

ADM

Dudas?



ADM

Dudas?



Formularios

- Los formularios son el elemento de interacción elemental de todo sistema, web o aplicación.
- Hay ciertas cuestiones que tenemos que recordar dependiendo del elemento que queremos usar para el ingreso de datos.
- La directiva `v-model`, dentro del mundo vue, nos permite crear el data binding de 2 vías o el enlace de datos de 2 vías.
- Controles como `inputs`, `textarea` y `select` poseen esta directiva para poder actualizar de manera correcta el elemento según el tipo de entrada.

VueJS

- En los elementos de tipo textarea la interpolacion no funciona

```
<textarea>{{text}}</textarea>
```



```
<textarea v-model="form_data.comentario"></textarea>
```



Checkbox simples con valor booleano:

```
<input type="checkbox" id="checkbox" v-model="checked">  
<label for="checkbox">{{ checked }}</label>
```

- Los valores que retorna la elección pueden ser **true o false**

VueJS

Múltiples checkboxes vinculados al mismo Array:

```
<div>
  <input type="checkbox" id="pepe" value="Pepe" v-model="nombres">
    <label for="Pepe">Pepe</label>
  <input type="checkbox" id="thor" value="Thor" v-model="nombres">
    <label for="thor">Thor</label>
  <input type="checkbox" id="Randy" value="Randy" v-model="nombres">
    <label for="randy">Randy</label>

  <span>Nombres: {{ nombres }}</span>
</div>
```

```
return {
  nombres: []
}
```

Para guardar estos valores, cada checkbox debe tener su propio **value**. Caso contrario devolverá `on` para todos los que se marquen

Recordar que **hay que definir estos valores como array dentro la lógica de Vuejs**

Radio

```
<input type="radio" id="uno" value="Negro" v-model="color">
  <label for="negro">Negro</label>

<input type="radio" id="Dos" value="Blanco" v-model="color">
  <label for="blanco">Blanco</label>

<span>Eligió: {{ color }}</span>
```

Selección simple

```
<select v-model="selected">
  <option disabled value="">Seleccione un elemento</option>
  <option> A</option>
  <option> B</option>
  <option> C</option>
</select>
<span>Seleccionado: {{ selected }}</span>
```

Desde la lógica la variable selected debe inicializarse

```
return {
  selected: ''
```

VueJS

- Si el valor inicial del `v-model` no coincide con ninguna de las opciones, el elemento `<select>` se representará en un estado “**unselected**”
- En iOS, esto hará que el usuario **no pueda seleccionar el primer elemento porque iOS no dispara un evento de tipo change**.
- Se recomienda dar una **opción deshabilitada** con un **value vacío**,

Selección de múltiple

```
<select v-model="selected" multiple>
  <option>A</option>
  <option>B</option>
  <option>C</option>
</select>
<span>Seleccionados: {{ selected }}</span>
```

Select dinámico con v-for

```
<select v-model="elegido">
  <option v-for="item in options" v-bind:value="item.value">
    {{ item.texto }}
  </option>
</select>
<span>Seleccionado: {{ elegido }}</span>
```

Desde la lógica :

```
return {
  elegido: 'A',
  options: [
    { texto: 'Uno', value: 'A' },
    { texto: 'Dos', value: 'B' },
    { texto: 'Tres', value: 'C' }
  ]
}
```

Vincular valores

- En los casos de **radio**, **checkbox** y **option** de select, los **valores** de vinculación del v-model suelen ser strings o booleanos

```
<input type="radio" v-model="elegido" value="a">
```

(elegido es una string "a" cuando está chequeado)

```
<input type="checkbox" v-model="toggle">
```

(`toggle` es verdadero o falso)

Vuejs

```
<select v-model="seleccion">  
  <option value="PC">Pc</option>  
</select>
```

(`seleccion` es un string "PC" cuando se selecciona la primera opción)

- Puede pasar que queramos **vincular el valor a una propiedad dinámica** en la instancia de Vue.
- Podemos usar **v-bind** para lograr eso. Además, el uso de **v-bind** nos permite vincular el valor de entrada a valores no string.

En select

```
<select v-model="selected">  
  <option v-bind:value="{ number: 123 }">123</option>  
</select>
```

objeto literal

Modificadores

.lazy De forma predeterminada, **v-model** sincroniza el input con los datos **después de cada evento de tipo input**

Podemos agregar el **modificador lazy** para realizar la sincronización **después del evento change**:

VueJS

```
<input v-model.lazy="msg" >
```

.number

Podemos configurar que las entradas del usuario se escriban automáticamente **como un número**, agregando el modificador **number** al **v-model** del elemento:

```
<input v-model.number="edad" type="number">
```

- Esto suele ser útil ya que con **type="number"**, el valor returned por el elemento HTML siempre es una **cadena de texto**

VueJS

.trim

Nos permite recortar los espacios de las entradas del usuario automáticamente, agregando el modificador trim al v-model del elemento (inicio-fin):

```
<input v-model.trim="msg">
```

Validar elementos

- Dependiendo del tipo de campo y de la información que queramos almacenar, hay que tener presente el **aviso al usuario** para facilitarle la tarea.

El título es obligatorio.

Debe seleccionar un elemento.

El año es obligatorio.

VueJS

- Indicarle al usuario, lo que nosotros esperamos de ese campo, le dará la **retroalimentación necesaria** y tendrá una **buenas experiencia**.
- Desde vuejs podemos hacerlo sin problemas.
- Cuando el usuario **quiera enviar el formulario**, si hay errores, le comunicaremos qué es lo que debe corregir.
- Sabemos que gracias a html5, el browser nos permite realizar **validaciones**, pero vamos a correrlo de esta responsabilidad para que **vue se encargue de esto**.

VueJS

```
<form v-on:submit.prevent="guardar" novalidate >
```

- El atributo **novalidate**, permite anular las validaciones típicas del browser por medio de los atributos de html5
- Podemos generar un **array de errores** e inicializarlo desde el data del componente form
- Iremos verificando si estos elementos, tienen contenido, o si son null, o si cumplen con determinada cantidad de caracteres, etc
- Si no cumple un campo con lo que estamos esperando que ingrese el usuario, haremos un push al array de errores y creamos el mensaje que verá el usuario.

VueJS

```
data:function(){  
    return {  
        titulo:null,  
        anio:null,  
        errores:[],  
    }  
}
```

Desde nuestro componente inicializamos las variables asociadas a los **v-model** y el **array que será responsable de ir mostrando los errores**

- Una vez que el usuario quiera enviar el form ejecutaremos las comprobaciones necesarias.

```
if (!this.titulo) {  
    this.errores.push('El título es obligatorio.');//  
}  
if (!this.anio) {  
    this.errores.push('El año es obligatorio.');//  
}
```

VueJS

- Desde la **propiedad template del componente** tendremos el html para mostrarlos
- Evaluamos si **this.errores.length es mayor a cero** en cuyo caso, significa que hay errores y los iteramos para que sean mostrados.

```
<div v-if="this.errores.length > 0" class="classerror">
<ul>
  <li v-for="x in errores" >{{x}}</li>
</ul>
</div>
```



VueJS

- Caso contrario, significa que no hay errores y mostrará el mensaje de "Enviado con éxito"

```
<div v-else class="enviado">  
    <span>Enviado con éxito</span>  
</div>
```

Enviado con éxito

- Una vez que se cumplan las validaciones y todo esté ok, verificaremos que **this.errores.length sea == 0**
- Recién en ese momento haremos el push del nuevo juego

VueJS

- Estos datos a futuro deberemos almacenarlos para que no se pierdan.

```
nuevoObj={  
    comentario: this.comentario,  
    titulo: this.titulo,  
    seleccion: this.seleccion,  
    anio: this.anio  
}  
  
this.arr.push(nuevoObj)
```

VueJS

```
<div v-if="this.arr.length > 0">
  <h2>Datos</h2>
  <ul>
    <li v-for="item in arr">
      {{item.titulo}}, {{item.comentario}},
      <span v-for="x in item.seleccion">{{x}}, </span>{{item.anio}}</li>
    </ul>
  </div>
```

- El condicional, nos permite controlar que el **titulo Datos** no se muestre si **no hay elementos que mostrar**
- En cuyo caso caemos en el **else** y le mostraremos al usuario que no hay nada en **localStorage**

```
<div v-else>
  <p>No hay datos que mostrar, empezá a cargar tus juegos!</p>
</div>
```



Dudas?

ADM

Dudas?

