



Clase 07

Diseño y Programación Web

Materia:

Aplicaciones para
Dispositivos Móviles

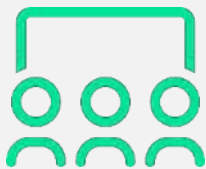
Docente contenidista: GARCIA, Mabel

Revisión: Coordinación

Contenido

Hooks del ciclo de vida de Vue	04
Organizar componentes.....	08
Comunicación de componentes hijos a padres por emit	18
Bibliografía	23

Clase 7



iTe damos la bienvenida a la materia
Aplicaciones para Dispositivos Móviles!

En esta clase vamos a ver los siguientes temas:

- Hooks del ciclo de vida de Vue (creación, montaje, actualización y destrucción).
- Organizar componentes.
- Comunicación entre componentes hijos a padres por emit.

Hooks del ciclo de vida de Vue

En Vue.js, cada instancia de componente pasa por una serie de etapas durante su ciclo de vida, desde la creación hasta la destrucción. Estas etapas proporcionan funciones especiales, ganchos (hooks) que nos permiten ejecutar código en momentos específicos del ciclo de vida del componente.

Las principales etapas del ciclo de vida de Vue.js 3 son:

- **Creación (Creation):**
 - **beforeCreate:** Se ejecuta antes de que se cree la instancia de Vue y antes de que se inicialicen las opciones del componente. Por ejemplo, en este gancho no podemos acceder a las propiedades declaradas en el data, su contenido si lo queremos referenciar será **undefined**.
 - **created:** Se ejecuta después de que se ha creado la instancia de Vue y se han inicializado las opciones del componente. En este punto, la instancia ya está disponible y se pueden acceder a las opciones de datos, métodos, props y eventos.
- **Montaje (Mounting):**
 - **beforeMount:** Se ejecuta antes de que el componente se monte en el DOM.
 - **mounted:** Se ejecuta después de que el componente se ha montado en el DOM. En este punto, el componente ya está visible en la página y se puede acceder al DOM y a los elementos del mismo.
- **Actualización (Updating):**
 - **beforeUpdate:** Se ejecuta antes de que el componente se actualice debido a cambios en los datos o props.
 - **updated:** Se ejecuta después de que el componente se ha actualizado debido a cambios en los datos o props. En este punto, el DOM ya ha sido actualizado para reflejar los cambios.
- **Destrucción (Destroying):**
 - **beforeUnmount:** Se ejecuta antes de que el componente se desmonte y destruya.

- **unmounted:** Se ejecuta después de que el componente se ha desmontado y destruido. En este punto, el componente ya no está en el DOM y se liberan los recursos.

Estos ganchos nos permiten realizar acciones específicas en cada etapa del ciclo de vida del componente, como inicializar datos, realizar llamadas a API, manipular el DOM, limpiar recursos, entre otros.

El gancho **mounted** se usa, por ejemplo, para ejecutar código después de que el componente haya terminado de representarse inicialmente y haya creado los nodos DOM.



```
app.js
1 const app = Vue.createApp({
2   data() {
3     return {
4       dato: '¡Hola desde Vue.js!'
5     };
6   }, // cierre del data
7   mounted() {
8     console.log('El componente se ha montado en el DOM.');
```

9 //Ejecutar código adicional después de que el componente se haya montado en el DOM

10 // Por ejemplo, hacer una solicitud HTTP, ejecutar funciones, etc.

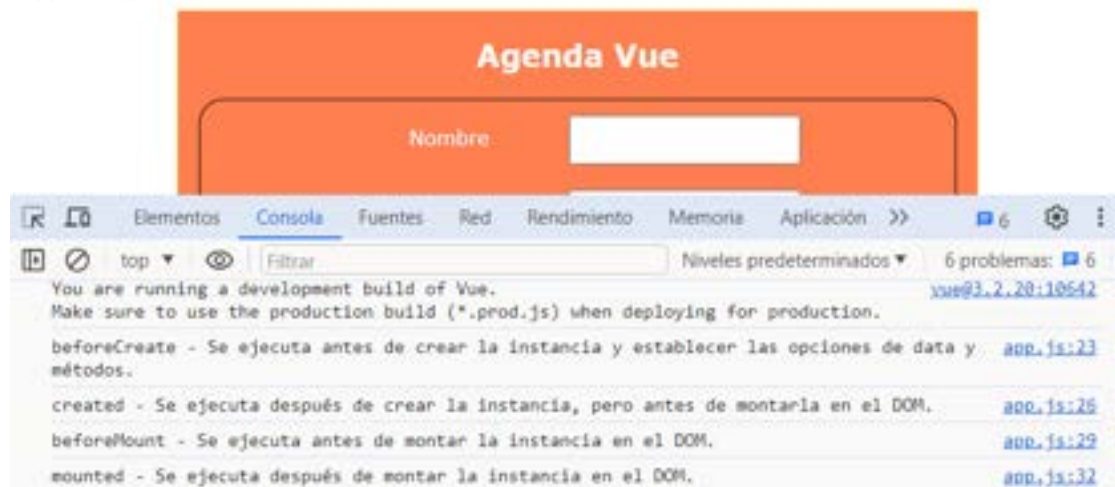
```
11   }
12 });
13
14 app.mount('#app');
```

Cuando Vue.js ejecuta estas funciones especiales, les pasa el contexto **this**. Esto significa que dentro de estas funciones, **podemos acceder a las propiedades y métodos de nuestro componente usando this**.

Hay que tener cuidado al usar funciones de flecha (`() => {}`). Las funciones de flecha **no tienen su propio this**, sino que **usan el this del código que las rodea**. Esto puede causar problemas, por lo que es mejor usar la sintaxis regular de funciones (`function() {}`) para definir los hooks del ciclo de vida.

Por ejemplo, si nuestro proyecto tuviera dos archivos .html, si navegamos de uno a otro, y llamamos a estas funciones en la instancia root, nos daremos cuenta que estas etapas se ejecutan, cada vez que se cargan los archivos.

Ejemplo-la instancia root



Cabe aclarar que no sólo la instancia principal tiene este ciclo de vida, los componentes que creamos también pasan por estos ganchos.

Por ejemplo, la **navegación** es un componente, el **formulario** es otro componente y la **lista** de los contactos que el usuario vaya agregando es otro componente. Si en cada componente llamamos a estos ganchos veríamos como pasan por el ciclo.

Para simplificar el ejemplo, mostraremos en cada componente un `console.log` imprimiendo el gancho **created**.

Ejemplo-la instancia root



Como se puede observar, en la consola los console log que hacen referencia a la instancia principal, nos indica que están en el archivo **app.js** y **cada uno de los componentes, ahora están desarrollados en archivos .js distintos**, facilitando también el debug de errores, si los hubiese.

Organizar componentes

Hasta ahora hemos trabajado la lógica dentro de un solo archivo `app.js` donde declaramos la instancia principal y empezamos a crear los primeros componentes también dentro de este archivo. Y también estábamos trabajando con un solo archivo `.html`.

Pero cuando nuestro sistema empieza a crecer y empezamos a desarrollar mayores funcionalidades, tenemos que tener nuestro código más estructurado y orientado a lo que veremos en un entorno de desarrollo más profesional, debemos acostumbrarnos a otro tipo de organización.

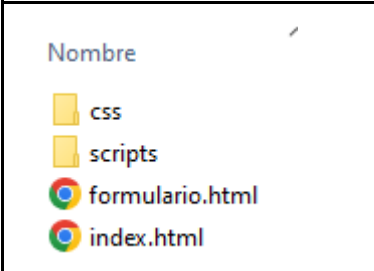
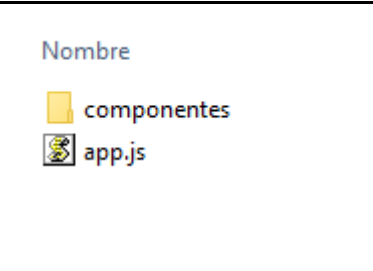
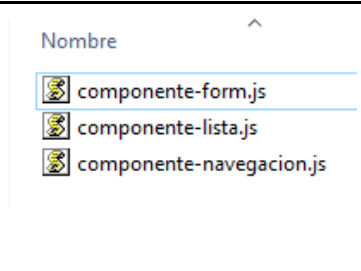
Las aplicaciones suelen crecer, suelen escalar en funcionalidades y no podemos tener todo en un mismo archivo, necesitamos una **mejor organización** de estos elementos.

Es importante comprender que hacer un código modular nos ayuda a realizar mantenimiento y desarrollar escalabilidad.

Este tipo de enfoque nos ayudará a entender la **separación de responsabilidades**, donde la instancia principal y los componentes son elementos separados, con sus propias responsabilidades y ciclos de vida. Esta separación es fundamental en la arquitectura de Vue.

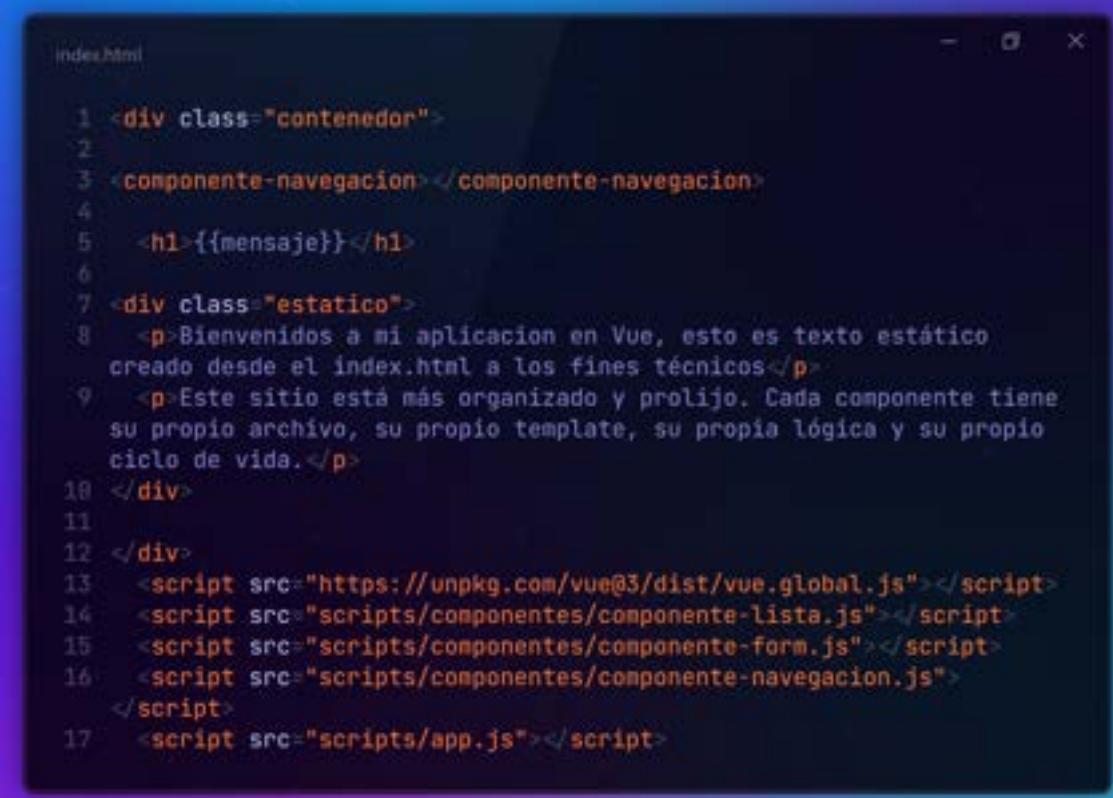
Además, otra de las ventajas que nos ofrece el trabajar de esta forma es que nos ayudará a **mejorar la legibilidad** y se nos hará más fácil cuando tengamos que encontrar o modificar partes específicas del código.

Veamos nuestro ejemplo anterior cómo está estructurado nivel físico.

root de la carpeta	carpeta scripts	carpeta componentes
		

Veamos los cambios a nivel código, para poder hacer esta separación de archivos.

En nuestro index, tendremos la incorporación del **componente navegación**, que es un elemento que estará en **ambos archivos**, en el **index.html** y en **formulario.html**. Además, presenta un enlace de datos a la propiedad **mensaje** que será nuestro **h1**. Esta propiedad está inicializada en la instancia root.



```
index.html
1 <div class="contenedor">
2
3 <componente-navegacion></componente-navegacion>
4
5 <h1{{mensaje}}</h1>
6
7 <div class="estatico">
8   <p>Bienvenidos a mi aplicacion en Vue, esto es texto estático
   creado desde el index.html a los fines técnicos</p>
9   <p>Este sitio está más organizado y prolijo. Cada componente tiene
   su propio archivo, su propio template, su propia lógica y su propio
   ciclo de vida.</p>
10 </div>
11
12 </div>
13 <script src="https://unpkg.com/vue@3/dist/vue.global.js"></script>
14 <script src="scripts/componentes/componente-lista.js"></script>
15 <script src="scripts/componentes/componente-form.js"></script>
16 <script src="scripts/componentes/componente-navegacion.js">
</script>
17 <script src="scripts/app.js"></script>
```

Antes del cierre del body tenemos que respetar el orden de la llamada a los archivos. Primero, por medio de la etiqueta script **referenciamos al cdn para incorporar Vue3**, luego cada **uno de los componentes** que vamos a utilizar para el proyecto y finalmente referenciamos al **archivo app.js** que tendrá la **creación de la instancia principal y el registro de los componentes para poder montar correctamente la instancia**.



En el archivo **app.js**, tendremos dentro del data de la instancia la propiedad **mensaje** que referenciamos tanto en el **index.html** como en el archivo **formulario.html** por medio del enlace de datos **{{mensaje}}**.

Luego **registramos** los componentes **ComponenteForm** y **ComponenteLista** y **ComponenteNavegacion** (que deben estar definidos en archivos separados). Esto se hace con el método **app.component()**.

```

1  const app = Vue.createApp({
2    data() {
3      return {
4        mensaje: "Ejemplo-la instancia root"
5      }
6    },
7    app.component('componente-form', ComponenteForm)
8    app.component('componente-lista', ComponenteLista)
9    app.component('componente-navegacion', ComponenteNavegacion)
10
11  app.mount('.contenedor');

```

El **primer argumento** de **app.component()** es una **cadena de texto que representa el nombre con el cual podremos usar el componente en nuestro HTML**. El **segundo argumento** es la **definición del componente**, que en este caso es la **constante** que creamos en el archivo js donde creamos en componente.

Entonces, cuando hacemos **app.component('componente-form, ComponenteForm)**, estamos diciendo: "Vue, registrá un nuevo componente llamado '**componente-form**' y usá esta definición de

componente (**ComponenteForm**) que tengo acá".

De esta forma, Vue sabe que cuando encuentre una etiqueta **<componente-form>** en nuestro HTML, debe renderizar el componente que registramos con esa definición.

Después de registrar los componentes, podemos usarlos en nuestro HTML como etiquetas personalizadas.

Esta separación en archivos y el registro de componentes nos permite organizar mejor nuestro código, reutilizar componentes fácilmente y mantener una estructura más escalable a medida que nuestras aplicaciones crezcan.

Retomemos el **componente-navegacion** y cómo lo tenemos creado dentro del archivo .js. Podríamos sólo tener el template con las etiquetas html necesarias para hacer la navegación, pero no sería un elemento funcional en donde Vue tenga participación, ni tampoco sería dinámico.

En este caso, vamos a inicializar dentro de la función data un array de **links** que **contendrá 2 objetos** (porque son 2 archivos html que debemos hacer navegables). Estos objetos tendrán 2 propiedades, por un lado, **el texto que el usuario verá** y por otro, el **href** que será la **URL** a la que apunta el enlace.

En el **template**, vamos a usar la directiva **v-for** para **renderizar dinámicamente los enlaces de la navegación a partir del array links**.

También agregamos una **clase dinámica por medio de v-bind**, llamada **activa** al enlace activo usando el método **esLinkActivo**.

```
componente-navegacion.js

1 const ComponenteNavegacion = {
2   data() {
3     return {
4       links: [
5         { text: 'Inicio', href: 'index.html' },
6         { text: 'Formulario', href: 'formulario.html' }
7       ] // Array para almacenar los enlaces de navegacion
8     }
9   },
10  template: `
11    <div>
12      <nav>
13        <ul>
14          <li v-for="item in links" :key="item.href">
15            <a :href="item.href" :class="{ activa:
16              esLinkActivo(item.href) }">{{ item.text }}</a>
17          </li>
18        </ul>
19      </nav>
20    </div>
21  `,
22  created() {
23    console.log("%cEl componenteNavegación se creó.", 'color: red;
24    background:yellow');
25  },
26  methods: {
27    esLinkActivo(href) {
28      const currentPath = window.location.pathname;
29      return currentPath.includes(href);
30    }
31  }
32 }
```

El método **esLinkActivo** lo usaremos para determinar si un enlace de navegación está activo o no, es decir, si corresponde a la página actualmente visitada por el usuario.

Funciona de la siguiente manera:

window.location.pathname devuelve la ruta actual de la URL del navegador, por ejemplo, **"/formulario.html"** si estamos en la página del formulario.

Por medio del método **includes** vamos a verificar **si la ruta actual incluye la ruta del enlace proporcionada como argumento href**. Si es así, significa que ese enlace está activo y se le aplicará nuestra clase CSS **'activa'** para resaltarlos visualmente como activo.

Ésta debe estar desarrollada en la hoja de estilos: **.activa**{color: red; background: yellow;}

En cuanto al **ComponenteForm** posee en su template el formulario, tal cual como lo hemos visto en la clase de localStorage. Sólo hemos optimizado el comportamiento que tenía.

Repasemos lo que hace el código ahora:

Data del Componente:

- El componente tiene una propiedad llamada **"dato"** que es un **objeto con campos para nombre, apellido y teléfono**, inicializados como cadenas vacías.
- También tiene una propiedad **"arr"** para almacenar los contactos guardados en localStorage y una propiedad **"mensaje"** para mostrar un mensaje al usuario.

Template del Componente:

- Incluye un **formulario** con campos para **nombre, apellido y teléfono**, vinculados a las propiedades del objeto **"dato"** usando **v-model**.
- Al hacer click en el botón **"Guardar"**, se llama al **método "guardar"** para almacenar los datos del formulario en **localStorage**.
- Usa un condicional mediante la directiva **v-if** para mostrar el componente llamado **"componente-lista"** si hay datos guardados en **"arr"**, de lo contrario, muestra el mensaje en la propiedad **"mensaje"**.

Método "guardar":

- Obtiene los datos guardados previamente en localStorage, si los hay.
- Agrega los nuevos datos del formulario al array **"arr"**.
- Guarda el array actualizado en localStorage y reinicia los campos del formulario a valores vacíos.

componente-formulario.js

```
1 const ComponenteForm = {
2   data: function(){
3     return {
4       dato: { nombre:"", apellido:"", telefono:"" },
5       arr:[],
6       mensaje:""
7     }
8   },
9   template:
10   `<div class="form-data">
11     <h2>Agenda Vue</h2>
12     <form @submit.prevent>
13       <div>
14         <label>Nombre</label>
15         <input type="text" v-model= "dato.nombre" />
16       </div>
17       <div>
18         <label>Apellido</label>
19         <input type="text" v-model= "dato.apellido" />
20       </div>
21       <div>
22         <label>Teléfono</label>
23         <input type="tel" v-model= "dato.telefono" />
24       </div>
25       <button @click="guardar(dato)">Guardar</button>
26     </form>
27     <template v-if="this.arr.length > 0">
28       <componente-lista :arr="this.arr"></componente-lista>
29     </template>
30     <template v-else>
31       <p>{{mensaje}}</p>
32     </template>
33   </div>`,
34   created() {
35     this.mostrar()
36     console.log('%El componenteForm se creó.', 'color:white;
37     background:blue');
38   },
39   methods:{
40     guardar(dato){
41       var localData = localStorage.getItem("local");
42       this.arr = localData ? JSON.parse(localData) : [];
43
44       this.arr.push({ nombre: dato.nombre, apellido: dato.apellido,
45         telefono: dato.telefono});
46       localStorage.setItem("local", JSON.stringify(this.arr));
47       this.dato = {nombre:"", apellido:"", telefono:""};
48     },
49     mostrar(){
50       if (localStorage.getItem("local")) {
51         this.arr= JSON.parse(localStorage.getItem("local"))
52       }else{
53         this.mensaje = "No hay datos que mostrar"
54       }
55     }
56   }
57 }
```


Método "mostrar":

- Verifica si hay datos almacenados en localStorage.
- Si existen, carga los datos en el array "arr"; de lo contrario, muestra el mensaje "No hay datos que mostrar" en la propiedad "mensaje".

Ciclo de Vida "created":

- Al crear el componente, se llama al método "**mostrar**" para cargar los datos almacenados previamente en localStorage y mostrarlos en caso de que existan.

	
Si hay datos los muestra el ComponenteLista .	Sin datos, mostramos la propiedad mensaje , debajo del formulario.

Por último, nos queda el **ComponenteLista** que es el **componente hijo** del **ComponenteForm**. Este componente no tiene methods, no tiene funcionalidad sólo recibe por **props** el array del formulario y se encarga de recorrerlo por medio de la directiva **v-for**.

```
componente-lista.js

1 const ComponenteLista = {
2   // Definición del componente
3   props: ["arr"],
4   template: `
5     <div class="ver">
6       <h1>Datos ingresados</h1>
7       <p v-for="x in arr">
8         {{x.nombre}}: {{x.apellido}}: {{x.telefono}}</p>
9     </div>`,
10 }
```

Finalmente, en nuestro archivo **formulario.html** tendremos la siguiente estructura:

```
formulario.html

1 <div class="contenedor">
2
3   <componente-navegacion></componente-navegacion>
4
5   <h1>{{mensaje}}</h1>
6
7   <componente-form></componente-form>
8
9 </div>
10
11 <script src="https://unpkg.com/vue@3/dist/vue.global.js"></script>
12 <script src="scripts/componentes/componente-lista.js"></script>
13 <script src="scripts/componentes/componente-form.js"></script>
14 <script src="scripts/componentes/componente-navegacion.js"></script>
15 <script src="scripts/app.js"></script>
```


El **componente-navegacion** incorporado, el **h1** proveniente de la instancia principal y el **componente-form**. Además, los scripts que se necesitan para el correcto funcionamiento del proyecto.

Entender los hooks del ciclo de vida de Vue.js 3 y organizar los componentes en archivos separados son prácticas esenciales para desarrollar aplicaciones más estructuradas y fáciles de mantener. Estas estrategias mejoran la legibilidad y la escalabilidad del código, facilitando la gestión y reutilización de componentes en el proyecto.

Comunicación de componentes hijos a padres por emit

Ya hemos visto como la creación de componentes facilita el código modular, y la división de responsabilidades. Y cómo los componentes padres pueden comunicarse, para pasar información a un componente hijo cuando hay una relación lógica entre ellos. Además, sabemos que los padres pasan **props** y estas **props** se convierten en **atributos personalizados de los componentes hijos**.

Todos las **props** forman una **vinculación unidireccional** entre la propiedad del hijo y la del padre: **cuando la propiedad del padre se actualiza, fluye hacia el hijo, pero no al revés**. Esto **evita que los componentes hijos muten accidentalmente el estado del padre**, lo que puede hacer que el flujo de datos de nuestro sistema sea más difícil de entender.

Además, **cada vez que se actualice el componente padre, todos los props del componente hijo se actualizarán con el último valor**. Esto significa que **no** debemos intentar mutar una **prop** dentro de un componente hijo. Si lo hacemos, Vue lo marca en la consola.

Suele haber dos casos en los que es tentador mutar una prop:

1. **La prop es usada para pasar un valor inicial; el componente hijo quiere utilizarla más adelante como una propiedad de datos local**. En este caso, es mejor definir una propiedad de datos local que utilice la prop como valor inicial
2. **La prop se pasa como un valor primario que necesita ser transformado**. En este caso, es mejor definir una propiedad computada usando el valor de la prop.

La mejor práctica es evitar estas mutaciones a menos que el padre y el hijo estén estrechamente acoplados por diseño. La mayor parte de las veces, el hijo debería emitir un evento para que el padre realice la mutación.



Los hijos tienen otra forma para enviar datos hacia un componentes padres. Lo hacen **emitiendo eventos**.

Vue.js permite que los componentes hijos emitan eventos utilizando el **método `this.$emit('nombreEvento', datos)`**. Esto significa que un componente hijo puede notificar a su componente padre sobre un evento específico junto con cualquier dato relevante.

Por ejemplo, supongamos que tenemos un formulario dentro de un componente hijo y queremos notificar a la instancia root (padre del componente) cuando se envíe el formulario. Podríamos emitir un evento de esta manera:

`this.$emit('formulario-enviado', { nombre: this.nombre, apellido: this.apellido });`

```
componente-formulario.js
1 const ComponenteFormulario = {
2   template: `
3     <form @submit.prevent="enviarFormulario">
4       <label for="nombre">Nombre:</label>
5       <input type="text" id="nombre" v-model="nombre">
6       <label for="apellido">Apellido:</label>
7       <input type="text" id="apellido" v-model="apellido">
8       <button type="submit">Enviar</button>
9     </form>
10 `
11   data() {
12     return {
13       nombre: '',
14       apellido: ''
15     };
16   },
17   methods: {
18     enviarFormulario() {
19       this.$emit('formulario-enviado', { nombre: this.nombre, apellido: this.apellido });
20       this.nombre = '';
21       this.apellido = '';
22     }
23   }
24 };
```

En este ejemplo, el **primer argumento** que le pasamos a **\$emit** es el **nombre del evento** que vamos a emitir. En este caso, estamos **emitiendo un evento personalizado llamado 'formulario-enviado'**.

Después del nombre del evento, **pasamos un objeto como segundo argumento de \$emit**. Este objeto contiene los datos que vamos a enviar junto con el evento.

Hay que tener presente que Vue permite emitir cualquier tipo de dato válido en Javascript como argumento de un evento personalizado.

En este caso, estamos enviando los datos del formulario, específicamente el nombre y el apellido. Estas variables **this.nombre** y **this.apellido** hacen referencia a los datos almacenados en las variables **nombre** y **apellido** dentro del componente de sus respectivos v-model.

Actualmente requiere que se declare el evento personalizado:



```
componente-formulario.js
1 emits: ['formulario-enviado']
```

Este **evento puede ser capturado y manejado por la instancia root para realizar acciones basadas en estos datos**, como por ejemplo, procesar el formulario y guardar la información en algún lugar.

En nuestro archivo app.js tendremos:



```
app.js
1 const app = Vue.createApp({
2   methods: {
3     manejarFormularioEnviado(datos) {
4       console.log('Datos del formulario:', datos);
5     }
6   }
7 });
8 app.component('componente-formulario', ComponenteFormulario);
9
10 app.mount('#app');
```

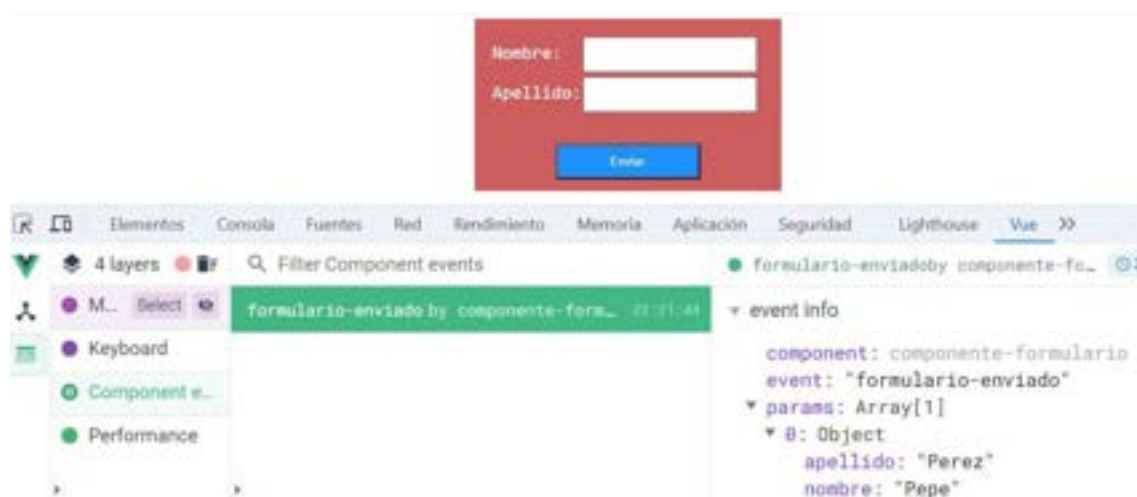
En la instancia principal tenemos el **método** que recibe un parámetro **datos**, que se espera contenga la información enviada desde el formulario.

Finalmente, en nuestro HTML, insertamos el componente `componente-formulario` y escuchamos el evento **formulario-enviado**:

```
index.html
1 <div id="app">
2   <componente-formulario
3     @formulario-enviado="manejarFormularioEnviado">
4   </componente-formulario>
5 </div>
```

El **@formulario-enviado** es el **evento** que el componente **componente-formulario** emitirá cuando se envíe el formulario. **Cuando este evento se emita, se ejecutará la función `manejarFormularioEnviado` en la instancia de Vue**, permitiendo la comunicación entre la instancia root y su componente hijo.

Desde el panel de Vue.js Devtools podemos ver la emisión del evento del componente, con sus respectivos datos enviados:

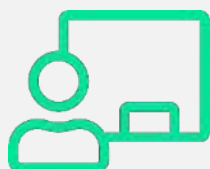


¡Estamos listos para implementar estos nuevos conceptos!



Hemos llegado así al final de esta clase en la que vimos:

- Hooks del ciclo de vida de Vue (creación, montaje, actualización y destrucción)
- Organizar componentes.
- Comunicación entre componentes hijos a padres por emit



Te esperamos en la **clase en vivo** de esta semana.
No olvides realizar el **desafío semanal**.
¡Hasta la próxima clase!

Bibliografía

Vue.js. (s. f.). Hooks del ciclo de vida. En Documentación Oficial de Vue.js. Recuperado de

<https://vuejs.org/guide/essentials/lifecycle.html#lifecycle-hooks/>

Vue.js. (s. f.). Eventos. En Documentación Oficial de Vue.js.

Recuperado de <https://vuejs.org/guide/components/events.html/>