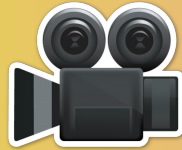




Clase 7. Vue JS

# ***Comunicación, Filtros de vista y Mixins***

***RECORDÁ PONER A GRABAR LA CLASE***





## ***OBJETIVOS DE LA CLASE***

- Conocer los distintos métodos de comunicación entre componentes.
- Crear filtros de vista y comprender el concepto de mixin.
- Entender cómo funciona la reactividad en Vue JS.

# ***CRONOGRAMA DEL CURSO***

## Clase 6



### Proyecto Vue Cli



USANDO VUE CLI 2



SERVIDOR DE DESARROLLO  
AUTOMÁTICO

## Clase 7



### Comunicación, Filtros de vista y Mixins



COMPONENTES ANIDADOS



FILTROS DE VISTA



PRIMERA ENTREGA  
PROYECTO FINAL

## Clase 8



### Formularios



FORMULARIO CON VUE



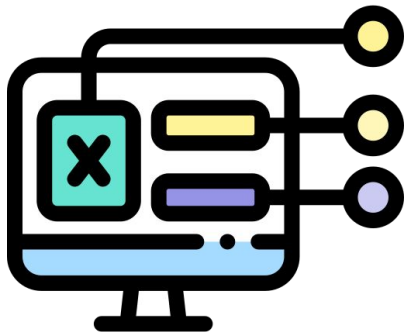
APLICANDO VUE-FORM



FORM CON VUE CLI Y SUS  
VALIDACIONES

# ***COMUNICACIÓN ENTRE COMPONENTES***

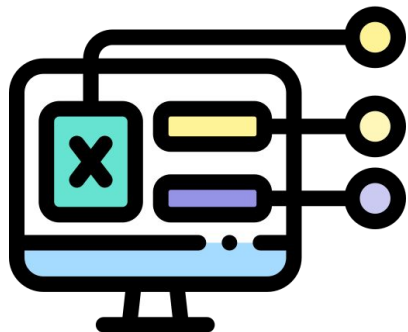
# ***COMUNICACIÓN ENTRE COMPONENTES***



La creación de una webapp basada en un framework como Vue, dispone de un montón de componentes web aislados.

Cada componente es reutilizable gracias a que lo pensamos de una forma más abstracta y desacoplada. Esto último nos permite quitarlo o añadirlo en una webapp Vue, sin perjudicar la funcionalidad de esta.

# ***COMUNICACIÓN ENTRE COMPONENTES***



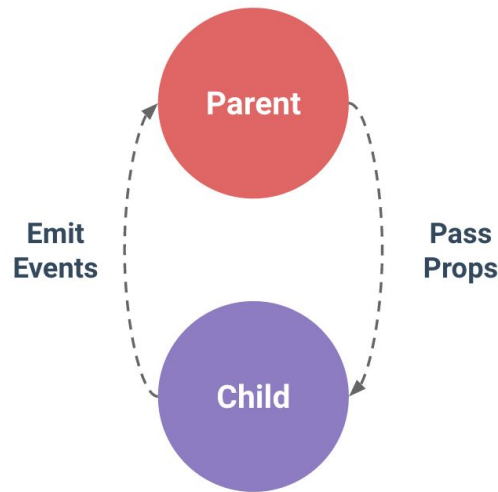
Aún así, es imposible crear componentes web con una independencia total del resto de los componentes dado que estos necesitan comunicarse entre sí, principalmente, para intercambiar datos o estados de la aplicación.

Esto nos fuerza a tener que aprender cómo lograr que la comunicación entre componentes sea adecuada, además de funcional, flexible y escalable.

# ***Comunicación entre componentes***

Particularmente en Vue.js, existen dos tipos de comunicación entre componentes:

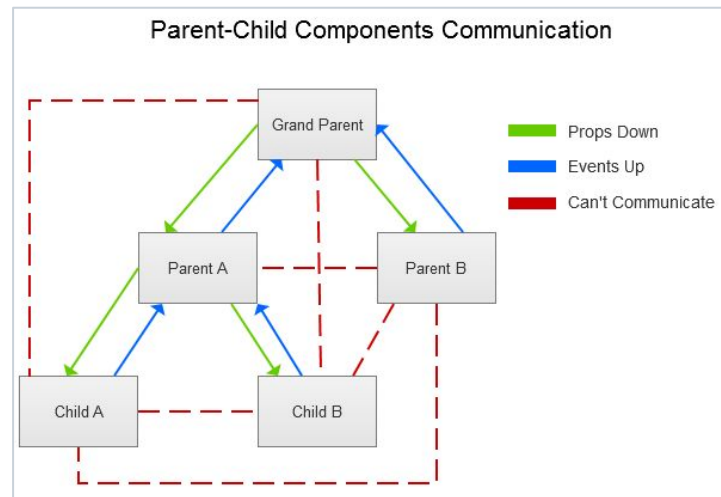
1. **Comunicación directa padre-hijo:** basada en las estrictas relaciones padre a hijo e hijo a padre
2. **Comunicación multicomponente:** en la que un componente puede "*hablar*" con cualquier otro, sin importar su relación





# COMUNICACIÓN DIRECTA: PADRE-HIJO

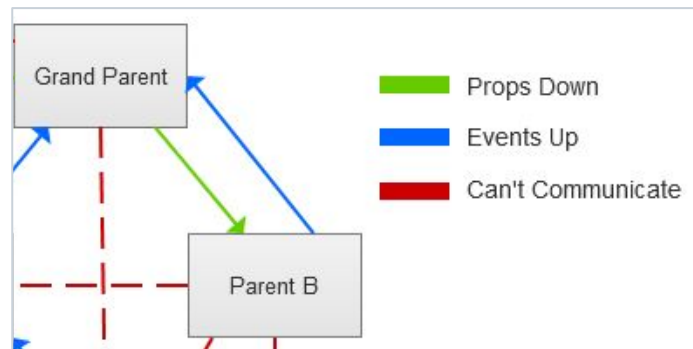
Y de ambas formas de comunicación mencionadas, el modelo de comunicación entre componentes con el que Vue.js es compatible de forma inmediata es el modelo **padre-hijo**, el cual involucra en su comunicación a **props** y **eventos** personalizados, tal como lo muestra el diagrama contigo: 🙌



# ***COMUNICACIÓN DIRECTA: PADRE-HIJO***

En este podemos apreciar que un padre se comunica solo con sus hijos directos y, estos últimos, solo pueden comunicarse con sus padres.

A través de una o más **props**, enviamos información del padre hacia el hijo y la misma, procesada, vuelve hacia el padre en forma de **evento**.



***PROPS***

# ***REPASO DE CONCEPTOS: PROPS***

En la clase pasada reflatamos, a modo de repaso, las tareas que las **props** cumplen en una aplicación Vue.

Profundicemos un poco más su función para tener en cuenta datos fundamentales de éstas, cuando se usan con un componente web.



# ***PROPS ESTÁTICAS vs DINÁMICAS***

Las props de un componente son los parámetros de entrada que éste permite para configurar su comportamiento por defecto.

Un componente padre puede instanciar y configurar un componente visual por medio de props.

En base a esto, podemos pasar datos del tipo **estáticos** o **dinámicos** a nuestras propiedades.



# ***PROPS ESTÁTICAS vs DINÁMICAS***

```
<script>
export default {
  name: 'blogPost',
  props: {
    publicacion: String,
    autor: String,
    fecha: Date
  }
  ...
}
```

Si tenemos definidas una serie de props (publicacion, autor, fecha) y necesitamos pasarlas a un componente web determinado, tenemos que optar por el uso de **v-bind** o su abreviatura, para comunicarlas al componente web.



# ***PROPS ESTÁTICAS vs DINÁMICAS***

Si no le indicamos la directiva `v-bind:` o su abreviatura `:`, el compilador entenderá que le estamos pasando un valor “hardcodeado” (estático), similar a la siguiente línea de código:

```
<blog-post titulo="Curso de Vue"></blog-post>
```

La forma correcta, sería 📖:

```
<blog-post v-bind:titulo="titulo"></blog-post>
```



# ***PROPS ESTÁTICAS vs DINÁMICAS***

Otra alternativa es indicar **variables** en lugar de las **props**, así los componentes web reaccionen a los cambios que ocurran en el padre, sin tener que involucrarnos con algún método. (datos dinámicos)

```
<blog-post v-bind:titulo="post.titulo ' (por ' + post.autor + ')"></blog-post>
```



# ***MANEJO DE NÚMEROS Y BOOLEANOS EN LAS PROPS***



# ***PROPS CON NÚMEROS***

Otro de los cuidados que debemos contemplar, es cuando tenemos que pasar por props un valor numérico a la propiedad de un componente:

```
<blog-post v-bind:megusta="21"></blog-post>
```

El compilador lo interpretará como un **string**. Por eso, si necesitamos que sea un valor numérico, autoincrementable en tiempo real, debemos recurrir a una prop dinámica.

```
<blog-post v-bind:megusta="post.likes"></blog-post>
```



# ***PROPS CON BOOLEANOS***

Cuando utilizamos un valor booleano a través de una prop, podemos simplemente indicar la propiedad. Esta, por defecto, marcará un valor del tipo `true` 🙌.

```
<blog-post post.is-published></blog-post>
```

Funciona exactamente igual que algunas propiedades de elementos HTML que no requieren especificar el valor booleano. Si está presente como atributo, por defecto es `true`.

```
<audio id="audio" controls>
```



# ***PROPS CON BOOLEANOS***

Ahora, si lo que deseamos es que dicho valor esté establecido en **false**, sí recomendamos pasarle un valor por defecto a dicha prop:

```
<blog-post post.is-published="false"></blog-post>
```

O indicarlo también de forma dinámica: 

```
<blog-post post.is-published="post.isPublished"></blog-post>
```

# ***MANEJO DE ARRAYS EN PROPS***



# ***MANEJO DE ARRAYS EN PROPS***

En el caso de los arrays, tenemos la opción de enviarlo a un componente web de forma dinámica.

A continuación, dos ejemplos de cómo hacerlo: 📌

```
<!-- Una opción de pasaje de array -->
<blog-post v-bind:comment-ids="[211, 213, 147]"></blog-post>

<!-- Otra opción de pasaje de array -->
<blog-post v-bind:comment-ids="post.commentIds"></blog-post>
```



# ***MANEJO DE ARRAYS EN PROPS***

Además, podemos también pasarle directamente los datos específicos de un objeto del tipo JSON, a través de la misma opción **v-bind**: 📝

```
<!-- Una opción de pasaje de array -->
<blog-post v-bind:autor="{autor: 'Coder House', fecha: '25-10-2021'}"></blog-post>

<!-- Otra opción de pasaje de array -->
<blog-post v-bind:autor="post.autor"></blog-post>
```

# ***EVENTOS***





# MANEJO DE EVENTOS

```
nombreCompleto: {  
  get: ()=> {  
    return 'Coder ' + 'House'  
  },  
  set: (nuevoValor)=> {  
    this.names = nuevoValor.split(' ')  
    this.nombre = names[0]  
    this.apellido = names[1]  
  }  
  ...  
}
```

Cuando trabajamos en una instancia o componente de Vue 2 (por ejemplo, con `methods`, `computed`, o `hooks` del ciclo de vida), utilizamos el objeto `this`, el cual nos permite referenciar al objeto de opciones, contenido en `Options API`.



# ***PROPIEDADES INTERNAS***

Pero no sólo podemos utilizar la palabra reservada `this` para acceder a `variables`, `props` o `computed`. También contamos con una serie de propiedades internas, que nos proporcionan acceso a diferentes secciones de una App

Vue: todo, anteponiendo el símbolo `$`.



# ***PROPIEDADES INTERNAS***

- `$el`
- `$props`
- `$data`
- `$options`
- `$slots`
- `$refs`
- `$attrs`

Si hacemos una analogía entre los nombres y lo que aprendimos hasta el momento con Vue, veremos que cada propiedad interna referencia directamente a una sección u objeto específico de este Framework.

¡Analicemos cada una de ellas!



# ***PROPIEDADES INTERNAS: `$el`***

Si queremos referenciar al elemento HTML raíz del `<template>`, podemos hacerlo a través de la propiedad de instancia `this.$el`.

```
...  
mounted() {  
  console.info(`El elemento raíz de este template, es: ${this.$el}`)  
}  
}  
</script>
```



# PROPIEDADES INTERNAS: `$props`

```
export default {  
  name: 'blogPost',  
  props: {  
    titulo: String,  
    publicacion: String,  
    autor: String,  
    fecha: Date,  
    commentIds: Array  
  },  
  mounted() {  
    console.info(`El título del post es: ${this.$props.titulo}`)  
    console.info(`Y su autor: ${this.$props.autor}`)  
  },  
}
```

Además de poder acceder a los props de un componente de la manera convencional: `this.propaconsultar`, también podemos hacerlo mediante `this.$props.fecha`.



# ***PROPIEDADES INTERNAS: \$data***

El acceso a propiedades almacenadas en `data()` se realiza de igual forma a lo que vimos con `$props`:

`(this.$data.valor ó this.valor)`

```
watch: {  
  question: (newQuestion, oldQuestion)=> {  
    this.$data.answer = 'Esperando que deje de escribir...'  
    console.log(`La pregunta formulada, es: ${this.$data.question}`)  
  }  
  ...  
}
```



# ***PROPIEDADES INTERNAS: \$options***

Cualquier opción que hayamos declarado dentro de la instancia de Vue, podrá ser accesible como una propiedad personalizada:

```
{...
  opcionPersonalizada: 'Opción Personalizada',
  created: function () {
    console.log(this.$options.opcionPersonalizada)
  }
})
```



# PROPIEDADES INTERNAS: **\$refs**

```
//TEMPLATE  
<template>  
  <base-input ref="usernameInput">  
    <input ref="input">  
  </base-input>  
</template>
```

```
//CÓDIGO DE ACCESO  
this.$refs.usernameInput  
this.$refs.input.focus()
```

Más allá de **props** y **events**, a veces necesitamos acceder puntualmente a un componente hijo (o secundario) mediante su atributo **id**, tal como se suele realizar en Vanilla JavaScript.

Contando con una propiedad **ref**, podemos aprovechar la propiedad **\$refs** para lograr este cometido.





# PROPIEDADES INTERNAS: *\$attrs*

```
<blog-post post.autor="post.autor"  
  v-bind="$attrs">  
  ...  
</blog-post>
```

A través de `$attrs` accedemos a los bindings de atributos del componente padre, exceptuando `class` y `style`, quienes no son reconocidos como props.

Si el componente no incluye declaración de props, esta propiedad te brinda todos los bindings de atributos del scope del componente padre y puede pasarse a un componente interno a través de la directiva `v-bind="$attrs"`.



***¡VAMOS A PRACTICAR!***



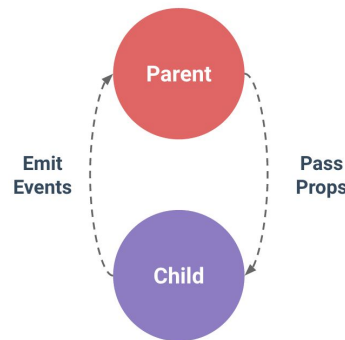
***BREAK***

**¡5/10 MINUTOS Y VOLVEMOS!**

# ***EVENTOS PERSONALIZADOS***

# EVENTOS PERSONALIZADOS

Tal como mencionamos al inicio de esta clase, los componentes padres se comunican con sus hijos a través de las **props**. En este caso, el flujo de información es unidireccional, por lo cual, los componentes padres pueden mutar las **props** a sus elementos hijos, pero si estos últimos también las mutan, no afecta para nada a los componentes padres.



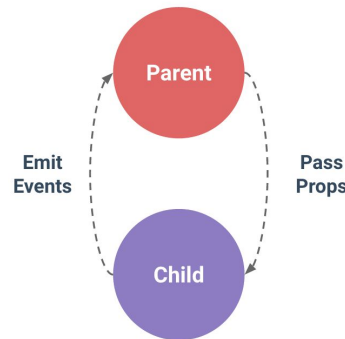
Entonces...

# EVENTOS PERSONALIZADOS

...

*¿Cómo podemos enviar información desde los componentes hijos hacia los componentes padres?*

¡Este proceso debemos realizarlo por intermedio de los eventos personalizados!



# ***NOMENCLATURA DE EVENTOS PERSONALIZADOS***



# ***EVENTOS PERSONALIZADOS***

Cuando debemos emitir acciones específicas, o datos a los componentes padres desde un componente hijo, debemos utilizar el método `$emit`.

Cuando definimos a este, debemos tener cuidado de que su nomenclatura sea establecida bajo el formato `camelCase`.

```
methods: {  
  enviarEvento() {  
    this.$emit('miEventoPersonalizado')  
  }  
}  
...
```





# ***EVENTOS PERSONALIZADOS***

Para el caso que un componente padre necesite escuchar determinados eventos de un componente hijo, una vez que definimos dicho evento, utilizamos `v-on` o su método abreviado `@` seguido al nombre de este.



```
...  
  <mi-componente @mi-evento-personalizado="ejecutarUnaAccion"></mi-componente>  
</div>  
</template>
```

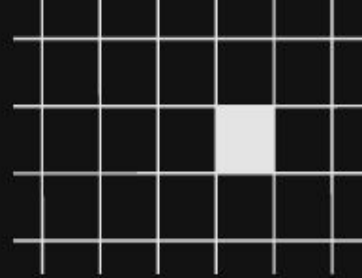
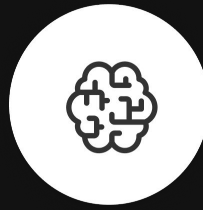


# DEMARCAR EVENTOS NATIVOS

En determinadas ocasiones, podemos llegar a encontrarnos con “colisiones” que acontecen entre un **evento nativo**, y uno **personalizado**, que nosotros generamos.

Vue cuenta con un modificador a integrar dentro del evento, que permite distinguir a la aplicación si lo que estamos escuchando es un evento personalizado, o uno nativo.

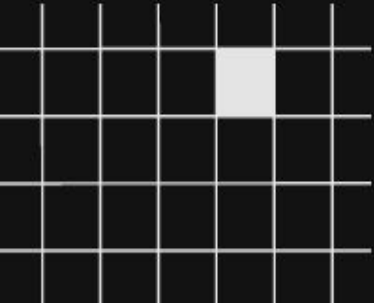
```
...  
<mi-componente @mi-evento-personalizado="ejecutarUnaAccion"></mi-componente>  
<base-input v-on:focus.native="onFocus"></base-input>  
</div>  
</template>
```

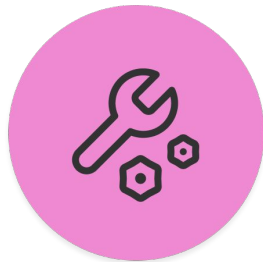


## ***¡PARA PENSAR!***

*Antes de avanzar, piensa en lo visto hasta ahora. ¿Recuerdas cómo se comunican los componentes padres con sus hijos?*

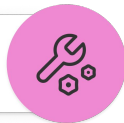
CONTESTA EN EL CHAT DE ZOOM





# ***ELABORA DOS COMPONENTES ANIDADOS***

Crea dos componentes anidados, que puedan comunicarse entre sí.



# ***COMUNICACIÓN ENTRE COMPONENTES ANIDADOS***

Crea dos componentes anidados, que cumplan el rol mencionado (padre - hijo).

En ellos, busca una forma simple para que el componente hijo pueda recibir y modificar datos que sean generados por el componente padre, pudiendo, a su vez, recapturarlos por parte del componente padre.

**Tiempo estimado:** 15 minutos.

***SLOTS***

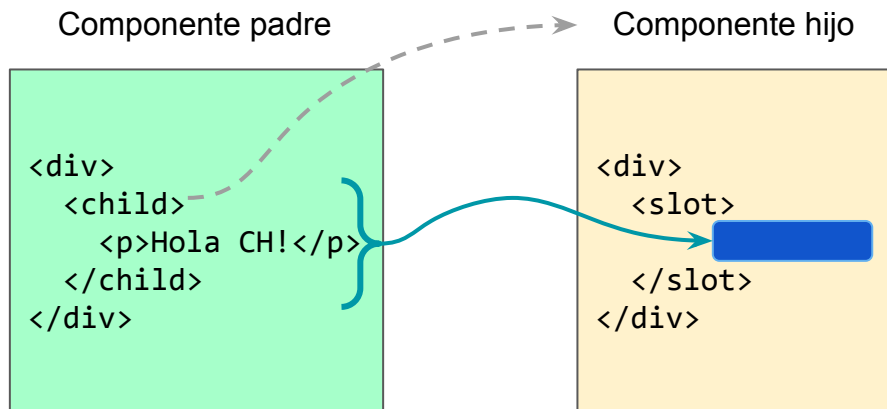


# ***SLOTS***

En el universo de Vue JS, se define a los **slots** como un mecanismo que se utiliza para agregar o insertar contenido HTML dentro de un componente web. Así como los props le envían a un componente web, variables y objetos, los slots se ocupan de insertar contenido en formato HTML dentro del componente en sí.



# SLOTS



Si deseáramos crear un componente a renderizar en un header y, a su vez, dentro de este agregar botones u otra información dinámica respecto a la página en la cual se encuentra, seguro que lo haríamos con props, pero los slots facilitan todo este proceso, pudiendo agregar contenido HTML desde afuera del componente.





# SLOTS

Tengamos presente que un slot es una etiqueta HTML propia de Vue JS. Cuando definimos a la misma, debemos hacerlo dentro del template de un componente web.

Con ello, ya le estamos indicando al transpilador de Vue que insertaremos contenido HTML desde afuera de este componente.

```
<template>
  <button @click="handleClick">
    <slot>
      <!-- Esperando por código externo... -->
    </slot>
  </button>
</template>

export default {
  ...
```



# SLOTS

→ Importamos el componente web que deseamos insertar vía slot.

→ Luego, todo el contenido que esté dentro de la etiqueta HTML de un componente con `<slot>`, será sustituido dentro del componente en el lugar en el cual fuera colocada dicha etiqueta.

```
<template>

  <my-button @click="handleClick">
    <i class="fas fa-cat"></i>
    Botón de ejemplo
  </my-button>
</template>

<script>

import MyButton from "@components/MyButton.vue";

export default {
  components: {
    MyButton
  },
  methods: {
    handleClick(info) {
      console.log("Evento click: " + info)
    }
  }
}
</script>
```



## ***PRECAUCIONES CON SLOTS***

El único inconveniente con el uso de slots es que, en aquellos equipos con varios desarrolladores de una misma aplicación, puede darse que accidentalmente se reemplace o sobrescriba un componente web, causando un error en el apartado donde este es importado con el fin de que sea insertado en un slot.

# ***NAMED SLOTS***



# ***NAMED SLOTS***

Es probable que el uso de múltiples slots sean de mucha más utilidad, en lugar de un único slot.

Esto sí es posible dentro de Vue, con la particularidad de que no podemos declarar más de un slot dentro de un bloque template, sin una distinción que permita diferenciarlo del resto. Esto, se conoce como **named slots**.



# NAMED SLOTS

```
<template>
  <div class="container">
    <header>
      <slot name="header"></slot>
    </header>
    <main>
      <slot name="main"></slot>
    </main>
    <footer>
      <slot name="footer"></slot>
    </footer>
  </div>
</template>
```

En una situación en la que necesitemos agregar slots en más de una sección, los agregamos definiendo en estos el atributo **name**, y referenciando en el mismo a qué tag HTML pertenece.

Vue detectará, al momento de importar los componentes, en cuál slot deberá volcar a cada uno.



# NAMED SLOTS

```
<div>
  <h1 slot="header">Encabezado principal</h1>
  <p slot="main">Párrafo para la sección Main</p>
  <div slot="footer">Información del contacto...</div>
</div>
```

Como alternativa, también podemos utilizar el atributo `slot`, en lugar del atributo `name`, para referenciar en qué sección del `template` debemos volcar los componentes importados que harán uso de slot para insertarse en el template en cuestión.



# NAMED SLOTS

Con la aplicación ya renderizada para el ambiente de Producción, el `template` armado que hizo uso de `slots`, se visualizará tal como muestra este bloque de código. 🙌

```
<div class="container">
  <header>
    <h1>Encabezado principal</h1>
  </header>
  <main>
    <p>Párrafo para la sección Main.</p>
  </main>
  <footer>
    <div>Información del contacto...</div>
  </footer>
</div>
```



# ***CONCEPTO DE REACTIVIDAD***



# ***CONCEPTO DE REACTIVIDAD***

La reactividad llevada al terreno del desarrollo web basado en frameworks o librerías hace referencia a un paradigma o forma característica de ejecutar acciones en base a determinadas situaciones.

Cuando un sistema es **Reactivo**, significa que el mismo cuenta con una mayor flexibilidad además de un bajo acoplamiento, lo cual facilita el desarrollo de éste y su escalabilidad, dado que estos sistemas, son susceptibles a los cambios.

# ***CARACTERÍSTICAS DE VUE***

Dos de las características más destacadas de Vue, son:

**Modularidad y Reactividad.**

En cuanto a Modularidad ponemos como ejemplo a **Angular**, como framework alternativo, quien ofrece todas las herramientas integradas, por lo tanto no necesitas acoplar nada desde afuera para sacar provecho del mismo.



# ***MODULARIDAD***

Pero esto conlleva a que tengamos con Angular un framework pesado, dado que nunca se utilizarán todas las herramientas que este trae integradas.

Así, los proyectos se harán más complejos y lentos, si es que hablamos de un desarrollo de mediana o gran escala.

Allí es donde Vue saca partida, **gracias a su modularidad.**



# ***MODULARIDAD***

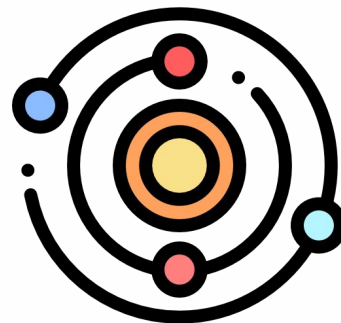
La modularidad es un gran fuerte en Vue, porque en sí, este framework ofrece lo mínimo indispensable para aplicarse a la lógica de la programación basada en javascript.

Luego, cualquier elemento que necesite complementar a este framework, se integrará de forma modular. Con esto ganamos agilidad dentro de un proyecto, el cual solo contiene cada pieza de software que es utilizada. Ni más, ni menos.



# ***REACTIVIDAD***

Y en cuanto a la capacidad de **Reactividad** de Vue, podemos definirla como una reacción que este genera ante, por ejemplo, el cambio de valor de una variable que afecta una vista. Cuando esto ocurre, Vue reacciona y actualiza la Vista con el nuevo valor de dicha variable.





# REACTIVIDAD

Un ejemplo para ello sería elaborar un contador en Vanilla JS. Dependemos de agregar en éste una línea de código que refleje en la vista de la aplicación el nuevo valor numérico por cada segundo que pasa.

Sabemos que esto no es necesario en Vue, ya que el framework se ocupará de este último paso. 💖

```
const contadorDeTiempo = () => {  
  setInterval(() => {  
    contador += 1  
    labelContador.innerText = contador  
  }, 1000);  
}
```

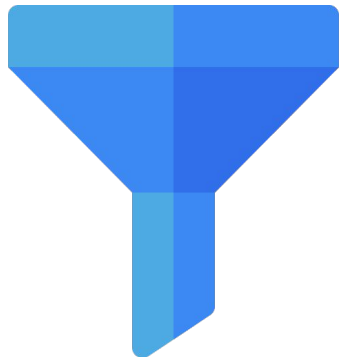
¡Viva la Reactividad!





# ***FILTROS DE VISTAS***

# ***FILTROS***



En Vue, se denomina **filtros** a la forma de crear transformaciones de datos del modelo, para que estos sean mostrados correctamente al usuario.

Vue cuenta con dos formas para declararlos:  
A través de un objeto **filter:**, dentro de la sección **<script>**, o invocando al método homónimo el cual viene integrado al objeto global **Vue**.



# ***FILTROS, vía OBJETO FILTER***

Dentro de la sección `<script>` puedes declarar el objeto `filters` y, dentro de este, agregar el o los filtros que consideres necesarios utilizar en tu aplicación.

Como siempre, deben retornar un valor sí o sí.

```
<script>
export default {
  filters: {
    capitalize: (value)=> {
      if (!value) return ''
      value = value.toString()
      return value.charAt(0).toUpperCase() + value.slice(1).toLowerCase()
    }
  }
}...
```



# ***FILTROS, vía OBJETO GLOBAL VUE***

Defines el objeto global `Vue`, junto al método `.filter()` que desees crear. Dentro de este, declaras una función anónima o arrow function, parametrizada, la cual procesará el filtro que desees aplicar para retornarte el valor ya transformado.

```
<script>
Vue.filter('formattedId', (value)=> {
  if (!value) return 0
  return value.toFixed(2)
})...
```



# ***FILTROS: IMPLEMENTAR***

En su implementación, podemos utilizarlos mediante las llaves dobles, como también aplicándolo a través de la directiva **v-bind**.

```
// Puedes aplicarlo mediante double moustaches
<p>{{ mensaje | capitalize(mensaje) }}</p>

//o a través de la directiva v-bind
<div v-bind:id="rawId | formattedId"></div>
```

Además de poder utilizarlos de forma simple, como también mediante el pasaje de parámetros dinámicos, para su transformación.

# ***MIXINS***



# ***MIXINS***

Los `mixin` son objetos que nos permiten distribuir funcionalidades reutilizables en Vue de una manera flexible, para algún componente de Vue.

Este puede contener cualquier opción de componente y, cuando un componente hace uso de un mixin, todas las opciones en este mixin se mezclarán entre las propias opciones del componente.



# MIXINS

Para crearlo, debemos definir un objeto con su propia funcionalidad.

Luego, definimos el componente que hará uso de este **mixin**, incluyendo a este dentro de dicha definición.

Finalmente, instanciamos el componente para poder hacer uso del mixin.

```
const myMixin = {
  created: function () {
    this.saludar()
  },
  methods: {
    saludar: function () {
      console.log('Saludos desde el mixin!')
    }
  }
}

// Componente que usará el mixin
const Component = Vue.extend({
  mixins: [myMixin]
})

const component = new Component()
// => "Saludos desde el mixin!"
```



# MIXINS Y FUSIONES

Cuando un `mixin`, y el componente en sí, contienen opciones o funcionalidades que se superponen, éstas se fusionarán utilizando una estrategia apropiada para evitar un conflicto en el código.

```
const mixin = {
  data: function () {
    return {
      message: 'hola',
      foo: 'abc'
    }
  }
}

new Vue({
  mixins: [mixin],
  data: function () {
    return {
      message: 'adiós',
      bar: 'def'
    }
  },
  created: function () {
    console.log(this.$data)
  }
})
// => { message: "adiós", foo: "abc",
bar: "def" }
```

# MIXIN GLOBAL

También es posible definir un mixin globalmente, aunque debemos tener precaución al crearlos porque cuando ya está aplicado de forma global este **mixin** afectará a cada instancia de Vue creada a posteriori.

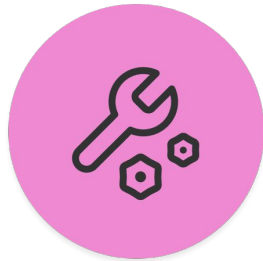
```
// inyectar un controlador para la opción personalizada `myOption`  
Vue.mixin({  
  created: function () {  
    const myOption =  
this.$options.myOption  
    if (myOption) {  
      console.log(myOption)  
    }  
  }  
})  
  
new Vue({  
  myOption: 'hola!'  
})  
// => "hola!"
```

Ejemplo  
en vivo



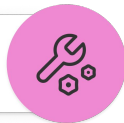
***¡VAMOS A PRACTICAR!***

***CODER HOUSE***



# ***FILTROS DE VISTA Y MIXINS***

Aplicarás lo aprendido sobre Filtros y Mixins en el siguiente ejercicio.

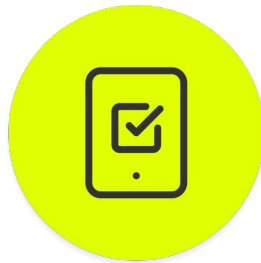


## ***CREA UN FILTRO DE VISTA Y UN MIXIN***

Deberás realizar un ejercicio donde crees un filtro personalizado para una Vista y luego lo apliques a la misma.

Por otro lado, crea al menos dos mixins y relaciónalos con alguna función o método de Vue, verificando a posteriori su correcto funcionamiento.

**Tiempo estimado:** 15 minutos.



# ***PRIMERA ENTREGA DEL PROYECTO FINAL***

Creación de un proyecto Vue CLI 2 con Bootstrap.

# PRIMERA ENTREGA DEL PROYECTO FINAL

**Formato:** nombra a tu proyecto utilizando la siguiente nomenclatura: **"1er Proyecto Final + Apellido"**.

**Sugerencia:** El proyecto debe ser entregado por la plataforma CH, en formato .ZIP (preferentemente), teniendo la precaución de **no incluir** la carpeta **/node\_modules/**.

Proyecto  
Final



1

## >>Objetivos Generales:

1. Crear el primer approach de lo que será tu proyecto final.
2. Deberá estar formateado con el framework Bootstrap.

## >>Objetivos Específicos:

- A. El proyecto deberá incluir: registro + login de usuario, listado, información y carrito.
- B. Debes definir la lógica y la estructura de la presentación de los datos en el listado de productos, la información (detalle) de los mismos, y del carrito de compras.

## >>Se debe entregar:

- Creación de componentes (*vista + lógica base + clases Bootstrap*): registro, login, listado, info y carrito
- El proyecto debe tener definida ya la lógica y estructura de representación de datos en listado, info y carrito, utilizando tablas dinámicas y tomando sus datos de archivos en formato JSON, o de propiedades creadas dentro de los componentes con información de prueba.
- **Por el momento, evita utilizar plataformas online para suministrar los datos remotos, así es más fácil poder evaluar la correcta definición de los mismos.**

***¿PREGUNTAS?***

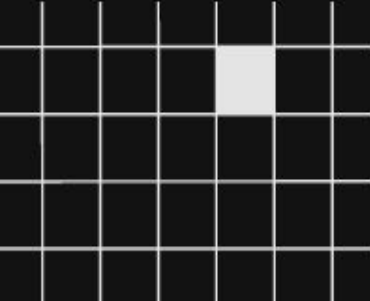






# ***¡MUCHAS GRACIAS!***

## **Resumen de lo visto en clase hoy:**

- Comunicación entre componentes con props
  - Manejo de Eventos del framework y personalizados
  - Slots
  - Concepto de Reactividad
  - Filtro de Vistas
  - Mixins
- 



***OPINA Y VALORA ESTA CLASE***