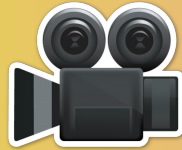




Clase 6. Vue JS

# ***Proyecto Vue CLI***

***RECORDÁ PONER A GRABAR LA CLASE***





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

- Crear componentes para Vue CLI.
- Manejar su estructura interna y el pasaje de parámetros por props.

# ***CRONOGRAMA DEL CURSO***

## Clase 5



### **VueJS en NodeJs**



USANDO VUE CLI 2



SERVIDOR DE DESARROLLO  
AUTOMÁTICO

## Clase 6



### **Proyecto Vue CLI**



COMPONENTES CON VUE  
CLI



INSTANCIANDO  
COMPONENTES Y PROPS

## Clase 7



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



COMPONENTES ANIDADOS



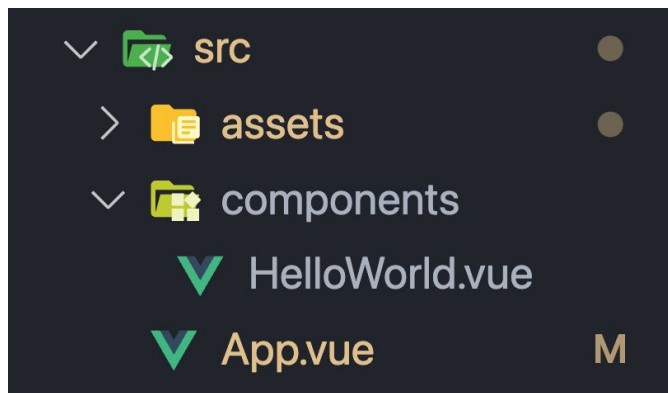
FILTROS DE VISTA



PRIMERA ENTREGA  
PROYECTO FINAL

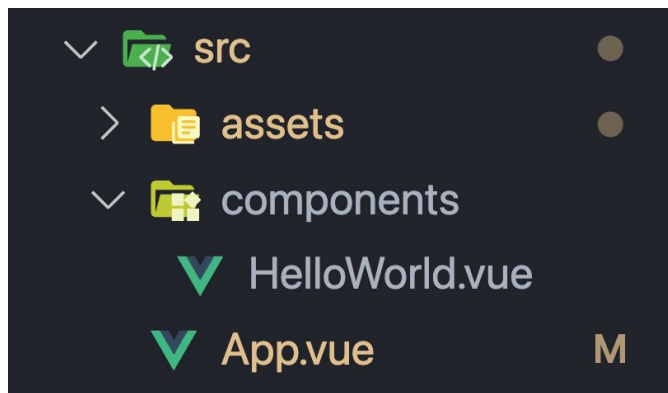
# ***ESTRUCTURA DE COMPONENTES WEB EN VUE/CLI***

# LOS ARCHIVOS .VUE



En la **Clase 04** hicimos mención de que Vue cuenta con su extensión de archivos (**.vue**) a través de la cual se generan los componentes web y que, en Vue/Cli, la extensión .vue es el formato predeterminado a indicar cuando los creamos.

# LOS ARCHIVOS .VUE



Y, también en la clase pasada, vimos que éstos se almacenan en la carpeta `src/components`, y hasta podemos crear subcarpetas si son demasiados los componentes a crear.



# SECCIONES DEL COMPONENTE

Podemos apreciar en el código de este componente web que todo su contenido se distribuye en tres secciones:

- `<template>`
- `<script>`
- `<style>`

Esto difiere bastante del paradigma de *web components* en Vue 2 que aprendimos algunas clases atrás.

```
<template>
  <div class="hello">
    <h1>{{ msg }}</h1>
  </div>
</template>

<script>
export default {
  name: 'HelloWorld',
  props: {
    msg: String
  }
}
</script>

<style scoped>
h3 {
  margin: 40px 0 0;
}
...
</style>
```





# SECCIÓN: <TEMPLATE>

```
<template>
  <div class="hello">
    <h1>{{ msg }}</h1>
  </div>
</template>
```

El o los tags HTML que conforman un componente web se encierran entre las etiquetas `<template>`. Además, debemos seguir evitando crear tags del tipo contenedor en un mismo nivel de anidamiento.

También, dentro de cada elemento HTML que agregamos, podemos definir una o más **props**, las cuales se transformarán en **atributos** cuando el componente se integre en un documento HTML.



## SECCIÓN: **<SCRIPT>** (1/2)

```
<script>
export default {
  name: 'HelloWorld',
  props: {
    msg: String
  }
}
</script>
```

La sección `<script>` soporta, por supuesto, código JavaScript.

Está pensada para volcar toda una lógica de funcionamiento a nivel Componente Web y, utilizar algoritmos de interacción interna, o para que el componente web interactúe con otros.

También se agrega el comando `export`, el atributo `name`, referenciando al nombre de este componente, y las `props`, que utilizará la aplicación. Todo, en formato objeto JSON.



## SECCIÓN: **<SCRIPT>** (2/2)

```
<script>
import ImageHelloWorld from './imageHW'
export default {
  name: 'HelloWorld',
  ...
```

En el caso de que tengamos que desarrollar componentes complejos y necesitemos incluir otros componentes dentro de éste, podemos importar los mismos referenciándolos mediante la palabra reservada `import` seguida del nombre del componente y la ruta donde está almacenado.



# SECCIÓN: <STYLE>

```
<style scoped>
#app {
  font-family: Avenir, Helvetica;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
  text-align: center;
  color: #2c3e50;
  margin-top: 60px;
}
</style>
```

Nos queda el tag donde volcamos todos los estilos CSS que deban utilizarse en el componente o a nivel **Vue** general. En cuanto a estilos a utilizar, podemos usar:

- estilos genéricos
- estilos sobre id's
- estilos sobre tags HTML
- generar clases CSS

## ***SECCIÓN: <STYLE>***

En aquellos casos que necesitemos crear uno o más estilos CSS sobre elementos de este componente, y que éstos últimos no deban ser alterados por un framework CSS u hoja de estilos externa, debemos agregar el atributo **scoped**, contiguo al tag **<style>**.

Con esto, le dejaremos en claro a **Webpack** cuál es el alcance de los estilos aplicados en este componente y hasta nos ayudará a prescindir de tener que crear Clases CSS locales por cada tag HTML.

```
<style scoped>
```

# ***SINGLE FILE COMPONENT***



**En resumen:** Vue permite generar componentes en un único archivo aunando la vista, el diseño y lógica, de manera tal que podamos reutilizarlo dónde y cuántas veces lo consideremos necesario.



# ***SINGLE FILE COMPONENT***



Estos componentes web son llamados **Single File Component** y, combinados al módulo Vue Loader y WebPack, generaremos el HTML, JS y Estilos CSS de cada módulo para llevarlos luego al ambiente de Producción.

***APP.VUE***



# ***APP.VUE***



Una vez que definimos todos los componentes que conformarán nuestra aplicación Vue/Cli, debemos continuar trabajando sobre el archivo `app.vue`.



Edita a continuación este archivo, en Visual Studio Code, para entender un poco su estructura.



# APP.VUE

```
src > App.vue > {} "App.vue"
1  <template>
2    <div id="app">
3      
4      <HelloWorld msg="Bienvenidos a CODERHOUSE."/>
5    </div>
6  </template>
7
8  <script>
9    import HelloWorld from '../components/HelloWorld.vue'
10
11    export default {
12      name: 'App',
13      components: {
14        HelloWorld
15      }
16    }
17  </script>
18
19  <style scoped>
20  #app {
21    font-family: Avenir, Helvetica, Arial, sans-serif;
22    -webkit-font-smoothing: antialiased;
23    -moz-osx-font-smoothing: grayscale;
24
25    text-align: center;
26    color: #2c3e50;
27    margin-top: 60px;
28  }
29  </style>
```

Como podemos ver en su estructura, no difiere de los otros componentes web que podemos desarrollar para darle vida a una aplicación Vue.

Cuenta también con las secciones `<template>`, `<script>` y `<style>`, las cuales concentran las vistas, lógica y estilos CSS, igual que el resto.



# APP.VUE: <TEMPLATE>

El apartado Template incluye la estructura base de este tipo de sección, además de poder incluir un componente web central, que gestione elementos HTML que aportarán a la Vista general de la aplicación.

Además, se ocupa de desplegar uno o más componentes web creados en archivos individuales.

```
✓ App.vue > {} "App.vue"
<template>
  <div id="app">
    
    <HelloWorld msg="Bienvenidos a CODERHOUSE."/>
  </div>
</template>
```



# APP.VUE: <TEMPLATE>

```
<template>
  <div id="app">
    
    <HelloWorld msg="Bienvenidos a CODERHOUSE."/>
  </div>
</template>
```

Como podemos apreciar, **app.vue** está haciendo uso en este ejemplo del componente **HelloWorld**, creado junto a la base de un nuevo proyecto Vue/Cli.

Además de invocarlo como un componente web, utiliza su atributo **msg**, para configurarle un texto a visualizar.



# APP.VUE: <SCRIPT>

El apartado Script cumple la función de concentrar todos los componentes web que incluye la aplicación, importándolos para construir la Vista.

Además, cuenta con propiedades como **name**, la cual le da el nombre general a la aplicación, y **components**, quien disponibiliza todos los componentes que la aplicación Vue utilizará.

```
<script>
import HelloWorld from './components/HelloWorld.vue'

export default {
  name: 'App',
  components: {
    HelloWorld
  }
}
</script>
```



# APP.VUE: <STYLE>

El apartado Style se ocupa de concentrar todos los estilos o clases CSS que luego se pueden aplicar a la Vista general y a cada uno de los web components que esta incluya.

👉 Recordemos que si un web component utiliza internamente el atributo scoped, sus estilos internos declarados serán priorizados por los estilos que se apliquen desde App.vue.

```
</script>

<style scoped>
#app {
  font-family: Avenir, Helvetica, Arial, sans-serif;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
  text-align: center;
  color: #2c3e50;
  margin-top: 60px;
}
</style>
```

***MAIN.JS***



# MAIN.JS (1/2)

```
Js main.js > ...  
import Vue from 'vue'  
import App from './App.vue'  
  
Vue.config.productionTip = false  
  
new Vue({  
  render: h => h(App),  
}).$mount('#app')
```

Finalmente llegamos al archivo `main.js`.

Este se ocupa de utilizar el módulo `import`, trayendo a `App.vue` y al framework `Vue` en sí.

Además, puede incluir parámetros de configuración generales y se ocupa de instanciar el objeto `Vue` pasándole `App` (*instancia donde fue importado `App.vue`*), como objeto...





# MAIN.JS (2/2)

```
Js main.js > ...  
import Vue from 'vue'  
import App from './App.vue'  
  
Vue.config.productionTip = false  
  
new Vue({  
  render: h => h(App),  
}).$mount('#app')
```

Al final de todo, vemos que invoca el método `$mount()`, parametrizándolo con el id `#app`.

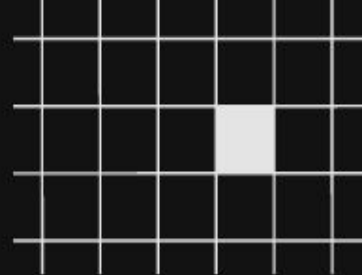
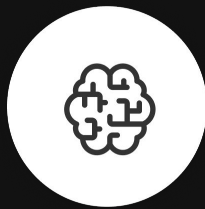
Este método se ocupa de montar la aplicación Vue. Va referenciado contigo a la instancia del objeto `Vue()` y podemos definirlo como el equivalente a la propiedad `el`, de Vue CDN.



# ***MAIN.JS: INTEGRACIÓN DE VUE***

```
import Vue from 'vue'  
import App from './App.vue'
```

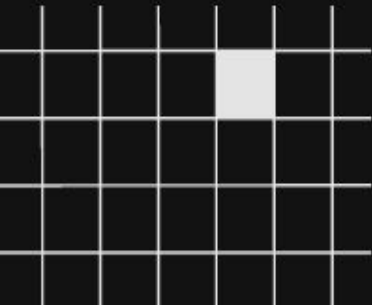
Como mencionamos antes, `main.js` es quien integra a Vue, con la diferencia de que en lugar de invocarlo vía CDN, lo trae dentro del proyecto mediante la importación de módulos de ECMAScript.



***¡PARA PENSAR!***

*¿Dónde creés que está Vue si no tiene un path definido?*

CONTESTA EN EL CHAT DE ZOOM





# MAIN.JS: INTEGRACIÓN DE VUE

¡Bien! Ahí es donde Node.js aporta lo suyo, concentrando la instalación de los módulos de cada herramienta, librería o complemento que tengamos que utilizar.

En la carpeta `/node_modules/vue/` encontrarás todo el contenido de Vue separado por módulos, y es allí desde donde es importado a nuestro proyecto.

vue

dist

i README.md

JS vue.common.dev.js

JS vue.common.js

JS vue.common.prod.js

JS vue.esm.browser.js

JS vue.esm.browser.min.js

JS vue.esm.js

JS vue.js

JS vue.min.js

JS vue.runtime.common.dev.js

JS vue.runtime.common.js

JS vue.runtime.common.prod.js

JS vue.runtime.esm.js

JS vue.runtime.js

JS vue.runtime.min.js

***INDEX.HTML***



# INDEX.HTML (1/2)

```
> index.html > ...
<!DOCTYPE html>
<html lang="">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width,initial-scale=1.0">
    <link rel="icon" href="<%= BASE_URL %>favicon.ico">
    <title><%= htmlWebpackPlugin.options.title %></title>
  </head>
  <body>
    <noscript>
      <strong>We're sorry but <%= htmlWebpackPlugin.options.title %>
        doesn't work properly without JavaScript enabled. Please enable it to
        continue.</strong>
    </noscript>
    <div id="app"></div>
    <!-- built files will be auto injected -->
  </body>
</html>
```

`index.html` es el archivo que se ocupará de recibir todos los componentes web de Vue, como también `App.vue` y `main.js` cuando Webpack haya transpilado todo a un código comprensible por el navegador.

👉 Aquí encontramos prácticamente todo lo que un archivo HTML contiene, con unas mínimas salvedades. Veamos a continuación:





# INDEX.HTML (2/2)

```
> index.html > ...
<!DOCTYPE html>
<html lang="">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width,initial-scale=1.0">
    <link rel="icon" href="<%= BASE_URL %>favicon.ico">
    <title><%= htmlWebpackPlugin.options.title %></title>
  </head>
  <body>
    <noscript>
      <strong>We're sorry but <%= htmlWebpackPlugin.options.title %>
        doesn't work properly without JavaScript enabled. Please enable it to
        continue.</strong>
    </noscript>
    <div id="app"></div>
    <!-- built files will be auto injected -->
  </body>
</html>
```

`<title>` tiene definido una ruta de nombre que Webpack completará, luego de transpilar el proyecto.

`<noscript>` tag HTML que solo es visible en aquellos navegadores web que no tengan JavaScript activo.

`<div id="app">` es el tag HTML con un ID especificado, donde el transpilador escribirá el código Vue, convertido previamente a HTML, CSS y JS puro.

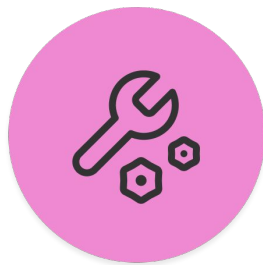
# ***TRANSFORMAR PROYECTOS DE VUE CDN A VUE-CLI***

Como pudimos ver, el cambio de Vue-CDN a Vue/Cli es un cambio radical en la forma de estructurar un proyecto.

En cuanto a programación, el mayor cambio lo notamos al tener que desarrollar los componentes web en archivos independientes, y luego importarlos hacia otros componentes web para su reutilización.

Finalmente, la forma en cómo distribuimos el proyecto Vue/Cli lo aprenderemos cuando veamos cómo mover un proyecto hacia el ambiente de producción.



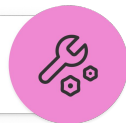


# ***COMPONENTES CON VUE CLI***

Crea componentes web para Vue/Cli.

# ***COMPONENTES EN VUE CLI***

Desafío  
generico



Deberás crear al menos dos componentes en un proyecto Vue/Cli, basándote en el paradigma de Single File Component.

Cada componente web deberá contar con las secciones template, script y style, y con un mínimo contenido en cada una de ellas.

Finalmente, importa tus componentes en App.vue para verlos funcionando.

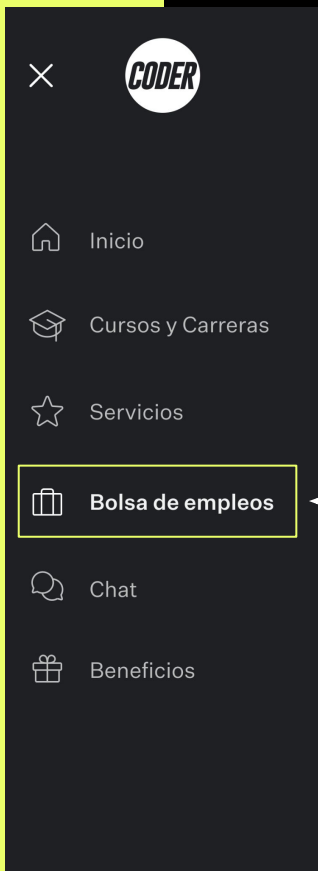
**Tiempo estimado:** 20 minutos.

***CODER HOUSE***



***BREAK***

**¡5/10 MINUTOS Y VOLVEMOS!**



Nuevo

# ¡Lanzamos la Bolsa de Empleos!

Un espacio para seguir **potenciando tu carrera** y que tengas más **oportunidades de inserción laboral**.

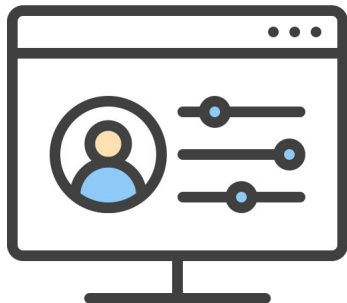
Podrás encontrar la **Bolsa de Empleos** en el menú izquierdo de la plataforma.

Te invitamos a conocerla y ¡postularte a tu futuro trabajo!

Conócela

***VETUR Y .EDITORCONFIG***

# ***VETUR Y .EDITORCONFIG***



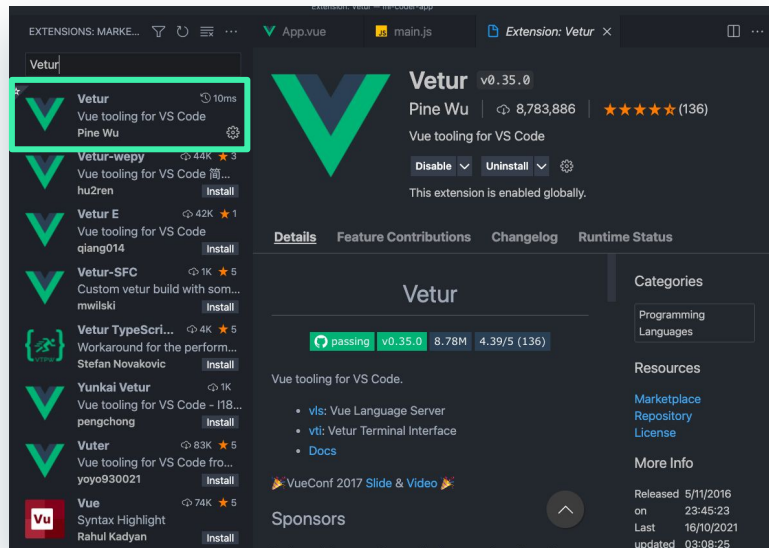
En la clase anterior nos dedicamos a preparar el ambiente de trabajo y a conocer el por qué de cada herramienta instalada.

Esto, permitió dejar listo el terreno para el cambio en la forma de codificar aplicaciones Vue desde hoy en adelante.

Para culminar la configuración de nuestro espacio de trabajo, te sugerimos, a continuación, sumar dos extensiones más.



# VETUR PARA VS CODE



En el apartado **Extensiones de VS Code**, escribe en el buscador: [Vetur](#). Esta provee un coloreado de sintaxis apropiado, para poder identificar correctamente el código de Vue que escribiremos de aquí en más.

De todas las opciones arrojadas como resultado, elijamos la del autor: *Pine Wu*.



# ***.EDITORCONFIG***

Como opcional, te sugerimos también  
agregar **EditorConfig** para VS Code.



Luego, agrega en el raíz del proyecto un  
archivo llamado **.editorconfig** con la  
siguiente configuración optimizada para  
Vue.

```
1 root = true
2
3 [*]
4 charset = utf-8
5 indent_style = space
6 indent_size = 2
7 insert_final_newline = true
8 trim_trailing_whitespace = true
```



# ***ARCHIVO DE CONFIGURACIÓN VUE.CONFIG.JS***

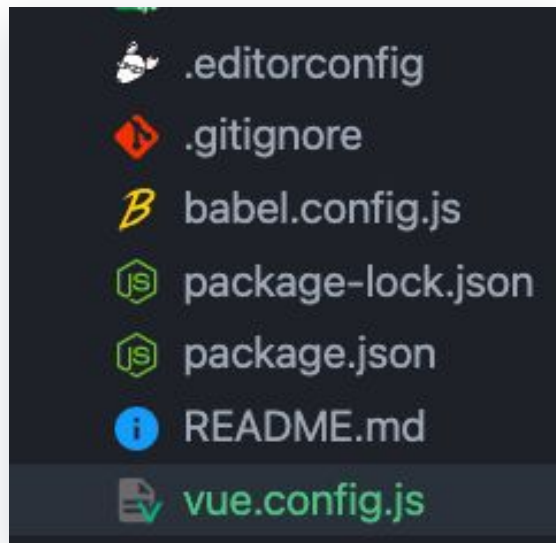
# ***VUE.CONFIG.JS***

Existen varias alternativas para configurar parámetros de funcionamiento de Vue, durante la etapa de desarrollo o despliegue de un proyecto.

Una de ellas, es el a través del archivo  
`vue.config.js`.



# VUE.CONFIG.JS



Este archivo es un archivo de configuración (*opcional*) para Vue, que será cargado y procesado por `vue/cli-service`, si lo encuentra presente en la carpeta raíz de nuestro proyecto Vue/Cli.



# VUE.CONFIG.JS

*¿Qué podemos configurar en este archivo?*

Parámetros generales que aporten al funcionamiento de Vue/Cli, manejo de alertas, errores, forma de uso de la Terminal, ignorar elementos, entre otras opciones.

**Este archivo cobrará gran protagonismo cuando preparemos el proyecto para el ambiente de producción.**

```
vue.config.js > ...
Vue.config.devtools = true
Vue.config.silent = true
Vue.config.errorHandler = function (err, vm, info) {
  // manejo de errores
  // `info` es información específica de Vue, por
  // ejemplo, en cual _hook_ del ciclo de vida fue
  // encontrado el error.
  // disponible en 2.2.0+
}
Vue.config.warnHandler = function (msg, vm, trace) {
  // `trace`: traza de jerarquía de componentes.
}
Vue.config.ignoredElements = [
  'my-custom-web-component',
  'another-web-component',
  // Use una `RegExp` para ignorar elementos que
  // comienzan con "ion-"
  /^ion-/
]
Vue.config.keyCodes = {
  v: 86,
  f1: 112, // camelCase no funcionará
  mediaPlayPause: 179,
  // puede usar kebab-case con comillas dobles
  "media-play-pause": 179,
  up: [38, 87]
}
```

# ***OPCIONES DE CONFIGURACIÓN GLOBAL PARA VUE/CLI***

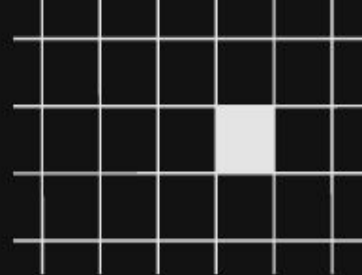
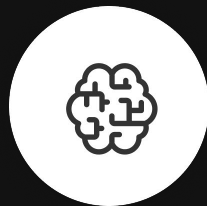
Además de poder parametrizar de forma global el funcionamiento de Vue/Cli a través de `vue.config.js`, contamos con otras opciones, como ser:

- La propiedad `Vue` en el archivo `package.json`
- La configuración global CLI en el archivo `.vuerc`

Cualquiera de estas opciones permitirán establecer configuraciones globales para el uso de Vue/Cli y podremos consultar las mismas mediante el comando de

Terminal: `vue config`.

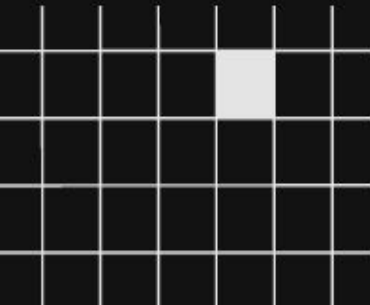
# ***USO DE PROPS***



## ***¡PARA PENSAR!***

*En las clases 2, 3, y 4, hicimos un uso intensivo de **props** a lo largo de cada tema explicado. Entonces:*

*¿Podemos considerar a las props como una funcionalidad equivalente en Vue, a lo que son las variables, constantes y/o arrays para con Vanilla JavaScript?*



CONTESTA LA ENCUESTA DE ZOOM





# USO DE PROPS

En la **Clase 04** conocimos las **props** y su uso dentro de la creación de un componente.

Se crean para definir nombres de propiedades y envían sus valores al componente web para, finalmente, convertirse en atributos configurables mediante **v-bind**.

```
//Declaración de props
props: ['cover', 'title', ...]

<!-- uso de props en el componente -->
<mi-componente class="card"
  :cover="portada"
  :title="título de la peli">
</mi-componente>
```





# PROPS TIPADAS

Otro detalle que aprendimos es que las props pueden ser tipadas. Si bien podemos declararlas en formato del tipo array, asumen un valor por defecto del tipo **string**.

En cambio, si las declaramos de forma tipada nos aseguramos que cada una de ellas reciba el valor correcto cuando se implementan en un desarrollo.

```
//props en formato array
props: ['cover', 'title', ...]

<!-- props tipadas -->
export default {
  name: ...,
  props: {
    error: Number,
    mensaje: String,
    icono: String,
    fechaError: Date
  }
}
```



# ***PROPS TIPADAS***

```
<!-- props tipadas -->
export default {
  name: ...,
  props: {
    error: Number,
    mensaje: String,
    icono: String,
    fechaError: Date
  }
}
```

Por lo tanto, en Vue/Cli recomendamos que las props sean declarados de forma tipada, armando un objeto de propiedades junto a su tipo de valor.

No solo desarrollaremos mejores componentes web, sino que también, cuando le enviemos un parámetro equivocado a éstas, el error aparecerá en la Consola JS de forma clara.

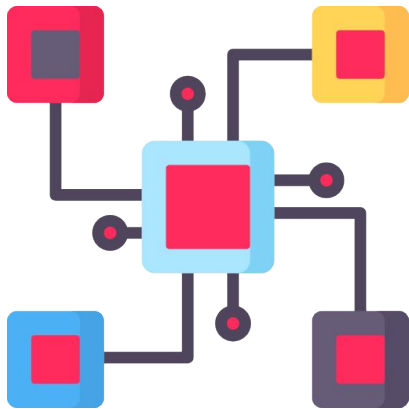
***APLICAR LA LÓGICA DE CÓDIGO***

# ***APLICAR LA LÓGICA DE CÓDIGO***

Dentro del bloque de código `<script>`, definiremos todos los valores y métodos necesarios para que nuestra app Vue/Cli, funcione correctamente.

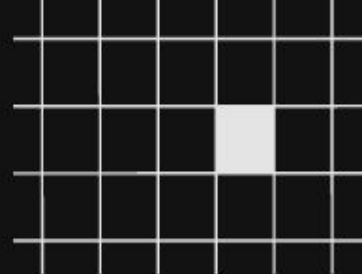
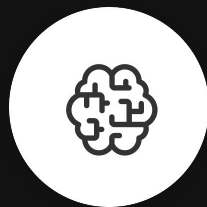
Y, como aquí trabajamos con componentes, debemos pensar bien el código para que, procese y extraiga los valores de cada dato que se deba compartir entre más de un componente web.

# ***DATA***



Recordemos que a través de **data** podemos definir todo tipo de variables, constantes y arrays, de uso interno para el componente.

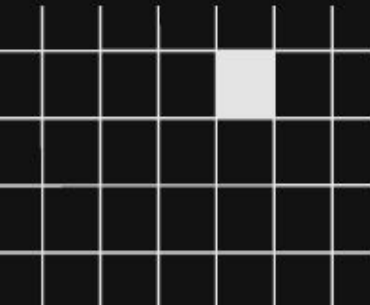
Y data llevado al espacio de Vue/Cli, deberá ser repensado para que funcione de forma interna o exportando datos a través de algún método acorde.



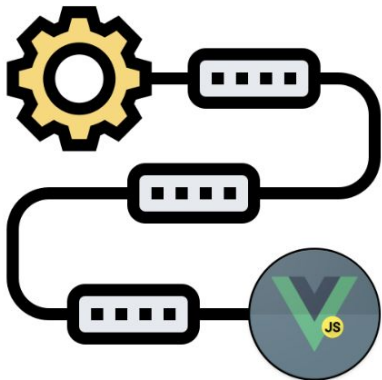
## ***¡PARA PENSAR!***

- 1. ¿Cómo puedo hacer dentro de un method, para conocer el total de elementos almacenados en un array declarado en data?*
- 2. Y en el caso que necesitemos agregar un elemento nuevo al array, ¿qué método de éste último deberíamos utilizar?*

CONTESTA EN EL CHAT DE ZOOM



# ***METHODS***



Los métodos nos permiten definir funciones de uso interno; todas las que necesitemos.

Estas se ocupan de interactuar luego con la Vista de nuestro componente web usando las directivas acordes para dicha tarea.

# ***PROPIEDADES COMPUTADAS***



Y las propiedades computadas son aquellas que se asociaban a componentes y que se le podían aplicar cálculos o transformaciones de datos de nuestra App Vue, gracias a que siempre retornan un valor con su ejecución, funcionando de forma similar a **getter**.

Además, las escribimos como una variable o propiedad, en lugar de como un método.





# VALORES EN PROPIEDADES COMPUTADAS

```
computed: {  
  nombreCompleto: {  
    get: () => {  
      return 'Coder ' + 'House'  
    }  
  }  
}
```

Cuando creamos una propiedad computada como las anteriores, estamos indicando que son de tipo **getter** por defecto. Es decir, solo son ejecutadas cuando el template pide obtener su valor.



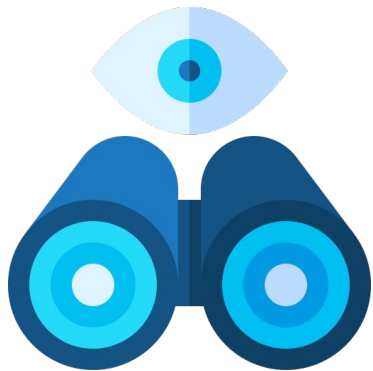
# VALORES EN PROPIEDADES COMPUTADAS

Y, en el caso que deseemos que la propiedad computada se ocupe de asignar el valor a alguna **propiedad** o **data**, debemos agregar en su declaración, la propiedad **set**, junto al nuevo valor que deseamos aplicar.

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

***WATCHERS***

# ***WATCHERS***



Vue proporciona una opción genérica para generar reacciones ante cambio de datos generados en las propiedades computadas.

Esta opción se denomina `watch` y es ideal para implementar en ámbitos donde ocurren operaciones asincrónicas o con mucha carga de tiempo del lado del servidor.



# WATCHERS

Por ejemplo, podemos activar un **watcher** para que una petición del tipo API Restful espere a que el usuario finalice la carga de datos en un campo de formulario.

Esta finalidad tal vez nos obligue a escribir más código, pero es realmente efectiva cuando debemos optimizar al máximo una aplicación Vue.

```
watch: {
  question: (newQuestion, oldQuestion)=> {
    this.answer = 'Esperando que deje de escribir...'
    ...
  },
  methods: {
    getAnswer: ()=> {
      if (this.question.indexOf('?') === -1) {
        this.answer = 'Las preguntas contienen signo de
interrogación. ;-)'
        return
      }
      this.answer = 'Pensando...'
      const vm = this
      axios.get('https://yesno.wtf/api')
        .then((response)=> {
          vm.answer = _.capitalize(response.data.answer)
        })
        .catch((error)=> {
          vm.answer = `La API no respondió: ${error}`
        })
    }
  }
}
```

Ejemplo  
en vivo

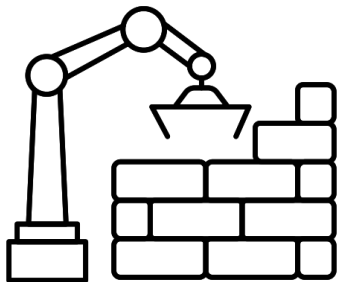


***¡VAMOS A PRACTICAR!***

***CODER HOUSE***

# ***VUE GENERATE COMPONENT***

# ***VUE GENERATE COMPONENT***



De la misma forma que aprendimos a crear componentes web con Vue-Cli, este fabuloso framework cuenta también con una opción automatizada, denominada **Vue Generate Component**.

La misma nos permite generar la estructura de un componente web de forma automática, dentro de nuestra aplicación, a través de la ventana Terminal.





# ***VUE GENERATE COMPONENT***

Para instalarlo, ejecutamos en la ventana Terminal, el comando:

```
npm install -g vue-generate-component
```

```
TERMINAL  PROBLEMS  OUTPUT  DEBUG CONSOLE  > node  +  -  [ ]  [ ]  ^  X
fernando@MacBook-Air bootstrap-app % npm install vue-generate-component
(( )) :: reify: timing arborist:ctor Completed in 0ms
```

# ***VUE GENERATE COMPONENT***

Luego de instalado, y desde nuestro proyecto, ejecutamos el comando: **vgc <nombre-del-componente>**

Finalizado el proceso, este comando creará 5 archivos diferentes dentro de nuestro proyecto, cada uno de ellos con el nombre elegido para el componente.

# ***VUE GENERATE COMPONENT***

Si, por ejemplo, nuestro componente creado con `vgc` lo llamamos `footer`, encontraremos los siguientes archivos generados:

- `footer.js`
- `footer.spec.js`
- `footer.html`
- `index.vue`
- `my-directive.directive.js`



# ***VUE GENERATE COMPONENT***

Al editar cada uno de estos archivos con VS Code, encontrarás la estructura del archivo correspondiente a un componente web con el código base creado.

Aquí, un ejemplo de cómo se ve el archivo `footer.js`. 🙌

Si estás acostumbrado a otros frameworks con herramientas similares, entonces podrás sacarle el máximo provecho a **vgc**.

```
export default {  
  name: 'footer',  
  props: [],  
  mounted() {},  
  data() {  
    return {};  
  },  
  methods: {},  
  computed: {}  
};
```

***VGC SINGLE FILE***

# ***VGC SINGLE FILE***

A través de VGC tenemos la posibilidad de crear también archivos de componente simples.

Para ello, debemos ejecutar a través de la Terminal, el comando:

```
vgc <nombre-del-componente> -s
```

Al finalizar el proceso, encontraremos dentro de nuestro proyecto una nueva carpeta con el nombre del componente y, dentro de ella...



# VGC SINGLE FILE

```
▼ ayuda
  <> ayuda.html
  JS ayuda.js
  ♂ ayuda.scss
  JS ayuda.spec.js
  ▼ index.vue
  > contacto
```

...cada uno de los archivos que conforma el componente, separados por tipo de lenguaje.

De esta forma, tenemos toda la lógica de un componente separada, más un archivo denominado `index.vue`, el cual referencia a cada sección que corresponde a un componente Vue.



# VGC SINGLE FILE

```
ayuda > ▼ index.vue > ...  
1   <template src="./ayuda.html"></template>  
2   <script src="./ayuda.js"></script>  
3   <style src="./ayuda.scss" scoped lang="scss"></style>  
4
```

Si buscas crear el componente en alguna carpeta o subcarpeta específica de tu proyecto, puedes especificarlo a través de la Terminal.

```
vgc <nombre-del-componente> -s --carpeta
```





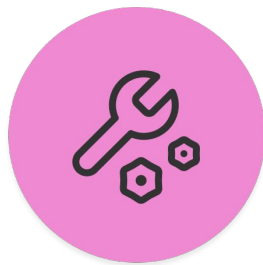
# ***RECOMENDACIONES SOBRE VGC***

Las herramientas de generación automática de componentes, crean muchos archivos con contenido pre-armado. En situaciones donde ya tenemos experiencia sobre cómo aprovechar estas Tools, no hay problema en sacar todo el provecho de ellas.



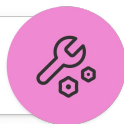
# ***RECOMENDACIONES SOBRE VGC***

Pero si aún te encuentras dando tus primeros pasos en la generación de componentes, te recomendamos que sigas creándolos desde cero, para ganar el conocimiento necesario y no perder tanto tiempo viendo qué código base del que se generó te sirve, y cuál no.



# ***INSTANCIA COMPONENTES Y PROPS***

Instancia los componentes creados anteriormente en App.vue.



## ***COMPONENTES EN VUE CLI***

Continúa trabajando con el ejercicio anterior.

Deberás instanciar cada uno de los componentes web creados en App.vue y hacer que los mismos se utilicen finalmente cuando ejecutamos nuestra App Vue.

Verifica que un componente pueda pasarle parámetros por props como también que se comuniquen mínimamente entre ellos.

**Tiempo estimado:** 20 minutos.



# ***DESAFÍO DE VUE CDN A VUE CLI***

Convierte tu desafío de la Clase 04 de Vue CDN a Vue Cli.

# DESAFÍO DE VUE CDN A VUE CLI

**Formato:** Sube tu desafío complementario a la Plataforma de Coderhouse. Su nombre deberá ser “Componentes vue cdn a vue cli + Tu Apellido”.

**Sugerencia:** N/A.

Desafío  
Complementario



>> **Consigna:** Recupera tu 2º Desafío genérico realizado en la Clase 04, y conviértelo a Vue Cli.

## >>Aspectos a incluir en el entregable:

En la clase 04 realizaste un desafío genérico generando múltiples instancias de un componente con Vue CDN, incluyendo pasaje de parámetros por props que cambiaban características de cada uno de los componentes que replicaste.

Toma ese mismo desafío, y conviértelo a Vue Cli, respetando las mismas funcionalidades y parametrizaciones por props, aplicadas oportunamente.

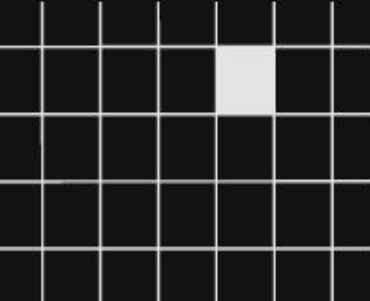
***¿PREGUNTAS?***





# ***¡MUCHAS GRACIAS!***

Resumen de lo visto en clase hoy:

- Single File Component
  - Transformación de proyectos a Vue/Cli
  - Uso de props con o sin tipado
  - Uso de data, methods, computed y watchers
- 





***OPINA Y VALORA ESTA CLASE***