sustantiva
CON•SENTIDO

Energiza!

COMPONENTES Y SU CICLO DE VIDA EN VUE

Contenidos

Componentes y su ciclo de vida en Vue

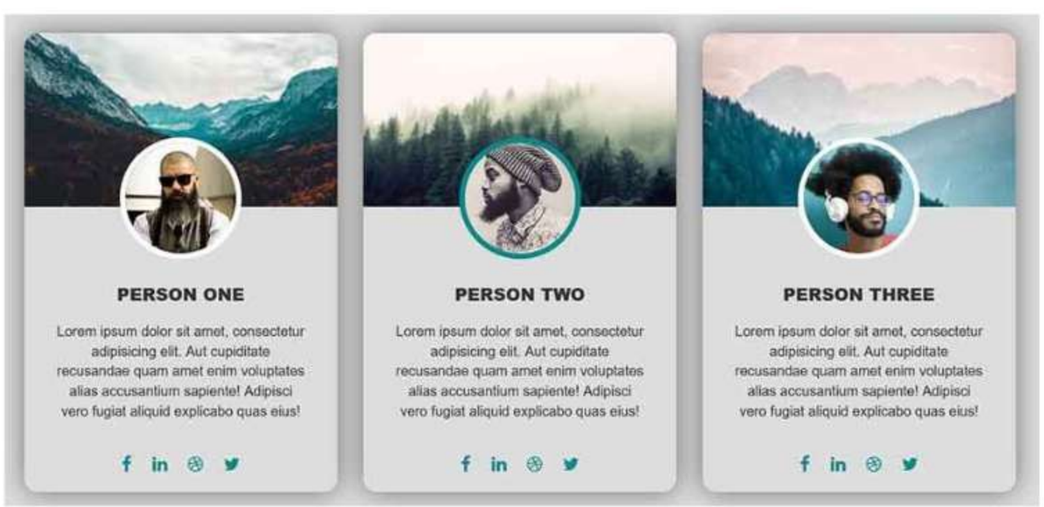
- Componentización
- Reutilización de componentes y Modularización
- Jerarquía de componentes
- Paso de datos a un componente mediante Props
- Enviar mensajes al componente padre
- Emitir eventos a componente padre
- Distribución de contenido en Slots
- Componentes dinámicos
- Ciclo de vida de un componente
- Cuándo usar los hooks
- Tipos de hooks y su función

Aplicar estilo a un componente

- Usando la etiqueta style
- Class binding
- Style binding

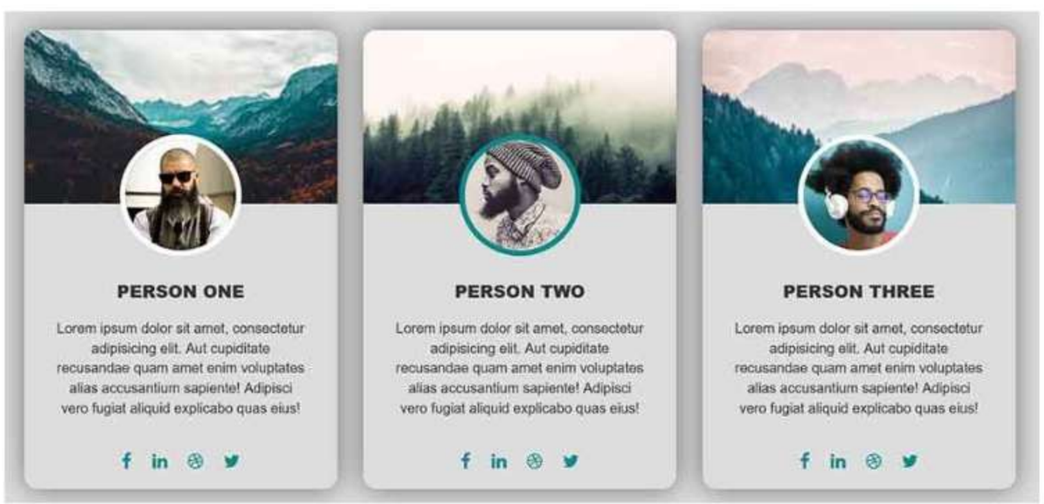
Componentización

La componentización tiene como objetivo facilitar el desarrollo de software mediante el uso de componentes reutilizables que se conectan entre sí mediante interfaces estándar. Los componentes en sí deben ajustarse a un modelo conocido que dicta cómo se conectan. A través de la componentización puedes dividir tu código en varias partes, tanto para una mejor facilidad de mantenimiento, como para su reutilización, llegando a llamarlas tantas veces como sea necesario, incluso teniendo que cambiar los datos.



Componentización

Podemos ver algunos elementos que se pueden dividir en componentes, como estas tarjetas, por ejemplo. Es un componente dinámico, ya que las tarjetas se ven iguales, solo cambia su contenido. Como tenemos 3 de estas tarjetas, podríamos ponerlas dentro de un for y pasar los datos de texto que están en estos objetos de matriz para crear cada una. Lo importante al dividir en componentes es poder identificar esto en el diseño, tratando de hacer algo reutilizable o simplemente separando su código para una mejor mantenibilidad, ya que hay componentes que no necesitan ser reutilizables, estos componentes se denominan instancia única.



Componentización

Aplicando el mismo concepto para una tabla podría tenerse el siguiente Array de Objetos:

```
personas: [
                {
                    id: 1,
                    nombre: 'Blanca',
                    apellido: 'Nieves',
                },
                {
                    id: 2,
                    nombre: 'Diego',
                    apellido: 'Pino',
                },
                {
                    id: 3,
                    nombre: 'Andrea',
                    apellido: 'Bouffanais',
                },
            ],
```

Se almacena como data object en la app que se montará en el index. Además importamos el componente que servirá para la componentización:

```
<script>
    import TablaPersonas from '@/components/TablaPersonas.vue'
    export default {
        name: 'app',
        components: {
            TablaPersonas,
        },
        data() {
            return {
//Aquí ingresamos el Array de objetos
            }
        },
    }
</script>
```

Componentización

Dentro del template del componente TablaPersonas se recorrerá con for cada persona del Array y ese llenarán las filas.

```
<template>
    <div id="tabla-personas">
        <table class="table">
            <thead>
                <tr>
                    <th>Nombre</th>
                    <th>Apellido</th>
                </tr>
            </thead>
            <tbody>
                <tr v-for="persona in personas" :key="persona.id">
                    <td>
                        {{ persona.nombre}}
                    </td>
                    <td>
                        {{ persona.apellido}}
                    </td>

                </tr>
            </tbody>
        </table>
    </div>
    </template>
```

No olvides exportar tu componente. Con props indicas el nombre y tipo de la propiedad que deseas utilizar.

```
<script>
    export default {
        name: 'tabla-personas',
        props: {
            personas: Array,
        },
      }
    </script>
```

Componentización

Para proporcionar a Vue una sugerencia para que pueda rastrear la identidad de cada nodo y, por lo tanto, reutilizar y reordenar los elementos existentes, debe proporcionar un atributo key único para cada elemento. Un valor ideal para key sería el ID único de cada elemento. Al funcionar como un atributo, necesitas usar v-bind para enlazarlo con valores dinámicos. Por último, en el template de tu app.vue debes enlazar la plantilla que será sobre la que se añada la tabla por su nombre (name).

```
<template>
    <div id="app" class="container">
        <div class="row">
            <div class="col-md-12">
                <h1>Personas</h1>
            </div>
        </div>
        <div class="row">
            <div class="col-md-12">
                <tabla-personas
:personas="personas"  />
            </div>
        </div>
    </div>
</template>
```

Al ejecutar tu servidor, se observan los resultados. Con esto le das dinamismo y reactividad a la página web a través de componentes en caso de incorporar más elementos al array de objetos.

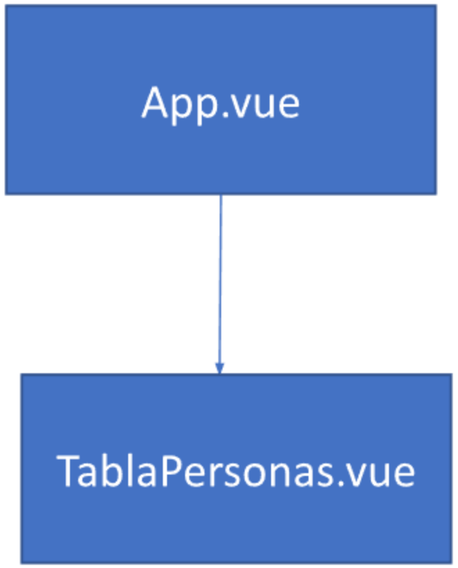
Personas

Nombre	Apellido
Blanca	Nieve
Diego	Pino
Andrea	Bouffanais

Jerarquía de componentes

Cada vez que cambias los datos del object data de la App.vue (componente padre), vuelve a renderizarse el componente TablaPersonas (componente hijo), ya que deben transmitirse los nuevos valores de la propiedad al Componente y, al mismo tiempo, vuelve a representar toda la jerarquía del componente.

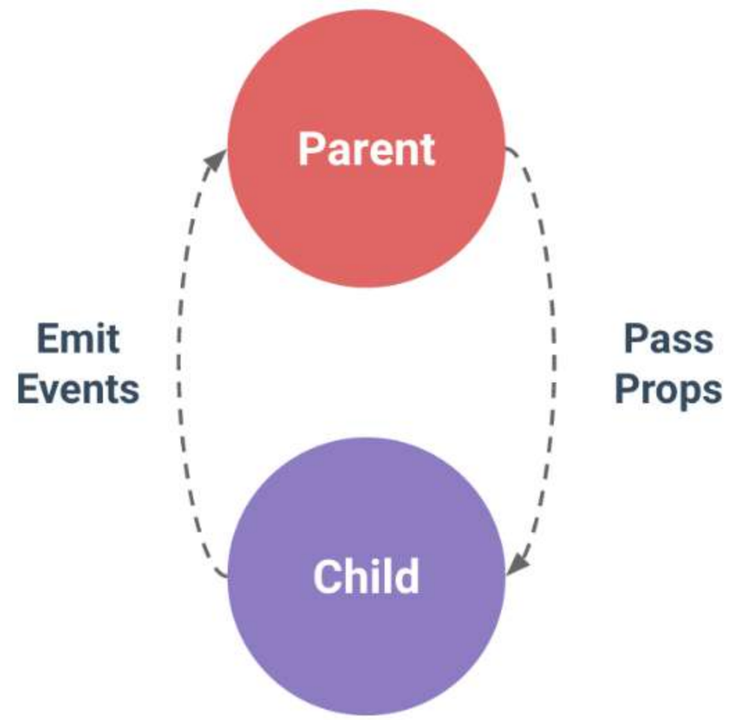
Esta es una jerarquía de componentes. En aplicaciones medianas y grandes, puede haber varias capas de dichos componentes que deben volver a renderizarse en el camino. Con esto, el rendimiento de la aplicación puede verse muy afectado y eso es algo que debe evitarse.



Paso de datos a un componente mediante Props

Para especificar el tipo de propiedad que deseas usar debes utilizar su nombre como clave de cada props y el tipo como valor.

Si el tipo de datos pasados no coincide con el tipo señalado en props, Vue envía una alerta (en modo de desarrollo) en la consola con una advertencia. Los tipos válidos que puedes utilizar son: **String, Number, Boolean, Array, Object, Date,Function, Symbol.**



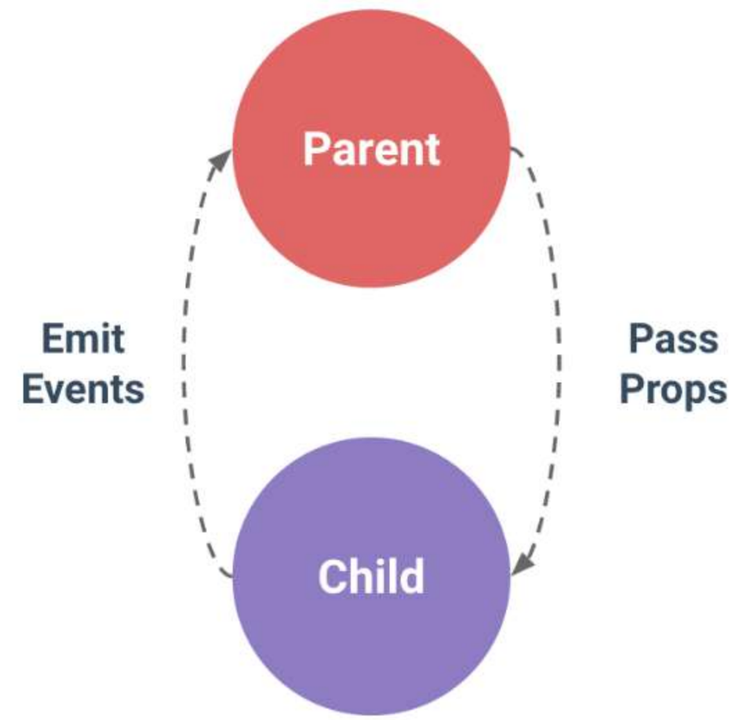
Como has podido ver, si no se pasaba un Array al componente hijo TablaPersonas.vue, entonces lanzaría un error ya que así se estableció al interior del <script> el tipo del data transmitido desde la app (componente padre).

```
<script>
   export default {
       name: 'tabla-personas',
       props: {
           personas: Array,
       },
     }
</script>
```

Enviar mensajes al componente Padre

Si queremos enviar datos de padres a hijos, podemos usar Props, pero para el caso inverso, tenemos que usar un enfoque diferente para lograrlo. Para esto se utiliza un método llamado $emit, el cual toma 2 argumentos: primero el evento personalizado y el segundo son los datos que estamos transmitiendo.

Intentemos cambiar el título de 'Personas' a 'Lista de Personas' de un modo sencillo empleando $emit y un botón incorporado a la página.



Personas

Nombre	Apellido
Blanca	Nieve
Diego	Pino
Andrea	Bouffanais

Emitir eventos al componente Padre

Podríamos implementar al interior del script del componente TablaPersonas, el método cambioTitulo(). El primer argumento del $emit, es el evento personalizado y segundo el dato al que deseamos cambiar.

```
<script>
export default {
    name: 'tabla-personas',
    props: {
        personas: Array,
    },
    methods: {
        cambioDeTitulo() {
            this.$emit('cambioTitulo', 'Lista de Personas')
        }
    },
}
</script>
```

Además, se añade un botón al final de la plantilla del mismo componente que contenga un evento click con el método como valor.

```
 <button type="button"
@click='cambioDeTitulo'>Cambiar Título</button>
```

Emitir eventos al componente Padre

Análogamente, se implementa al interior del script de la app, el método cambiarTitulo, en cual se señala la acción a ejecutarse: cambiar el primer el primer título <h1> por el contenido o argumento del mismo método.

```
 methods: {
        cambiarTitulo(contenido) {

document.getElementsByTagName("h1")[0].textContent=contenido;
        },
    }
```

Por último, al interior del componente importado dentro de la plantilla de la app, se añade la escucha al evento personalizado (cambioTitulo), tomando por valor el método implementado en la app y que toma por argumento el dato emitido desde el componente hijo.

```
<tabla-personas :personas="personas"
@cambioTitulo="cambiarTitulo($event)" />
```

Emitir eventos al componente Padre

Personas

Nombre	Apellido
Blanca	Nieve
Diego	Pino
Andrea	Bouffanais

Cambiar Título

Lista de Personas

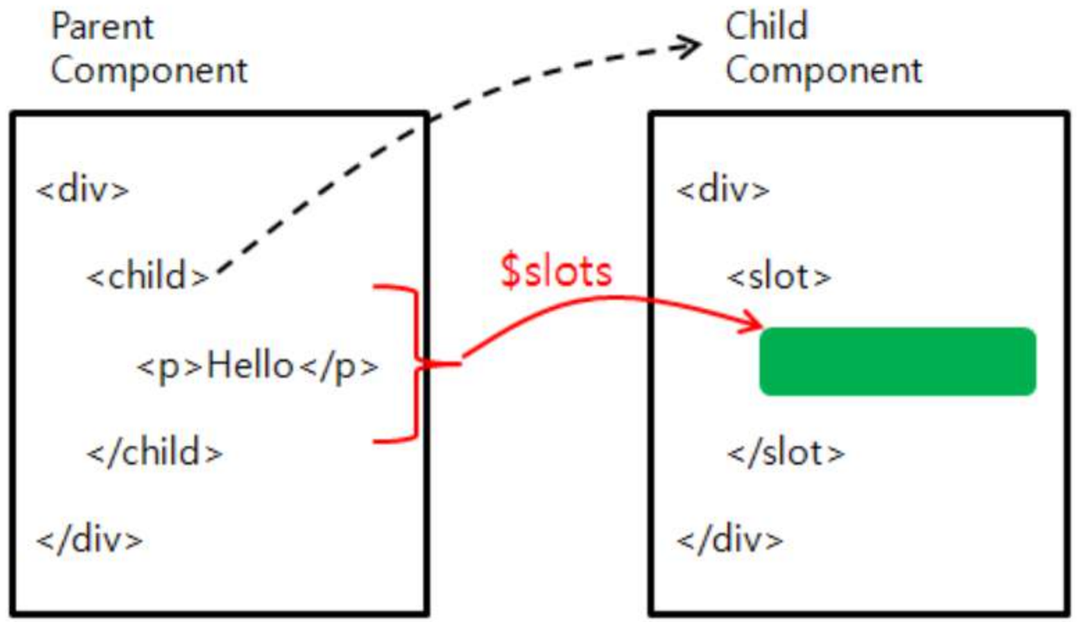
Nombre	Apellido
Blanca	Nieve
Diego	Pino
Andrea	Bouffanais

Cambiar Título

Luego de hacer click en el botón, se aprecian los cambios.

Distribución de contenidos en Slots

Los slots son un mecanismo de Vue JS que sirve para insertar contenido HTML dentro de los componentes. Es decir, con los props puedes pasar objetos y variables javascript a los componentes y con los **slots puedes insertar contenido HTML** dentro de otros componentes. Slot es una etiqueta especial de Vue que se inserta en el template de componente hijo para señalar que allí se insertará contenido “de afuera”.



Distribución de contenidos en Slots

Siguiendo con el mismo ejemplo, imagina que queremos pasar desde el componente padre al componente hijo una imagen. Para esto agregamos al interior del componente importado en la aplicación, otro template con #nombre y el source de lo que queremos incluir. Quedaría así:

```
<template>
    <div id="app" class="container">
        <div class="row">
            <div class="col-md-12">
                <h1>Personas</h1>
            </div>
        </div>
        <div class="row">
            <div class="col-md-12">
          ---<tabla-personas :personas="personas" @cambioTitulo="cambiarTitulo($event)">
                    <template #imagen>
                        <img
src="https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcTLPswIpr29vKQOgqc85teB017VTo4vtkiX
DA&usqp=CAU">
                    </template>
                </tabla-personas> ------
            </div>
        </div>
    </div>
</template>
```

Distribución de contenidos en Slots

Mientras que en el componente TablaPersonas, debes traer desde afuera el contenido para incorporarlo en el slot donde más te guste al interior del template. Recuerda llamarlo por el #nombre que le asignaste en la aplicación. Quedaría así:

```
<div>
    <slot name="imagen" />
</div>
```

Y el resultado:

Componentes Dinámicos

Los componentes dinámicos de Vue permiten a los usuarios cambiar entre dos o más componentes sin enrutamiento e incluso conservar el estado de los datos cuando se vuelve al componente inicial.

La idea central es permitir a los usuarios montar y desmontar dinámicamente componentes en la interfaz de usuario sin usar enrutadores.

Vue ofrece un elemento de plantilla especial para componentes dinámicos llamado simplemente **component**. La sintaxis se ve así:

```
<component v-bind:is=”currentComponent”/>
```

Componentes Dinámicos

Imagina que quieres incorporar un botón al final de nuestro ejemplo que te permita moverte entre dos componentes diferentes.

Test.vue

```
<template>
    <div><h1>Soy Test.vue</h1>
    </div>
</template>
<script>
export default {
  name: 'Test-1',
  props: {
    msg: String
  }
}
</script>
```

Test.vue

```
<template>
    <div><h1>Soy Test2.vue</h1>
    </div>
</template>
<script>
export default {
  name: 'Test-2',
  props: {
    msg: String
  }
}
</script>
```

Componentes Dinámicos

Luego agrega el tag component en el template de la app. Como valor de v-bind:is se tendrá el componente actual que estarás retornando desde el data() de la app y que se empleará como componente dinámico. En button implementamos un evento click que ejecute el método para cambiar de componente.

```
<component v-bind:is="componente" />
<button v-on:click="cambiarComponente">Cambiar a otra página</button>
```

En el script de la app agregas los componentes importados y en data() señalas el valor por default del componente inscrito en el tag del componente dinámico.

```
import TablaPersonas from '@/components/TablaPersonas.vue'
import Test from './components/Test.vue'
import Test2 from './components/Test2.vue'
export default {
    name: 'app',
    components: {
        TablaPersonas,
        Test,
        Test2,
    },
```

```
data() {
        return {
            componente:"Test2",
//...}
}
```

Componentes Dinámicos

Por último, en la app desarrollas el método que se ejecutará al hacer click en el botón que acabas de implementar y con el cual irás alternando el valor del “componente” en el tag de componente dinámico.

```
methods: {
        cambiarPagina() {
            if (this.componente === "Test2") {
                this.componente = "Test";
            } else {
                this.componente = "Test2";
            }
        }
    }
```

Componentes Dinámicos

Observamos el cambio entre los componentes importados al hacer click. Obviamente, puedes darle más contenido a los componentes para darle una mejor utilidad.

Personas

Nombre	Apellido
Blanca	Nieve
Diego	Pino
Andrea	Bouffanais

Cambiar Título

Soy Test2.vue

Cambiar a otra página

Personas

Nombre	Apellido
Blanca	Nieve
Diego	Pino
Andrea	Bouffanais

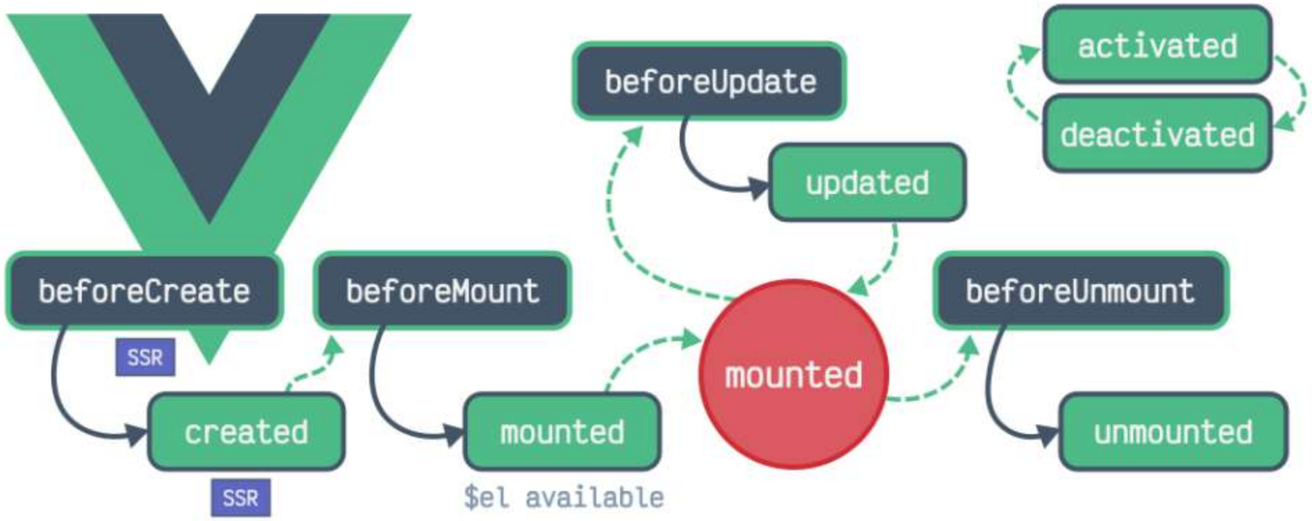
Cambiar Título

Soy Test.vue

Cambiar a otra página

Ciclo de vida de un componente y Hooks

Cada instancia de Vue tiene un ciclo de vida y cada Hook nos permiten ejecutar código en ciertas etapas del ciclo de vida de una instancia o componente de vue.



Cada fase presentada en este diagrama es un Hook. Estos son métodos especiales que nos dan ideas sobre cómo funcionan las cosas detrás de escena del marco Vue. Estos métodos le permiten saber cuándo se crea, agrega, actualiza o destruye un componente en la instancia de Vue.

Ciclo de vida de un componente y Hooks

Cada fase presentada en este diagrama es un Hook. Se presenta cada uno en la siguiente tabla.

Hook	Cuándo se llama
beforeCreate	Al inicializar y antes de procesar opciones
created	Después de crearse. Los datos ya están disponibles
beforeMount	Inmediatamente antes de la fase de montaje en el DOM
mounted	Al montarse en la página. No se garantiza el montaje de hijos.
beforeUpdate	Cuando cambian los datos (antes de modificar el DOM)
updated	Cuando cambian los datos y el DOM ha sido modificado
beforeUnmount	Justo antes del desmontaje, pero siendo aún funcional
unmounted	Justo cuando un componente (y todos sus hijos) han sido desmontados
activated	Cuando un componente dinámico con <keep-alive> es activado
desactivated	Cuando un componente dinámico con <keep-alive> es desactivado

Ciclo de vida de un componente y Hooks

Un ejemplo de uso del Hook mounted. Se puede usar para ejecutar código después de que el componente haya terminado el renderizado inicial y haya creado los nodos DOM:

```
export default {
  mounted() {
    console.log(`El componente está montado`)
  }
}
```

Usando la etiqueta Style

Para mantener las configuraciones de estilo al lado del componente, podemos agregar un elemento <style> en su interior. Sin embargo, estos estilos al ser de carácter global, se pueden alterar desde fuera del componente, pudiendo complicarse la solución al problema. Para esto, el atributo de *scope* puede ser útil: **adjunta un selector de atributo de datos HTML único a todos sus estilos, evitando que colisionen globalmente.**

Usando la etiqueta Style

Al crear el componente puedes indicar que los cambios en estilo que se realicen, tengan alcance de aplicación sólo dentro del archivo en el que halla situado.

```
<template>
    <p>Hola Mundo</p>
</template>

<script>
export default {
  name: 'hola-mundo',
 }
</script>

<style scoped>
p{
    color:green;
}
</style>
```

Class binding

Con v-bind:class puedes aplicar varias clases a los elementos. Si bien esto es posible con la manipulación del DOM, Vue.js te permite representar dinámicamente las clases con líneas de código concisas y aprovecha la estructura de modelo de datos. Siguiendo con el ejemplo de la diapositiva anterior, podríamos agregar un checkbox que permita cambiar el color al texto “Hola Mundo”.

v-model en el interior del checkbox actúa como un verificador de checked y este estado se transmite al class binding (v-bind:class). Por lo tanto, si está chequeado el checkbox, la claseRojo es verdadera y por lo tanto se enlaza al elemento p que la contiene.

```
<template>
  <div id="app">
    <p v-bind:class="{'claseRojo':indicador}">Hola Mundo</p>
    Cambiar color a rojo<input type="checkbox" v-model="indicador"/>
  </div>
</template>
```

Class binding

Posteriormente, debes implementar el verificador del checked que actúa como v-model, ya que hasta ahora sólo está enlazando una variable 'indicador' que no tiene significado booleano. Por lo tanto, puedes retornar del data() un valor para el indicador.

```
<script>
export default {
  data(){
    return{
      indicador:false,
    }
  },
  name: 'hola-mundo',
  props: {
    msg: String
  }
}
</script>
```

Con esto, el indicador estará en falso como valor predeterminado, luego el v-model dentro del checkbox será NO chequeado y transmitirá este estado a la claseRojo como falso, por lo cual, no se aplicará. Pero aún falta implementar la claseRojo…

Class binding

Como el color por defecto para p será verde, siempre y cuando el estado inicial de 'indicador' sea falso (es decir, checkbox no chequeado), entonces al chequear o hacer click en el checkbox, podrás ver que se cambiará v-model a true y con esto transmitirá el estado para que la claseRoja se aplique.

```
<style scoped>
p{
    color:green;
}
.claseRojo{
  color:red;
}
</style>
```

Así se vería:

Hola Mundo

Cambiar color a rojo

Style binding

De modo análogo, podemos enlazar estilos dinámicos a los elementos del componente. Podríamos implementar dos botones que ayuden a cambiar el tamaño de la fuente. Para esto al interior del template insertamos los respectivos botones. Al elemento p se añadió el v-bind:style con la propiedad fontSize y un valor de tamaño que debe retornar el data por defecto. Notar que a tamaño se le suma la unidad 'px' para darle formato apropiado según css estándar.

```
<template>
  <div id="app">
    <p v-bind:class="{ 'claseRojo': indicador }" v-bind:style="{ fontSize: tamaño + 'px' }">Hola Mundo</p>
    Cambiar color a rojo<input type="checkbox" v-model="indicador" />
    <div>
    <button v-on:click=“tamaño++">
      Aumentar tamaño de Fuente
    </button>
    <button v-on:click=“tamaño--">
      Disminuir tamaño de Fuente
    </button>
   </div>
  </div>
</template>
```

Style binding

Por otro lado, cada vez que se realice un click en los botones (v-on:click), se realizará un aumento unitario (en pixeles) de la variable “tamaño”.

```
<template>
  <div id="app">
    <p v-bind:class="{ 'claseRojo': indicador }" v-bind:style="{ fontSize: tamaño + 'px' }">Hola Mundo</p>
    Cambiar color a rojo<input type="checkbox" v-model="indicador" />
    <div>
    <button v-on:click=“tamaño++">
      Aumentar tamaño de Fuente
    </button>
    <button v-on:click=“tamaño--">
      Disminuir tamaño de Fuente
    </button>
   </div>
  </div>
</template>
```

Style binding

Ahora solo falta crear el valor de ‘tamaño’ por defecto que retornará data().

```
<script>
export default {
  data() {
    return {
      indicador: false,
      tamaño: 30
    }
  },
  name: 'hola-mundo',
  props: {
    msg: String
  }
}
</script>
```

Así se vería:

Bootcamp
Academy

sustantiva