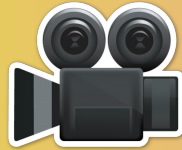




Clase 8. Vue JS

Formularios

RECORDÁ PONER A GRABAR LA CLASE





OBJETIVOS DE LA CLASE

- Aprender a desarrollar un formulario avanzado en Vue JS con validaciones simples y personalizadas por campo.

CRONOGRAMA DEL CURSO

Clase 7



Comunicación, Filtros de vista y Mixins



COMPONENTES ANIDADOS



FILTROS DE VISTA



EJEMPLO EN VIVO



ENTREGA INTERMEDIA

Clase 8



Formularios



FORMULARIO CON VUE



EJEMPLO EN VIVO



APLICANDO VUE-FORM



FORM CON VUE CLI Y SUS VALIDACIONES

Clase 9



Router y Life Cycle Hooks



IMPLEMENTANDO ROUTER



EJEMPLO EN VIVO



LIFE CYCLE HOOKS

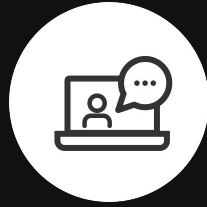
FORMULARIOS Y VUE

FORMULARIOS Y VUE

La interacción entre un Formulario Web y Vue se da de la misma forma que con el resto de los ejercicios que venimos realizando hasta el momento.



Si deseamos obtener datos de inputs, selects o checkboxes para enlazarlos con nuestro modelo, debemos utilizar las mismas herramientas que con otros elementos html. ¡Veamos!

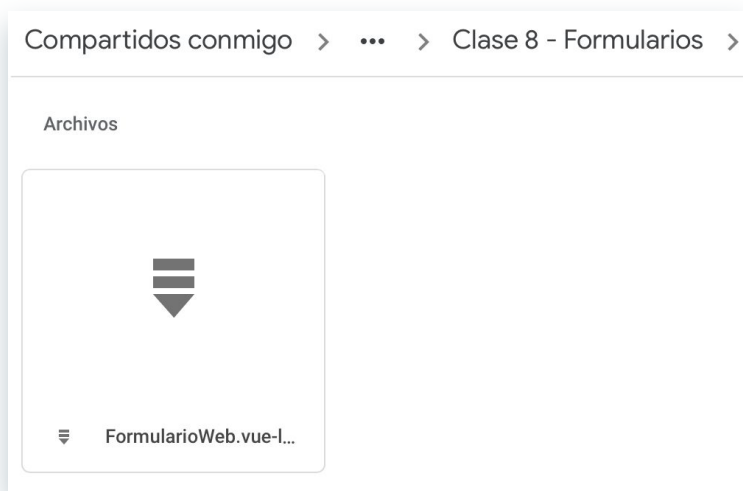


EJEMPLO EN VIVO

Basaremos las explicaciones de las siguientes diapositivas con un ejemplo prearmado que te compartimos como material adicional.



FORMULARIOS Y VUE



- Ejecuta el comando **vue create** para crear un nuevo proyecto Vue/Cli, de igual forma como lo venimos haciendo hasta ahora.
- Luego, ubica en la carpeta de Drive de la clase de hoy, un archivo comprimido llamado **FormularioWeb.vue-logo.zip**. Descárgalo, descomprimelo, y agrégalo al proyecto creado.



FORMULARIOS Y VUE

```
//App.vue > <style>  
.logo-small {  
  width: 6em;  
}
```

→ Copia la imagen `logo.jpg` a la carpeta `assets/`, y reemplaza la que viene por defecto con el proyecto Vue/Cli base. Si quieres, puedes estilizarla con una 🖱️ clase CSS, para reducir sus dimensiones.

→ Solo nos resta referenciar el archivo del formulario web en `App.vue`, para terminar de configurar la base del proyecto.



FORMULARIOS Y VUE

→ El formulario integrado a tu proyecto Vue, deberá lucir como el de la imagen contigua. 🧐

→ Luego agregaremos otros elementos de formulario pero, con esto, ya tenemos la base necesaria para comenzar a interactuar con el mismo desde nuestro código.

The image shows a web browser window at localhost:8080 displaying a form for 'CODER HOUSE'. The form includes the following elements:

- Tu nombre**: A text input field with the placeholder 'Nombre completo'.
- Edad**: A text input field with the placeholder 'Edad'.
- Email**: A text input field with the placeholder 'tu@email.com'.
- Selecciona tu curso**: A section with four radio button options: JavaScript, React, Angular, and Vue.
- Comentarios**: A large text area for comments.
- Tipo de documento**: A section with three radio button options: DNI, Pasaporte, and VISA.
- ENVIAR**: A blue button at the bottom right of the form.

<INPUT> Y V-MODEL



<input> y V-MODEL

A screenshot of a web form. At the top center is the CODEK HOUSE logo, which consists of the words 'CODEK' and 'HOUSE' stacked vertically inside a black circle. Below the logo, the text 'Tu nombre' is centered. Underneath this text is a rectangular text input field with a light gray border. Inside the input field, the placeholder text 'Nombre completo' is visible on the left side.

- Hagamos interactuar el primer elemento `<input>` con `v-model` y un tag dedicado para saber que el enlace de datos bidireccional es funcional también en el input.
- Sumemos una etiqueta del tipo `paragraph` a continuación de este elemento input.



<input> y V-MODEL

→ Agreguemos en el método `data()` una propiedad acorde, como por ejemplo `nombre`, e instanciamos la misma entre `input` y el `paragraph` creado.

A este último, podemos estilizarlo como el código de ejemplo de aquí abajo. 🙏

```
<input type="text" class="form-control" id="inputNombre" placeholder="Nombre completo" v-model="nombre">
<br>
<p>Validar nombre: <span class="text-success fw-bold">{{ nombre }}</span></p>
```



<input> y V-MODEL

← → ↻ ⓘ localhost:8080

CODER HOUSE

Tu nombre

Nombre completo

Validar nombre:

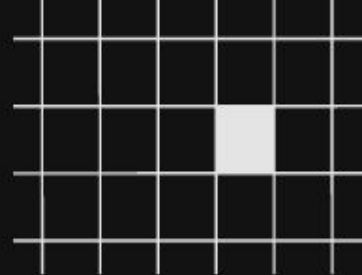
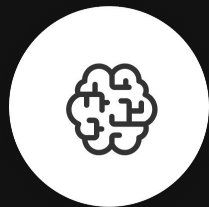
→ Como resultado, el contenido que tipeamos en el `input type` en cuestión deberá replicarse en el paragraph que agregamos justo debajo de la caja de texto.

<input> y V-MODEL



En cada input type donde se deba ingresar datos por teclado encontraremos que `v-model` funcionará por igual.

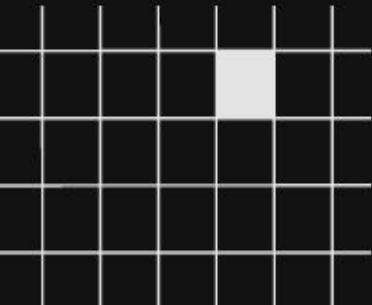
Ahora, en el caso del segundo input type que espera un dato del tipo numérico, debemos declarar dentro de `data()` una propiedad acorde y, de igual forma, debemos proceder para los input types del tipo `boolean`.



¡PARA PENSAR!

Ya que estamos familiarizados con v-model y los componentes de Vue/Cli. ¿Cómo te imaginas que debemos controlar múltiples opciones de check?

RESPONDE EN EL CHAT DE ZOOM





<input> boolean y V-MODEL

→ Adaptemos los input del tipo `checkbox` para que dispongan cada uno de un atributo `id` y de un atributo `value`, además del `v-model`.

Este último dispondrá de un objeto común a todos los otros checkboxes llamado, por ejemplo, `chequeados`...

```
//Elementos check
<div class="form-check">
  <input class="form-check-input" type="checkbox" value="javascript" id="javascript" v-model="chequeados">
  <label class="form-check-label" for="javascript">
    JavaScript
  </label>
  ...
```



<input> boolean y V-MODEL

Selecciona tu curso

- ☐ JavaScript
- ☐ React
- ☐ Angular
- ☐ Vue

Confirmados: []

...el mismo será del tipo array.

→ Lo instanciamos dentro del método `data()` y lo sumamos junto a la directiva `v-model` en cada uno de los input check, como también en el tag paragraph que definimos debajo de estos.



<input> boolean y V-MODEL

Selecciona tu curso

- ☐ JavaScript
- ☐ React
- ☐ Angular
- ☐ Vue

Confirmados: []

✓ Así comprobamos cómo se carga cada uno de los elementos seleccionados dentro del array, con el valor definido en sus atributos **value** para poder hacer match, a posteriori, cuando debamos recuperar un dato almacenado y mostrarlo en pantalla.

SELECT Y V-MODEL



SELECT y V-MODEL

→ Agreguemos un elemento `select` al formulario.

Sabemos que éstos manejan un listado de ítems,
conteniendo cada opción dentro del tag `<option>`.

```
//Elemento Select
<select class="form-select" aria-label="Default select example">
  <option selected></option>
  <option value="1">Uno</option>
  <option value="2">Dos</option>
  <option value="3">Tres</option>
</select>
```



SELECT y V-MODEL

→ Agregando la directiva `v-model` en el elemento `select`, veamos cómo se comporta el mismo en base a lo que seleccione el usuario.

```
<select class="form-select"... v-model="pais">  
  <option value="1">Uno</option>  
  ...
```



SELECT y V-MODEL

```
<option>Uno</option>
```

Y, en aquellos casos en donde el tag `option` cuente con un valor, sin el atributo `id` definido 🖱️, `v-model` leerá el dato de `value`.

A screenshot of a web form. At the top, there is a text input field. Below it, the label "País" is centered. Underneath the label is a dropdown menu with a downward arrow icon on the right. Below the dropdown menu, the label "Confirmados:" is visible. A mouse cursor is pointing at the bottom right corner of the form area.

SELECT MÚLTIPLE

En los casos donde tenemos la necesidad de establecer un `select` de múltiples valores a seleccionar, debemos aplicar la misma lógica definida para el caso de los elementos `checkbox`:

Establecer un array como propiedad y `v-model` se ocupará de llenarlo con cada opción seleccionada de este.



BREAK

¡5/10 MINUTOS Y VOLVEMOS!

SELECT DINÁMICO

SELECT DINÁMICO



En aquellos casos donde dispongamos de un array de elementos o de un array de objetos JSON, podemos aprovechar la directiva `v-for` para cargar de forma dinámica este contenido en un elemento `select`.

```
data() {  
  return {  
    nombre: '',  
    pais: '',  
    listaPaises: [  
      {  
        id: 'A',  
        pais: 'Argentina'  
      },  
      {  
        id: 'B',  
        pais: 'Uruguay'  
      },  
      {  
        id: 'C',  
        pais: 'Perú'  
      },  
      {  
        id: 'D',  
        pais: 'Colombia'  
      }  
    ]  
  }  
}
```

Ejemplo
en vivo



SELECT DINÁMICO

De acuerdo a cómo funciona el tag `<option>`,
podemos cargar en un elemento `select` tanto un
array de elementos o un array de objetos,
aprovechando la forma más conveniente para
presentar a éstos.



`v-for` genera cada elemento `<option>` con su `id`; a partir del `id` indicado en el array y su valor; a partir del dato primario.

La propiedad `index` a iterar es ideal para completar `v-bind:key`.

```
<select class="form-select" v-model="pais">
  <option selected></option>
  <option v-for="pais in listaPaíses"
    v-bind:value="pais.id"
    v-bind:key="pais.index">
    {{ pais.pais }}
  </option>
</select>
```

SELECT DINÁMICO

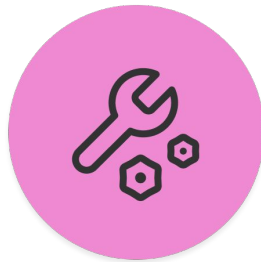
Email

tu@email.com

País

Tu selección:

Selecciona tu curso

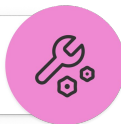


FORMULARIOS CON VUE

Crea un formulario base en Vue, utilizando la directiva v-model.

FORMULARIO DE LOGIN

Desafío
generico



Sobre cualquier proyecto de Vue-Cli que ya tengas iniciado, crea un web component con una plantilla de formulario pensando en un login (2 campos, 1 botón).

Agrega un paragraph o elemento HTML similar, para establecer la directiva v-model que obtenga los datos de ambos campos y los muestre en el elemento HTML definido.

Tiempo estimado: 10 minutos

CODER HOUSE

MODIFICADORES



EVENTOS JS

```
event.preventDefault()  
event.stopPropagation()  
...
```

El objeto global **Event**, nativo de JavaScript, cuenta con una serie de métodos de uso exclusivo para controlar eventos en una aplicación web.

MODIFICADORES



Las operaciones predeterminadas por los eventos muchas veces pueden o deben ser controladas por nuestra aplicación para aplicar decisiones diferentes a un evento determinado, lo cual puede controlarse por el código de nuestra aplicación.

MODIFICADORES



Por esto, los modificadores de Vue fueron pensados para ser aplicados a lo largo del ciclo de vida de una aplicación web construida con este framework. Y, para un orden efectivo, Vue separa los modificadores en tres frentes:

- Modificadores de eventos
- Modificadores de entrada
- Modificadores clave

MODIFICADORES DE EVENTOS

MODIFICADORES

Asociados a la directiva **v-on**, los eventos JS pueden controlarse a través de diferentes directivas del tipo **postfixes**, denotadas por un punto seguido a la directiva mencionada. Entre las diferentes opciones disponibles encontramos a:

- **.stop**
- **.prevent**
- **.capture**
- **.self**
- **.once**



.STOP

La directiva postfix `.stop` cumple la misma función que `.stopPropagation()` de Vanilla JS, impidiendo la propagación de un evento `click`, una vez accionado:

```
<input type="submit" class="btn btn-primary"
v-on:click.stop="enviarFormulario">
```

.PREVENT

La directiva postfix `.prevent` podemos utilizarla, en el caso del formulario de datos, para evitar que se recargue la página cuando es disparado el evento `onsubmit`. este:

```
<form action=""
  v-on:submit.prevent="onSubmit">
  ...
```



.CAPTURE

La directiva postfix `.capture` se implementa generalmente cuando agregamos un `event Listener` durante la ejecución de nuestra aplicación. Iniciado el evento, capture nos permitirá capturar el momento en el cual este ocurra.

```
<input type="submit" class="btn btn-primary" value="ENVIAR"  
v-on:click.capture="eventoAcapturar">
```



.SELF

.self inicia un trigger para el manejo de eventos solo si la propiedad **.target** del evento en cuestión, corresponde al elemento en sí mismo.

```
<input type="submit"  
  value="ENVIAR"  
  v-on:click.self="capturarAlgo">
```

.ONCE

.once permite que un método o función asociada a un evento se active una vez cuando es invocado. **.once** es muy poco utilizado en JS.

```
<input type="submit"  
  value="ENVIAR"  
  v-on:click.once="enviarFormulario">
```


MODIFICADORES DE ENTRADA

MODIFICADORES DE ENTRADA



También contamos con la posibilidad de controlar o modificar en tiempo real las entradas de datos de los input types, según lo que necesitamos obtener o almacenar.

- `.number`
- `.trim`
- `.lazy`



.NUMBER

A través del modificador `.number`, podemos alterar el tipo de datos ingresado en un campo, directamente desde el elemento HTML en sí, aprovechando a `v-model`.

En el ejemplo siguiente, vemos cómo cambiar el dato ingresado en el input edad a un valor numérico como realmente debe ser.

```
<input type="number" class="form-control" id="inputEdad"  
placeholder="Edad" v-model.number="edad">
```



.TRIM

La directiva `.trim` elimina los espacios creados antes o después de un texto. Es común aplicarlo en elementos HTML como ser un campo de comentarios.

```
<textarea id="areaComentarios" rows="3"
  v-model.trim="comentario">
</textarea>
```

.LAZY

`v-model` sincroniza forma automática la información cargada, con el modelo. Si deseamos retrasar la carga de datos y que Vue sincronice, luego, debemos usar `.lazy`.

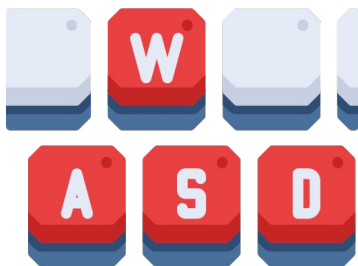
```
<input type="text" placeholder="Nombre"
  v-model.lazy="nombre">
```

MODIFICADORES CLAVE

MODIFICADORES CLAVE

Estos modificadores nos permiten escuchar los eventos de teclado. A partir de esto, podremos controlar cuando ocurre determinado evento (`enter`, `tab`, `delete`, etcétera), a la vez que evitar tener que recordar o buscar los `keyCodes` relacionados con cada tecla o clic del mouse.

MODIFICADORES CLAVE



Dentro de los modificadores que podemos utilizar, se encuentran:

- .enter
- .tab
- .delete
- .esc
- .space
- .up
- .down
- .left
- .right



MODIFICADORES CLAVE

En este ejemplo, simulamos la opción de incluir un elemento `<button>` en lugar de `<input type="submit">`, para el envío de los datos cargados en el formulario.

```
<button class="btn btn-primary"  
v-on:enter="enviarFormulario">ENVIAR</button>
```


TECLAS DEL SISTEMA



También contamos con la posibilidad de detectar las teclas del sistema, como ser:

- `.ctrl`
- `.alt`
- `.shift`
- `.meta` (*corresponde a la tecla WIN*)



TECLAS DEL SISTEMA

Con ellas podremos combinar acciones que parten de la combinación de más de una tecla, como por ejemplo **Ctrl + Clic**:

```
<button @click.ctrl="sendInformation">Enviar</button>
```

O también usar combinaciones como **Alt + S**, para guardar alguna tarea sobre la aplicación web.

```
<button @keyup.alt:s="saveElement">Guardar</button>
```

VALIDACIONES DE FORMULARIO

VALIDACIONES DE FORMULARIO

Si bien HTML5 tiene validaciones de formulario avanzadas y resuelve muy bien cada uno de los datos que debemos agregar en el formulario, puede que en algunas oportunidades deseemos realizar estas mismas validaciones desde nuestro código JS.

IMPLEMENTANDO VALIDACIONES

Veamos, a continuación, cómo implementar validaciones personalizadas en `Vue/Cli`.





VALIDACIONES DE FORMULARIO

En primera instancia, nos apoyaremos en el evento `submit` del tag `form`. Utilizaremos la directiva de Vue para invocarlo, llamando previamente a un método personalizado.

```
<form id="app" @submit="validarContenidoCargado"  
      action="/accionPosteriorAvalidar"  
      method="POST">
```



VALIDACIONES DE FORMULARIO

Declaramos un array de errores en `data()`, denominado `errors[]`, y armamos un método que recibe como parámetro el objeto global `Event` de JS.

Nos concentraremos en validar sólo tres campos de formulario: `nombre`, `edad` e `email`. Si los tres campos tienen datos correctos, invocamos un `return` para finalizar la funcionalidad de este método.

```
methods: {  
  validarContenidoCargado: (e)=> {  
    if (this.nombre && this.edad && this.email) {  
      return true  
    }  
    ...  
  }  
  ...  
}
```



VALIDACIONES DE FORMULARIO

Luego de la validación rápida, si uno o más no cumplen con la información esperada, llenamos el array `errors[]`, con un mensaje específico por cada campo que no cumpla la condición. Finalmente, invocamos a `e.preventDefault()` para que no se ejecute el action del formulario.

```
methods: {  
  validarContenidoCargado: (e)=> {  
    if (this.nombre && this.edad && this.email) {  
      return true  
    }  
    if (this.nombre === '') {this.errors.push('El nombre es obligatorio.')}  
    if (this.email === '') {this.errors.push('El correo electrónico es obligatorio.')}  
    if (this.edad === 0) {this.errors.push('La edad debe ser mayor a 0 (cero).')}  
    e.preventDefault()  
  }  
}
```




VALIDACIONES DE FORMULARIO

Finalmente, en alguna parte de nuestro template, podemos apoyarnos en el condicional `v-if` para validar si el total de elementos del array `errors[]` es mayor a 0 (cero), entonces armamos una lista desordenada con la directiva `v-for`, y cargamos en cada `list-item`, la descripción del error almacenada en este array.


```
<div v-if="errors.length > 0">
  <p>Errores detectados: </p>
  <ul class="text-warning fw-bold">
    {{ <li v-for="error in errors" v-bind:key="error.index">{{ error }}</li> }}
  </ul>
</div>
```




VALIDACIONES DE FORMULARIO


Estas son solo algunas de las posibles validaciones que podemos realizar sobre un Formulario utilizando Vue/Cli.

Si utilizas algún framework CSS como Bootstrap, podrás recurrir a componentes HTML del estilo [Alerts](#), para estilizar los mensajes de validación para con el usuario.

 An example alert with an icon

 An example success alert with an icon

 An example warning alert with an icon

 An example danger alert with an icon

USO DE WATCH CON ERRORES



USO DE WATCH CON ERRORES

Otra opción para aplicar validaciones en campos, es apoyarnos en el uso de `watch`, verificando cada campo que deba tener un dato válido, e invocando a un método que muestre un `mensaje en pantalla`, `alerta`, o que `valide a través de expresiones regulares` el dato cargado en el elemento input que estamos controlando.

```
watch: {  
  email(value){  
    this.email = value  
    this.validarEmail(value)  
  }  
},  
...
```

VUE-FORM

VUE-FORM



El ecosistema Vue cuenta con sendos complementos que simplifican tareas pesadas y, entre estos, encontramos a herramientas como **Vue-Form**.

A través de este, podemos desarrollar formularios avanzados con validación predefinida como también mediante un esquema personalizado. Veamos cómo implementarlo.



VUE-FORM

Para disponer de **Vue-Form** en un proyecto Vue/Cli, recurrimos en primera instancia a la Terminal y ejecutamos el siguiente comando para su instalación:

```
_> npm install vue-form
```

Una vez instalado en nuestro proyecto, debemos importarlo previo a comenzar a utilizarlo.



VUE-FORM

```
import Vue from 'vue'
import App from './App.vue'
import 'bootstrap'
import 'bootstrap/dist/css/bootstrap.min.css'
import VueForm from 'vue-form'
```

Para instanciarlo en nuestro proyecto, abrimos el archivo `index.js` y agregamos la línea de importación de `Vue-Form`, tal como muestra el ejemplo de código anterior.



VUE-FORM

```
import 'bootstrap/dist/css/bootstrap.min.css'  
import VueForm from 'vue-form'  
  
Vue.use(VueForm)
```

En el mismo archivo `index.js`, declaramos a continuación el método `Vue.use()` para comenzar a utilizar `VueForm`.

VUE-FORM TEMPLATE



VUE-FORM TEMPLATE

```
<vue-form :state="formstate" @submit.prevent="onSubmit">
```

Al definir el apartado `<template>`, debemos utilizar la etiqueta de formulario propia de `VueForm`, denominada `<vue-form>`. La misma cuenta con el atributo `:state`, para manejar su estado y con la directiva `@submit` para controlar el envío del formulario.



VUE-FORM TEMPLATE

Dentro del tag `<vue-form>` volcamos la estructura de lo que serán los campos de formulario, sumando en estos una serie de tags en los cuales definimos:

- cómo será el modelo de datos.
- si el campo es o no requerido.
- qué mensaje se mostrará cuando la información ingresada sea correcta.
- o qué mensaje se verá cuando no lo sea.

```
<div id="app">
  <vue-form :state="formstate" @submit.prevent="onSubmit">
    <validate tag="label">
      <span>Nombre *</span>
      <input v-model="model.name" required name="name" />
      <field-messages name="name">
        <div>Correcto!</div>
        <div slot="required">Nombre es un campo requerido</div>
      </field-messages>
    </validate>
    <validate tag="label">
      <span>Email</span>
      <input v-model="model.email" name="email" type="email"
required />
      <field-messages name="email">
        <div slot="required">Email es un campo requerido</div>
        <div slot="email">Email no es válido</div>
      </field-messages>
    </validate>
    <button type="submit">ENVIAR</button>
  </vue-form>
</div>
```



VUE-FORM TEMPLATE

```
<validate tag="label">
  <span>Nombre *</span>
  <input v-model="model.name" required name="name" />
  <field-messages name="name">
    <div>Correcto!</div>
    <div slot="required">Nombre es un campo requerido</div>
  </field-messages>
</validate>
```

Nombre *



Correcto!

Dentro del tag `<validate>` podemos ver la estructura que conforma la etiqueta del campo, el input type en cuestión con su atributo requerido y, de forma oculta, los mensajes de confirmación del dato ingresado o un mensaje para cuando se pasó por alto su información.



VUE-FORM TEMPLATE

```
<span>Nombre *</span>
<input v-model="model.name" required name="name" />
<field-messages name="name">
  <div>Correcto!</div>
  <div slot="required">Nombre es un campo requerido</div>
</field-messages>
```

Nombre *

Nombre es un campo requerido

Email *

Correcto!

`<field-messages>` establece el área donde se definen los tags correspondientes al tipo de devolución de la validación sobre el campo.

Si existe en el `input` el atributo `required`, se activará la validación y mostrará el mensaje correspondiente si la misma pasa, o no.

<SCRIPT>



SCRIPT

```
<script>
export default {
  name: 'VueForm',
  data: {
    formstate: {},
    model: {
      name: '',
      email: '',
      ...
    }
  }
}
```

En el apartado `<script>`, definimos el objeto `formstate`, más el modelo de datos que corresponde al formulario en sí, detallando por supuesto cada uno de los campos que lo conforman.

Vue-Form nos devolverá a través del objeto `formstate` un array de propiedades y valores booleanos que identifican el estado de la validación de la información cargada, o por cargar.



SCRIPT

```
formstate: {  
  ...  
  "$error": {  
    //campos con error  
  }  
  ...  
}
```

Además de la devolución que recibimos vía **formstate**, encontramos, al final de la misma, una propiedad denominada **\$error**. En esta, se listarán todos los campos que devolvieron un error durante el proceso de validación.



SCRIPT

`formstate` cuenta también con un objeto llamado `$submittedState`, el cual mantendrá toda la información del formulario procesado y enviado, por si necesitamos hacer algo con la misma posterior al envío de dichos datos.

```
formstate: {  
  ...  
  "$error": {  
    //campos con error  
  },  
  "$submittedState": {  
    //El formulario enviado  
    quedará clonado en  
    este objeto  
  }  
  ...  
}
```



SCRIPT

En el apartado `methods` declaramos un método que se ocupa de procesar el formulario. Aquí aprovechamos el objeto `formstate`, donde podemos consultar el valor booleano de su propiedad `$invalid`.

Si la misma devuelve `true`, alertamos al usuario con un mensaje acorde. Caso contrario, enviamos los datos.

```
methods: {  
  onSubmit: function () {  
    if(this.formstate.$invalid) {  
      // alertar al usuario  
      por campos con valores  
      no válidos  
      return  
    }  
    //procesar y enviar  
    el Formulario  
  }  
}
```

VALIDACIONES PERSONALIZADAS

VALIDACIONES PERSONALIZADAS



Más allá de las validaciones estándar que se incluye junto a cada input type, a través de Vue-Form contamos con la posibilidad de integrar validaciones personalizadas.

A través de las mismas, podemos definir una validación específica para determinado o determinados campos de formulario.



VALIDACIONES PERSONALIZADAS

```
validators: {  
  matches: (value, attrValue) => {  
    If (!attrValue) {  
      return true  
    }  
    return value === attrValue  
  }...  
}
```

Aquí aprovechamos el objeto `validators: {}`.

Éste cuenta con la posibilidad de definir una función con parámetros a través de la cual establecer la validación para determinado campo de formulario.



VALIDACIONES PERSONALIZADAS

Más allá de establecer una validación predeterminada, también podemos evaluar el ingreso de datos en un campo de formulario como, por ejemplo, la fortaleza de una contraseña ingresada en un `input type password` integrando a una validación tan específica el uso de expresiones regulares.

```
...  
'composicion-passwd': (value) => {  
  return /^(?=^.{8,}$)((?=.*\d)|(?=.*\W+))(?![\.\n])(?=.*[A-Z])(?=.*[a-z]).*$/.test(value)  
}  
}
```



VALIDACIONES PERSONALIZADAS

Ejemplo de formulario de login

Nombre *

Password *

Confirmar Password *

ENVIAR

De esta forma, conseguimos establecer un mensaje predeterminado, como el utilizado en el primero y tercer campo de este ejemplo, o algo muy específico como la notificación del segundo campo.



VALIDACIONES PERSONALIZADAS

Ejemplo de formulario de login

Nombre *

Password *

Confirmar Password *

ENVIAR

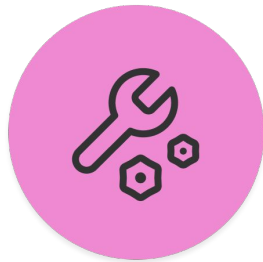
Como podemos apreciar, Vue-Form es una muy valiosa herramienta que complementa de forma excelente a Vue/Cli.

Ejemplo
en vivo



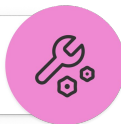
¡VAMOS A PRACTICAR!

CODER HOUSE



APLICA VALIDACIONES DE VUE-FORM

Utiliza las validaciones aprendidas hasta aquí para potenciar tu desafío anterior.



APLICANDO VALIDACIONES DE VUE-FORM

Sobre el desafío anterior basado en un formulario de login, agrega validaciones a ambos campos para que no puedan estar vacíos antes de pulsar el botón LOGIN.

Integra también el modificador de teclas para que, al pulsar **Enter** en cada campo, el foco avance al siguiente.

Tiempo estimado: 15 minutos



PROYECTO VUE CLI CON FORMULARIO

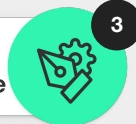
Instancia un formulario Vue/Cli con validaciones.

PROYECTO VUE CLI CON FORMULARIO

Formato: Sube tu desafío a la plataforma de Coderhouse. Deberá tener como nombre “Desafío Formulario Vue-Cli + Tu Apellido”.

Sugerencia: Aprovecha y aplica todos los conocimientos aprendidos hoy.

Desafío
entregable



Consigna: Crea un proyecto utilizando Vue/Cli en el cual instancias un formulario de datos. En el mismo deberás agregar, al menos, cuatro input de diferentes tipos (**type**) y le aplicarás validaciones en los cuatro campos. Una vez que los datos ingresados pasen las validaciones pertinentes, carga los mismos en un objeto JSON para que, luego, este objeto se refleje en un Componente `<table>` (tabla reactiva), ubicada en el apartado inferior del Formulario, mediante filtros de Vista.

El formulario se debe vaciar y permitir agregar otros datos adicionales en la Vista de la tabla inferior.

Aspectos a incluir en el entregable:

Ten presente crear dos componentes diferentes (formulario) y (tabla). Ambos deberán verse reflejados en modo funcional, en un único documento HTML.

¿PREGUNTAS?



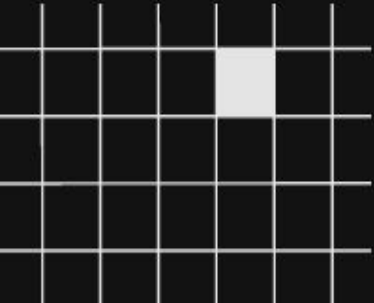


- [HTML Input types](#) | **W3Schools**
- [HTML Form Attributes](#) | **W3Schools**
- [HTML Input types attributes](#) | **W3Schools**
- [HTML Input types](#) | **Mozilla Developer Network**
- [Bootstrap Vue Form](#) | **Vue JS**
- [Vue Form Wizard](#) | **Styde.net**
- [Manejo de eventos](#) | **Vue JS**



¡MUCHAS GRACIAS!

Resumen de lo visto en clase hoy:

- Formularios web
 - Uso de formularios en Vue con validaciones
 - Modificadores number, trim, lazy, prevent
 - Formulario avanzado con vue-form
 - Validaciones personalizadas en vue-form
- 



OPINA Y VALORA ESTA CLASE