



Clase 03

Diseño y Programación Web

Materia:

Aplicaciones para
Dispositivos Móviles

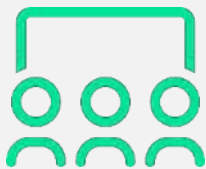
Docente contenidista: GARCIA, Mabel

Revisión: Coordinación

Contenido

Directiva v-on	04
Modificadores de Eventos	07
Modificadores de Teclas.....	08
Enlace de datos.....	11
Manipulación de Datos con Métodos en Formularios Vue.....	13
Bibliografía	23

Clase 3



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

En esta clase vamos a ver los siguientes temas:

- Directiva v-on.
- Modificadores de eventos.
- Modificadores de teclas.
- Enlaces de datos.(directiva v-model).
- Manipulación de datos con métodos en formulario (agregar, borrar, validar campo, contar elementos, establecer clases CSS con operador ternario).

Directiva v-on

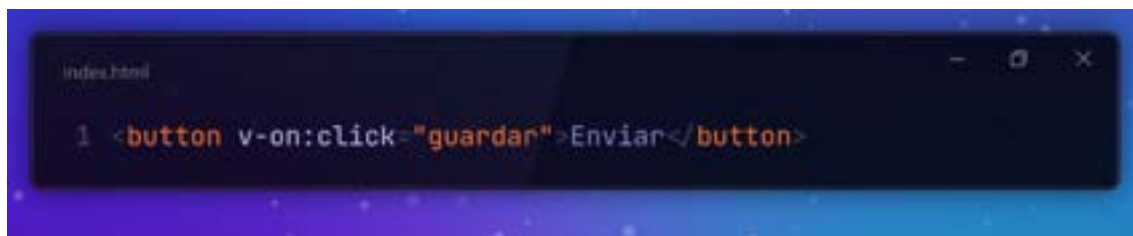
La directiva v-on se utiliza para escuchar eventos del DOM y ejecutar expresiones o métodos en respuesta a ellos.

Estos eventos pueden vincularse a cualquier elemento del HTML, un botón, un div, un formulario, etc.

La sintaxis es **v-on:nombreEvento="método"**.

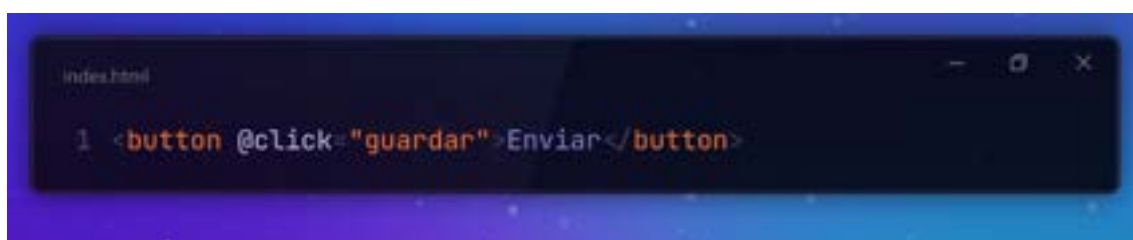
El **nombre del evento** puede ser cualquiera de los ya conocidos por nosotros en javascript, por ejemplo el click, mouseover, keypress, focus, etc.

Desde la vista, desde nuestro index.html tendremos una etiqueta a la cual asociaremos esta directiva para poder escuchar el evento click y asociar un método, una función.

A screenshot of a code editor window titled 'index.html'. The code shows a single line: `1 <button v-on:click="guardar">Enviar</button>`. The text is color-coded: `<button` is blue, `v-on:click="guardar"` is orange, and `>Enviar</button>` is blue.

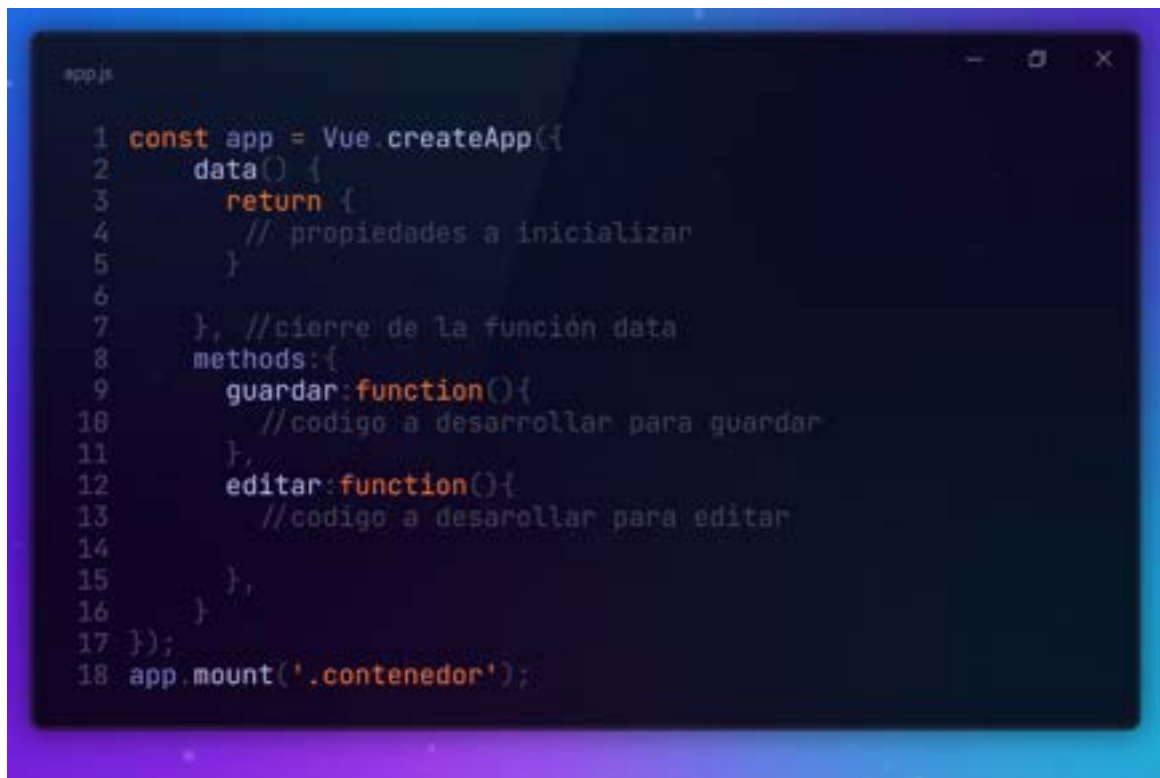
La directiva **v-on**, tiene una forma abreviada para escribirse.

La sintaxis es **@nombreEvento="metodo"**.

A screenshot of a code editor window titled 'index.html'. The code shows a single line: `1 <button @click="guardar">Enviar</button>`. The text is color-coded: `<button` is blue, `@click="guardar"` is orange, and `>Enviar</button>` is blue.

Esta función, este método **"guardar"** tiene que estar desarrollado desde nuestro archivo app.js.

Para que podamos trabajarlo desde Vue, necesitamos indicar dentro de las opciones de la instancia root una nueva propiedad llamada **methods**.



```
1 const app = Vue.createApp({
2   data() {
3     return {
4       // propiedades a inicializar
5     }
6   }, // cierre de la función data
7   methods: {
8     guardar: function() {
9       // código a desarrollar para guardar
10     },
11     editar: function() {
12       // código a desarrollar para editar
13     },
14   },
15 },
16 );
17 app.mount('.contenedor');
```

Dentro del objeto **methods**, crearemos las funciones, es decir, tantos métodos como queramos separados por comas.

Es muy importante tener presente que cualquier variable, o método que escribimos en el HTML, debe estar declarado en el código de nuestro archivo app.js, caso contrario la consola arrojará error.

Por ejemplo, tomemos el caso de un formulario. El comportamiento del **evento submit**, al enviar los datos es recargar la página. Para que nosotros podamos manipular este comportamiento y no perder esos datos que contiene el formulario, Vue nos da la posibilidad de trabajar con **modificadores de eventos**.

En casos más simples, podemos optar por utilizar manejadores en línea (inline handlers), por ejemplo:

```
app.js

1 data() {
2   return {
3     contador: 0
4   }
5 }
```

En el index tendremos el siguiente código:

```
index.html

1 <button @click="contador++">Sumar 1</button>
2 <p>Sumar: {{ contador }}</p>
```

En este caso, el evento click está directamente vinculado a una expresión en línea (contador++), **incrementando la variable contador** al hacer click en el botón.

Modificadores de Eventos

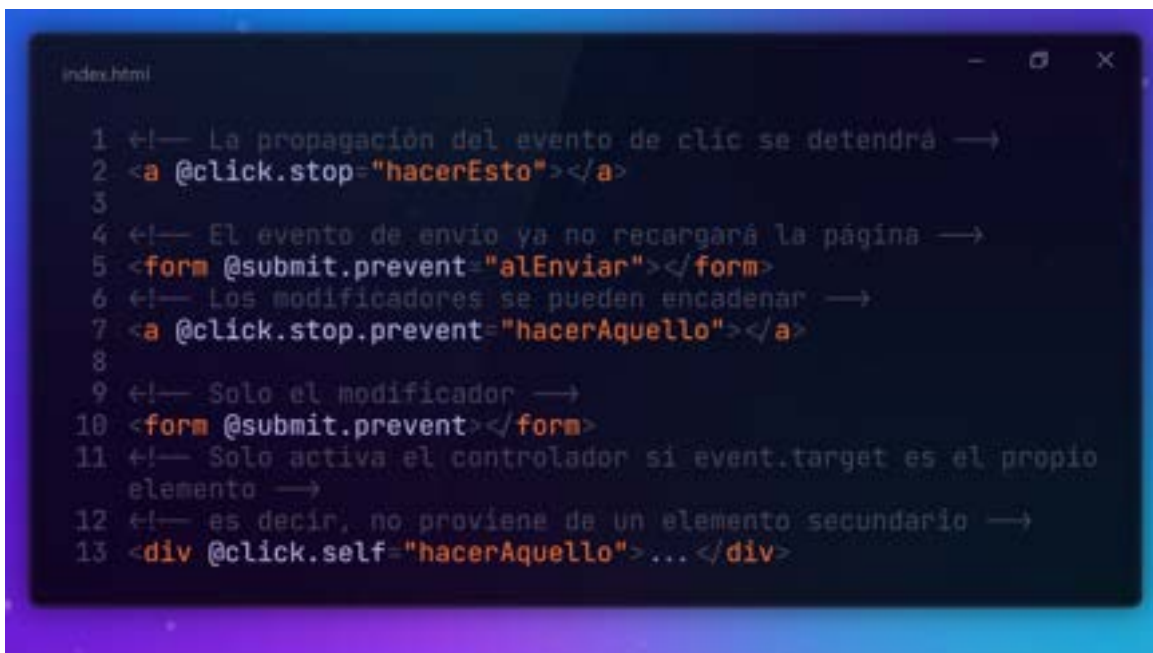
Suele ser muy habitual cuando trabajamos con eventos llamar a **event.preventDefault()** o **event.stopPropagation()** dentro de los controladores de eventos. Aunque podemos hacer esto fácilmente dentro de los métodos, sería mejor si **los métodos pudieran ser puramente sobre lógica de datos, en lugar de tener que lidiar con detalles de eventos DOM.**

Para solucionar este problema, Vue proporciona **modificadores de eventos para v-on.**

Por ejemplo:

- .stop
- .prevent
- .self
- .capture
- .once
- .passive

La sintaxis para usar estos **modificadores** es muy sencilla, **después del nombre del evento se coloca un punto.**



```
index.html
1 <!-- La propagación del evento de clic se detendrá -->
2 <a @click.stop="hacerEsto"></a>
3
4 <!-- El evento de envío ya no recargará la página -->
5 <form @submit.prevent="alEnviar"></form>
6 <!-- Los modificadores se pueden encadenar -->
7 <a @click.stop.prevent="hacerAquellos"></a>
8
9 <!-- Solo el modificador -->
10 <form @submit.prevent></form>
11 <!-- Solo activa el controlador si event.target es el propio elemento -->
12 <!-- es decir, no proviene de un elemento secundario -->
13 <div @click.self="hacerAquellos">...</div>
```

Retomando el caso del formulario, el modificador **".prevent"** se utiliza para prevenir el comportamiento predeterminado de un evento, en este caso, **evitar que el formulario recargue la página al ser enviado.**

```
index.html
1 <form v-on:submit.prevent="guardar">
2
3 <!-- controles del form -->
4
5 </form>
```

Modificadores de Teclas

Cuando escuchamos **eventos del teclado**, a menudo necesitamos buscar teclas específicas. Vue permite agregar **modificadores de teclas** para **v-on** o **@** cuando **escucha estos eventos**.

Podríamos asignar el evento **keyup** (se dispara cuando el usuario deja de presionar una tecla) si el usuario presiona determinada tecla.

```
index.html
1 <!-- Solo llama a 'submit' cuando la tecla es 'Enter' -->
2 <input @keyup.enter="submit" />
```

Se puede usar cualquier nombre de tecla válido expuesto a través de **KeyboardEvent.key** como modificadores convirtiéndolos a **kebab-case**.

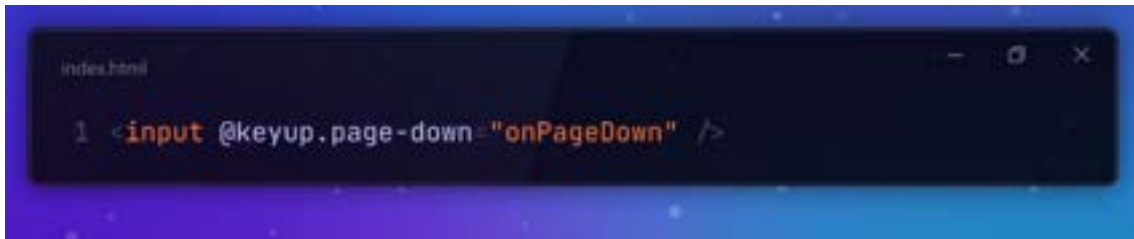
KeyboardEvent.key es una propiedad que se utiliza en JavaScript para **representar la clave (tecla) que fue presionada en un evento de teclado**.

Cuando se presiona una tecla en el teclado, **se genera un evento** de teclado (keydown, keyup o keypress). **Estos eventos tienen varias propiedades** que brindan información sobre la tecla presionada y **KeyboardEvent.key** es una de ellas.

El **valor de KeyboardEvent.key es una cadena de texto** que representa la clave presionada. Por ejemplo:

- Si se presiona la tecla "A", el valor de **KeyboardEvent.key** será "a".
- Si se presiona la tecla "Enter", el valor de **KeyboardEvent.key** será "Enter".

- Si se presiona la tecla de flecha hacia arriba, el valor de `KeyboardEvent.key` será "ArrowUp".



En el ejemplo anterior, sólo se llamará al controlador si **\$event.key** es igual a '**PageDown**'.

Vue ofrece poder trabajar con **alias** para las teclas más utilizadas:

- `.enter`
- `.tab`
- `.delete` (captura ambas teclas "Delete" y "Backspace")
- `.esc`
- `.space`
- `.up`
- `.down`
- `.left`
- `.right`

Vue también nos da la posibilidad de trabajar **con teclas modificadoras del sistema**.

Podemos usar los siguientes modificadores para activar detectores de eventos del mouse o del teclado sólo cuando se presiona la tecla modificadora correspondiente:

- `.ctrl`
- `.alt`
- `.shift`
- `.meta` (Tecla Windows- tecla Command en Mac)

Por ejemplo:

A screenshot of a code editor window titled 'index.html'. The code is as follows:

```
1 +!— Alt + Enter —>
2 <input @keyup.alt.enter="borrar" />
3
4 +!— Ctrl + Click —>
5 <div @click.ctrl="seleccionar">+!—Contenido —></div>
```

La documentación nos aclara que las **teclas de modificación son distintas de las teclas normales**. Cuando se utilizan con eventos de "keyup", estas teclas deben estar presionadas en el momento en que se emite el evento.

En otras palabras, "**keyup.ctrl**" sólo se activará si soltamos una tecla mientras mantenemos presionado ctrl. No se activará si soltamos únicamente la tecla ctrl.

Cabe destacar, que **este tipo de combinaciones suelen encontrarse en sistemas donde se necesita una interacción específica**, como en juegos o aplicaciones multimedia para controlar eventos precisos.

Enlace de datos

Cuando trabajamos con formularios tenemos que tener presente cómo vamos a tratar los datos.

Hay un concepto fundamental en Vue que tiene que ver con el **enlace de datos**.

Hasta este momento, hemos visto cómo los datos, inicializados desde el modelo, fluyen de manera unidireccional hacia la vista, lo que conocemos como un **"enlace de una vía"**. En este proceso, **los cambios en el modelo se reflejan en la vista, pero no al revés**. Este tipo de enlace es eficaz cuando deseamos **mostrar información** sin necesidad de actualizaciones constantes desde la vista al modelo.

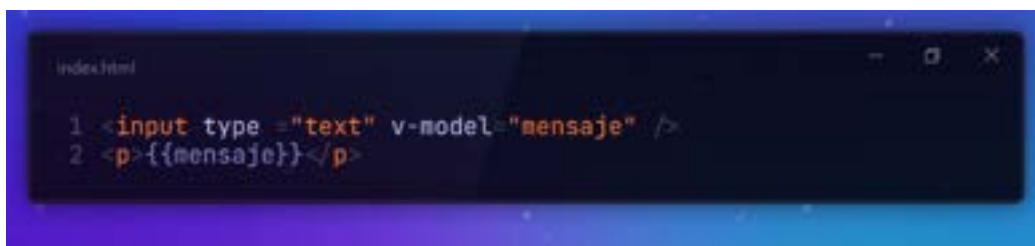
Para que el **enlace de datos de 2 vías** ocurra necesitamos de la participación del usuario.

Con la **directiva v-model** podemos **crear enlace de datos bidireccionales**. Esta directiva se usa en controles de tipo **input**, **textarea** y **select** en **formularios**.

La directiva busca automáticamente la manera correcta de actualizar el elemento según el tipo de entrada.

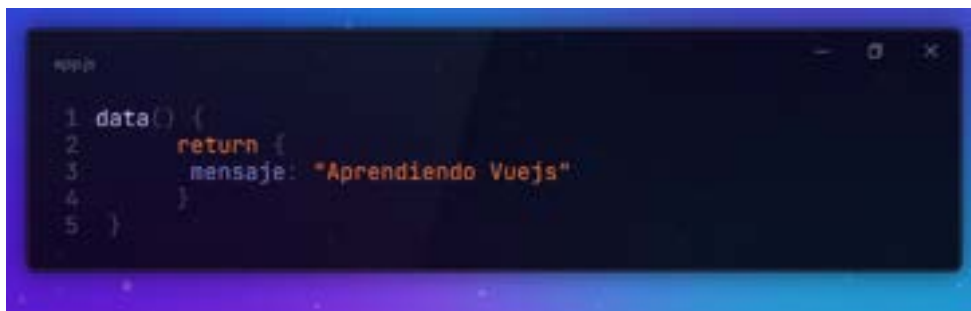
Al usar v-model, le decimos a Vue qué variable debe unirse con esa entrada, en este caso el "mensaje".

En el index tendremos la directiva **v-model** que mostrará su valor dentro del input y por otro lado la **interpolación** para que podamos notar sus cambios.



```
index.html
1 <input type="text" v-model="mensaje" />
2 <p>{{mensaje}}</p>
```

En nuestra lógica, tendremos inicializada la variable con el contenido.

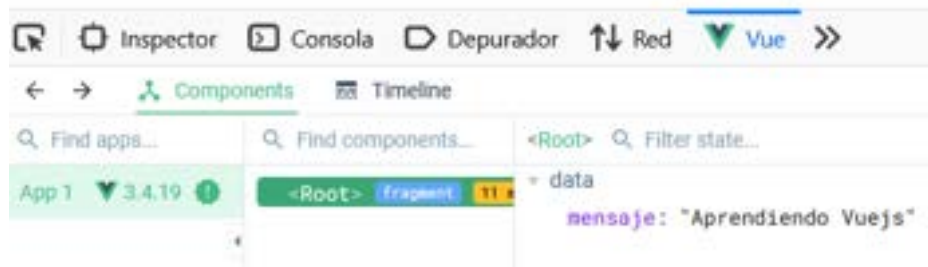


```
script.js
1 data() {
2   return {
3     mensaje: "Aprendiendo Vuejs"
4   }
5 }
```

En el navegador veremos representado el contenido:

Aprendiendo Vuejs

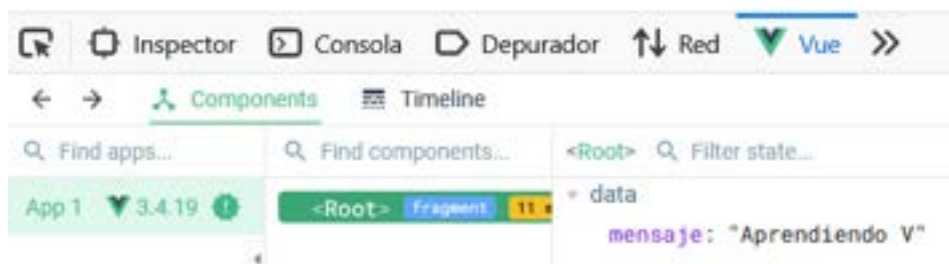
Aprendiendo Vuejs



Entonces, la directiva v-model la tenemos vinculada a la variable `mensaje` del modelo. Con esto, **cualquier cambio realizado en el campo de texto se refleja automáticamente en la variable `mensaje`.**

Aprendiendo V

Aprendiendo V



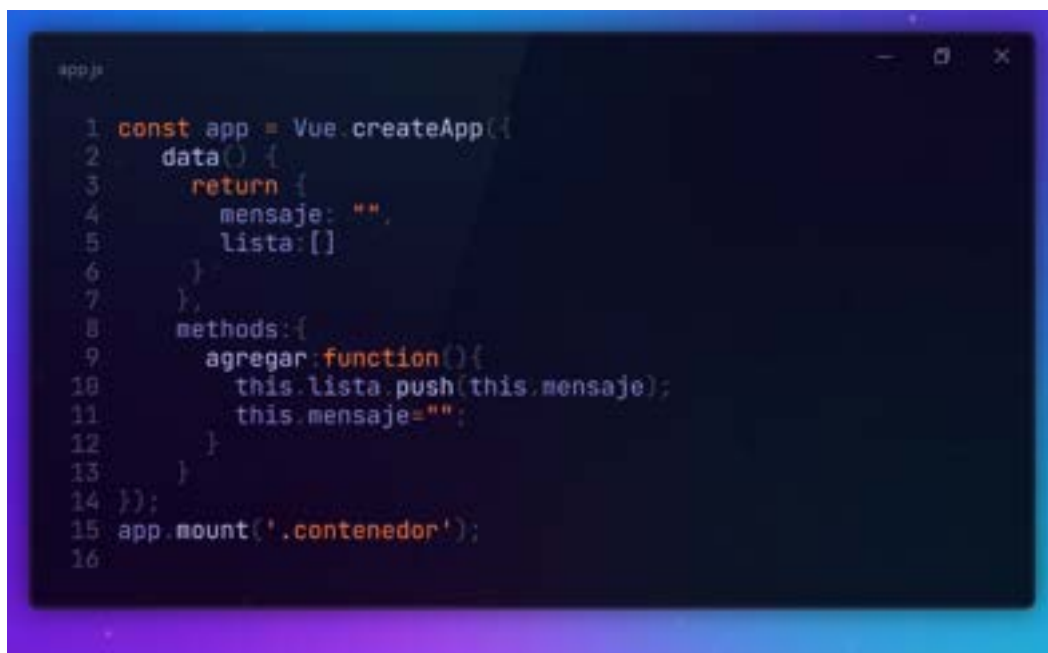
Si borramos parte del contenido, veremos cómo se va actualizando la vista y el modelo.
Más adelante iremos profundizando en estos conceptos reactivos.

Manipulación de Datos con Métodos en Formularios Vue

Los formularios son una herramienta fundamental para facilitar la interacción con el usuario. Cuando deseamos que el usuario ingrese datos y visualice la información a medida que la proporciona, es necesario determinar qué tipo de estructura tendrán (Arrays, objetos, etc).

Como punto de partida, es fundamental recordar que las variables deben ser inicializadas dentro de la función data. Pueden tener valores predeterminados o valores vacíos. Además, cualquier función que creamos debe ubicarse dentro de la propiedad "methods".

Por ejemplo, nuestro primer método va a ser el responsable de ir guardando cada mensaje que el usuario escriba en el input.

A screenshot of a code editor with a dark background and blue and orange syntax highlighting. The code is written in JavaScript and uses the Vue.js framework. It defines a Vue application with a data object containing 'mensaje' (an empty string) and 'lista' (an empty array). A method named 'agregar' is defined, which pushes the current 'mensaje' value into the 'lista' array and then resets 'mensaje' to an empty string. The application is mounted to a DOM element with the class 'contenedor'.

```
1 const app = Vue.createApp({
2   data() {
3     return {
4       mensaje: "",
5       lista: []
6     }
7   },
8   methods: {
9     agregar: function() {
10       this.lista.push(this.mensaje);
11       this.mensaje = "";
12     }
13   }
14 });
15 app.mount('.contenedor');
16
```

Partimos de una propiedad inicializada desde el modelo y vamos agregar cada **mensaje** nuevo que el usuario escriba a un **array** llamado **"lista"**.

Para poder acceder a las propiedades de la función data desde las funciones, tenemos que usar **"this"**. Más adelante veremos que también usaremos **"this"** para poder acceder a métodos.

Por medio del método **.push** de javascript, vamos a ir agregando estos mensajes al final del **array lista**. Además, vaciaremos el valor del input.

Con el modificador **.prevent** anulamos el comportamiento por defecto del formulario cuando se realiza el submit.

Cada vez que el usuario **deje de presionar la tecla "enter" (evento keyup)**, se ejecutará la función **agregar** y el **mensaje** se agrega por medio del **push** al array **lista**.

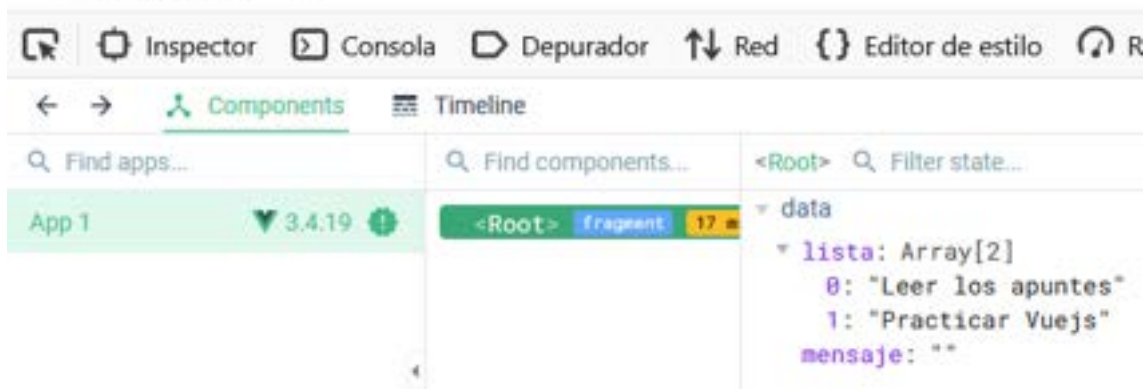
Por medio de la **directiva v-for**, recorreremos cada ítem que posea el array y además mostraremos su posición dentro del array con el segundo argumento **"index"**.

```
index.html
1 <form @submit.prevent>
2   <input type="text" v-model="mensaje"
3     @keyup.enter="agregar">
4 </form>
5
6 <p>{{ mensaje }}</p>
7
8 <ul>
9   <li v-for="(x, index) in lista">
10     <span>{{x}} → {{index}}</span>
11   </li>
12 </ul>
```

En el navegador veremos que esos datos se muestran correctamente, y en la consola observamos que efectivamente ahora forman parte del **array "lista"**.

Listado de Tareas

- Leer los apuntes --> 0
- Practicar Vuejs --> 1



Si queremos lograr más interacción y que el usuario **pueda borrar cada uno de los datos** que fue ingresando, necesitamos **crear otra función y desde la vista un botón asociado esta función** dentro del ámbito del **"v-for"**.

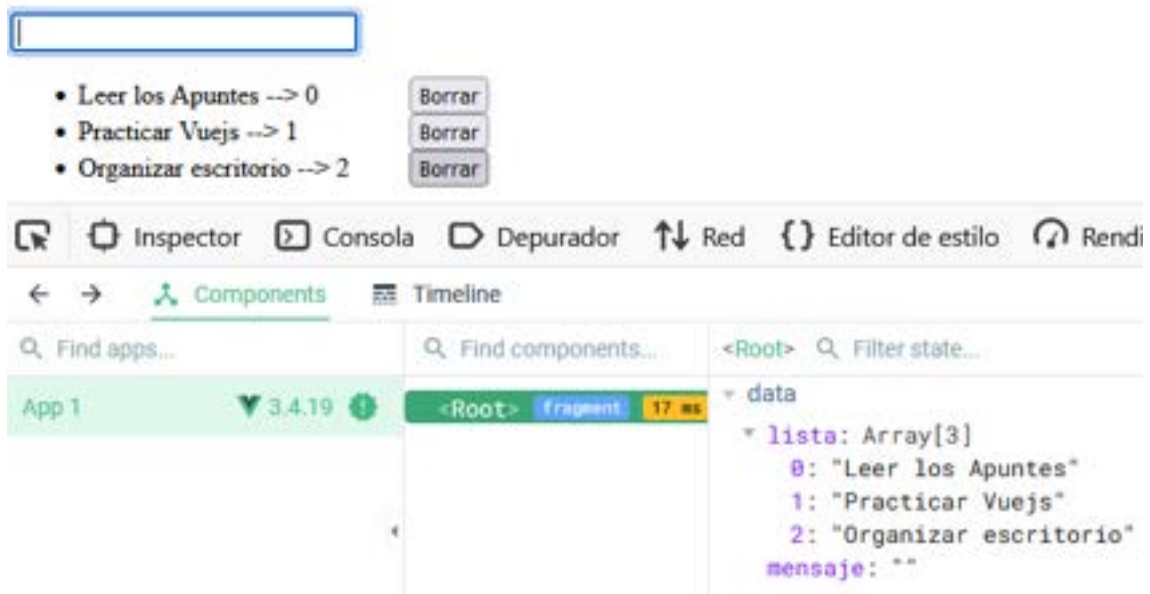
```
index.html
1 <form @submit.prevent>
2   <input type="text" v-model="mensaje"
3     @keyup.enter="agregar">
4 </form>
5
6 <p>{{ mensaje }}</p>
7
8 <ul>
9   <li v-for="(x, index) in lista">
10     <span>{{x}} → {{index}} </span>
11     <button @click="borrar(index)">Borrar</button>
12   </li>
13 </ul>
```

Retomemos lo aprendido anteriormente: Cada vez que definamos una función dentro de la propiedad **"methods"**, **es necesario separar cada función con una coma**.

```
app.js
1 const app = Vue.createApp({
2   data() {
3     return {
4       mensaje: "",
5       lista: []
6     }
7   },
8   methods: {
9     agregar: function() {
10       this.lista.push(this.mensaje);
11       this.mensaje = "";
12     },
13     borrar: function(index) {
14       this.lista.splice(index, 1)
15     }
16   }
17 });
18 app.mount('.contenedor');
```

El **método splice** nos permite elegir **el elemento del array que queramos eliminar (index)**, como **segundo argumento** le indicamos **cuántos elementos a eliminar a partir del número del index**. En este caso, ese elemento solo.

Listado de Tareas

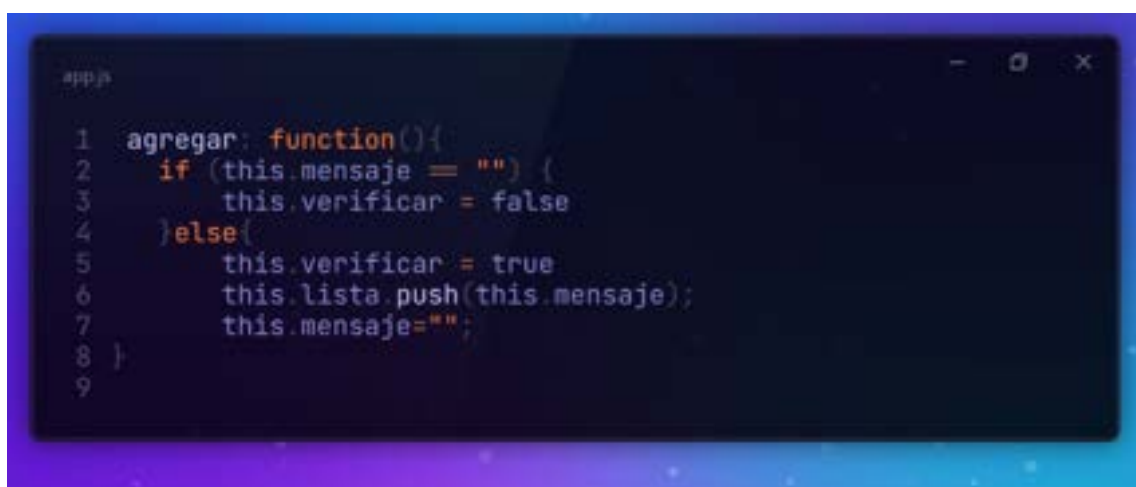


Una vez eliminado el elemento, los elementos que quedan aún en el array se seguirán mostrando.

Este ejemplo nos permite incorporar los conceptos vistos, pero podemos darle más funcionalidades para seguir aplicando lógica y convertirlo en un ejemplo más dinámico y reactivo.

La lógica nos indica que un usuario no debería poder guardar un elemento si no tiene contenido. Para lograr esto, debemos verificar el contenido del input y mostrarle al usuario que ese campo debe completarse.

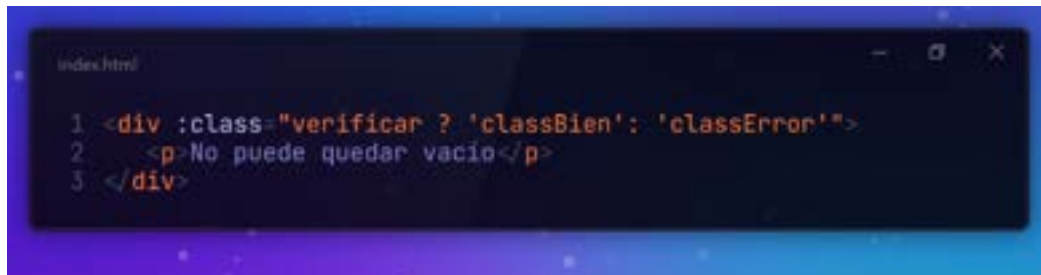
Para esto crearemos una propiedad más, llamada **"verificar"** y la inicializamos con valor **true**.



Si la variable **mensaje** está vacía, cambiamos el valor de la variable **verificar** a **false**. Caso contrario, **verificar** es **true** y podemos hacer push para que se agregue a la **lista**.

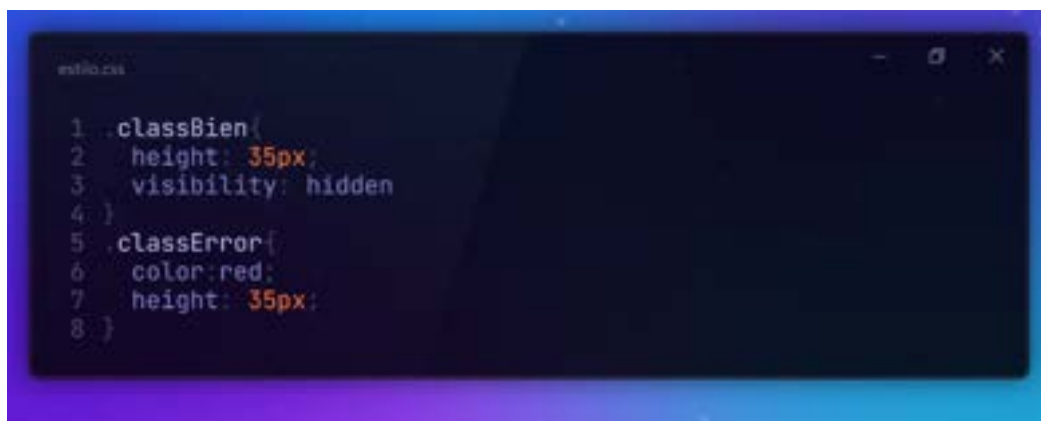
Para poder darle una retroalimentación al usuario, usaremos **v-bind** asociado al atributo **class** y dependiendo del valor de la variable **"verificar"** le podremos mostrar un **mensaje de error**, sólo si el usuario ejecuta la función y el input no tiene contenido.

En el index.html tendremos:



```
index.html
1 <div :class="verificar ? 'classBien': 'classError'">
2   <p>No puede quedar vacío</p>
3 </div>
```

Si **verificar** es **true**, aplicará la clase CSS **"classBien"**, caso contrario aplicará la clase **"classError"**.



```
estilo.css
1 .classBien{
2   height: 35px;
3   visibility: hidden
4 }
5 .classError{
6   color:red;
7   height: 35px;
8 }
```

Nuestro CSS se limitará a establecer la altura de la caja para evitar que los elementos siguientes en el DOM se desplacen. En caso de error, el texto se mostrará en color: red; de lo contrario, utilizaremos visibility: hidden para ocultar el texto.

Si el usuario ejecuta la función y el input no tiene contenido, se verá lo siguiente en el navegador:

No puede quedar vacío

Podemos seguir dando mayor lógica a la función **"agregar"**, por ejemplo, informarle al usuario cuántos datos fue cargando en esta lista.

Desde la función data podemos inicializar una nueva variable llamada **"total"** con valor 0 (cero).

A medida que se haga el **push**, establecer que el valor de **total** va a ser el **.length** de lo que tenga el array **lista**.

```
app.js
1 this.total= this.lista.length;
```

Cuando el usuario borre algún elemento deberíamos ir restando del valor de **"total"**

```
app.js
1 borrar: function(index){
2   this.lista.splice(index,1)
3   this.total --
4 }
```

En nuestro index debemos mostrar este dato por medio de la interpolación:

```
index.html
1 <p> Datos Ingresados: {{total}}</p>
```

Hasta ahora tenemos una lista que muestra elementos con contenido, caso contrario le muestra el mensaje que debe completar el input. Se pueden borrar cada uno de los mensajes y actualizar la cantidad total de elementos ingresados.

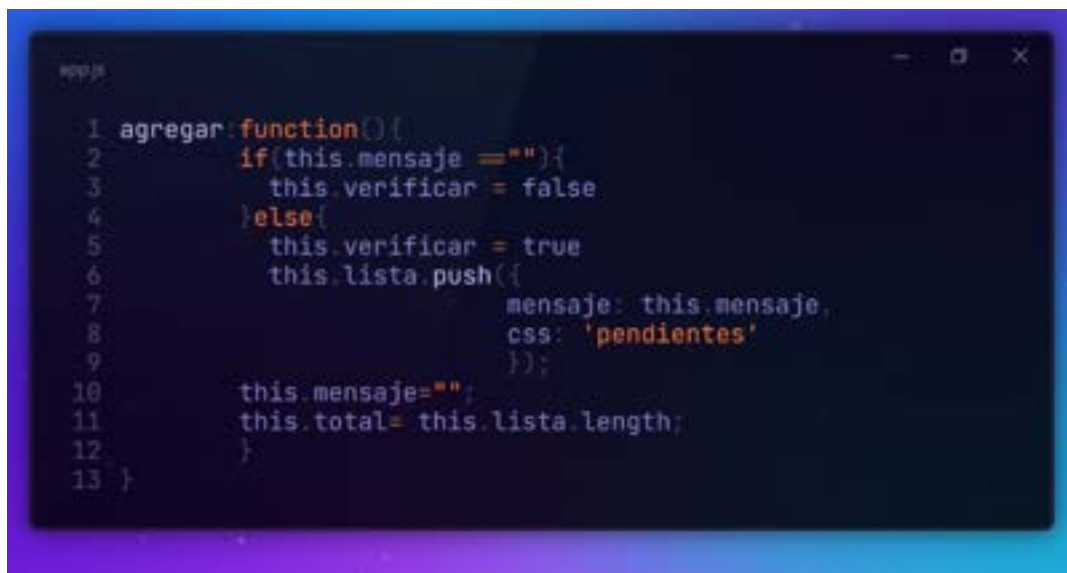
Si avanzamos un poco más allá, también podríamos convertir a este array en un **array de objetos** y hacer que cada uno de estos mensajes puedan ser marcados como **terminados o pendientes de forma dinámica**.

Por defecto el usuario ingresará tareas pendientes, asociadas a una clase CSS "**pendientes**". Y si selecciona alguna tarea cambiará la clase a "**terminadas**".

En este caso, vamos a crear un objeto que contenga 2 propiedades, **mensaje** y **css**.

Cuando el usuario vaya cargando las nuevas tareas, iremos haciendo **push** de estos objetos **al array lista**.

Modificaremos la **función agregar** para poder lograr esto.



```
1 agregar: function() {
2   if (this.mensaje === "") {
3     this.verificar = false
4   } else {
5     this.verificar = true
6     this.lista.push({
7       mensaje: this.mensaje,
8       css: 'pendientes'
9     });
10    this.mensaje = "";
11    this.total = this.lista.length;
12  }
13 }
```

Esta clase "**pendientes**" debe estar escrita en nuestro CSS, en nuestro caso tiene color:red;

Cuando se realice la iteración en el index gracias a **v-bind**, Vue **podrá darle esta clase** por defecto. Además cuando el usuario haga click sobre la tarea vamos a **desarrollar la función toggle**.



```
1 <li v-for="(item, index) in lista" :key="index"
2   :class="item.css" >
3   <span @click="toggle(index)">{{item.mensaje}}, indice:
4     {{index}}</span>
5   <button @click="borrar(index)">Borrar</button>
6 </li>
```

Desde la lógica, la **función toggle** nos va a permitir cambiar el estado de una tarea entre “pendientes” y “terminadas” ajustando a su vez un contador. La clase “terminadas” le dará un background color verde y un color de tipografía blanco.

```
app.js
1 toggle: function(index) {
2   if (this.lista[index].css == "pendientes") {
3     this.sumarTerminadas++; // Incrementa la cuenta de tareas
    terminadas
4     this.lista[index].css = 'terminadas'; // Cambia el
    estado CSS a 'terminadas'
5   } else {
6     this.sumarTerminadas--; // Decrementa la cuenta de tareas
    terminadas
7     this.lista[index].css = 'pendientes'; // Cambia el
    estado CSS a 'pendientes'
8   }
9 }
```

this.lista[index] --> Nos va a permitir afectar a la posición de ese elemento que estamos haciendo click.

Esto nos lleva a replantear un poco la lógica, al borrar también tenemos que contemplar esta actualización en los contadores.

```
app.js
1 borrar: function(index){
2   if(this.lista[index].css == 'terminadas'){
3     this.sumarTerminadas --;
4   }
5   this.lista.splice(index,1)
6   this.total --;
7 }
```

Finalmente nuestra **función data** nos quedó de esta forma, con todas las **variables inicializadas** para darle más lógica a esta Lista de Pendientes.

```
app.js
1  mensaje: "H",
2  lista: [],
3  total: 0,
4  verificar: true,
5  sumarTerminadas: 0
```

En el navegador veremos como se muestran estos elementos trabajados:

Tareas Pendientes

Leer Apuntes, indice: 0
Hacer los desafios, indice: 1
Meditar, indice: 2
Hacer ejercicio, indice: 3

Borrar
Borrar
Borrar
Borrar

Datos Ingresados: 4

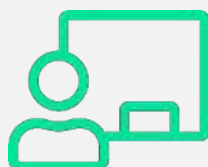
Terminadas : 1

¡Ahora vamos a practicar!



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

- Directiva v-on.
- Modificadores de eventos.
- Modificadores de teclas.
- Enlaces de datos.(directiva v-model).
- Manipulación de datos con métodos en formulario (agregar, borrar, validar campo, contar elementos, establecer clases css con operador ternario).



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.). Directiva v-on: Métodos y Modificadores. En Documentación Oficial de Vue.js. Recuperado de <https://vuejs.org/guide/essentials/event-handling.html>

Vue.js. (s.f.). Directiva v-model. En Documentación Oficial de Vue.js. Recuperado de <https://vuejs.org/api/built-in-directives.html#v-model>