Por último, el resultado de este ejercicio sería:

Productos 4

- CocaCola 1000
- PepsiCola 1000
- Sprite 1000
- Fanta 1100

Total: 4100