Vue JS

Germán Caballero

Version 1.0.0 2019-06-09

Contenidos

Capítulo 1. Vue

Vue es un framework de código abierto, hecho con JavaScript y que nos va a permitir construir SPAs. Fue creado por Evan You, un extrabajador de Google que trabajaba con AngularJs, y ahora es mantenido por la comunidad, no tiene una empresa grande que lo respalde al contrario que React (Facebook) o Angular (Google) y por lo tanto su desarrollo va a seguir el camino que le marquen los desarrolladores que lo usan.

Vue se usa sobre todo en Ásia, y entre las empresas que lo están usando en la actualidad podemos encontrarnos a Gitlab y Alibaba.

1.1. Requisitos básicos

Para empezar a trabajar con Vue necesitaremos:

- Instalar Node y Npm: Node
- Un editor de código: VSCode, Atom, Sublime Text, WebStorm...
- Un navegador: Chrome, Firefox
- Recomendado instalar la extensión **Vetur** en VSCode.

Capítulo 2. Uso progresivo de Vue

Es un framework progresivo ya que se ha construido para poder añadirlo a proyectos que no se han hecho con Vue de una forma sencilla progresivamente, sin la necesidad que se haya creado el proyecto usando Vue desde un inicio.

2.1. Instancia de Vue

Al importar el script de Vue en las aplicaciones, necesitamos crear una **instancia de Vue** ,para poder trabajar con el en la vista, la cual recibe como parámetro un objeto donde vamos a configurar la lógica de dicha instancia de Vue.

En ese objeto podemos añadir varias propiedades entre las que se encuentran algunas como:

- el: le damos como valor el selector CSS del elemento donde vamos a trabajar con Vue.
- data: un objeto con los datos que usaremos en la aplicación.
- methods: un objeto con funciones que podremos usar en la aplicación.
- computed: un objeto con las *computed properties*.
- watch: un objeto con los objetos *watchers*.
- filters: un objeto con funciones de filtro o *pipes*.
- props: un objeto en el que se definen las propiedades que recibe un componente.
- components: un objeto con los componentes que podemos usar donde se indica el nombre que habrá que usar como etiqueta y que componente tiene que renderizar cuando la usemos.
- template: se le asigna el código HTML que tiene que pintar en el elemento al que apunta el selector CSS que hemos puesto en *el*.
- métodos del ciclo de vida

2.2. Lab: añadir Vue a una página HTML

En este laboratorio vamos a ver como añadir Vue a una página normal de HTML y trabajar sobre ella con este framework.

2.2.1. Importamos Vue

Como hemos comentado antes, Vue es muy sencillo de usar, y nos basta con añadir el script de CDN a nuestro proyecto para poder empezar a usarlo. Este enlace lo podemos encontrar en https://vuejs.org/v2/guide/.

Empezamos creando la página HTML y añadimos el script de Vue.

/vuejs-uso-progresivo-lab/index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <script src="https://cdn.jsdelivr.net/npm/vue"></script>
  <title>Document</title>
</head>
<body>
</body>
</html>
```

2.2.2. Instancia de Vue

Una vez que tenemos Vue importado en nuestra aplicación, necesitamos crear una **instancia de Vue** para poder trabajar con el y para crearla usamos `new Vue()`.

Vamos a crear una archivo javascript que importaremos en el HTML de la aplicación, y añadiremos una etiqueta que será sobre la cual vamos a trabajar desde la instancia de Vue.

/vuejs-uso-progresivo-lab/index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <script src="https://cdn.jsdelivr.net/npm/vue"></script>
  <title>Document</title>
</head>
<body>
  <div>
    <h1>Uso de Vue progresivamente</h1>
    <div id="app"></div>
  </div>
  <script src="app.js"></script>
</body>
</html>
```

El siguiente paso es crear en el script la instancia de Vue, e indicar sobre que etiqueta HTML se va a usar, para lo que usaremos la opción `el` a la cual se le asigna el selector de id de dicha etiqueta.

/vuejs-uso-progresivo-lab/app.js

```
new Vue({
  el: '#app'
})
```

Ahora vamos a pasarle a la instancia otra opción para indicar lo que se tiene que mostrar en la etiqueta que hemos señalado con el atributo `el`. El nuevo atributo que se va a usar es `template` al que le vamos a dar como valor el código HTML que hay que mostrar.

/vuejs-uso-progresivo-lab/app.js

```
new Vue({
  el: '#app',
  template: `<h2>Hola Koz</h2>`
})
```

Si abrimos la página en el navegador, podremos ver aquello que hemos pasado como valor al atributo `template`.

2.3. Lab: añadir Vue a una página HTML

En este laboratorio vamos a ver como crear un componente de Vue para reutilizarlo entre distintas instancias de Vue sobre una misma página de HTML.

2.3.1. Primer componente

Empezamos creando una página HTML en la que vamos a añadir dos etiquetas a controlar con dos instancias de Vue distintas, en las que queremos mostrar el mismo mensaje.

/vuejs-uso-progresivo-componente-lab/index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <script src="https://cdn.jsdelivr.net/npm/vue"></script>
  <title>Document</title>
</head>
<body>
  <div>
    <h1>Primer Componente</h1>
    <div id="app1"></div>
    <hr>
    <div id="app2"></div>
  </div>
  <script src="app.js"></script>
</body>
</html>
```

Lo siguiente es crear el script `app.js` en el que vamos a crear dos instancias de Vue, una para controlar `app1` y la otra para el `app2`.

/vuejs-uso-progresivo-componente-lab/app.js

```
new Vue({
  el: '#app1'
})

new Vue({
  el: '#app2'
})
```

Ahora vamos a crear un componente que nos va a permitir reutilizar el código y mostrarlo tantas veces como queramos ponerlo.

Para crear el componente, antes de crear las instancias de Vue, hay que usar el método `Vue.component()` al que se le van a pasar como parámetros, el nombre del componente y el objeto que define como es el componente con las opciones indicadas al inicio de este capítulo.

/vuejs-uso-progresivo-componente-lab/app.js

```
Vue.component('app-saludo', {
  template: `<h2>Hola mundo!</h2>`
})

new Vue({
  el: '#app1'
})

new Vue({
  el: '#app2'
})
```

Por último, solo tenemos que poner la etiqueta con el nombre que le hemos dado en el componente, entre las etiquetas del HTML donde están actuando las instancias de Vue.

/vuejs-uso-progresivo-componente-lab/index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <script src="https://cdn.jsdelivr.net/npm/vue"></script>
  <title>Document</title>
</head>
<body>
  <div>
    <h1>Primer Componente</h1>
    <div id="app1">
      <app-saludo></app-saludo>
    </div>
    <hr>
    <div id="app2">
      <app-saludo></app-saludo>
    </div>
  </div>
  <script src="app.js"></script>
</body>
</html>
```

Ahora al abrir el navegador, deberíamos de ver el texto del componente dos veces.

Capítulo 3. Vue-Cli

Vue-Cli es un cliente de Vue que nos va a permitir crear proyectos Vue completamente configurados con solo unos pocos comandos. De esta forma empezar a trabajar con Vue es muy rápido y sencillo.

3.1. Instalación

Para instalarlo hay que ejecutar el siguiente comando:

```
$ npm install -g @vue/cli
```

3.2. Estructura del proyecto

Al crear el proyecto nos encontramos con los siguientes archivos y carpetas:

- public: carpeta que tiene que servir el servidor.
  - favicon.ico: icono de la aplicación.
  - index.html: página en la que va a vivir la aplicación.
- src: carpeta donde vamos a trabajar nosotros para construir toda nuestra aplicación.
  - assets: carpeta donde vamos a poner los assets de la aplicación como los estilos globales, imagenes…
  - components: carpeta en la que vamos a crear nuestros componentes.
  - App.vue: componente raíz de la aplicación.
  - main.js: archivo principal que se va a ejecutar al abrir la aplicación y que indica que componente es el principal de la aplicación.
- .browserslistrc: navegadores en los que debe de funcionar la aplicación. Suele ser usado por herramientas como *autoprefixer* para saber que transformaciones tiene que aplicar en el CSS.
- .editorconfig: configuración a tener en cuenta por el editor para trabajar con los archivos.
- .eslintrc.js: se añade configuración del linter.
- .gitignore: archivo donde se añaden aquellas carpetas y archivos que queremos que se ignoren al trabajar con **git**, por ejemplo la carpeta **node_modules**.
- babel.config.js: archivo de configuración donde se le indica a **babel** que tiene que usar para poder trabajar con los archivos que tienen una sintaxis que no entienden los navegadores y transformar el código a una versión de JavaScript que si entienden.
- package.json: archivo donde se encuentra información del proyecto (nombre, descripción, autor, versión…), dependencias que son necesarias, scripts…
- package-lock.json: archivo que contienen todas las versiones de las dependencias y las dependencias de estas para que al instalarlas sea lo más concreto posible.

3.3. Lab: Creación de un proyecto con vue-cli

En este laboratorio vamos a ver como crear un proyecto de Vue usando **vue-cli**.

Para crear un proyecto desde 0, vamos a lanzar el siguiente comando.

```
$ vue create vuejs-vue-cli-lab
```

Una vez lanzado el comando, nos da a elegir unas opciones, donde vamos a seleccionar la segunda, **Manually select features**.

```
? Please pick a preset: (Use arrow keys)
  default (babel, eslint)
❯ Manually select features
```

Ahora nos permite seleccionar que funcionalidades queremos añadir a nuestro proyecto, entre las que se encuentra el **router**, **vuex**, **pwa**...

Vamos a seleccionar aquellas funcionalidades que queremos añadir a nuestro proyecto, de momento solo vamos a seleccionar **CSS Pre-processors** a parte de las que ya se encuentran seleccionadas.

```
? Please pick a preset: Manually select features
? Check the features needed for your project: (Press <space> to select, <a> to toggle al
l, <i> to invert selection)
 ◉ Babel
 ◉ TypeScript
 ◉ Progressive Web App (PWA) Support
 ◉ Router
 ◉ Vuex
❯◉ CSS Pre-processors
 ◉ Linter / Formatter
 ◉ Unit Testing
 ◉ E2E Testing
```

Ahora nos va a ir pidiendo que configuración queremos usar de las funcionalidades que hemos elegido.

Seleccionamos **Sass/SCSS (with node-sass)**.

```
? Please pick a preset: Manually select features
? Check the features needed for your project: Babel, CSS Pre-processors, Linter
? Pick a CSS pre-processor (PostCSS, Autoprefixer and CSS Modules are supported by defau
lt):
  Sass/SCSS (with dart-sass)
 Sass/SCSS (with node-sass)
  Less
  Stylus
```

Seleccionamos **ESLint** + **Prettier**.

```
? Please pick a preset: Manually select features
? Check the features needed for your project: Babel, CSS Pre-processors, Linter
? Pick a CSS pre-processor (PostCSS, Autoprefixer and CSS Modules are supported by defau
lt): Sass/SCSS (with node-sass)
? Pick a linter / formatter config: (Use arrow keys)
  ESLint with error prevention only
  ESLint + Airbnb config
  ESLint + Standard config
 ESLint + Prettier
```

Seleccionamos **Lint on save**.

```
? Please pick a preset: Manually select features
? Check the features needed for your project: Babel, CSS Pre-processors, Linter
? Pick a CSS pre-processor (PostCSS, Autoprefixer and CSS Modules are supported by defau
lt): Sass/SCSS (with node-sass)
? Pick a linter / formatter config: Prettier
? Pick additional lint features: (Press <space> to select, <a> to toggle all, <i> to inv
ert selection)
 Lint on save
  Lint and fix on commit
```

Seleccionamos **In dedicated config files**.

```
? Please pick a preset: Manually select features
? Check the features needed for your project: Babel, CSS Pre-processors, Linter
? Pick a CSS pre-processor (PostCSS, Autoprefixer and CSS Modules are supported by defau
lt): Sass/SCSS (with node-sass)
? Pick a linter / formatter config: Prettier
? Pick additional lint features: (Press <space> to select, <a> to toggle all, <i> to inv
ert selection)Lint on save
? Where do you prefer placing config for Babel, PostCSS, ESLint, etc.? (Use arrow keys)
 In dedicated config files
  In package.json
```

Y por último, le indicamos que no queremos guardar el preset. En caso de hacerlo, la siguiente vez que vayamos a crear un proyecto, se mostrará como otra opción más a parte de la de **default** y **Manually select features**.

```
? Please pick a preset: Manually select features
? Check the features needed for your project: Babel, CSS Pre-processors, Linter
? Pick a CSS pre-processor (PostCSS, Autoprefixer and CSS Modules are supported by defau
lt): Sass/SCSS (with node-sass)
? Pick a linter / formatter config: Prettier
? Pick additional lint features: (Press <space> to select, <a> to toggle all, <i> to inv
ert selection)Lint on save
? Where do you prefer placing config for Babel, PostCSS, ESLint, etc.? In dedicated conf
ig files
? Save this as a preset for future projects? (y/N) N
```

Una vez seleccionadas todas estas opciones, ahora empieza a instalar las dependencias y crear los archivos del proyecto, y cuando termina de hacer esto, podemos lanzar el siguiente comando dentro de la carpeta del proyecto para levantar el servidor de desarrollo.

```
$ npm run serve
```

Ahora al entrar en localhost:8080 podremos ver el proyecto inicial.

Welcome to Your Vue.js App

For a guide and recipes on how to configure / customize this project,
check out the vue-cli documentation.

Installed CLI Plugins

babel eslint

Essential Links

Core Docs Forum Community Chat Twitter News

Ecosystem

vue-router vuex vue-devtools vue-loader awesome-vue

Capítulo 4. Componentes

Al final construir un proyecto con las instancias de Vue que hemos visto no va a ser fácil ya que vamos a tener mucho código dentro de cada instancia, y código de distintos lenguajes mezclados. Por lo tanto lo más fácil va a ser sacar los componentes a archivos `.vue` independientes.

Estos archivos constan tres partes:

- La plantilla o código HTML, que pondremos entre etiquetas `template`.
- El componente que pondremos entre etiquetas `script`. Si queremos usar componentes dentro de otro, tendremos que importar estos componentes y añadirlos a la propiedad `components`.
- Los estilos del componente entre las etiquetas `style`.

Tenemos dos formas de crear los componentes:

- **Localmente**: solo se va a usar en pocos componentes, y tenemos que importarlo y añadirlo a la propiedad `components` en cada uno de ellos.
- **Globalmente**: se va a usar en muchos componentes, y tenemos que crearlo con `Vue.component(⋯)` antes de crear la instancia de la aplicación.

4.1. Lab: Componentes locales

En este laboratorio, vamos a ver como crear un componente y usarlo como componente local dentro de otro.

Empezamos por crear el proyecto lanzando el siguiente comando y seleccionamos la configuración por defecto.

```
$ vue create vuejs-componentes-locales-lab
```

Una vez creado el proyecto, eliminamos el componente `HelloWorld` y dejamos vacío el componente `App`.

Vamos a crear un componente `CmpLocal` en la carpeta `components`.

/vuejs-componentes-locales-lab/src/components/CmpLocal.vue

```
<template>
  <h1>Esto es un componente local</h1>
</template>

<script>
export default {

}
</script>

<style>

</style>
```

Una vez que tenemos el componente, vamos a añadirlo al componente `App` como componente local, y para ello, lo que tenemos que hacer es ponerlo dentro de la propiedad `components`.

/vuejs-componentes-locales-lab/src/App.vue

```
<template>
  <div>
    <CmpLocal />
  </div>
</template>

<script>
import CmpLocal from './components/CmpLocal.vue'

export default {
  components: {
    CmpLocal
  }
}
</script>

<style>

</style>
```

4.1.1. Álias para el componente local

Si queremos usar el componente con otro nombre distinto, podríamos poner dentro de la propiedad `components` como clave el nombre que queremos usar a la hora de mostrar este componente y como valor el nombre de la clase del componente.

/vuejs-componentes-locales-lab/src/App.vue

```
<template>
  <div>
    <cmp-local />
  </div>
</template>

<script>
import CmpLocal from './components/CmpLocal.vue'

export default {
  components: {
    'cmp-local': CmpLocal
  }
}
</script>

<style>

</style>
```

4.2. Lab: Componentes globales

En este laboratorio, vamos a ver como crear un componente global y usarlo en otro componente.

Empezamos por crear el proyecto lanzando el siguiente comando y seleccionamos la configuración por defecto.

```
$ vue create vuejs-componentes-globales-lab
```

Una vez creado el proyecto, eliminamos el componente `HelloWorld` y dejamos vacío el componente `App`.

Vamos a crear un componente `CmpGlobal` dentro de la carpeta `components`.

/vuejs-componentes-globales-lab/src/components/CmpGlobal.vue

```
<template>
  <h1>Esto es un componente global</h1>
</template>

<script>
export default {

}
</script>

<style>

</style>
```

Una vez que lo tenemos, vamos a ir al archivo `main.js` y justo antes de crear la instacia de Vue de la aplicación, vamos a llamar a la función `Vue.component()` a la que le pasaremos como parámetros, el nombre de la etiqueta del componente y la clase que representa al componente.

/vuejs-componentes-globales-lab/src/main.js

```
import Vue from 'vue'
import App from './App.vue'
import CmpGlobal from './components/CmpGlobal.vue';

Vue.config.productionTip = false

Vue.component('CmpGlobal', CmpGlobal);

new Vue({
  render: h => h(App),
}).$mount('#app')
```

Una vez que lo hemos declarado como un componente global, ya podemos usarlo en cualquier lugar de la aplicación, sin necesidad de importarlo en los scripts de los demás componentes.

/vuejs-componentes-globales-lab/src/App.vue

```
<template>
  <div>
    <CmpGlobal />
  </div>
</template>

<script>
export default {

}
</script>

<style>

</style>
```

Una vez hecho todo esto, si habrímos la aplicación en el navegador, deberíamos de poder ver el componente.

Capítulo 5. Ciclo de vida

Los componentes en Vue tienen unos métodos que se van a ir ejecutando en ciertos momentos de la vida de los componentes, y nos van a permitir ejecutar código cuando corresponda.

Estos métodos del ciclo de vida son los siguientes.

5.1. beforeCreate

El método `beforeCreate()` se ejecuta al crearse la instancia del componente, pero en este método todavía no se han inicializado los datos, métodos... por lo que no podemos acceder a ellos. Aquí podríamos inicializar el estado externo de la aplicación (por ejemplo el *localStorage*) o librerías externas.

5.2. created

El método `created()` se ejecuta una vez que ya se han inicializado los datos, métodos... peto todavía no tenemos acceso al DOM porque no se ha renderizado el componente. Aquí será donde se realicen las operaciones asíncronas como las *peticiones HTTP*.

5.3. beforeMount

El método `beforeMount()` se ejecuta justo antes de renderizar el componente por primera vez. Se pueden inicializar variables aunque es mejor hacerlo en el método anterior. Es de los que menos se usan y puede servir para trazar cuando se va a renderizar un componente.

5.4. mounted

El método `mounted()` se ejecuta nada más poner el componente en el DOM, o renderizarlo. Aquí podemos poner el código que necesite modificar el DOM nada más incluir los componentes, por ejemplo si necesitamos crear algún elemento HTML usando una librería externa como JQuery.

5.5. beforeUpdate

El método `beforeUpdate()` se ejecuta justo antes de realizar el renderizado del componente cuando cambia el estado de este. Podemos acceder al nuevo estado antes de que se renderice el componente. Es de los que menos se usan y nos puede servir para trazar los cambios de estado.

5.6. updated

El método `updated()` se ejecuta justo después de volver a renderizar el componente porque el estado de este ha sufrido algún cambio. Al igual que en el método `mounted` es un buen sitio donde actualizar los elementos (creados con librerías externas a Vue) con los datos que han cambiado.

5.7. beforeDestroy

El método `beforeDestroy()` se ejecuta justo antes de destruir el componente y se suele usar para eliminar los listeners, desuscribirnos de observables... En este método todavía tenemos acceso a los métodos y datos que hay en el componente.

5.8. destroyed

El método `destroyed()` se ejecuta cuando el componente (incluidos los componentes hijos) se ha destruido. Aquí será donde podemos limpiar el estado global de la aplicación, por ejemplo eliminar datos que se hayan guardado en el navegador (como en el *localStorage*).

Capítulo 6. Data

El atributo **data** de los componentes es el que va a almacenar el estado local a ese componente, es decir, contendrá los datos que vayamos a usar. Este atributo es una función que devuelve un objeto con ese estado.

```
<template>
  <div></div>
</template>

<script>
export default {
  data() {
    return {
      tituloPagina: 'Un título',
      menuOculto: true
    }
  }
}
</script>
```

Capítulo 7. Methods

El atributo **methods** es un objeto que contiene las funciones que vamos a llamar desde el propio componente para realizar ciertas acciones cuando el usuario interacciona con el.

```
<template>
  <div></div>
</template>

<script>
export default {
  methods: {
    mostrarMsg(msg) {
      // Aquí va el código para mostrar un mensaje por consola
    },
    guardarDatos() {
      // Aquí va el código necesario para guardar los datos
    }
  }
}
</script>
```

Capítulo 8. Data Binding

Vue nos permite desarrollar plantillas HTML en las que podemos usar código HTML junto a una sintaxis del estilo de librerías como Handlebars en la que podremos insertar datos dinámicos.

Vue cogera estas plantillas y generará el código HTML final que se mostrará en el navegador, sustituyendo esa sintaxis especial por el valor que obtenga en cada caso.

8.1. String Interpolation

El **String Interpolation** se usa para renderizar el valor de una variable en las plantillas de los componentes.

No sirve para darle valor a las etiquetas o a los atributos de estas etiquetas.

Los datos que se usan son solo de lectura, es decir, no podemos modificarlos directamente dentro del String Interpolation.

Se usa con `{{ nombreVariable }}`.

8.1.1. Lab: String Interpolation

En este laboratorio vamos a ver los distintos casos en los que podríamos usar el String Interpolation dentro de nuestros componentes.

Empezamos por crear el proyecto lanzando el siguiente comando y seleccionamos la configuración por defecto.

```
$ vue create vuejs-data-binding-string-interpolation-lab
```

Una vez creado el proyecto, eliminamos el componente `HelloWorld` y dejamos vacío el componente `App`.

Ahora vamos a crearnos un componente `CmpStringInterpolation` dentro de la carpeta `components`.

Dentro del componente, vamos a añadir *nombre* dentro de la propiedad `data`.

/vuejs-data-binding-string-interpolation-lab/src/components/CmpStringInterpolation.vue

```
<template>
  <div>

  </div>
</template>

<script>
export default {
  data() {
    return {
      nombre: 'Bruce'
    }
  }
}
</script>

<style>

</style>
```

Ahora vamos a usar en la plantilla del componente el String Interpolation para mostrar el nombre, y para ello tendremos que ponerlo entre `{{ }}`.

/vuejs-data-binding-string-interpolation-lab/src/components/CmpStringInterpolation.vue

```
<template>
  <div>
    <p>Me llamo {{nombre}}</p>
  </div>
</template>

<script>
export default {
  data() {
    return {
      nombre: 'Bruce'
    }
  }
}
</script>

<style>

</style>
```

Una vez que tenemos nuestro componente, vamos a importarlo en el componente raíz para que se muestre en la aplicación cuando se abra en el navegador.

/vuejs-data-binding-string-interpolation-lab/src/App.vue

```
<template>
  <div>
    <CmpStringInterpolation />
  </div>
</template>

<script>
import CmpStringInterpolation from './components/CmpStringInterpolation.vue';

export default {
  components: {
    CmpStringInterpolation
  }
}
</script>

<style>

</style>
```

Dentro de las llaves también podemos añadir expresiones que se van a evaluar y mostrarán el resultado de ellas.

En este caso, vamos a añadir otro nombre, y lo vamos a mostrar dentro del String Interpolation en una expresión que crea un array con varias posiciones vacias, las une por otra expresión que se le pasa al método `join` y por último le concatenamos el nuevo nombre.

/vuejs-data-binding-string-interpolation-lab/src/components/CmpStringInterpolation.vue

```
<template>
  <div>
    <p>Me llamo {{nombre}}</p>
    <p>{{new Array(17).join(1 - 'what') + ` ${nombreSuperheroe}!`}}</p>
  </div>
</template>

<script>
export default {
  data() {
    return {
      nombre: 'Bruce',
      nombreSuperheroe: 'Batman'
    }
  }
}
</script>

<style>

</style>
```

8.2. Property Binding

El String Interpolation no se puede usar para darle valores a los atributos de las etiquetas HTML, pero para eso tenemos la directiva `v-bind`.

Para usarla solo hay que añadir como argumento de la directiva el atributo al que queremos asignarle un valor, quedando la sintaxis como `v-bind:atributo="valor"`.

Esta directiva tiene una sintaxis abreviada para poder usarla y evitar usar `v-bind` todo el tiempo. En su lugar vamos a dejar los dos puntos delante del atributo y se le asigna el valor, `:atributo="valor"`.

8.2.1. Lab: Property Binding

En este laboratorio vamos a ver como darles valores dinámicos a los atributos de las etiquetas usando la directiva `v-bind`.

Empezamos por crear el proyecto lanzando el siguiente comando y seleccionamos la configuración por defecto.

```
$ vue create vuejs-data-binding-propiedades-lab
```

Una vez creado el proyecto, eliminamos el componente `HelloWorld` y dejamos vacío el componente `App`.

Vamos a crear un componente `CmpPropiedades` en la carpeta `components` en el que vamos a añadir una propiedad texto que queremos añadir como valor del atributo `placeholder` de un campo de texto.

/vuejs-data-binding-propiedades-lab/src/components/CmpPropiedades.vue

```
<template>
  <div>
    <input type="text" >
  </div>
</template>

<script>
export default {
  data() {
    return {
      nombre: 'Charly Falco'
    }
  }
}
</script>

<style>

</style>
```

Para asignarle el valor de `nombre` como valor del atributo `placeholder` del input, tenemos que usar la directiva `v-bind`.

/vuejs-data-binding-propiedades-lab/src/components/CmpPropiedades.vue

```
<template>
  <div>
    <input type="text" v-bind:placeholder="nombre">
  </div>
</template>

<script>
export default {
  data() {
    return {
      nombre: 'Charly Falco'
    }
  }
}
</script>

<style>

</style>
```

Antes de continuar, vamos a ir al componente `App` para poner este otro y asi mostrarlo al abrir el navegador.

/vuejs-data-binding-propiedades-lab/src/App.vue

```
<template>
  <CmpPropiedades />
</template>

<script>
import CmpPropiedades from './components/CmpPropiedades';
export default {
  components: {
    CmpPropiedades
  }
}
</script>

<style>

</style>
```

Por último, para dejar el código más claro, podemos usar la abreviación de dicha directiva, que sería quitar el `v-bind` dejando el resto tal cual está.

/vuejs-data-binding-propiedades-lab/src/components/CmpPropiedades.vue

```
<template>
  <div>
    <input type="text" :placeholder="nombre">
  </div>
</template>

<script>
export default {
  data() {
    return {
      nombre: 'Charly Falco'
    }
  }
}
</script>

<style>

</style>
```

Si vamos al navegador, deberíamos de seguir viendo el input con el nombre como placeholder.

8.3. Event Binding

Con la directiva `v-on` vamos a detectar eventos sobre las etiquetas HTML y ejecutar una función cuando ocurra dicho evento.

A la directiva se le añade `:` seguidos del nombre del evento que se va a detectar (es una directiva con parámetro) y se le asigna como valor la función que se va a ejecutar. Este método que se va a ejecutar hay que añadirlo en el objeto de **methods** que hay en el componente.

Por defecto no hay que ponerle los paréntesis a la función, salvo que le tengamos que pasar algún valor como parámetro a dicha función.

Siempre que no se mandan parámetros, va a llegar el objeto **Event** como parámetro de la función. Pero si nosotros le mandamos parámetros, también tendremos que mandarle este objeto como parámetro añadiendo `$event` en la llamada a la función.

Al igual que con las propiedades podemos usar una sintaxis abreviada con esta directiva, y vamos a sustituir `v-on:` por `@`.

8.3.1. Lab: Event Binding

En este laboratorio vamos a ver como ejecutar una función cuando se detecta un evento sobre una etiqueta.

Empezamos por crear el proyecto lanzando el siguiente comando y seleccionamos la configuración por defecto.

```
$ vue create vuejs-data-binding-eventos-lab
```

Una vez creado el proyecto, eliminamos el componente `HelloWorld` y dejamos vacío el componente `App`.

Vamos a crear un componente `CmpEventos` dentro de la carpeta `components` donde vamos a añadir un botón junto a un párrafo que muestre si la calefacción está encendida o no, por lo tanto también necesitamos añadir una propiedad en el estado del componente.

/vuejs-data-binding-eventos-lab/src/components/CmpEventos.vue

```
<template>
  <div>
    <button type="button">Toggle</button>
    <p>Calefacción: {{calefaccionEncendida ? 'ON' : 'OFF'}}</p>
  </div>
</template>

<script>
export default {
  data() {
    return {
      calefaccionEncendida: false
    }
  }
}
</script>

<style>

</style>
```

Ahora vamos a añadir este componente en el componente de la aplicación para comprobar que se muestra correctamente.

/vuejs-data-binding-eventos-lab/src/App.vue

```
<template>
  <div>
    <CmpEventos />
  </div>
</template>

<script>
import CmpEventos from './components/CmpEventos';
export default {
  components: {
    CmpEventos
  }
}
</script>

<style>

</style>
```

Ahora vamos a añadir un método en el componente que se va a encargar de cambiar el valor del estado de la calefacción.

/vuejs-data-binding-eventos-lab/src/components/CmpEventos.vue

```
<template>
  <div>
    <button type="button">Toggle</button>
    <p>Calefacción: {{calefaccionEncendida ? 'ON' : 'OFF'}}</p>
  </div>
</template>

<script>
export default {
  data() {
    return {
      calefaccionEncendida: false
    }
  },
  methods: {
    cambiarEstado() {
      this.calefaccionEncendida = !this.calefaccionEncendida;
    }
  }
}
</script>

<style>

</style>
```

Y ahora al pulsar el botón queremos llamar al método que acabamos de crear para cambie el estado de la calefacción. Para detectar eventos vamos a usar la directiva `v-on` a la que le vamos a añadir como parámetro el nombre del evento que queremos detectar, en este caso el `click`, y a esto le asignaremos el nombre de la función a ejecutar.

/vuejs-data-binding-eventos-lab/src/components/CmpEventos.vue

```
<template>
  <div>
    <button type="button" v-on:click="cambiarEstado">Toggle</button>
    <p>Calefacción: {{calefaccionEncendida ? 'ON' : 'OFF'}}</p>
  </div>
</template>

<script>
export default {
  data() {
    return {
      calefaccionEncendida: false
    }
  },
  methods: {
    cambiarEstado() {
      this.calefaccionEncendida = !this.calefaccionEncendida;
    }
  }
}
</script>

<style>

</style>
```

Ahora al pulsar el botón debería de ir cambiando el estado de la calefacción.

En lugar de poner el nombre de la directiva, Vue nos permite añadir los evento usando `@` seguido del nombre del evento, quedando el código más claro.

/vuejs-data-binding-eventos-lab/src/components/CmpEventos.vue

```
<template>
  <div>
    <button type="button" @click="cambiarEstado">Toggle</button>
    <p>Calefacción: {{calefaccionEncendida ? 'ON' : 'OFF'}}</p>
  </div>
</template>

<script>
export default {
  data() {
    return {
      calefaccionEncendida: false
    }
  },
  methods: {
    cambiarEstado() {
      this.calefaccionEncendida = !this.calefaccionEncendida;
    }
  }
}
</script>

<style>

</style>
```

Muchas veces necesitaremos acceder al elemento sobre el que ha ocurrido el evento, por ejemplo para obtener el valor de alguna de sus propiedades, o quizás al propio evento. Siempre que no pongamos los paréntesis en la directiva `v-on` recibiremos el evento como primer parámetro en el método.

/vuejs-data-binding-eventos-lab/src/components/CmpEventos.vue

```
<template>
  <div>
    <button type="button" @click="cambiarEstado">Toggle</button>
    <p>Calefacción: {{calefaccionEncendida ? 'ON' : 'OFF'}}</p>
  </div>
</template>

<script>
export default {
  data() {
    return {
      calefaccionEncendida: false
    }
  },
  methods: {
    cambiarEstado(event) {
      console.log(event);
      this.calefaccionEncendida = !this.calefaccionEncendida;
    }
  }
}
</script>

<style>

</style>
```

En el caso de que tengamos que poner los paréntesis porque necesitamos pasarle un parámetro al método entonces podemos añadir como parámetro el evento pasandoselo como `$event`.

/vuejs-data-binding-eventos-lab/src/components/CmpEventos.vue

```
<template>
  <div>
    <button type="button" @click="cambiarEstado($event)">Toggle</button>
    <p>Calefacción: {{calefaccionEncendida ? 'ON' : 'OFF'}}</p>
  </div>
</template>

<script>
export default {
  data() {
    return {
      calefaccionEncendida: false
    }
  },
  methods: {
    cambiarEstado(event) {
      console.log(event);
      this.calefaccionEncendida = !this.calefaccionEncendida;
    }
  }
}
</script>

<style>

</style>
```

8.4. Two Way Data Binding

La directiva `v-model` nos va a permitir sincronizar propiedades que están en *data* con las que están en la vista (en ambos sentidos), es decir, que si una propiedad cambia en el *script* (controlador) también va a cambiar en la vista, y viceversa.

Para ello solo hay que añadir en los inputs la directiva e igualarle cual es la variable que queremos sincronizar, `v-model="variable`.

8.4.1. Lab: Two Way Data Binding

En este laboratorio vamos a ver como usar el two way binding que habilitamos con la directiva `v-model` para que se actualicen los datos en la vista y también lo hagan en el script.

Empezamos por crear el proyecto lanzando el siguiente comando y seleccionamos la configuración por defecto.

```
$ vue create vuejs-data-binding-two-way-binding-lab
```

Una vez creado el proyecto, eliminamos el componente `HelloWorld` y dejamos vacío el componente `App`.

Vamos a añadir el componente `CmpTwoWayBinding`, que crearemos a continuación, dentro del componente de la aplicación.

/vuejs-data-binding-two-way-binding-lab/src/App.vue

```
<template>
  <CmpTwoWayBinding />
</template>

<script>
import CmpTwoWayBinding from './components/CmpTwoWayBinding';
export default {
  components: {
    CmpTwoWayBinding
  }
}
</script>

<style>

</style>
```

Una vez añadido, vamos a poner un párrafo y un input en la plantilla del componente, además de añadir una propiedad `serie` que mostraremos en el párrafo.

/vuejs-data-binding-two-way-binding-lab/src/components/CmpTwoWayBinding.vue

```
<template>
  <div>
    <p>Serie: {{serie}}</p>
    <input type="text">
  </div>
</template>

<script>
export default {
  data() {
    return {
      serie: 'Game of Thrones'
    }
  }
}
</script>

<style>

</style>
```

Una vez que estamos mostrando nuestra propiedad en el párrafo vamos a añadir la directiva `v-model` para poder modificar el valor de la `serie`. Lo único que tenemos que hacer es asignarle a la directiva la propiedad que queremos modificar.

/vuejs-data-binding-two-way-binding-lab/src/components/CmpTwoWayBinding.vue

```
<template>
  <div>
    <p>Serie: {{serie}}</p>
    <input type="text" v-model="serie">
  </div>
</template>

<script>
export default {
  data() {
    return {
      serie: 'Game of Thrones'
    }
  }
}
</script>

<style>

</style>
```

Después de añadir el two way binding, si modificamos el valor de la serie a través del input, debería de modificarse en todos los sitios donde se estaba mostrando.

Pero además, si el valor de la serie cambia por otra acción del usuario, el input en el que hemos aplicado la directiva también tiene que obtener dicho cambio.

Vamos a añadir un botón que va a volver a poner el nombre inicial de la serie cuando pulsemos sobre el.

/vuejs-data-binding-two-way-binding-lab/src/components/CmpTwoWayBinding.vue

```
<template>
  <div>
    <p>Serie: {{serie}}</p>
    <input type="text" v-model="serie">
    <button type="button" @click="resetearSerie">Resetear serie</button>
  </div>
</template>

<script>
export default {
  data() {
    return {
      serie: 'Game of Thrones'
    }
  },
  methods: {
    resetearSerie() {
      this.serie = 'Game of Thrones';
    }
  }
}
</script>

<style>

</style>
```

Ahora podemos darnos cuenta que el `v-model` hace que obtengamos los cambios de la propiedad que le hemos asignado, además de cambiarla cuando nosotros cambiamos su valor en el campo de texto.

Capítulo 9. Estilos

Entre las etiquetas `style` de los componentes podemos añadir los estilos, pero estos estilos se aplicarán a todos los componentes.

En el caso de que los estilos los queramos aplicar localmente al componente en el que se encuentra tendremos que poner la propiedad `scoped` en la etiqueta `style`. Esta propiedad lo que hace es añadir un atributo con un código aleatorio y que va a usar para diferenciar los elementos del componente del resto de la aplicación, de esta forma al aplicar los estilos a estos elementos, no afectarán al resto.

Otras formas de aplicar los estilos son:

- Usando el atributo `style` junto a la directiva `v-bind`, lo que nos permite darle como valor un objeto en el que las claves son las propiedades CSS, y los valores aquellos que queramos ponerles.
- Usando el atributo `class` junto a la directiva `v-bind`, lo que nos permite darle como valor un objeto en el que las claves son las clases definidas en los estilos, y los valores son booleanos que indican si la clase se tiene que aplicar o no.

9.1. Lab: Estilos

En este laboratorio vamos a ver como aplicar estilos en nuestros componentes desde la etiqueta `style`.

Empezamos por crear el proyecto lanzando el siguiente comando y seleccionamos la configuración por defecto.

```
$ vue create vuejs-estilos-lab
```

Una vez creado el proyecto, eliminamos el componente `HelloWorld` y dejamos vacío el componente `App`.

Vamos a crear un componente `CmpEstilos` en el que vamos a añadir un título principal, y en el que vamos a poner que dicho título sea de color azul.

/vuejs-estilos-lab/src/components/CmpEstilos.vue

```
<template>
  <div>
    <h1>Un título</h1>
  </div>
</template>

<script>
export default {

}
</script>

<style>
  h1 {
    color: blue;
  }
</style>
```

Ahora vamos a ir al componente principal de la aplicación, donde vamos a poner este otro componente para mostrarlo.

/vuejs-estilos-lab/src/App.vue

```
<template>
  <div>
    <CmpEstilos />
  </div>
</template>

<script>
import CmpEstilos from './components/CmpEstilos';

export default {
  components: {
    CmpEstilos
  }
}
</script>

<style>

</style>
```

Una vez mostrado el componente, en el navegador, tenemos que ver que se están aplicando correctamente los estilos.

Pero si ponemos otro título, esta vez en el componente `App`, también se aplicará el estilo que hemos puesto antes.

/vuejs-estilos-lab/src/App.vue

```
<template>
  <div>
    <h1>Otro título</h1>
    <CmpEstilos />
  </div>
</template>

<script>
import CmpEstilos from './components/CmpEstilos';

export default {
  components: {
    CmpEstilos
  }
}
</script>

<style>

</style>
```

Ahora los dos títulos aparecen de color azul, y esto se debe a que todos los estilos que ponemos entre las etiquetas `style` se aplican globalmente.

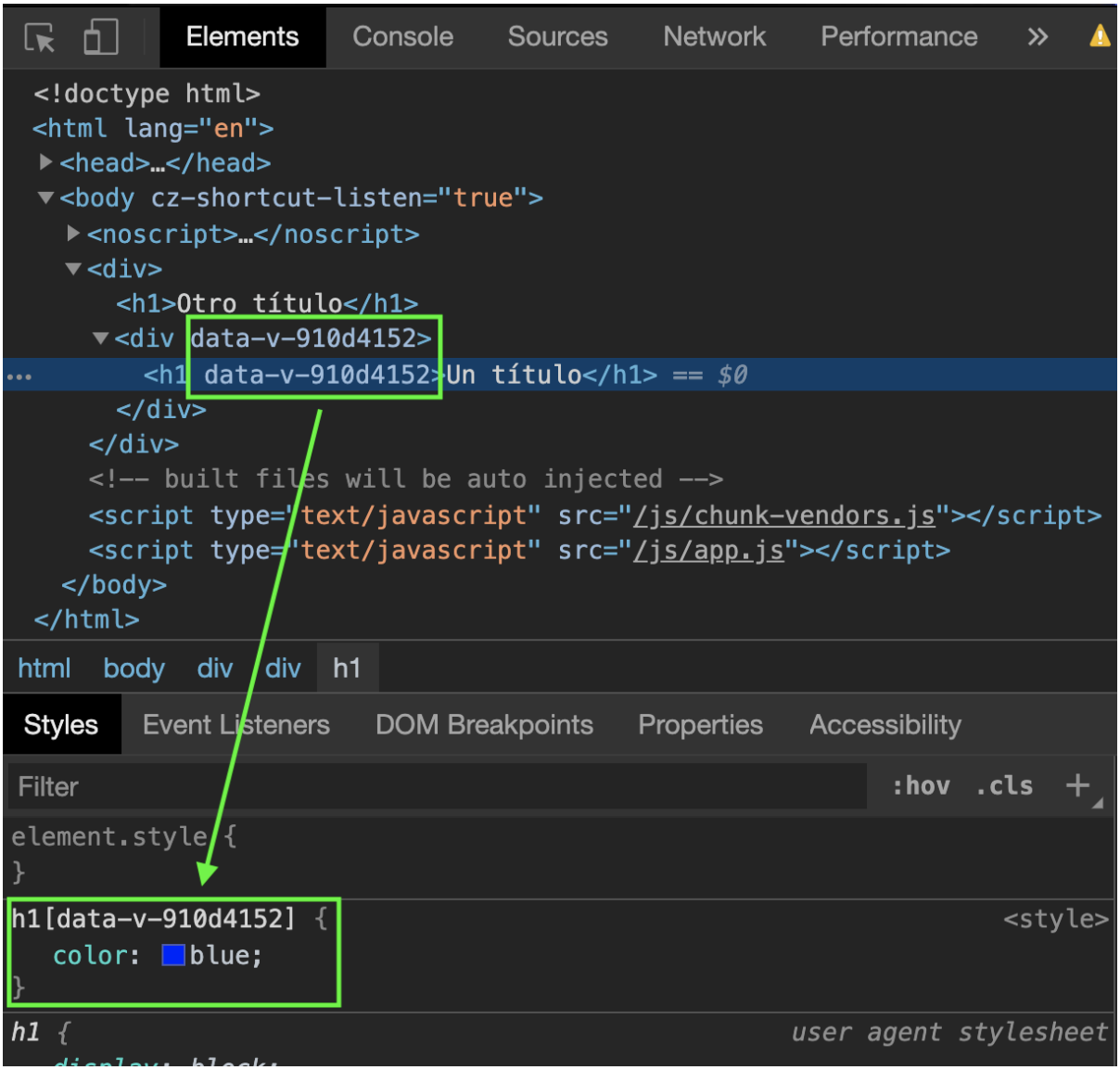
Para evitar que pase esto, y que los estilos solo se apliquen al componente en el que se han definido, tenemos que añadir el atributo `scoped` en la etiqueta `style`.

/vuejs-estilos-lab/src/components/CmpEstilos.vue

```
<template>
  <div>
    <h1>Un título</h1>
  </div>
</template>

<script>
export default {

}
</script>

<style>
  h1 {
    color: blue;
  }
</style>
```

De esta forma, se ha añadido un atributo tanto a los estilos como a las etiquetas de este

componente, evitando que se aplique a aquellos elementos que no se encuentran dentro de este archivo.



Ahora el otro título tiene que aparecer con la letra de color negra.

9.2. Lab: Estilos inline

En este laboratorio vamos a ver como aplicar estilos en nuestros componentes de forma dinámica a través de la propiedad `style` y `class`.

Empezamos por crear el proyecto lanzando el siguiente comando y seleccionamos la configuración por defecto.

```
$ vue create vuejs-estilos-inline-lab
```

Una vez creado el proyecto, eliminamos el componente `HelloWorld` y dejamos vacío el componente `App`.

Vamos a crear un componente `CmpEstilos` dentro de la carpeta `components` en el que vamos a poner

dos párrafos a los que les vamos a aplicar los mismos estilos de distintas formas.

/vuejs-estilos-inline-lab/src/components/CmpEstilos.vue

```
<template>
  <div>
    <p>Le aplicamos estilos con el atributo style</p>
    <p>Le aplicamos estilos con el atributo class</p>
  </div>
</template>

<script>
export default {

}
</script>

<style>

</style>
```

Ahora vamos a mostrar este componente en el componente inicial de la aplicación.

/vuejs-estilos-inline-lab/src/App.vue

```
<template>
  <div>
    <CmpEstilos />
  </div>
</template>

<script>
import CmpEstilos from './components/CmpEstilos';

export default {
  components: {
    CmpEstilos
  }
}
</script>

<style>

</style>
```

Vamos a añadir unos estilos al primer párrafo para que el color del fondo sea negro y la letra naranja, y para ello, vamos a usar la directiva `v-bind` junto al atributo `style`.

El valor que tenemos que asignarle es un objeto cuyas claves sean las propiedades CSS a aplicar (`color` y `backgroundColor`) y el valor aquel que queramos darle.

Las propiedades CSS que tienen más de una palabra hay que escribirlas en **camelcase** (`textDecoration`) o como **string** (`'text-decoration'`).

/vuejs-estilos-inline-lab/src/components/CmpEstilos.vue

```
<template>
  <div>
    <p :style="{backgroundColor: 'black', color: 'orange'}">Le aplicamos estilos con el atributo style</p>
    <p>Le aplicamos estilos con el atributo class</p>
  </div>
</template>

<script>
export default {

}
</script>

<style>

</style>
```

Ahora si miramos el navegador, se tienen que estar aplicando esos estilos correctamente.

Podemos hacer lo mismo pero con el atributo `class`, y este recibe un objeto con el nombre de las clases como claves y un booleano indicando si se tienen que aplicar o no.

Vamos a añadir unas clases dentro de la etiqueta `style` del componente.

/vuejs-estilos-inline-lab/src/components/CmpEstilos.vue

```
<template>
  <div>
    <p :style="{backgroundColor: 'black', color: 'orange'}">Le aplicamos estilos con el atributo style</p>
    <p>Le aplicamos estilos con el atributo class</p>
  </div>
</template>

<script>
export default {

}
</script>

<style scoped>
  .letraNaranja {
    color: orange;
  }

  .fondoNegro {
    background-color: black;
  }
</style>
```

Y ahora vamos a darle los estilos al párrafo correspondiente como se ha explicado arriba.

/vuejs-estilos-inline-lab/src/components/CmpEstilos.vue

```
<template>
  <div>
    <p :style="{backgroundColor: 'black', color: 'orange'}">Le aplicamos estilos con el atributo style</p>
    <p :class="{fondoNegro: true, letraNaranja: true}">Le aplicamos estilos con el atributo class</p>
  </div>
</template>

<script>
export default {

}
</script>

<style scoped>
  .letraNaranja {
    color: orange;
  }

  .fondoNegro {
    background-color: black;
  }
</style>
```

Y al igual que el otro párrafo, este también tiene que estar aplicando los estilos correctamente.

La ventaja de usar estos métodos, es que los valores asignados a las propiedades pueden venir de los datos guardados en el componente.

Vamos a añadir en `data` las propiedades `color` y `activado` que vamos a usar para asignarlas como valores en los estilos de antes.

/vuejs-estilos-inline-lab/src/components/CmpEstilos.vue

```
<template>
  <div>
    <p :style="{backgroundColor: 'black', color: color}">Le aplicamos estilos con el atributo style</p>
    <p :class="{fondoNegro: activado, letraNaranja: true}">Le aplicamos estilos con el atributo class</p>
  </div>
</template>

<script>
export default {
  data() {
    return {
      color: 'yellow',
      activado: false
    }
  }
}
</script>

<style scoped>
  .letraNaranja {
    color: orange;
  }

  .fondoNegro {
    background-color: black;
  }
</style>
```

Y tras este cambio, el color de la letra del primer párrafo será amarillo, mientras que el fondo de color negro del segundo párrafo no se aplica porque le hemos pasado como valor un `false`.

Capítulo 10. Directivas

Las **directivas** añaden comportamiento dinámico a la aplicación usando unos atributos en las etiquetas. Estos atributos le indican a VUE que tiene que hacer en ciertas partes del código.

10.1. Directivas de interpolación

Las directivas de interpolación son aquellas que nos permiten inyectar contenido dentro de los componentes o etiquetas HTML que usamos dentro de estos.

10.1.1. v-text

La directiva `v-text` nos permite inyectar texto que viene de una variable en una etiqueta. Para usarla solo hay que igualarle a esta directiva la variable que contiene el texto a mostrar.

10.1.2. v-html

Si queremos inyectar código HTML dentro de una etiqueta, no podemos hacerlo con el *String Interpolation* porque lo que va a hacer es mostrar el código HTML tal cual está escrito.

La directiva `v-html` nos permite asignarla a una etiqueta HTML y pasarle como valor código HTML que se va a mostrar como HTML y no como texto. Pero hay que tener cuidado con esta directiva para que no nos puedan inyectar código malicioso.

10.1.3. Lab: Directivas de interpolación

En este laboratorio vamos a ver como inyectar tanto texto como código HTML dentro de nuestros componentes usando las directivas apropiadas para ello.

Empezamos por crear el proyecto lanzando el siguiente comando y seleccionamos la configuración por defecto.

```
$ vue create vuejs-directivas-de-interpolacion-lab
```

Una vez creado el proyecto, eliminamos el componente `HelloWorld` y dejamos vacío el componente `App`.

Vamos a crear un componente `CmpDirectivas` en el que vamos a añadir dos datos de texto, y vamos a añadir dos etiquetas vacías donde después inyectaremos aquello que tienen que mostrar.

/vuejs-directivas-de-interpolacion-lab/src/components/CmpDirectivas.vue

```
<template>
  <div>
    <p id="p1"></p>
    <div></div>
  </div>
</template>

<script>
export default {
  data() {
    return {
      texto: 'El botón tiene trampa',
      contenido: '<button onclick="document.getElementById(\'p1\').innerHTML=\'\'">Cuidado con esto</button>'
    }
  }
}
</script>

<style>

</style>
```

Una vez que tenemos este componente, vamos a añadirlo dentro del componente raíz de la aplicación.

/vuejs-directivas-de-interpolacion-lab/src/App.vue

```
<template>
  <div>
    <CmpDirectivas />
  </div>
</template>

<script>
import CmpDirectivas from './components/CmpDirectivas';

export default {
  components: {
    CmpDirectivas
  }
}
</script>

<style>

</style>
```

Lo siguiente que vamos a hacer es inyectar con la directiva `v-text` el valor de `texto` dentro del párrafo, y para ello solo tenemos que añadir dicha directiva en la etiqueta correspondiente.

/vuejs-directivas-de-interpolacion-lab/src/components/CmpDirectivas.vue

```
<template>
  <div>
    <p id="p1" v-text="texto"></p>
    <div></div>
  </div>
</template>

<script>
export default {
  data() {
    return {
      texto: 'El botón tiene trampa',
      contenido: '<button onclick="document.getElementById(\'p1\').innerHTML=\'\'">Cuidado con esto</button>'
    }
  }
}
</script>

<style>

</style>
```

Ahora ya debería de mostrarse el texto en el navegador.

Lo siguiente es inyectar el código HTML que hemos puesto en la otra variable, y para ello, vamos a poner la directiva `v-html` en el `div` del componente y le vamos a asignar como valor el `contenido`.

/vuejs-directivas-de-interpolacion-lab/src/components/CmpDirectivas.vue

```
<template>
  <div>
    <p id="p1" v-text="texto"></p>
    <div v-html="contenido"></div>
  </div>
</template>

<script>
export default {
  data() {
    return {
      texto: 'El botón tiene trampa',
      contenido: '<button onclick="document.getElementById(\'p1\').innerHTML=\'\'">Cuidado con esto</button>'
    }
  }
}
</script>

<style>

</style>
```

Ahora ya deberíamos de poder ver el botón en el navegador, pero hay que tener cuidado con esta directiva, porque si lo pulsamos nos podemos dar cuenta de que nos han inyectado código JS con el botón, por lo que al pulsarlo nos vacía el contenido del párrafo.

10.2. Directivas condicionales

Las directivas condicionales son aquellas que nos permiten cambiar el DOM dependiendo de una condición dada.

10.2.1. v-if

La directiva `v-if` añade o elimina un elemento del DOM dependiendo de si el valor que se le da es un `true` o un `false`.

Si queremos varios elementos de una vez con esta directiva, podemos meterlos entre etiquetas `template` y ponerle el `v-if` a esta etiqueta.

10.2.2. v-else

La directiva `v-else` se usa junto al `v-if` y se mostrará cuando no se cumpla la condición del `v-if`. Esta directiva hay que ponerla justo despues de la directiva `v-if` a la que va asociada.

10.2.3. v-else-if

La directiva `v-else-if` nos permite un control mayor de opciones a la hora de mostrar elementos, porque ahora podemos tener más de una condición. Esta directiva tiene que ir justo después de la directiva `v-if`, y la directiva `v-else` debe de ir detrás de la última directiva de este tipo que vayamos a poner.

10.2.4. v-show

La directiva `v-show` actua como el `v-if`, pero en este caso elimina o añade el elemento, sino que le añade o le quita el estilo `display: none` a las etiquetas.

10.2.5. Lab: Directivas condicionales

En este laboratorio vamos a ver como usar las directivas condicionales para mostrar unos componentes u otros dependiendo de la condición que les pasemos.

Empezamos por crear el proyecto lanzando el siguiente comando y seleccionamos la configuración por defecto.

```
$ vue create vuejs-directivas-condicionales-lab
```

Una vez creado el proyecto, eliminamos el componente `HelloWorld` y dejamos vacío el componente `App`.

Vamos a crear un componente `CmpDirectivas` dentro de la carpeta `components`. Dentro del cual vamos a poner un contador de una nota con dos botones para incrementar y decrementar.

/vuejs-directivas-condicionales-lab/src/components/CmpDirectivas.vue

```
<template>
  <div>
    <div>
      <button type="button" @click="cambiarNota(-1)">-1</button>
      <span>{{nota}}</span>
      <button type="button" @click="cambiarNota(1)">+1</button>
    </div>
  </div>
</template>

<script>
export default {
  data() {
    return {
      nota: 5
    }
  },
  methods: {
    cambiarNota(cantidad) {
      this.nota += cantidad;
    }
  }
}
</script>

<style>

</style>
```

Una vez que tenemos el contador, vamos a mostrar este componente en al componente raíz de la aplicación, el `App`.

/vuejs-directivas-condicionales-lab/src/App.vue

```
<template>
  <div>
    <CmpDirectivas />
  </div>
</template>

<script>
import CmpDirectivas from './components/CmpDirectivas';

export default {
  components: {
    CmpDirectivas
  }
}
</script>

<style>

</style>
```

Con lo que hemos hecho hasta ahora deberíamos de poder incrementar y decrementar la nota.

Ahora vamos a añadir unos párrafos en la plantilla del componente para mostrar si la nota corresponde a un `aprobado`, `suficiente`, `suspenso` o es una nota `no válida`.

Para ello vamos a usar las directivas `v-if`, `v-else-if` y `v-else` añadiendoles las condiciones necesarias.

/vuejs-directivas-condicionales-lab/src/components/CmpDirectivas.vue

```
<template>
  <div>
    <div>
      <button type="button" @click="cambiarNota(-1)">-1</button>
      <span>{{nota}}</span>
      <button type="button" @click="cambiarNota(1)">+1</button>
    </div>
    <p v-if="nota > 5 && nota < 11">Aprobado</p>
    <p v-else-if="nota > -1 && nota < 5">Suspenso</p>
    <p v-else-if="nota === 5">Suficiente</p>
    <p v-else>No válida</p>
  </div>
</template>

<script>
export default {
  data() {
    return {
      nota: 5
    }
  },
  methods: {
    cambiarNota(cantidad) {
      this.nota += cantidad;
    }
  }
}
</script>

<style>

</style>
```

Ahora cuando vamos modificando el valor de la nota deben de irse creando o eliminando los párrafos según las condiciones dadas a las directivas condicionales.

En este caso, hemos usado directivas que crean o eliminan los elementos, pero también tenemos la directiva `v-show` que se encarga solo de ocultarlos o mostrarlos con la propiedad de CSS `display: none`.

Vamos a añadirla a los botones para que no se nos permita llegar a las notas inválidas.

/vuejs-directivas-condicionales-lab/src/components/CmpDirectivas.vue

```
<template>
  <div>
    <div>
      <button type="button" v-show="nota > 0" @click="cambiarNota(-1)">-1</button>
      <span>{{nota}}</span>
      <button type="button" v-show="nota < 10" @click="cambiarNota(1)">+1</button>
    </div>
    <p v-if="nota > 5 && nota < 11">Aprobado</p>
    <p v-else-if="nota > -1 && nota < 5">Suspenso</p>
    <p v-else-if="nota === 5">Suficiente</p>
    <p v-else>No válida</p>
  </div>
</template>

<script>
export default {
  data() {
    return {
      nota: 5
    }
  },
  methods: {
    cambiarNota(cantidad) {
      this.nota += cantidad;
    }
  }
}
</script>

<style>

</style>
```

Ahora cuando lleguemos al `10` el botón del `+1` se ocultará, y ocurrira lo mismo con el botón de `-1` cuando lleguemos al `0`.

10.3. Directivas de listas

Las directivas de listas son aquellas que nos permiten recorrer arrays u objetos y mostrar tantos elementos HTML como elementos hubiera.

10.3.1. v-for

Con la directiva `v-for` podemos replicar un elemento HTML o componente tantas veces como elementos haya en un array (u objeto iterable). La sintaxis es `<li v-for="elem in elementos">{{ elem }}</li>`.

Podemos obtener la posición en la que se encuentra cada elemento con `v-for="(elem, pos) in elementos"`.

También es posible iterar sobre objetos JavaScript, y en este caso el primer argumento que vamos a obtener es el valor, y el segundo será la clave. En este caso también podemos añadir un tercer argumento donde obtendremos el número de iteración.

Otra de las características de esta directiva, es que podemos iterar sobre un rango si en lugar de un array ponemos un número. En este caso el primer valor que se va a mostrar siempre es 1.

10.3.2. Atributo key

En las listas de elementos, habría que añadir un atributo `key` en el cual vamos a poner un valor único. Este atributo es necesario ya que ayuda a Vue a reutilizar o reordenar los elementos de la lista cuando esta cambia.

10.3.3. Lab: Directivas de listas

En este laboratorio vamos a ver como usar la directiva `v-for` para crear un componente por cada elemento de los que tenemos en una lista.

Empezamos por crear el proyecto lanzando el siguiente comando y seleccionamos la configuración por defecto.

```
$ vue create vuejs-directivas-de-listas-lab
```

Una vez creado el proyecto, eliminamos el componente `HelloWorld` y dejamos vacío el componente `App`.

Vamos a añadir un array de mascotas dentro del componente raíz de nuestra aplicación. Cada mascota empezará teniendo el tipo de mascota que es, y una imagen.

/vuejs-directivas-de-listas-lab/src/App.vue

```
<template>
  <div>

  </div>
</template>

<script>
export default {
  data() {
    return {
      mascotas: [
        { tipo: 'Perro', img:
'https://vignette.wikia.nocookie.net/reinoanimalia/images/a/a9/Husky_siberiano_20.png/revision/latest?cb=201505130447508&p
ath-prefix=es' },
        { tipo: 'Ninfa', img: 'https://arrobaparktienda.com/lavoladera/wp-content/uploads/2014/10/ninfa-o-carolina.jpg'
},
        { tipo: 'Tortuga', img: 'https://misanimales.com/wp-content/uploads/2017/10/edad-de-una-
tortuga.jpg?width=1200&enable=upscale' },
      ]
    }
  }
}
</script>

<style>

</style>
```

Ahora vamos a crear un componente `Mascota` dentro de la carpeta `components` donde vamos a mostrar los datos de cada mascota.

Dentro de este componente vamos a indicarle que se va a recibir un objeto como propiedad y que usaremos para mostrar los datos.

/vuejs-directivas-de-listas-lab/src/components/Mascota.vue

```
<template>
  <div :style="{width: '300px', border: '1px solid black'}">
    <img :src="mascota.img" :alt="altImgMascota" :style="{width: '100%'}">
    <h4 :style="{textAlign: 'center'}">{{mascota.tipo}}</h4>
  </div>
</template>

<script>
export default {
  props: {
    mascota: {
      type: Object,
      required: true
    }
  },
  computed: {
    altImgMascota() {
      return 'Imagen de ' + this.mascota.tipo;
    }
  }
}
</script>

<style>

</style>
```

Ahora que tenemos el componente `Mascota`, vamos a mostrar un componente de estos por cada mascota que tenemos en el array, y para ello, vamos a importar este componente dentro del componente `App` y le vamos a añadir la directiva `v-for` de la que obtendremos cada mascota y que tendremos que ir pasando como propiedad al componente que le hemos puesto la directiva.

/vuejs-directivas-de-listas-lab/src/App.vue

```
<template>
  <div>
    <Mascota v-for="mascota in mascotas" :mascota="mascota" />
  </div>
</template>

<script>
import Mascota from './components/Mascota';

export default {
  components: {
    Mascota
  },
  data() {
    return {
      mascotas: [
        { tipo: 'Perro', img:
'https://vignette.wikia.nocookie.net/reinoanimalia/images/a/a9/Husky_siberiano_20.png/revision/latest?cb=20150513044750&p
ath-prefix=es' },
        { tipo: 'Ninfa', img: 'https://arrobaparktienda.com/lavoladera/wp-content/uploads/2014/10/ninfa-o-carolina.jpg'
},
        { tipo: 'Tortuga', img: 'https://misanimales.com/wp-content/uploads/2017/10/edad-de-una-
tortuga.jpg?width=1200&enable=upscale' },
      ]
    }
  }
}
</script>

<style>

</style>
```

Al añadir la directiva, si vamos al navegador, podemos ver que salen las mascotas, pero también se nos muestra un error en el que se indica que tenemos que añadir una propiedad `key` cuando trabajamos con listas.

Esta propiedad `key` tiene que ser un valor único que no se repita dentro de la lista que se está generando/modificando, porque va a ser aquello que le indique a Vue si tiene que pintar toda la lista de nuevo, o solo tiene que añadir/modificar/eliminar un solo elemento de la lista.

Esto lo podemos solucionar poniendo la posición de cada mascota en el array como valor para `key`.

/vuejs-directivas-de-listas-lab/src/App.vue

```
<template>
  <div>
    <Mascota v-for="(mascota, pos) in mascotas" :key="pos" :mascota="mascota" />
  </div>
</template>

<script>
import Mascota from './components/Mascota';

export default {
  components: {
    Mascota
  },
  data() {
    return {
      mascotas: [
        { tipo: 'Perro', img:
'https://vignette.wikia.nocookie.net/reinoanimalia/images/a/a9/Husky_siberiano_20.png/revision/latest?cb=20150513044750&p
ath-prefix=es' },
        { tipo: 'Ninfa', img: 'https://arrobaparktienda.com/lavoladera/wp-content/uploads/2014/10/ninfa-o-carolina.jpg'
},
        { tipo: 'Tortuga', img: 'https://misanimales.com/wp-content/uploads/2017/10/edad-de-una-
tortuga.jpg?width=1200&enable=upscale' },
      ]
    }
  }
}
</script>

<style>

</style>
```

Una vez añadida esta propiedad, ya se ha debido de quitar el error que teníamos, pero todavía no está del todo bien, porque además de que el valor que se asigna a la `key` debe de ser único en la lista, también tiene que ser el mismo para cada elemento que se asigna, es decir que si al componente que muestra el *perro* se le ha asignado la posición `0`, su posición no debería de cambiar nunca, pero en nuestro caso, si añadimos una nueva mascota al inicio del array, la posición del *perro* habrá cambiado provocando que Vue no sepa como renderizar de forma optima la lista.

Para solucionar este problema, la mejor opción siempre es la de usar un `id` como valor del `key`, por tanto vamos a añadir un valor único a cada mascota, y vamos a usar dicho valor como `key`, en lugar de usar la posición como estabamos haciendo.

/vuejs-directivas-de-listas-lab/src/App.vue

```
<template>
  <div>
    <Mascota v-for="mascota in mascotas" :key="mascota.id" :mascota="mascota" />
  </div>
</template>

<script>
import Mascota from './components/Mascota';

export default {
  components: {
    Mascota
  },
  data() {
    return {
      mascotas: [
        { id: 104, tipo: 'Perro', img:
'https://vignette.wikia.nocookie.net/reinoanimalia/images/a/a9/Husky_siberiano_20.png/revision/latest?cb=20150513044750&p
ath-prefix=es' },
        { id: 391, tipo: 'Ninfa', img: 'https://arrobaparktienda.com/lavoladera/wp-content/uploads/2014/10/ninfa-o-
carolina.jpg' },
        { id: 83, tipo: 'Tortuga', img: 'https://misanimales.com/wp-content/uploads/2017/10/edad-de-una-
tortuga.jpg?width=1200&enable=upscale' },
      ]
    }
  }
}
</script>

<style>

</style>
```

10.3.4. Lab: Directivas de listas con objetos

En este laboratorio vamos a ver como usar la directiva `v-for` para iterar sobre un objeto de javascript y mostrar todas sus propiedades con sus valores en una lista.

Empezamos por crear el proyecto lanzando el siguiente comando y seleccionamos la configuración por defecto.

```
$ vue create vuejs-directivas-de-listas-objetos-lab
```

Una vez creado el proyecto, eliminamos el componente `HelloWorld` y dejamos vacío el componente `App`.

Vamos a añadir un objeto que representa a una serie dentro del componente raíz de nuestra aplicación.

/vuejs-directivas-de-listas-objetos-lab/src/App.vue

```
<template>
  <div>
  </div>
</template>

<script>
export default {
  data() {
    return {
      serie: {
        titulo: 'Bitten',
        temporadas: 3,
        finalizada: true
      }
    }
  }
}
</script>

<style>

</style>
```

Una vez que tenemos el objeto, vamos a añadir en la plantilla una lista, donde vamos a crear un elemento `li` por cada propiedad que tiene el objeto `serie`. En este caso, dentro de la directiva `v-for` nos va a ir devolviendo el valor y la propiedad que usaremos para mostrarlas dentro del `li` y para generar el valor del atributo `key`.

/vuejs-directivas-de-listas-objetos-lab/src/App.vue

```
<template>
  <div>
    <ul>
      <li v-for="(val, key) in serie" :key="key+val">{{key}}: {{val}}</li>
    </ul>
  </div>
</template>

<script>
export default {
  data() {
    return {
      serie: {
        titulo: 'Bitten',
        temporadas: 3,
        finalizada: true
      }
    }
  }
}
</script>

<style>

</style>
```

Una vez que tenemos esto, si vamos al navegador, veremos que se muestran todas las propiedades del objeto junto a sus valores.

10.3.5. Lab: Directivas de listas para un rango

En este laboratorio vamos a ver como usar la directiva `v-for` para crear una lista de números pasandole a la directiva el número final en lugar de un array u objeto.

Empezamos por crear el proyecto lanzando el siguiente comando y seleccionamos la configuración por defecto.

```
$ vue create vuejs-directivas-de-listas-lab
```

Una vez creado el proyecto, eliminamos el componente `HelloWorld` y dejamos vacío el componente `App`.

Dentro del componente `App` vamos a añadir una lista desordenada en la que vamos a añadir tantos elementos `li` como el número que le hayamos pasado a la directiva.

/vuejs-directivas-de-listas-rangos-lab/src/App.vue

```
<template>
  <div>
    <ul>
      <li v-for="num in 4" :key="num">{{num}}</li>
    </ul>
  </div>
</template>

<script>
export default {

}
</script>

<style>

</style>
```

Una vez que tenemos el paso anterior, si vamos al navegador podrémos ver una lista que muestra los números del 1 al 4.

- No es un rango en realidad, porque no se puede pasar un rango como tal indicando que queremos que muestre los números del 3 al 7 por ejemplo.
- Siempre se van a mostrar del 1 al número que hayamos puesto.
- El primero número que nos devuelve el rango es el 1 y no el 0.

10.4. Crear una directiva local

En Vue podemos crear nuestras propias directivas, y para ello tenemos que añadirlas en la propiedad `directives` del componente, la cual recibe el nombre de la directiva como clave y un objeto con unos métodos que nos van a permitir indicar que tiene que hacer la directiva que se está creando.

A las directivas le podemos pasar un **parámetro** poniendo `:param` detrás del nombre.

También es posible usar nuestros propios **modificadores** en las directivas, se usan igual que los parámetros, solo que esta vez se le pasan `.modificador`.

A estos elementos podemos acceder en los métodos del ciclo de vida de las directivas:

- `bind`: se llama una única vez cuando la directiva se asigna al elemento.
- `inserted`: se llama cuando el elemento al que se ha asignado la directiva se ha insertado en el nodo padre.
- `update`: se llama cuando el elemento que usa la directiva, se actualiza.
- `componentUpdated`: se llama después de que se haya actualizado el elemento que contiene la

directiva y sus hijos.

- `unbind`: se llama una única vez cuando se va a desasignar la directiva del elemento.

Y estas funciones que añadimos en la directiva se reciben como parámetro:

- `el`: elemento al que se ha aplicado la directiva.
- `binding`: objeto que contiene las siguientes propiedades:
  - `name`: nombre de la directiva.
  - `value`: el valor pasado a la directiva o el valor resultante de la expresión.
  - `oldValue`: el antiguo valor de la directiva. Solo está disponible en los métodos de `update` y `componentUpdated`.
  - `expression`: expresión asignada a la directiva.
  - `arg`: el argumento que se le ha pasado a la directiva.
  - `modifiers`: objeto con los modificadores pasados a la directiva.
- `vnode`: nodo virtual producido por el compilador de Vue.
- `oldVnode`: antiguo nodo virtual al que solo tienen acceso los métodos `update` y `componentUpdated`.

Si necesitamos pasar datos entre los distintos métodos de la directiva deberíamos de hacerlo a través de la propiedad `dataset` de los elementos.

10.4.1. Lab: Directivas locales

En este laboratorio vamos a ver como crear nuestra propia directiva que le añada cierta funcionalidad a los componentes o etiquetas HTML usadas en la aplicación.

Empezamos por crear el proyecto lanzando el siguiente comando y seleccionamos la configuración por defecto.

```
$ vue create vuejs-directivas-locales-lab
```

Una vez creado el proyecto, eliminamos el componente `HelloWorld` y dejamos vacío el componente `App`.

Vamos a crear un componente `CmpDirectivaLocal` en la carpeta `components` que vamos a importar y añadir en la plantilla del componente `App`.

/vuejs-directivas-locales-lab/src/App.vue

```
<template>
  <div>
    <CmpDirectivaLocal />
  </div>
</template>

<script>
import CmpDirectivaLocal from './components/CmpDirectivaLocal';

export default {
  components: {
    CmpDirectivaLocal
  }
}
</script>

<style>

</style>
```

Ahora vamos a ir al componente que hemos creado antes donde vamos a añadir una propiedad `directives` en la que pondremos nuestras directivas personalizadas.

/vuejs-directivas-locales-lab/src/components/CmpDirectivaLocal.vue

```
<template>
  <div>

  </div>
</template>

<script>
export default {
  directives: {

  }
}
</script>

<style>

</style>
```

Cada directiva es un objeto donde la clave es el nombre de esta, y el valor es un objeto en que pondremos los hooks necesarios para añadir la funcionalidad de la directiva.

/vuejs-directivas-locales-lab/src/components/CmpDirectivaLocal.vue

```
<template>
  <div>

  </div>
</template>

<script>
export default {
  directives: {
    marcar: {

    }
  }
}
</script>

<style>

</style>
```

El nombre de la directiva no tiene que llevar el `v-`.

En nuestro caso, vamos a usar el método `bind`, por lo tanto vamos a ponerlo dentro del objeto que define nuestra directiva.

/vuejs-directivas-locales-lab/src/components/CmpDirectivaLocal.vue

```
<template>
  <div>

  </div>
</template>

<script>
export default {
  directives: {
    marcar: {
      bind(el, binding) {

      }
    }
  }
}
</script>

<style>

</style>
```

Lo primero que vamos a hacer es cambiar de color el fondo de la etiqueta a la que le asignemos la directiva, y para ello, como tenemos esta etiqueta en el parámetro `el`, podemos cambiarlo accediendo a sus estilos y dentro de estos a la propiedad del color de fondo.

/vuejs-directivas-locales-lab/src/components/CmpDirectivaLocal.vue

```
<template>
  <div>

  </div>
</template>

<script>
export default {
  directives: {
    marcar: {
      bind(el, binding) {
        el.style.backgroundColor = 'yellow';
      }
    }
  }
}
</script>

<style>

</style>
```

Una vez que lo tenemos, vamos a aplicar esta directiva a un párrafo. Solo tenemos que poner en la etiqueta `v-` seguido del nombre que hemos dado a la directiva.

/vuejs-directivas-locales-lab/src/components/CmpDirectivaLocal.vue

```
<template>
  <div>
    <p v-marcar>Este texto aparecerá marcado</p>
  </div>
</template>

<script>
export default {
  directives: {
    marcar: {
      bind(el, binding) {
        el.style.backgroundColor = 'yellow';
      }
    }
  }
}
</script>

<style>

</style>
```

Ahora si vamos al navegador, deberíamos de ver el párrafo con el fondo de color amarillo como le hemos indicado en la directiva.

Ahora vamos a añadir otro párrafo en el que vamos a reutilizar esta misma directiva, pero en este caso le vamos a indicar el color con el que queremos que se marque el párrafo nuevo.

/vuejs-directivas-locales-lab/src/components/CmpDirectivaLocal.vue

```
<template>
  <div>
    <p v-marcar>Este texto aparecerá marcado</p>
    <p v-marcar="'blue'">Este texto aparecerá marcado en azul</p>
  </div>
</template>

<script>
export default {
  directives: {
    marcar: {
      bind(el, binding) {
        el.style.backgroundColor = 'yellow';
      }
    }
  }
}
</script>

<style>

</style>
```

Podemos acceder al valor asignado desde el parámetro `binding` que recibe la función del `bind` de nuestra directiva, por lo que en lugar de asignar siempre el color amarillo, vamos a asignar el color que le pasamos, y en caso de no pasarle ninguno se usará por defecto el amarillo.

/vuejs-directivas-locales-lab/src/components/CmpDirectivaLocal.vue

```
<template>
  <div>
    <p v-marcar>Este texto aparecerá marcado</p>
    <p v-marcar="'blue'">Este texto aparecerá marcado en azul</p>
  </div>
</template>

<script>
export default {
  directives: {
    marcar: {
      bind(el, binding) {
        const color = binding.value || 'yellow';
        el.style.backgroundColor = color;
      }
    }
  }
}
</script>

<style>

</style>
```

Ahora ya deberían de aparecer los dos párrafos coloreados, cada uno con su color.

El siguiente paso es hacer que solo se marquen cuando pasemos el ratón por encima de los párrafos, y para ello vamos a añadir unos listeners sobre los elementos en los que se ha usado la directiva que cambien el color de fondo a blanco (color por defecto), o al color dado como valor cuando se detectan los eventos `mouseout` y `mouseover`.

/vuejs-directivas-locales-lab/src/components/CmpDirectivaLocal.vue

```
<template>
  <div>
    <p v-marcar>Este texto aparecerá marcado</p>
    <p v-marcar="'blue'">Este texto aparecerá marcado en azul</p>
  </div>
</template>

<script>
export default {
  directives: {
    marcar: {
      bind(el, binding) {
        const colorPorDefecto = 'white';
        const color = binding.value || 'yellow';
        el.style.backgroundColor = colorPorDefecto;
        el.addEventListener('mouseover', () => {
          el.style.backgroundColor = color;
        })
        el.addEventListener('mouseout', () => {
          el.style.backgroundColor = colorPorDefecto;
        })
      }
    }
  }
}
</script>

<style>

</style>
```

Ahora solo se deberían de pintar cuando el ratón se encuentra sobre los párrafos.

10.4.2. Añadiendo un modificador

Lo siguiente que vamos a hacer es añadir un modificador `delayed` a nuestra directiva. Si se le añade el modificador, el cambio de color cuando pasamos el ratón por encima no tiene que ser instantaneo.

En este caso, vamos a añadir dicho modificador en un nuevo párrafo, detrás de la directiva.

/vuejs-directivas-locales-lab/src/components/CmpDirectivaLocal.vue

```
<template>
  <div>
    <p v-marcar>Este texto aparecerá marcado</p>
    <p v-marcar="'blue'">Este texto aparecerá marcado en azul</p>
    <p v-marcar.delayed="'red'">Este texto aparecerá marcado en rojo</p>
  </div>
</template>

<script>
export default {
  directives: {
    marcar: {
      bind(el, binding) {
        const colorPorDefecto = 'white';
        const color = binding.value || 'yellow';
        el.style.backgroundColor = colorPorDefecto;
        el.addEventListener('mouseover', () => {
          el.style.backgroundColor = color;
        })
        el.addEventListener('mouseout', () => {
          el.style.backgroundColor = colorPorDefecto;
        })
      }
    }
  }
}
</script>

<style>

</style>
```

Una vez que tenemos el modificador aplicado, vamos a añadir la funcionalidad extra que se va a añadir. Para comprobar que llega el modificador tendremos que acceder a `binding.modifiers` y en el caso de que se encuentre el que hemos añadido vamos a hacer que el cambio de color se realice dentro de una función `setTimeout` de javascript.

/vuejs-directivas-locales-lab/src/components/CmpDirectivaLocal.vue

```
<template>
  <div>
    <p v-marcar>Este texto aparecerá marcado</p>
    <p v-marcar="'blue'">Este texto aparecerá marcado en azul</p>
    <p v-marcar.delayed="'red'">Este texto aparecerá marcado en rojo</p>
  </div>
</template>

<script>
export default {
  directives: {
    marcar: {
      bind(el, binding) {
        const colorPorDefecto = 'white';
        const color = binding.value || 'yellow';
        el.style.backgroundColor = colorPorDefecto;
        el.addEventListener('mouseover', () => {
          if (binding.modifiers.delayed) {
            setTimeout(() => {
              el.style.backgroundColor = color;
            }, 500);
          } else {
            el.style.backgroundColor = color;
          }
        })
        el.addEventListener('mouseout', () => {
          el.style.backgroundColor = colorPorDefecto;
        })
      }
    }
  }
}
</script>

<style>

</style>
```

10.4.3. Añadiendo un argumento

Por último vamos a añadir un argumento para indicar a que parte del párrafo queremos cambiarle el color, si al fondo o a la letra. Para ello vamos a pasarle como argumento detrás de la directiva `background` o `font`.

Añadimos otros dos párrafos con la directiva y un argumento para cada uno de ellos.

/vuejs-directivas-locales-lab/src/components/CmpDirectivaLocal.vue

```
<template>
  <div>
    <p v-marcar>Este texto aparecerá marcado</p>
    <p v-marcar="'blue'">Este texto aparecerá marcado en azul</p>
    <p v-marcar.delayed="'red'">Este texto aparecerá marcado en rojo</p>
    <p v-marcar:background.delayed="'green'">Este texto aparecerá marcado en verde</p>
    <p v-marcar:color.delayed="'orange'">Este texto aparecerá con la letra en naranja</p>
  </div>
</template>

<script>
export default {
  directives: {
    marcar: {
      bind(el, binding) {
        const colorPorDefecto = 'white';
        const color = binding.value || 'yellow';
        el.style.backgroundColor = colorPorDefecto;
        el.addEventListener('mouseover', () => {
          if (binding.modifiers.delayed) {
            setTimeout(() => {
              el.style.backgroundColor = color;
            }, 500);
          } else {
            el.style.backgroundColor = color;
          }
        })
        el.addEventListener('mouseout', () => {
          el.style.backgroundColor = colorPorDefecto;
        })
      }
    }
  }
}
</script>

<style>

</style>
```

Ahora vamos a comprobar que argumento nos ha llegado accediendo a `binding.arg` y dependiendo de este valor cambiaremos el color de una propiedad o de otra.

/vuejs-directivas-locales-lab/src/components/CmpDirectivaLocal.vue

```
<template>
  <div>
    <p v-marcar>Este texto aparecerá marcado</p>
    <p v-marcar="'blue'">Este texto aparecerá marcado en azul</p>
    <p v-marcar.delayed="'red'">Este texto aparecerá marcado en rojo</p>
    <p v-marcar:background.delayed="'green'">Este texto aparecerá marcado en verde</p>
    <p v-marcar:color.delayed="'orange'">Este texto aparecerá con la letra en naranja</p>
  </div>
</template>

<script>
export default {
  directives: {
    marcar: {
      bind(el, binding) {
        const colorPorDefecto = 'white';
        console.log(binding)
        const color = binding.value || 'yellow';
        el.style.backgroundColor = colorPorDefecto;
        el.addEventListener('mouseover', () => {
          if (binding.modifiers.delayed) {
            setTimeout(() => {
              if (binding.arg === 'color') {
                el.style.color = color;
              } else {
                el.style.backgroundColor = color;
              }
            }, 500);
          } else {
            if (binding.arg === 'color') {
              el.style.color = color;
            } else {
              el.style.backgroundColor = color;
            }
          }
        })
        el.addEventListener('mouseout', () => {
          el.style.backgroundColor = colorPorDefecto;
        })
      }
    }
  }
}
</script>

<style>

</style>
```

Una vez añadida esta funcionalidad, al pasar el ratón por el último párrafo se tiene que cambiar la letra de color, pero al pasarlo por cualquier otro párrafo se tiene que cambiar el fondo de color.

10.5. Directivas globales

Las directivas las podemos declarar como directivas globales en caso de que necesitemos usarlas en varios sitios. Para declarar una directiva como global tenemos que usar `Vue.directive(nombre, config)` antes de crear la instancia de Vue.

10.5.1. Lab: Directivas globales

En este laboratorio vamos a ver como crear una directiva global que haga parpadear el elemento al que se le asigne.

Empezamos por crear el proyecto lanzando el siguiente comando y seleccionamos la configuración por defecto.

```
$ vue create vuejs-directivas-globales-lab
```

Una vez creado el proyecto, eliminamos el componente `HelloWorld` y dejamos vacío el componente `App`.

Lo primero que vamos a hacer es crear un archivo `directivas-globales.js` en la carpeta `src` donde vamos a crear la directiva global. Importaremos `Vue` para poder llamar a la función `Vue.directive` a la que le vamos a pasar dos párametros, el primero el nombre de la directiva y el segundo el objeto que define el comportamiento de la directiva.

/vuejs-directivas-globales-lab/src/directivas-globales.js

```
import Vue from 'vue'

Vue.directive('blink', {

})
```

Ahora vamos a añadir el método `bind` donde vamos a usar la función `setInterval` para que se cambie el color del elemento cada cierto tiempo. Además, esta función devuelve un identificador que vamos a usar después.

/vuejs-directivas-globales-lab/src/directivas-globales.js

```
import Vue from 'vue'

Vue.directive('blink', {
  bind(el) {
    const color = 'black';
    const intervalId = setInterval(() => {
      el.style.backgroundColor = el.style.backgroundColor === 'white' ? color : 'white';
    }, 300)
  }
})
```

Ahora vamos a ir al archivo `main.js` porque al igual que los componentes globales, las directivas globales se tienen que cargar antes de crear la instancia de Vue.

/vuejs-directivas-globales-lab/src/main.js

```
import Vue from 'vue'
import App from './App.vue'
import './directivas-globales';

Vue.config.productionTip = false

new Vue({
  render: h => h(App),
}).$mount('#app')
```

Ahora vamos a ir al componente de la aplicación donde vamos a añadir un párrafo al que le vamos a aplicar esta directiva.

/vuejs-directivas-globales-lab/src/App.vue

```
<template>
  <div>
    <p v-blink>Este párrafo parpadea</p>
  </div>
</template>

<script>
export default {

}
</script>

<style>

</style>
```

Ahora nos encontramos con un problema, y es que si ese párrafo se eliminara de la aplicación, el intervalo de la directiva se seguiría ejecutando, por lo tanto vamos a evitar esto llamando a la función `clearInterval` dentro del método `unbind` de la directiva global.

/vuejs-directivas-globales-lab/src/directivas-globales.js

```
import Vue from 'vue'

Vue.directive('blink', {
  bind(el) {
    const color = 'black';
    const intervalId = setInterval(() => {
      el.style.backgroundColor = el.style.backgroundColor === 'white' ? color : 'white';
    }, 300)
  },
  unbind(el) {

  }
})
```

Pero antes de poder llamar al método indicado, tenemos que poder pasar de alguna forma el identificador, por lo tanto vamos a añadirlo dentro del atributo `dataset` del elemento, para después leerlo y poder eliminar el intervalo.

/vuejs-directivas-globales-lab/src/directivas-globales.js

```
import Vue from 'vue'

Vue.directive('blink', {
  bind(el) {
    const color = 'black';
    const intervalId = setInterval(() => {
      el.style.backgroundColor = el.style.backgroundColor === 'white' ? color : 'white';
    }, 300)
    el.dataset.intervalId = intervalId;
  },
  unbind(el) {
    clearInterval(el.dataset.intervalId);
  }
})
```

Con esto ya tendríamos nuestra directiva, para comprobar que se elimina correctamente el intervalo, vamos a añadir un botón que se encargue de eliminar el párrafo al que hemos puesto la directiva.

/vuejs-directivas-globales-lab/src/App.vue

```
<template>
  <div>
    <button type="button" @click="mostrar = !mostrar">Ocultar</button>
    <p v-blink v-if="mostrar">Este párrafo parpadea</p>
  </div>
</template>

<script>
export default {
  data() {
    return {
      mostrar: true
    }
  }
}
</script>

<style>

</style>
```

Además vamos a poner un `console.log` dentro del intervalo de la directiva para ver que cuando se elimina, deja de salir el mensaje por consola.

/vuejs-directivas-globales-lab/src/directivas-globales.js

```
import Vue from 'vue'

Vue.directive('blink', {
  bind(el) {
    const color = 'black';
    const intervalId = setInterval(() => {
      el.style.backgroundColor = el.style.backgroundColor === 'white' ? color : 'white';
      console.log('Se pinta!')
    }, 300)
    el.dataset.intervalId = intervalId;
  },
  unbind(el) {
    clearInterval(el.dataset.intervalId);
  }
})
```

Capítulo 11. Computed

A veces tenemos que usar algunos valores que dependen de otras propiedades para calcular dicho valor. Podriamos controlar nosotros el cambio de estas propiedades, pero si la aplicación es grande, puede que sea poco mantenible.

Las **computed properties** nos ayudan exactamente a calcular el nuevo valor de una propiedad que depende de otra. En este caso, estas propiedades van a ser funciones que devuelven ese valor, y se ponen dentro de la propiedad `computed` del componente.

La principal diferencia entre las *computed properties* y los *methods* es que las primeras funciones solo se van a ejecutar cuando cambie la propiedad de la que dependen para calcular el valor mientras que las segundas se van a ejecutar siempre que se renderice el componente.

Capítulo 12. Lab: Computed Props

En este laboratorio vamos a ver como usar las computed props para mostrar el número de clicks que se han hecho sobre un botón.

Empezamos por crear el proyecto lanzando el siguiente comando y seleccionamos la configuración por defecto.

```
$ vue create vuejs-computed-props-lab
```

Una vez creado el proyecto, eliminamos el componente `HelloWorld` y dejamos vacío el componente `App`.

Ahora vamos a crearnos un componente, donde vamos a añadir una propiedad `precio` dentro de los datos del componente y un método que va a ir aumentando el precio.

/vuejs-computed-props-lab/src/components/CmpComputedProps.vue

```
<template>
  <div>

  </div>
</template>

<script>
export default {
  data() {
    return {
      precio: 0
    }
  },
  methods: {
    incrementar() {
      this.precio += 1;
    }
  }
}
</script>

<style>

</style>
```

Ahora vamos a mostrar el precio y pondremos un botón que llame al método `incrementar` en la plantilla del componente.

/vuejs-computed-props-lab/src/components/CmpComputedProps.vue

```
<template>
  <div>
    <p>Precio: {{precio}}</p>
    <button type="button" @click="incrementar">+</button>
  </div>
</template>

<script>
export default {
  data() {
    return {
      precio: 0
    }
  },
  methods: {
    incrementar() {
      this.precio += 1;
    }
  }
}
</script>

<style>

</style>
```

Antes de continuar, vamos a mostrar este componente en el componente `App`.

/vuejs-computed-props-lab/src/App.vue

```
<template>
  <div>
    <CmpComputedProps />
  </div>
</template>

<script>
import CmpComputedProps from './components/CmpComputedProps.vue'

export default {
  components: {
    CmpComputedProps
  }
}
</script>

<style>

</style>
```

El siguiente paso es añadir una computed prop que le añada al precio el simbolo de la moneda.

Para poner la computed prop, vamos a añadir la propiedad `computed` dentro del objeto de la instancia del componente, y dentro de dicho objeto, vamos a añadir un método con el nombre de la computed prop.

Dentro del método de la computed prop, vamos a devolver los datos como queramos verlos, de momento vamos a poner el simbolo de `€` detrás del precio.

/vuejs-computed-props-lab/src/components/CmpComputedProps.vue

```
<template>
  <div>
    <p>Precio: {{precio}}</p>
    <button type="button" @click="incrementar">+</button>
  </div>
</template>

<script>
export default {
  data() {
    return {
      precio: 0
    }
  },
  methods: {
    incrementar() {
      this.precio += 1;
    }
  },
  computed: {
    precioConSimbolo() {
      return this.precio + '€';
    }
  }
}
</script>

<style>

</style>
```

Este método que hemos usado se va a usar como si fuera una propiedad igual que las que tenemos en los datos, por lo que vamos a mostrarla en la plantilla del componente de la misma forma.

/vuejs-computed-props-lab/src/components/CmpComputedProps.vue

```
<template>
  <div>
    <p>Precio: {{precio}}</p>
    <p>Precio con símbolo: {{precioConSimbolo}}</p>
    <button type="button" @click="incrementar">+</button>
  </div>
</template>

<script>
export default {
  data() {
    return {
      precio: 0
    }
  },
  methods: {
    incrementar() {
      this.precio += 1;
    }
  },
  computed: {
    precioConSimbolo() {
      return this.precio + '€';
    }
  }
}
</script>

<style>

</style>
```

Ahora si probamos a darle al botón de incrementar, se va a ir actualizando tanto el precio que habíamos puesto al principio, como el precio con el símbolo que hemos puesto en la computed prop. Y esta computed prop solo se va a actualizar cada vez que se actualice alguno de los datos que se usan dentro del método.

Lo que vamos a hacer es añadir otro dato `simbolo` que seleccionaremos pulsando distintos botones y que irá actualizando también la computed prop que tenemos.

/vuejs-computed-props-lab/src/components/CmpComputedProps.vue

```
<template>
  <div>
    <p>Precio: {{precio}}</p>
    <p>Precio con símbolo: {{precioConSimbolo}}</p>
    <button type="button" @click="incrementar">+</button>
    <button type="button" @click="cambiarSimbolo('€')">€</button>
    <button type="button" @click="cambiarSimbolo('$')">$</button>
  </div>
</template>

<script>
export default {
  data() {
    return {
      precio: 0,
      simbolo: '€'
    }
  },
  methods: {
    incrementar() {
      this.precio += 1;
    },
    cambiarSimbolo(s) {
      this.simbolo = s;
    }
  },
  computed: {
    precioConSimbolo() {
      return this.precio + this.simbolo;
    }
  }
}
</script>

<style>

</style>
```

Ahora vemos que la computed prop va actualizandose cada vez que cambia, el precio o el simbolo.

Capítulo 13. Lab: Desestructurar una Computed Property

En este laboratorio vamos a ver como desestructurar las computed props en los métodos `get` y `set` para controlar lo que tiene que devolver o lo que tiene que asignarle como valor si le igualamos algún dato.

Empezamos por crear el proyecto lanzando el siguiente comando y seleccionamos la configuración por defecto.

```
$ vue create vuejs-computed-props-desestructuracion-lab
```

Una vez creado el proyecto, eliminamos el componente `HelloWorld` y dejamos vacío el componente `App`.

Vamos a crear el componente `CmpDesestructuracionComputedProps` dentro de la carpeta `components`, y vamos a añadir en la propiedad `data`, dos propiedades que mostraremos en la plantilla del componente.

/vuejs-computed-props-desestructuracion-lab/src/components/CmpDesestructuracionComputedProps.vue

```
<template>
  <div>
    <p><strong>Color:</strong> {{color}}</p>
    <p><strong>Sabor:</strong> {{sabor}}</p>
  </div>
</template>

<script>
export default {
  data() {
    return {
      sabor: 'piña',
      color: 'azul'
    }
  }
}
</script>

<style>

</style>
```

Ahora vamos a añadir este componente dentro del componente App para mostrarlo.

/vuejs-computed-props-desestructuracion-lab/src/App.vue

```
<template>
  <div>
    <CmpDesestructuracionComputedProps />
  </div>
</template>

<script>
import CmpDesestructuracionComputedProps from './components/CmpDesestructuracionComputedProps';

export default {
  components: {
    CmpDesestructuracionComputedProps
  }
}
</script>

<style>

</style>
```

Una vez que se muestran los datos en la pantalla del navegador, vamos a crearnos una computed prop que muestre el valor de las dos propiedades.

/vuejs-computed-props-desestructuracion-lab/src/components/CmpDesestructuracionComputedProps.vue

```
<template>
  <div>
    <p><strong>Color:</strong> {{color}}</p>
    <p><strong>Sabor:</strong> {{sabor}}</p>
    <hr>
    <p>{{sugus}}</p>
  </div>
</template>

<script>
export default {
  data() {
    return {
      sabor: 'piña',
      color: 'azul'
    }
  },
  computed: {
    sugus() {
      return `Sugus: ${this.sabor} (${this.color})`;
    }
  }
}
</script>

<style>

</style>
```

Hasta aquí, hemos creado una computed prop como hemos visto en el laboratorio anterior, pero lo que vamos a hacer es desestructurar la computed prop que hemos creado en los método `get` y `set` que nos van a permitir controlar que se devuelve cuando se pide el valor, y como tienen que cambiar los valores de las propiedades cuando se le asigne un valor directamente a la computed prop que hemos creado.

Ahora nuestra computed prop pasará a ser un objeto con dos métodos.

Empezamos por el método `get`, que tendrá lo que queremos mostrar, que es lo mismo que teníamos en la computed prop antes de desestructurarla.

/vuejs-computed-props-desestructuracion-lab/src/components/CmpDesestructuracionComputedProps.vue

```
<template>
  <div>
    <p><strong>Color:</strong> {{color}}</p>
    <p><strong>Sabor:</strong> {{sabor}}</p>
    <hr>
    <p>{{sugus}}</p>
  </div>
</template>

<script>
export default {
  data() {
    return {
      sabor: 'piña',
      color: 'azul'
    }
  },
  computed: {
    sugus: {
      get() {
        return `Sugus: ${this.sabor} (${this.color})`;
      }
    }
  }
}
</script>

<style>

</style>
```

El siguiente paso es coger y añadir el método `set` dentro de la computed prop desestructurada, en la que vamos a añadir la lógica necesaria para obtener los valores de los datos a partir del valor que se le asigna a la computed prop. El valor asignado a la computed prop lo recibimos como parámetro del `set`.

/vuejs-computed-props-desestructuracion-lab/src/components/CmpDesestructuracionComputedProps.vue

```
<template>
  <div>
    <p><strong>Color:</strong> {{color}}</p>
    <p><strong>Sabor:</strong> {{sabor}}</p>
    <hr>
    <p>{{sugus}}</p>
  </div>
</template>

<script>
export default {
  data() {
    return {
      sabor: 'piña',
      color: 'azul'
    }
  },
  computed: {
    sugus: {
      get() {
        return `Sugus: ${this.sabor} (${this.color})`;
      },
      set(val) {
        const colores = ['azul', 'rojo', 'naranja', 'amarillo'];
        const sabores = ['piña', 'fresa', 'naranja', 'limon'];
        if (colores.includes(val)) {
          this.color = val;
          const pos = colores.indexOf(val);
          this.sabor = sabores[pos];
        } else if (sabores.includes(val)) {
          this.sabor = val;
          const pos = sabores.indexOf(val);
          this.color = colores[pos];
        } else {
          this.color = '???';
          this.sabor = '???';
        }
      }
    }
  }
}
</script>

<style>

</style>
```

Y una vez que tenemos la computed prop completamente desestructurada, vamos a añadir un

campo de texto en el que iremos asignando a nuestra computed prop aquello que se escriba en el.

/vuejs-computed-props-desestructuracion-lab/src/components/CmpDesestructuracionComputedProps.vue

```
<template>
  <div>
    <p><strong>Color:</strong> {{color}}</p>
    <p><strong>Sabor:</strong> {{sabor}}</p>
    <hr>
    <p>{{sugus}}</p>
    <div>
      <label for="sugus">Introduce el color/sabor del sugus:</label>
      <input type="text" id="sugus" @input="sugus = $event.target.value">
    </div>
  </div>
</template>

<script>
export default {
  data() {
    return {
      sabor: 'piña',
      color: 'azul'
    }
  },
  computed: {
    sugus: {
      get() {
        return `Sugus: ${this.sabor} (${this.color})`;
      },
      set(val) {
        const colores = ['azul', 'rojo', 'naranja', 'amarillo'];
        const sabores = ['piña', 'fresa', 'naranja', 'limon'];
        if (colores.includes(val)) {
          this.color = val;
          const pos = colores.indexOf(val);
          this.sabor = sabores[pos];
        } else if (sabores.includes(val)) {
          this.sabor = val;
          const pos = sabores.indexOf(val);
          this.color = colores[pos];
        } else {
          this.color = '???';
          this.sabor = '???';
        }
      }
    }
  }
}
</script>
```

```
<style>

</style>
```

Y ahora si probamos a escribir cualquier valor de los que tenemos en los arrays del método `set`, deberían de mostrarse los datos correctamente.

Capítulo 14. Watchers

Otra posible solución al problema que resuelven las *computed properties* es el uso de los **watchers**. Con los *watchers* Vue va a detectar los cambios sobre una propiedad, y va a ejecutar una función en la que se pondrá el código que necesitemos ejecutar. Estas funciones llevan el mismo nombre que la propiedad que tiene que cambiar para que se ejecuten, y se añaden dentro del objeto `watch` de los componentes.

La diferencia entre los *watchers* y las *computed properties* es que estos primeros se usan con el código asíncrono, es decir, para tareas que tienen que ejecutarse una vez que se modifica una propiedad y que no importa el tiempo que tarden en ejecutarse. Mientras que las segundas están pensadas para usarse con código síncrono y que tiene que ocurrir justo después de que cambie la propiedad.

14.1. Lab: Watchers

En este laboratorio vamos a ver como realizar una validación del valor introducido en un campo de texto cada vez que este cambia.

Empezamos por crear el proyecto lanzando el siguiente comando y seleccionamos la configuración por defecto.

```
$ vue create vuejs-watchers-lab
```

Una vez creado el proyecto, eliminamos el componente `HelloWorld` y dejamos vacío el componente `App`.

Vamos a crear el componente `CmpWatchers` dentro de la carpeta `components`, y vamos a añadir en la propiedad `data`, dos propiedades que mostraremos en la plantilla del componente. Además de añadir un campo de texto que se encargará de modificar una de ellas.

/vuejs-watchers-lab/src/components/CmpWatchers.vue

```
<template>
  <div>
    <div>
      <label for="nombre">Nombre:</label>
      <input type="text" v-model="nombre">
    </div>
    <p>Estado: {{estado}}</p>
  </div>
</template>

<script>
export default {
  data() {
    return {
      nombre: '',
      estado: ''
    }
  }
}
</script>

<style>

</style>
```

Ahora vamos a mostrar este componente en nuestra aplicación añadiendolo en nuestro componente `App`.

/vuejs-watchers-lab/src/App.vue

```
<template>
  <div>
    <CmpWatchers />
  </div>
</template>

<script>
import CmpWatchers from './components/CmpWatchers';

export default {
  components: {
    CmpWatchers
  }
}
</script>

<style>

</style>
```

Lo siguiente es añadir el watcher a nuestro componente, por tanto añadimos la propiedad `watch` y dentro de ella una función con el nombre de la propiedad que queremos vigilar, de tal forma que cuando dicha propiedad se modifique se va a ejecutar esta función. Esta función va a recibir como parámetros el nuevo valor y el antiguo valor de la propiedad observada.

/vuejs-watchers-lab/src/components/CmpWatchers.vue

```
<template>
  <div>
    <div>
      <label for="nombre">Nombre:</label>
      <input type="text" v-model="nombre">
    </div>
    <p>Estado: {{estado}}</p>
  </div>
</template>

<script>
export default {
  data() {
    return {
      nombre: '',
      estado: ''
    }
  },
  watch: {
    nombre(newVal, oldVal) {

    }
  }
}
</script>

<style>

</style>
```

Dentro del watcher pondremos aquella tarea que queramos que se ejecute cuando la propiedad con el mismo nombre del watcher cambie. En este caso vamos a realizar una validación para comprobar si el nombre es un nombre válido o no lo es. Vamos a simular la asincronía con la función `setTimeout` y cambiaremos el valor de la propiedad `estado` que se ha definido y que depende de la validación.

/vuejs-watchers-lab/src/components/CmpWatchers.vue

```
<template>
  <div>
    <div>
      <label for="nombre">Nombre:</label>
      <input type="text" v-model="nombre">
    </div>
    <p>Estado: {{estado}}</p>
  </div>
</template>

<script>
export default {
  data() {
    return {
      nombre: '',
      estado: ''
    }
  },
  watch: {
    nombre(newVal, oldValue) {
      const nombresValidos = ['brooklyn', 'hudson', 'bronx', 'goliath', 'lexington', 'broadway', 'demona', 'angela'];
      this.estado = 'Comprobando en la BBDD...';
      setTimeout(() => {
        this.estado = nombresValidos.includes(newVal.toLowerCase()) ? 'Valido' : 'Invalido';
      }, 1500);
    }
  }
}
</script>

<style>

</style>
```

Y con esto cada vez que se cambie el nombre, se comprobará si es un nombre de los que se encuentran en el array de nombres válidos, si es así el estado pasará a ser válido, y si no lo es entonces será invalido.

Capítulo 15. Propiedades

Para poder reutilizar los componentes en distintos sitios tenemos que poder hacerle llegar los datos que se van a usar dentro, y estos datos van a ser las **propiedades**

Para indicar que el componente va a recibir unas propiedades externas, vamos a añadir una propiedad `props`, dentro del componente, a la que le vamos a asignar como valor un array con los nombres de las propiedades que va a recibir o un objeto que nos proporciona más funcionalidades como poder validar los datos que le llegan.

Propiedades en array

```
export default {
  props: ['titulo', 'descripcion', 'finalizada']
}
```

Por cada propiedad dentro del objeto podemos añadir las siguientes propiedades:

- **type**: el tipo de dato o un array con los tipos posibles para el valor que nos van a pasar como propiedad. Podemos poner los siguientes valores:
  - **String**
  - **Boolean**
  - **Number**
  - **Array**
  - **Object**
- **default**: el valor por defecto a usar cuando no le pasan esta propiedad al componente. Cuando el valor por defecto es un array u objeto, tenemos que asignarle una función que devuelva dicho array u objeto.
- **required**: booleano que indica si la propiedad es obligatoria u opcional.
- **validator**: función realizar validaciones adicionales sobre el valor que nos mandan. Esta función recibe como parámetro el valor, y tiene que devolver un booleano para indicar si es un valor válido o no.

Propiedades en objeto

```
export default {
  props: {
    titulo: {
      type: String,
      required: true
    },
    descripcion: {
      type: String,
    },
    estrenada: {
      type: Boolean,
      default: false
    }
  }
}
```

15.1. Lab: Propiedades

En este laboratorio vamos a ver como pasar datos a los componentes y como podemos validarlos.

Empezamos por crear el proyecto lanzando el siguiente comando y seleccionamos la configuración por defecto.

```
$ vue create vuejs-propiedades-lab
```

Una vez creado el proyecto, eliminamos el componente `HelloWorld` y dejamos vacío el componente `App`.

Ahora vamos a crear un componente `CmpPersona` dentro de la carpeta `components` en el que vamos a recibir las propiedades y las vamos a mostrar. Por lo tanto empezamos añadiendo la propiedad `props` dentro del componente.

/vuejs-propiedades-lab/src/components/CmpPersona.vue

```
<template>

</template>

<script>
export default {
  props: {

  }
}
</script>

<style>

</style>
```

Este componente va a recibir como propiedades un nombre que tiene que ser obligatorio, y tiene que cumplir la validación que comprueba que sea un **Stark**.

/vuejs-propiedades-lab/src/components/CmpPersona.vue

```
<template>
  <div>
    <p>Nombre: {{nombre}}</p>
  </div>
</template>

<script>
export default {
  props: {
    nombre: {
      type: String,
      required: true,
      validator(value) {
        return ['robb', 'arya', 'rickon', 'sansa', 'bran', 'tony'].includes(value.toLowerCase());
      }
    }
  }
}
</script>

<style>

</style>
```

Una vez que hemos añadido la propiedad en el componente, vamos a importarlo en el componente raíz de la aplicación, en el cual lo mostraremos sin llegar a pasarle ningún valor para el nombre.

/vuejs-propiedades-lab/src/App.vue

```
<template>
  <div>
    <CmpPersona />
  </div>
</template>

<script>
import CmpPersona from './components/CmpPersona'

export default {
  components: {
    CmpPersona
  }
}
</script>

<style>

</style>
```

Si miramos la consola del navegador, en ella podemos ver que hay un error en el que nos dice que el componente no está recibiendo la propiedad nombre.

Vamos a corregirlo añadiendo la propiedad nombre al mostrar el componente, pero le vamos a dar como valor un nombre que no espera recibir.

/vuejs-propiedades-lab/src/App.vue

```
<template>
  <div>
    <CmpPersona nombre="Ramsay" />
  </div>
</template>

<script>
import CmpPersona from './components/CmpPersona'

export default {
  components: {
    CmpPersona
  }
}
</script>

<style>

</style>
```

Ahora en la consola nos sale otro error, esta vez distinto al anterior, y en el que nos dice que el valor que le hemos dado a la propiedad no cumple la validación personalizada que hemos puesto.

Una vez que hemos visto estos errores, ahora vamos a darle un valor correcto para que dejen de salir estos mensajes por la consola.

/vuejs-propiedades-lab/src/App.vue

```
<template>
  <div>
    <CmpPersona nombre="Arya" />
  </div>
</template>

<script>
import CmpPersona from './components/CmpPersona'

export default {
  components: {
    CmpPersona
  }
}
</script>

<style>

</style>
```

Una vez puesto un nombre correcto, ya podemos ver que se muestra correctamente y no salen mensajes de error por la consola.

Ahora vamos a añadir otra propiedad, esta vez va a ser un booleano con un valor por defecto a `false`.

/vuejs-propiedades-lab/src/components/CmpPersona.vue

```
<template>
  <div>
    <p :style="{textDecoration: tachado}">Nombre: {{nombre}}</p>
  </div>
</template>

<script>
export default {
  props: {
    nombre: {
      type: String,
      required: true,
      validator(value) {
        return ['robb', 'arya', 'rickon', 'sansa', 'bran', 'tony'].includes(value.toLowerCase());
      }
    },
    estaMuerto: {
      type: Boolean,
      default: false
    }
  },
  computed: {
    tachado() {
      return this.estaMuerto ? 'line-through' : 'none';
    }
  }
}
</script>

<style>

</style>
```

Si miramos el navegador, el componente anterior no sale tachado porque está aplicando el valor por defecto de la nueva propiedad.

Vamos a ir al componente `App` y vamos a añadir otro componente `CmpPersona` pasandole un nombre válido y esta nueva propiedad como `true`.

Al ser una propiedad booleana con un valor por defecto a `false`, no hace falta asignarle el valor, sino que con poner el nombre de la propiedad se toma como `true`, y si no se pone, entonces se toma como `false`.

/vuejs-propiedades-lab/src/App.vue

```
<template>
  <div>
    <CmpPersona nombre="Arya" />
    <CmpPersona nombre="Robb" estaMuerto />
  </div>
</template>

<script>
import CmpPersona from './components/CmpPersona'

export default {
  components: {
    CmpPersona
  }
}
</script>

<style>

</style>
```

Ahora en el navegador, este segundo nombre debería de aparecer tachado.

Capítulo 16. Comunicación entre componentes

Hasta ahora no hemos visto como comunicar los componentes entre si. Podemos mandar propiedades desde los componentes superiores hacía los componentes inferiores, pero estas propiedades no deberían ser modificadas por los componentes que las reciben ya que entonces el componente que había pasado inicialmente dicha propiedad no se va a enterar del cambio del valor.

Una vez que el componente que pasa la propiedad la modifica, todos aquellos componentes que la estaban recibiendo, van a volver a recibirla, pero esta vez con el valor actualizado.

Por tanto, si no podemos modificar las propiedades, o mejor dicho, no debemos hacerlo, ¿cómo podemos avisar al componente que la ha pasado inicialmente para que cambie su valor?.

Pues tenemos varias opciones que vamos a ver a continuación.

16.1. Emitir eventos personalizados

Para avisar a los componentes superiores de que tienen que cambiar el valor de la propiedad podemos emitir eventos personalizados en los que vamos a mandar el nuevo valor de esta.

Para ello Vue nos da acceso al método `this.$emit(nombreEvento, valorAEmitir)`.

Una vez que se emite el evento, necesitaremos detectarlo en el componente superior al igual que hemos detectado cualquier evento hasta ahora, usando la directiva `v-on`.

16.2. Patrón EventBus

En la anterior sección hemos comentado como comunicar distintos componentes, pero en el caso de que estos componentes que tienen que recibir los nuevos valores estén muy separados en el árbol de componentes, se vuelve muy dificil de manejar todos los eventos y propiedades, y para estos casos vamos a usar el patrón **EventBus**.

Ahora lo que vamos a hacer es crear una nueva instancia de Vue que vamos a exportar para comunicar componentes lejanos.

En aquellos componentes que tienen que enviar los datos vamos a importar la nueva instancia y emitiremos el vento usando su método `$emit` como hemos hecho en la sección anterior.

Por último, para escuchar el evento en los componentes que tienen que recibir el nuevo valor vamos a usar el método `this.$on(nombreEvento, funcionAEjecutar)` para poner un listener y que escuche el evento que hemos emitido.

Al igual que tenemos el método `$on` para poner listeners, también tenemos el método `$off` para quitarlos cuando dejen de ser necesarios, por ejemplo porque el componente se va a destruir.

Al método `$off` le vamos a pasar como parámetro el nombre del evento del cual queremos eliminar

los listeners.

16.2.1. Lab: Comunicación entre componentes

En este laboratorio vamos a ver como comunicar varios componentes usando los eventos personalizados.

Empezamos por crear el proyecto lanzando el siguiente comando y seleccionamos la configuración por defecto.

```
$ vue create vuejs-comunicacion-entre-componentes-lab
```

Una vez creado el proyecto, eliminamos el componente `HelloWorld` y dejamos vacío el componente `App`.

Vamos a empezar por crear los componentes que necesitaremos para este laboratorio.

El primer componente que vamos a crear es `CmpLeia` en el que vamos a recibir dos propiedades, su nombre y el nombre de su hermano, y las vamos a mostrar.

/vuejs-comunicacion-entre-componentes-lab/src/components/CmpLeia.vue

```
<template>
  <div>
    <p>Me llamo {{nombre}}</p>
    <p>Mi hermano se llama {{nombreHermano}}</p>
  </div>
</template>

<script>
export default {
  props: {
    nombre: {
      type: String,
      required: true
    },
    nombreHermano: {
      type: String,
      required: true
    }
  }
}
</script>

<style>

</style>
```

El siguiente componente que vamos a crear es `CmpLuke` que también va a recibir dos propiedades y

las va a mostrar.

/vuejs-comunicacion-entre-componentes-lab/src/components/CmpLuke.vue

```
<template>
  <div>
    <p>Me llamo {{nombre}}</p>
    <p>Mi hermana se llama {{nombreHermana}}</p>
  </div>
</template>

<script>
export default {
  props: {
    nombre: {
      type: String,
      required: true
    },
    nombreHermana: {
      type: String,
      required: true
    }
  }
}
</script>

<style>

</style>
```

Ahora vamos a crear el componente `CmpAnakin` que se encarga de mostrar los dos anteriores y pasarles las propiedades que necesitan. Además mostrará los nombres que va a pasar como propiedades a los componentes anteriores.

/vuejs-comunicacion-entre-componentes-lab/src/components/CmpAnakin.vue

```
<template>
  <div>
    <p>Mi hija se llama {{nombreHija}}</p>
    <p>Mi hijo se llama {{nombreHijo}}</p>
    <CmpLeia :nombre="nombreHija" :nombreHermano="nombreHijo" />
    <CmpLuke :nombre="nombreHijo" :nombreHermana="nombreHija" />
  </div>
</template>

<script>
import CmpLeia from './CmpLeia';
import CmpLuke from './CmpLuke';

export default {
  components: {
    CmpLeia,
    CmpLuke
  },
  data() {
    return {
      nombreHija: 'Leia',
      nombreHijo: 'Luke'
    }
  }
}
</script>

<style>

</style>
```

Y por último vamos a mostrar este último componente en el componente `App`.

/vuejs-comunicacion-entre-componentes-lab/src/App.vue

```
<template>
  <CmpAnakin />
</template>

<script>
import CmpAnakin from './components/CmpAnakin';

export default {
  components: {
    CmpAnakin
  }
}
</script>

<style>

</style>
```

Ahora ya deberíamos de poder ver nuestros componentes en el navegador.

Lo siguiente que vamos a hacer es poner un input en el componente `CmpLuke` y ver como hacerle llegar el nuevo nombre al componente `CmpAnakin` para que modifique el nombre y se actualicen los componentes que lo estaban recibiendo.

Vamos a empezar poniendo dicho input en el componente, y añadiremos un método en el cual vamos a emitir nuestro evento cuando se ejecute.

/vuejs-comunicacion-entre-componentes-lab/src/components/CmpLuke.vue

```
<template>
  <div>
    <input type="text" :placeholder="nombre" v-model="nuevoNombre">
    <button type="button" @click="cambiarNombre">Cambiar nombre</button>
    <p>Me llamo {{nombre}}</p>
    <p>Mi hermana se llama {{nombreHermana}}</p>
  </div>
</template>

<script>
export default {
  props: {
    nombre: {
      type: String,
      required: true
    },
    nombreHermana: {
      type: String,
      required: true
    }
  },
  data() {
    return {
      nuevoNombre: ''
    }
  },
  methods: {
    cambiarNombre() {

    }
  }
}
</script>

<style>

</style>
```

Para emitir el evento vamos a usar el método `$emit` al que le vamos a pasar el nombre del evento que queremos emitir, y el dato a enviar.

/vuejs-comunicacion-entre-componentes-lab/src/components/CmpLuke.vue

```
<template>
  <div>
    <input type="text" :placeholder="nombre" v-model="nuevoNombre">
    <button type="button" @click="cambiarNombre">Cambiar nombre</button>
    <p>Me llamo {{nombre}}</p>
    <p>Mi hermana se llama {{nombreHermana}}</p>
  </div>
</template>

<script>
export default {
  props: {
    nombre: {
      type: String,
      required: true
    },
    nombreHermana: {
      type: String,
      required: true
    }
  },
  data() {
    return {
      nuevoNombre: ''
    }
  },
  methods: {
    cambiarNombre() {
      this.$emit('nombreCambiado', this.nuevoNombre)
    }
  }
}
</script>

<style>

</style>
```

Ahora tenemos que detectar este evento sobre la etiqueta de este componente para poder ejecutar un método que va a recibir el nuevo nombre y que nos va a permitir cambiarlo en el componente de `CmpAnakin`.

/vuejs-comunicacion-entre-componentes-lab/src/components/CmpAnakin.vue

```
<template>
  <div>
    <p>Mi hija se llama {{nombreHija}}</p>
    <p>Mi hijo se llama {{nombreHijo}}</p>
    <CmpLeia :nombre="nombreHija" :nombreHermano="nombreHijo" />
    <CmpLuke :nombre="nombreHijo" :nombreHermana="nombreHija" @nombreCambiado="cambiarNombreHijo" />
  </div>
</template>

<script>
import CmpLeia from './CmpLeia';
import CmpLuke from './CmpLuke';

export default {
  components: {
    CmpLeia,
    CmpLuke
  },
  data() {
    return {
      nombreHija: 'Leia',
      nombreHijo: 'Luke'
    }
  },
  methods: {
    cambiarNombreHijo(nuevoNombre) {
      this.nombreHijo = nuevoNombre;
    }
  }
}
</script>

<style>

</style>
```

Y al cambiar el nombre en el componente superior, todos aquellos que recibian ese dato como propiedad también van a recibir el nombre actualizado.

16.2.2. Lab: Comunicación entre componentes con el EventBus

En este laboratorio vamos a ver como usar el patrón EventBus para comunicar varios componentes.

Empezamos por crear el proyecto lanzando el siguiente comando y seleccionamos la configuración por defecto.

```
$ vue create vuejs-comunicacion-entre-componentes-eventbus-lab
```

Una vez creado el proyecto, eliminamos el componente `HelloWorld` y dejamos vacío el componente `App`.

Vamos a empezar por crear los componentes que necesitaremos para este laboratorio.

El primer componente que vamos a crear es `CmpLeia` en el que vamos a recibir dos propiedades, su nombre y el nombre de su hermano, y las vamos a mostrar.

/vuejs-comunicacion-entre-componentes-eventbus-lab/src/components/CmpLeia.vue

```
<template>
  <div>
    <p>Me llamo {{nombre}}</p>
    <p>Mi hermano se llama {{nombreHermano}}</p>
  </div>
</template>

<script>
export default {
  props: {
    nombre: {
      type: String,
      required: true
    },
    nombreHermano: {
      type: String,
      required: true
    }
  }
}
</script>

<style>

</style>
```

El siguiente componente que vamos a crear es `CmpLuke` que también va a recibir dos propiedades y las va a mostrar.

/vuejs-comunicacion-entre-componentes-eventbus-lab/src/components/CmpLuke.vue

```
<template>
  <div>
    <p>Me llamo {{nombre}}</p>
    <p>Mi hermana se llama {{nombreHermana}}</p>
  </div>
</template>

<script>
export default {
  props: {
    nombre: {
      type: String,
      required: true
    },
    nombreHermana: {
      type: String,
      required: true
    }
  }
}
</script>

<style>

</style>
```

Ahora vamos a crear el componente `CmpAnakin` que se encarga de mostrar los dos anteriores y pasarles las propiedades que necesitan. Además mostrará los nombres que va a pasar como propiedades a los componentes anteriores.

/vuejs-comunicacion-entre-componentes-eventbus-lab/src/components/CmpAnakin.vue

```
<template>
  <div>
    <p>Mi hija se llama {{nombreHija}}</p>
    <p>Mi hijo se llama {{nombreHijo}}</p>
    <CmpLeia :nombre="nombreHija" :nombreHermano="nombreHijo" />
    <CmpLuke :nombre="nombreHijo" :nombreHermana="nombreHija" />
  </div>
</template>

<script>
import CmpLeia from './CmpLeia';
import CmpLuke from './CmpLuke';

export default {
  components: {
    CmpLeia,
    CmpLuke
  },
  data() {
    return {
      nombreHija: 'Leia',
      nombreHijo: 'Luke'
    }
  }
}
</script>

<style>

</style>
```

Y por último vamos a mostrar este último componente en el componente `App`.

/vuejs-comunicacion-entre-componentes-eventbus-lab/src/App.vue

```
<template>
  <CmpAnakin />
</template>

<script>
import CmpAnakin from './components/CmpAnakin';

export default {
  components: {
    CmpAnakin
  }
}
</script>

<style>

</style>
```

Ahora ya deberíamos de poder ver nuestros componentes en el navegador.

Lo siguiente que vamos a hacer es crear el EventBus que es una nueva instancia de Vue que hay que exportar para después poder usarla en otros componentes. Esta instancia la vamos a poner en el archivo `main.js` antes de la instancia de la aplicación.

/vuejs-comunicacion-entre-componentes-eventbus-lab/src/main.js

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

Vue.config.productionTip = false

export const EventBus = new Vue({});

new Vue({
  render: h => h(App),
}).$mount('#app')
```

Una vez que tenemos la instancia, vamos a ir al componente `CmpLeia` el cual va a avisar al componente `CmpAnakin` que ha cambiado su nombre, pero esta vez usando el EventBus que acabamos de crear.

En el primer componente vamos a importar la nueva instancia de Vue, y vamos a añadir un campo de texto con un botón que al pulsarlo llamará a un método.

/vuejs-comunicacion-entre-componentes-eventbus-lab/src/components/CmpLeia.vue

```
<template>
  <div>
    <input type="text" :placeholder="nombre" v-model="nuevoNombre">
    <button type="button" @click="cambiarNombre">Cambiar nombre</button>
    <p>Me llamo {{nombre}}</p>
    <p>Mi hermano se llama {{nombreHermano}}</p>
  </div>
</template>

<script>
import { EventBus } from '../main';

export default {
  props: {
    nombre: {
      type: String,
      required: true
    },
    nombreHermano: {
      type: String,
      required: true
    }
  },
  data() {
    return {
      nuevoNombre: ''
    }
  },
  methods: {
    cambiarNombre() {

    }
  }
}
</script>

<style>

</style>
```

En el método que hemos añadido, vamos a coger el EventBus y vamos a usar su método `$emit` para emitir un evento propio y así poder enviar el nuevo nombre.

/vuejs-comunicacion-entre-componentes-eventbus-lab/src/components/CmpLeia.vue

```
<template>
  <div>
    <input type="text" :placeholder="nombre" v-model="nuevoNombre">
    <button type="button" @click="cambiarNombre">Cambiar nombre</button>
    <p>Me llamo {{nombre}}</p>
    <p>Mi hermano se llama {{nombreHermano}}</p>
  </div>
</template>

<script>
import { EventBus } from '../main';

export default {
  props: {
    nombre: {
      type: String,
      required: true
    },
    nombreHermano: {
      type: String,
      required: true
    }
  },
  data() {
    return {
      nuevoNombre: ''
    }
  },
  methods: {
    cambiarNombre() {
      EventBus.$emit('nombreCambiado', this.nuevoNombre)
    }
  }
}
</script>

<style>

</style>
```

Ahora vamos a ir al componente `CmpAnakin` para poner un listener sobre el evento que se va a emitir desde el componente anterior. Para ello tenemos que importar la instancia del EventBus dentro del componente.

/vuejs-comunicacion-entre-componentes-eventbus-lab/src/components/CmpAnakin.vue

```
<template>
  <div>
    <p>Mi hija se llama {{nombreHija}}</p>
    <p>Mi hijo se llama {{nombreHijo}}</p>
    <CmpLeia :nombre="nombreHija" :nombreHermano="nombreHijo" />
    <CmpLuke :nombre="nombreHijo" :nombreHermana="nombreHija" />
  </div>
</template>

<script>
import { EventBus } from '../main';
import CmpLeia from './CmpLeia';
import CmpLuke from './CmpLuke';

export default {
  components: {
    CmpLeia,
    CmpLuke
  },
  data() {
    return {
      nombreHija: 'Leia',
      nombreHijo: 'Luke'
    }
  }
}
</script>

<style>

</style>
```

Una vez importada, vamos a añadir un listener sobre el evento, y esta vez, como no se está emitiendo a través de la propia instancia de la aplicación, no podremos usar la directiva `v-on` sobre la etiqueta del componente que se encarga de emitir el evento.

Lo que si podemos hacer es poner un listener a través del EventBus usando `$on(nombreEvento, funcionAEjecutar)`. Este listener lo tenemos que poner cuando se crea el componente, por lo que meteremos esta instrucción en el método del ciclo de vida `created`.

/vuejs-comunicacion-entre-componentes-eventbus-lab/src/components/CmpAnakin.vue

```
<template>
  <div>
    <p>Mi hija se llama {{nombreHija}}</p>
    <p>Mi hijo se llama {{nombreHijo}}</p>
    <CmpLeia :nombre="nombreHija" :nombreHermano="nombreHijo" />
    <CmpLuke :nombre="nombreHijo" :nombreHermana="nombreHija" />
  </div>
</template>

<script>
import { EventBus } from '../main';
import CmpLeia from './CmpLeia';
import CmpLuke from './CmpLuke';

export default {
  components: {
    CmpLeia,
    CmpLuke
  },
  data() {
    return {
      nombreHija: 'Leia',
      nombreHijo: 'Luke'
    }
  },
  created() {
    EventBus.$on('nombreCambiado', (nuevoNombre) => {
      this.nombreHija = nuevoNombre;
    })
  }
}
</script>

<style>

</style>
```

Ahora si probamos a cambiar el nombre, deberían de obtener la actualización todos los componentes.

Por último, al igual que hemos añadido un listener sobre el evento, vamos a quitarlo antes de que el componente se elimine. Para ello usaremos la función `$off(nombreEvento)` en el método del ciclo de vida `beforeDestroy`.

/vuejs-comunicacion-entre-componentes-eventbus-lab/src/components/CmpAnakin.vue

```
<template>
  <div>
    <p>Mi hija se llama {{nombreHija}}</p>
    <p>Mi hijo se llama {{nombreHijo}}</p>
    <CmpLeia :nombre="nombreHija" :nombreHermano="nombreHijo" />
    <CmpLuke :nombre="nombreHijo" :nombreHermana="nombreHija" />
  </div>
</template>

<script>
import { EventBus } from '../main';
import CmpLeia from './CmpLeia';
import CmpLuke from './CmpLuke';

export default {
  components: {
    CmpLeia,
    CmpLuke
  },
  data() {
    return {
      nombreHija: 'Leia',
      nombreHijo: 'Luke'
    }
  },
  created() {
    EventBus.$on('nombreCambiado', (nuevoNombre) => {
      this.nombreHija = nuevoNombre;
    })
  },
  beforeDestroy() {
    EventBus.$off('nombreCambiado')
  }
}
</script>

<style>

</style>
```

Y ahora ya tendríamos el laboratorio terminado.

Capítulo 17. Slots

Hemos visto como podemos reutilizar componentes pasandoles propiedades con algunos valores, pero hay casos en los que queremos pasarles trozos de HTML a los componentes para hacerlos reusables y pasar este código HTML como propiedades a los componentes quedaría un poco feo.

Las etiquetas **slot** nos van a permitir indicar dentro de un componente donde se tiene que poner el código HTML que se le ha pasado entre las etiquetas cuando se ha usado en el componente superior para mostrarlo.

17.1. Múltiples slots

Si la estructura en que se muestra tiene la posibilidad de cambiar, por ejemplo, que una imagen aparezca antes que el título, entonces tenemos que diferenciarlo añadiendo más etiquetas `slot`. En este caso cada etiqueta `slot` tiene que llevar un atributo `name` que va a servirnos para indicar que parte del HTML hay que inyectar en cada `slot`.

Y en la parte del código HTML que se va a inyectar en cada `slot` hay que añadir un atributo `slot` con el valor que hemos dado al `name` en el componente que queremos reutilizar.

17.2. Slot por defecto

Es posible que queramos dejar algo de contenido por defecto cuando no se lo pasamos a un `slot`, y para ello hay que poner lo que queramos que salga por defecto entre las etiquetas `slot`.

17.3. Lab: Slots

En este laboratorio vamos a ver como utilizar las etiquetas `slot` para poder pasarle a un componente distintas estructuras de HTML para mostrar dentro de el.

Empezamos por crear el proyecto lanzando el siguiente comando y seleccionamos la configuración por defecto.

```
$ vue create vuejs-slots-lab
```

Una vez creado el proyecto, eliminamos el componente `HelloWorld` y dejamos vacío el componente `App`.

Vamos a crear los componentes `CmpSlots` y `Card` dentro de la carpeta `components`.

Vamos a empezar por rellenar el componente `Card`, en el que de momento pondremos una etiqueta `slot` en la plantilla. Además añadiremos unos estilos para el contenedor de esta etiqueta.

/vuejs-slots-lab/src/components/Card.vue

```
<template>
  <div class="card-container">
    <slot></slot>
  </div>
</template>

<script>
export default {

}
</script>

<style scoped>
  .card-container {
    width: 300px;
    border: 1px solid lightgray;
    box-shadow: 20px 20px 20px darkgray;
    text-align: center;
    border-radius: 5px;
  }

  .card-container img {
    width: 70%;
  }
</style>
```

Una vez creado el componente `Card`, vamos al componente `CmpSlots` en el que lo mostraremos, y pondremos entre las etiquetas `Card` el contenido que queremos que se muestre dentro del componente.

/vuejs-slots-lab/src/components/CmpSlots.vue

```
<template>
  <div>
    <Card>
      <h3>Título</h3>
      <img src="https://www.stickpng.com/assets/images/58482acecef1014c0b5e4a1e.png" alt="Logo de Vue">
      <p>Lorem ipsum dolor, sit amet consectetur adipisicing elit. Unde eaque molestias omnis inventore architecto, culpa sapiente officiis earum id animi! Quia, itaque non. Ab at aliquid hic magnam ullam esse!</p>
    </Card>
  </div>
</template>

<script>
import Card from './Card';

export default {
  components: {
    Card
  }
}
</script>

<style>

</style>
```

Lo siguiente es coger este componente y ponerlo dentro del componente `App` para mostrarlo en el navegador.

/vuejs-slots-lab/src/App.vue

```
<template>
  <div>
    <CmpSlots />
  </div>
</template>

<script>
import CmpSlots from './components/CmpSlots';

export default {
  components: {
    CmpSlots
  }
}
</script>

<style>

</style>
```

Ahora ya deberíamos de poder ver el componente Card, con los datos que le hemos pasado desde el componente superior.

Con estos slots, ahora podríamos coger la estructura de etiquetas que hemos pasado al componente Card, y cambiarla de orden o el tipo de etiquetas.

/vuejs-slots-lab/src/components/CmpSlots.vue

```
<template>
  <div>
    <Card>
      <img src="https://www.stickpng.com/assets/images/58482acecef1014c0b5e4a1e.png" alt="Logo de Vue">
      <h2>Título</h2>
    </Card>
  </div>
</template>

<script>
import Card from './Card';

export default {
  components: {
    Card
  }
}
</script>

<style>

</style>
```

17.3.1. Añadiendo más slots

Es posible que dentro del componente donde tenemos la etiqueta `slot` queramos dejar contenido que tiene que aparecer siempre que se vaya a mostrar este componente, como en este caso una división horizontal del contenido. Para ello, podemos crear distintas secciones añadiendo más slots. Lo único a tener en cuenta es que para luego poder indicar que contenido tiene que ir en cada sección, tenemos que añadirles a estas etiquetas un atributo `name` con un valor.

/vuejs-slots-lab/src/components/Card.vue

```
<template>
  <div class="card-container">
    <slot name="header"></slot>
    <slot name="body"></slot>
    <hr>
    <slot name="footer"></slot>
  </div>
</template>

<script>
export default {

}
</script>

<style scoped>
  .card-container {
    width: 300px;
    border: 1px solid lightgray;
    box-shadow: 20px 20px 20px darkgray;
    text-align: center;
    border-radius: 5px;
  }

  .card-container img {
    width: 70%;
  }
</style>
```

Una vez que hemos modificado el componente, ahora tendremos que ir al componente superior donde le vamos a indicar que partes del contenido que le hemos pasado a `Card` tienen que ir en que sección, y para ello solo tenemos que añadir el atributo `slot` y darle como valor el mismo que habíamos puesto en el atributo `name` de cada sección.

/vuejs-slots-lab/src/components/CmpSlots.vue

```
<template>
  <div>
    <Card>
      <h2 slot="header">Título</h2>
      <img slot="body" src="https://www.stickpng.com/assets/images/58482acecef1014c0b5e4a1e.png" alt="Logo de Vue">
      <p slot="footer">Lorem ipsum dolor, sit amet consectetur adipisicing elit. Unde eaque molestias omnis inventore
architecto, culpa sapiente officiis earum id animi! Quia, itaque non. Ab at aliquid hic magnam ullam esse!</p>
    </Card>
  </div>
</template>

<script>
import Card from './Card';

export default {
  components: {
    Card
  }
}
</script>

<style>

</style>
```

17.3.2. Sección con contenido por defecto

Estas etiquetas también nos permiten dejar un contenido por defecto que se va a mostrar cuando a una de las secciones no se le pase nada de contenido. Lo único que hay que hacer es, pasarle el contenido a mostrar por defecto entre las etiquetas `slot` del componente.

/vuejs-slots-lab/src/components/Card.vue

```
<template>
  <div class="card-container">
    <slot name="header"></slot>
    <slot name="body"></slot>
    <hr>
    <slot name="footer">
      <p>Footer de Card</p>
    </slot>
  </div>
</template>

<script>
export default {

}
</script>

<style scoped>
  .card-container {
    width: 300px;
    border: 1px solid lightgray;
    box-shadow: 20px 20px 20px darkgray;
    text-align: center;
    border-radius: 5px;
  }

  .card-container img {
    width: 70%;
  }
</style>
```

Una vez puesto el contenido por defecto, vamos a quitar el contenido para dicha sección del componente superior.

/vuejs-slots-lab/src/components/CmpSlots.vue

```
<template>
  <div>
    <Card>
      <h2 slot="header">Título</h2>
      <img slot="body" src="https://www.stickpng.com/assets/images/58482acecef1014c0b5e4a1e.png" alt="Logo de Vue">
    </Card>
  </div>
</template>

<script>
import Card from './Card';

export default {
  components: {
    Card
  }
}
</script>

<style>

</style>
```

Tras realizar este cambio, deberíamos de poder ver en el navegador el contenido que hemos puesto por defecto para la sección para cuando no se le pasa contenido a dicha sección.

Capítulo 18. Template

En Vue hay ocasiones en las que nos podemos encontrar con que queremos añadir una misma directiva a varias etiquetas por ejemplo para ocultarlas todas ellas de una vez, pero no queremos envolver todas estas etiquetas con otra etiqueta más (como un `div`) porque nuestro código se puede llegar a llenar de etiquetas que no son necesarias. En estos casos, en lugar de usar una etiqueta de las típicas, podemos usar la etiqueta `template` para envolverlas, y esta etiqueta no se va a mostrar a la hora de renderizarse la aplicación.

Esta etiqueta no puede usarse como nodo raíz de un componente.

18.1. Lab: Template

En este laboratorio vamos a ver como usar la etiqueta `template` para añadir o eliminar varias etiquetas HTML dependiendo de una condición usando la directiva `v-if`.

Empezamos por crear el proyecto lanzando el siguiente comando y seleccionamos la configuración por defecto.

```
$ vue create vuejs-template-lab
```

Una vez creado el proyecto, eliminamos el componente `HelloWorld` y dejamos vacío el componente `App`.

Vamos a crear el componente `CmpTemplate` dentro de la carpeta `components`. En dicho componente, vamos a añadir una serie de datos, que mostraremos en la plantilla del componente.

/vuejs-template-lab/src/components/CmpTemplate.vue

```
<template>
  <div>
    <button type="button">{{mostrarNoticia ? 'Ocultar' : 'Mostrar'}} noticia</button>
    <h1>{{noticia.titulo}}</h1>
    <p>{{noticia.contenido}}</p>
  </div>
</template>

<script>
export default {
  data() {
    return {
      mostrarNoticia: true,
      noticia: {
        titulo: 'Sube la venta de pitos y flautas',
        contenido: 'Lorem ipsum dolor sit amet, consectetur adipiscing elit. Maecenas cursus lacinia sagittis. Nullam
mattis sem eget mauris congue, sit amet venenatis nibh porta. Nullam a leo at lorem ullamcorper tempor vel et nisl.
Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Nullam pharetra ex iaculis
tellus sollicitudin, eu sollicitudin libero sollicitudin.'
      }
    }
  }
}
</script>

<style>

</style>
```

Vamos a añadir este componente en el `App` para mostrarlo.

/vuejs-template-lab/src/App.vue

```
<template>
  <div>
    <CmpTemplate />
  </div>
</template>

<script>
import CmpTemplate from './components/CmpTemplate';

export default {
  components: {
    CmpTemplate
  }
}
</script>

<style>

</style>
```

Una vez que se muestra, vamos a añadir la funcionalidad al botón para ocultar o mostrar los

elementos HTML que tenemos, además tenemos que añadir la directiva sobre estos elementos para que funcione correctamente.

/vuejs-template-lab/src/components/CmpTemplate.vue

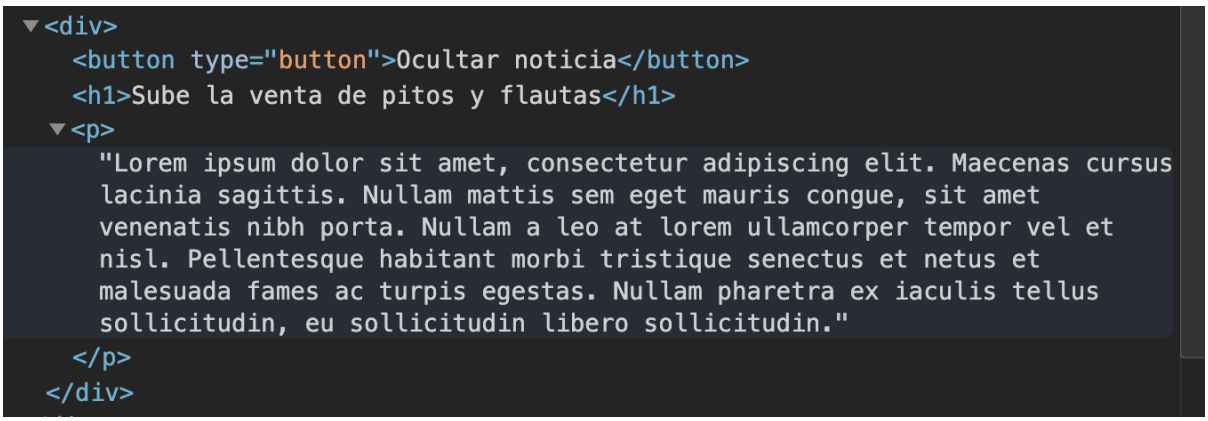
```
<template>
  <div>
    <button type="button" @click="toggleNoticia">{{mostrarNoticia ? 'Ocultar' : 'Mostrar'}} noticia</button>
    <h1 v-if="mostrarNoticia">{{noticia.titulo}}</h1>
    <p v-if="mostrarNoticia">{{noticia.contenido}}</p>
  </div>
</template>

<script>
export default {
  data() {
    return {
      mostrarNoticia: true,
      noticia: {
        titulo: 'Sube la venta de pitos y flautas',
        contenido: 'Lorem ipsum dolor sit amet, consectetur adipiscing elit. Maecenas cursus lacinia sagittis. Nullam
mattis sem eget mauris congue, sit amet venenatis nibh porta. Nullam a leo at lorem ullamcorper tempor vel et nisl.
Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Nullam pharetra ex iaculis
tellus sollicitudin, eu sollicitudin libero sollicitudin.'
      }
    }
  },
  methods: {
    toggleNoticia() {
      this.mostrarNoticia = !this.mostrarNoticia
    }
  }
}
</script>

<style>

</style>
```

El problema que nos encontramos es que tenemos que añadir la misma directiva en varios elementos de la plantilla del componente, pero en lugar de hacerlo de esa forma, podemos envolver estos elementos con la etiqueta `template` y añadirle la directiva a esta etiqueta para que se añada o elimine el contenido dependiendo de la condición.

/vuejs-template-lab/src/components/CmpTemplate.vue

```
<template>
  <div>
    <button type="button" @click="toggleNoticia">{{mostrarNoticia ? 'Ocultar' : 'Mostrar'}} noticia</button>
    <template v-if="mostrarNoticia">
      <h1>{{noticia.titulo}}</h1>
      <p>{{noticia.contenido}}</p>
    </template>
  </div>
</template>

<script>
export default {
  data() {
    return {
      mostrarNoticia: true,
      noticia: {
        titulo: 'Sube la venta de pitos y flautas',
        contenido: 'Lorem ipsum dolor sit amet, consectetur adipiscing elit. Maecenas cursus lacinia sagittis. Nullam
mattis sem eget mauris congue, sit amet venenatis nibh porta. Nullam a leo at lorem ullamcorper tempor vel et nisl.
Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Nullam pharetra ex iaculis
tellus sollicitudin, eu sollicitudin libero sollicitudin.'
      }
    }
  },
  methods: {
    toggleNoticia() {
      this.mostrarNoticia = !this.mostrarNoticia
    }
  }
}
</script>

<style>

</style>
```

Al usar la etiqueta `template` en lugar de otra como el `div`, si inspeccionamos nuestra aplicación con las herramientas de desarrollador del navegador, podemos ver que no esta etiqueta no ha afectado al DOM.



Capítulo 19. Filtros

Los **filtros** son funciones que se encargan de transformar un valor a la hora de mostrarlo en la vista, pero sin llegar a modificar el valor de la propiedad.

En Vue, no hay filtros predefinidos, sino que los tenemos que crear nosotros, y lo podemos hacer creandolos de forma local (aquellos filtros que solo se vayan a usar en un componente), o de forma global (aquellos filtros que se puedan usar en distintos componentes de la aplicación).

Al igual que con los componentes, aquí podemos crear filtros:

- locales: solo los podemos usar en los componentes donde los hemos definido.
- globales: podemos usarlos en toda la aplicación sin necesidad de importar el filtro.

19.1. Lab: Filtros locales

En este laboratorio vamos a ver como crear un filtro local a un componente.

Empezamos por crear el proyecto lanzando el siguiente comando y seleccionamos la configuración por defecto.

```
$ vue create vuejs-filtros-locales-lab
```

Una vez creado el proyecto, eliminamos el componente `HelloWorld` y dejamos vacío el componente `App`.

Vamos a crear un componente `CmpFiltroLocal` dentro de la carpeta `components`, en el que vamos a crear nuestro filtro.

Lo primero vamos a añadir dentro de este componente, dos propiedades dentro de `data`, un string y un número, y los vamos a mostrar en el componente.

/vuejs-filtros-locales-lab/src/components/CmpFiltroLocal.vue

```
<template>
  <div>
    <p>Nombre: {{nombre}}</p>
    <p>Precio: {{precio}}</p>
  </div>
</template>

<script>
export default {
  data() {
    return {
      nombre: 'Charly',
      precio: 3
    }
  }
}
</script>

<style>

</style>
```

Ahora vamos a mostrar este componente en el componente principal de la aplicación.

/vuejs-filtros-locales-lab/src/App.vue

```
<template>
  <div>
    <CmpFiltroLocal />
  </div>
</template>

<script>
import CmpFiltroLocal from './components/CmpFiltroLocal';

export default {
  components: {
    CmpFiltroLocal
  }
}
</script>

<style>

</style>
```

Y ahora vamos a añadir nuestro filtro local en el componente donde teníamos los datos. Para añadir un filtro local, tenemos que añadir una propiedad `filters` y dentro de esta una función con

el nombre que queramos ponerle al filtro, la cual va a recibir el valor de la propiedad a la que se aplica el filtro. Dentro de la función añadimos la lógica necesaria para transformar los valores según queremos mostrarlos.

/vuejs-filtros-locales-lab/src/components/CmpFiltroLocal.vue

```
<template>
  <div>
    <p>Nombre: {{nombre}}</p>
    <p>Precio: {{precio}}</p>
  </div>
</template>

<script>
export default {
  data() {
    return {
      nombre: 'Charly',
      precio: 3
    }
  },
  filters: {
    doble(value) {
      if (typeof(value) === 'string') {
        return value.repeat(2);
      }
      return value * 2;
    }
  }
}
</script>

<style>

</style>
```

Una vez que tenemos el filtro, podemos aplicarlo dentro del string interpolation donde se están mostrando los valores. Lo único que hay que hacer es añadir detrás del valor al que se lo vamos a aplicar `| nombreFiltro`.

/vuejs-filtros-locales-lab/src/components/CmpFiltroLocal.vue

```
<template>
  <div>
    <p>Nombre: {{nombre | doble}}</p>
    <p>Precio: {{precio | doble}}</p>
  </div>
</template>

<script>
export default {
  data() {
    return {
      nombre: 'Charly',
      precio: 3
    }
  },
  filters: {
    doble(value) {
      if (typeof(value) === 'string') {
        return value.repeat(2);
      }
      return value * 2;
    }
  }
}
</script>

<style>

</style>
```

Una vez añadido, si miramos la aplicación vamos a ver el doble del número, y en el caso del string, este se muestra repetido dos veces.

19.2. Lab: Filtros globales

En este laboratorio vamos a crear un filtro global para usar en la aplicación.

Empezamos por crear el proyecto lanzando el siguiente comando y seleccionamos la configuración por defecto.

```
$ vue create vuejs-filtros-globales-lab
```

Una vez creado el proyecto, eliminamos el componente `HelloWorld` y dejamos vacío el componente `App`.

Cuando los filtros son globales, al igual que los componentes globales, los vamos a definir antes de

crear la instancia de Vue de nuestra aplicación, por lo tanto, en nuestro caso, lo vamos a implementar en un archivo `filtro-global.js`.

Dentro del archivo, vamos a importar Vue para poder usar la función `Vue.filter` a la que le vamos a pasar como primer parámetro el nombre del filtro, y como segundo parámetro la función que se encarga de obtener el valor y devolverlo transformado.

/vuejs-filtros-globales-lab/src/filtro-global.js

```
import Vue from 'vue';

Vue.filter('uppercase', function(value) {
  return value.toUpperCase();
});
```

Una vez que tenemos el filtro creado, vamos a importar este archivo, dentro del `main.js` antes de crear la instancia de la aplicación.

/vuejs-filtros-globales-lab/src/main.js

```
import Vue from 'vue'
import App from './App.vue'
import './filtro-global';

Vue.config.productionTip = false

new Vue({
  render: h => h(App),
}).$mount('#app')
```

Y una vez que ya lo tenemos importado, ya podemos usarlo en todos los componentes de la aplicación, así que vamos a ir al componente raíz de la aplicación, vamos a añadir un dato de tipo string, y le vamos a aplicar el filtro que hemos creado.

/vuejs-filtros-globales-lab/src/App.vue

```
<template>
  <div>
    <p>{{msg | uppercase}}</p>
  </div>
</template>

<script>


export default {
  data() {
    return {
      msg: 'Hola mundo!'
    }
  }
}
</script>

<style>

</style>
```

Una vez que lo tenemos, si abrimos la aplicación en el navegador, el mensaje mostrado tiene que salir en mayúsculas.

19.3. Lab: Filtros con parámetros

En este laboratorio vamos a ver como añadirle parámetros a los pipes para que sean más dinámicos.

Empezamos por crear el proyecto lanzando el siguiente comando y seleccionamos la configuración por defecto.

```
$ vue create vuejs-filtros-con-parametros-lab
```

Una vez creado el proyecto, eliminamos el componente `HelloWorld` y dejamos vacío el componente `App`.

Vamos a crear un componente `CmpFiltroConParams` dentro de la carpeta `components`, y antes de continuar lo vamos a importar en nuestro componente raíz.

/vuejs-filtros-con-parametros-lab/src/App.vue

```
<template>
  <div>
    <CmpFiltroConParams />
  </div>
</template>

<script>
import CmpFiltroConParams from './components/CmpFiltroConParams';

export default {
  components: {
    CmpFiltroConParams
  }
}
</script>

<style>

</style>
```

Ahora vamos a crear un dato de tipo string dentro del componente creado, y vamos a empezar por mostrar dicho dato.

/vuejs-filtros-con-parametros-lab/src/components/CmpFiltroConParams.vue

```
<template>
  <div>
    <p>{{msg}}</p>
  </div>
</template>

<script>
export default {
  data() {
    return {
      msg: 'Hola mundo!'
    }
  }
}
</script>

<style>

</style>
```

Lo siguiente es crear un filtro nuevo `slice` que se va a encargar de obtener un substring de un string. Para ello vamos a añadir la función `slice` dentro de la propiedad `filters` del componente, y le vamos a pasar como parámetro el valor que hay que transformar.

/vuejs-filtros-con-parametros-lab/src/components/CmpFiltroConParams.vue

```
<template>
  <div>
    <p>{{msg}}</p>
  </div>
</template>

<script>
export default {
  data() {
    return {
      msg: 'Hola mundo!'
    }
  },
  filters: {
    slice(value) {

    }
  }
}
</script>

<style>

</style>
```

Cuando queremos añadir parámetros a un filtro, solo tenemos que añadirlos en la función detrás del primer parámetro, por lo que vamos a pasarle dos parámetros más a la función y vamos a rellenarla con el código necesario para obtener el substring que se encuentra entre las posiciones dadas como parámetros.

/vuejs-filtros-con-parametros-lab/src/components/CmpFiltroConParams.vue

```
<template>
  <div>
    <p>{{msg}}</p>
  </div>
</template>

<script>
export default {
  data() {
    return {
      msg: 'Hola mundo!'
    }
  },
  filters: {
    slice(value, start, end) {
      if (end) {
        return value.slice(start, end);
      }
      if (start) {
        return value.slice(start);
      }
      return value;
    }
  }
}
</script>

<style>

</style>
```

Y por último a la hora de usar el pipe con parámetros, tenemos que llamar a la función del pipe y pasarlos sin tener en cuenta el parámetro `value` que lo va a recibir siempre.

/vuejs-filtros-con-parametros-lab/src/components/CmpFiltroConParams.vue

```
<template>
  <div>
    <p>{{msg | slice}}</p>
    <p>{{msg | slice(5, 10)}}</p>
    <p>{{msg | slice(5)}}</p>
  </div>
</template>

<script>
export default {
  data() {
    return {
      msg: 'Hola mundo!'
    }
  },
  filters: {
    slice(value, start, end) {
      if (end) {
        return value.slice(start, end);
      }
      if (start) {
        return value.slice(start);
      }
      return value;
    }
  }
}
</script>

<style>

</style>
```

Una vez aplicado el pipe si miramos el resultado en el navegador, deberíamos de ver:

```
Hola mundo!
mundo
mundo!
```

Capítulo 20. Formularios

Los formularios son una de las partes más importantes de las aplicaciones porque son los elementos con los que vamos a recoger información enviada por el usuario. Con ellos vamos a poder hacer el login, rellenar los datos para hacer una reserva...

En Vue para trabajar con los formularios, vamos a usar la directiva **v-model** que nos ayuda con algunos de los distintos tipos de inputs que nos podemos encontrar.

20.1. Input

En los inputs normales (los de tipo **text**, **number**, **email**...) vamos a usarla para darle el valor por defecto y recoger los cambios que vayamos realizando.

20.2. Radio

En los inputs de tipo **radio**, le daremos como valor a la propiedad **value** un string, y al **v-model** le asignaremos el nombre de la variable donde hay que guardar el valor seleccionado.

Los distintos inputs que pertenezcan al mismo grupo de **radio buttons** tienen que llevar como valor la misma variable, la cual se va a ir sobreescribiendo según vayamos pulsando las distintas opciones.

20.3. Checkbox

Con los inputs de tipo **checkbox** ocurre lo mismo que con los de tipo **radio**, solo que esta vez en lugar de ser un string la variable donde vamos a guardar aquellos elementos en los que pulsamos, será un array para que permita seleccionar varios de ellos.

20.4. Select

Para los inputs de tipo **select**, es decir, los desplegables, tendremos que pasarle al **v-model** una variable cuyo valor sea un string. Este string tiene que coincidir con alguno de los valores asignados a las propiedades **value** de las etiquetas **option**.

20.5. Lab: Formularios

En este laboratorio vamos a ver como crear un formulario de registro con distintos tipos de inputs.

Empezamos por crear el proyecto lanzando el siguiente comando y seleccionamos la configuración por defecto.

```
$ vue create vuejs-formularios-lab
```

Una vez creado el proyecto, eliminamos el componente `HelloWorld` y dejamos vacío el componente

`App`.

Vamos a crear un componente `CmpFormulario` dentro de la carpeta `components`.

En este componente vamos a crear un formulario en el que vamos a poner los siguientes campos:

- Uno para el nombre de tipo **text**.
- Uno para el email de tipo **email**.
- Uno para la contraseña de tipo **password**.
- Uno para el pais de tipo **select**.
- Uno para indicar si eres estudiante o no de tipo **radio**.
- Uno para elegir tus hobbies de tipo **checkbox**.

/vuejs-formularios-lab/src/components/CmpFormulario.vue

```
<template>
  <div>
    <form>
      <div>
        <label for="nombre">Nombre</label>
        <input type="text" name="nombre" id="nombre">
      </div>
      <div>
        <label for="email">Email</label>
        <input type="email" name="email" id="email">
      </div>
      <div>
        <label for="password">Contraseña</label>
        <input type="password" name="password" id="password">
      </div>
      <div>
        <label for="pais">País</label>
        <select name="pais" id="pais">
          <option value="es">España</option>
          <option value="fr">Francia</option>
          <option value="it">Italia</option>
          <option value="ge">Alemania</option>
        </select>
      </div>
      <div>
        <label for="estudiante">Eres estudiante?</label>
        <input type="radio" name="estudiante" id="estudiante" value="si">Si
        <input type="radio" name="estudiante" id="estudiante" value="no">No
      </div>
      <div>
        <label for="hobbies">Hobbies</label>
        <input type="checkbox" name="hobbies" id="hobbies" value="tenis">Tenis
        <input type="checkbox" name="hobbies" id="hobbies" value="futbol">Futbol
        <input type="checkbox" name="hobbies" id="hobbies" value="natacion">Natación
```

```
        <input type="checkbox" name="hobbies" id="hobbies" value="teatro">Teatro
        <input type="checkbox" name="hobbies" id="hobbies" value="series">Series
        <input type="checkbox" name="hobbies" id="hobbies" value="cine">Cine
      </div>
      <button type="submit">Registrarme</button>
    </form>
    <hr>
    <pre></pre>
  </div>
</template>

<script>
export default {

}
</script>

<style>

</style>
```

Una vez que tenemos la estructura del formulario, vamos a añadir este componente dentro del componente raíz para mostrarlo.

/vuejs-formularios-lab/src/App.vue

```
<template>
  <CmpFormulario />
</template>

<script>
import CmpFormulario from './components/CmpFormulario';

export default {
  components: {
    CmpFormulario
  }
}
</script>

<style>

</style>
```

Lo siguiente es añadir los datos donde se van a ir guardando los valores de los inputs del formulario. Para ello vamos a añadir la propiedad **data** dentro del componente.

Vamos a empezar añadiendo los datos para el nombre, el email y la contraseña, ya que el valor que les tenemos que asignar es un string. Y tenemos que asignar estas variables en los inputs correspondientes a estos campos usando la directiva **v-model**.

/vuejs-formularios-lab/src/components/CmpFormulario.vue

```
<template>
  <div>
    <form>
      <div>
        <label for="nombre">Nombre</label>
        <input type="text" name="nombre" id="nombre" v-model="nombre">
      </div>
      <div>
        <label for="email">Email</label>
        <input type="email" name="email" id="email" v-model="email">
      </div>
      <div>
        <label for="password">Contraseña</label>
        <input type="password" name="password" id="password" v-model="password">
      </div>
      <div>
        <label for="pais">País</label>
        <select name="pais" id="pais">
          <option value="es">España</option>
          <option value="fr">Francia</option>
          <option value="it">Italia</option>
          <option value="ge">Alemania</option>
        </select>
      </div>
      <div>
        <label for="estudiante">Eres estudiante?</label>
        <input type="radio" name="estudiante" id="estudiante" value="si">Si
        <input type="radio" name="estudiante" id="estudiante" value="no">No
      </div>
      <div>
        <label for="hobbies">Hobbies</label>
        <input type="checkbox" name="hobbies" id="hobbies" value="tenis">Tenis
        <input type="checkbox" name="hobbies" id="hobbies" value="futbol">Futbol
        <input type="checkbox" name="hobbies" id="hobbies" value="natacion">Natación
        <input type="checkbox" name="hobbies" id="hobbies" value="teatro">Teatro
        <input type="checkbox" name="hobbies" id="hobbies" value="series">Series
        <input type="checkbox" name="hobbies" id="hobbies" value="cine">Cine
      </div>
      <button type="submit">Registrarme</button>
    </form>
    <hr>
    <pre></pre>
  </div>
</template>

<script>
export default {
  data() {
    return {
```

```
      nombre: 'Charly Falco',
      email: 'cfalco@gmail.com',
      password: 'cfalco123'
    }
  }
}
</script>

<style>

</style>
```

Ahora para ir comprobando que si modificamos los datos, estos van actualizandose en el estado del componente, vamos a añadir una computed prop que nos devuelva como JSON todo el estado. Y dentro de la etiqueta **pre** mostraremos el valor de esta propiedad.

/vuejs-formularios-lab/src/components/CmpFormulario.vue

```
<template>
  <div>
    <form>
      <div>
        <label for="nombre">Nombre</label>
        <input type="text" name="nombre" id="nombre" v-model="nombre">
      </div>
      <div>
        <label for="email">Email</label>
        <input type="email" name="email" id="email" v-model="email">
      </div>
      <div>
        <label for="password">Contraseña</label>
        <input type="password" name="password" id="password" v-model="password">
      </div>
      <div>
        <label for="pais">País</label>
        <select name="pais" id="pais">
          <option value="es">España</option>
          <option value="fr">Francia</option>
          <option value="it">Italia</option>
          <option value="ge">Alemania</option>
        </select>
      </div>
      <div>
        <label for="estudiante">Eres estudiante?</label>
        <input type="radio" name="estudiante" id="estudiante" value="si">Si
        <input type="radio" name="estudiante" id="estudiante" value="no">No
      </div>
      <div>
        <label for="hobbies">Hobbies</label>
        <input type="checkbox" name="hobbies" id="hobbies" value="tenis">Tenis
        <input type="checkbox" name="hobbies" id="hobbies" value="futbol">Futbol
```

```
        <input type="checkbox" name="hobbies" id="hobbies" value="natacion">Natación
        <input type="checkbox" name="hobbies" id="hobbies" value="teatro">Teatro
        <input type="checkbox" name="hobbies" id="hobbies" value="series">Series
        <input type="checkbox" name="hobbies" id="hobbies" value="cine">Cine
      </div>
      <button type="submit">Registrarme</button>
    </form>
    <hr>
    <pre>{{datosForm}}</pre>
  </div>
</template>

<script>
export default {
  data() {
    return {
      nombre: 'Charly Falco',
      email: 'cfalco@gmail.com',
      password: 'cfalco123'
    }
  },
  computed: {
    datosForm() {
      return JSON.stringify(this.$data, null, 4)
    }
  }
}
</script>

<style>

</style>
```

Ahora deberíamos de ver los datos que se están almacenando en el componente como un JSON debajo del formulario.

Lo siguiente es añadir la parte del desplegable. Para ello, tenemos que añadir a data otra propiedad de tipo string. Si el valor es un string vacio, no aparecerá ninguna de las opciones preseleccionada, en cambio, si le damos como valor el valor de una de las opciones, esta aparecerá seleccionada inicialmente.

/vuejs-formularios-lab/src/components/CmpFormulario.vue

```
<template>
  <div>
    <form>
      <div>
        <label for="nombre">Nombre</label>
        <input type="text" name="nombre" id="nombre" v-model="nombre">
      </div>
      <div>
```

```
        <label for="email">Email</label>
        <input type="email" name="email" id="email" v-model="email">
      </div>
      <div>
        <label for="password">Contraseña</label>
        <input type="password" name="password" id="password" v-model="password">
      </div>
      <div>
        <label for="pais">País</label>
        <select name="pais" id="pais" v-model="pais">
          <option value="es">España</option>
          <option value="fr">Francia</option>
          <option value="it">Italia</option>
          <option value="ge">Alemania</option>
        </select>
      </div>
      <div>
        <label for="estudiante">Eres estudiante?</label>
        <input type="radio" name="estudiante" id="estudiante" value="si">Si
        <input type="radio" name="estudiante" id="estudiante" value="no">No
      </div>
      <div>
        <label for="hobbies">Hobbies</label>
        <input type="checkbox" name="hobbies" id="hobbies" value="tenis">Tenis
        <input type="checkbox" name="hobbies" id="hobbies" value="futbol">Futbol
        <input type="checkbox" name="hobbies" id="hobbies" value="natacion">Natación
        <input type="checkbox" name="hobbies" id="hobbies" value="teatro">Teatro
        <input type="checkbox" name="hobbies" id="hobbies" value="series">Series
        <input type="checkbox" name="hobbies" id="hobbies" value="cine">Cine
      </div>
      <button type="submit">Registrarme</button>
    </form>
    <hr>
    <pre>{{datosForm}}</pre>
  </div>
</template>

<script>
export default {
  data() {
    return {
      nombre: 'Charly Falco',
      email: 'cfalco@gmail.com',
      password: 'cfalco123',
      pais: ''
    }
  },
  computed: {
    datosForm() {
      return JSON.stringify(this.$data, null, 4)
    }
```

```
  }
}
</script>

<style>

</style>
```

El siguiente paso es añadir otra propiedad de tipo string dentro de data para controlar si somos estudiantes o no. En este caso cada input de tipo **radio** tiene que tener el mismo **v-model** para que al seleccionar uno se deseleccione el otro.

/vuejs-formularios-lab/src/components/CmpFormulario.vue

```
<template>
  <div>
    <form>
      <div>
        <label for="nombre">Nombre</label>
        <input type="text" name="nombre" id="nombre" v-model="nombre">
      </div>
      <div>
        <label for="email">Email</label>
        <input type="email" name="email" id="email" v-model="email">
      </div>
      <div>
        <label for="password">Contraseña</label>
        <input type="password" name="password" id="password" v-model="password">
      </div>
      <div>
        <label for="pais">País</label>
        <select name="pais" id="pais" v-model="pais">
          <option value="es">España</option>
          <option value="fr">Francia</option>
          <option value="it">Italia</option>
          <option value="ge">Alemania</option>
        </select>
      </div>
      <div>
        <label for="estudiante">Eres estudiante?</label>
        <input type="radio" name="estudiante" id="estudiante" value="si" v-model="estudiante">Si
        <input type="radio" name="estudiante" id="estudiante" value="no" v-model="estudiante">No
      </div>
      <div>
        <label for="hobbies">Hobbies</label>
        <input type="checkbox" name="hobbies" id="hobbies" value="tenis">Tenis
        <input type="checkbox" name="hobbies" id="hobbies" value="futbol">Futbol
        <input type="checkbox" name="hobbies" id="hobbies" value="natacion">Natación
        <input type="checkbox" name="hobbies" id="hobbies" value="teatro">Teatro
        <input type="checkbox" name="hobbies" id="hobbies" value="series">Series
        <input type="checkbox" name="hobbies" id="hobbies" value="cine">Cine
      </div>
      <button type="submit">Registrarme</button>
    </form>
```

```
    <hr>
    <pre>{{datosForm}}</pre>
  </div>
</template>

<script>
export default {
  data() {
    return {
      nombre: 'Charly Falco',
      email: 'cfalco@gmail.com',
      password: 'cfalco123',
      pais: '',
      estudiante: 'si',
    }
  },
  computed: {
    datosForm() {
      return JSON.stringify(this.$data, null, 4)
    }
  }
}
</script>

<style>

</style>
```

Y para terminar, la parte de los hobbies es igual que la anterior, salvo porque la propiedad que va a mantener el estado de los hobbies que tienen que aparecer marcado tiene que ser un array en lugar de un string.

/vuejs-formularios-lab/src/components/CmpFormulario.vue

```
<template>
  <div>
    <form>
      <div>
        <label for="nombre">Nombre</label>
        <input type="text" name="nombre" id="nombre" v-model="nombre">
      </div>
      <div>
        <label for="email">Email</label>
        <input type="email" name="email" id="email" v-model="email">
      </div>
      <div>
        <label for="password">Contraseña</label>
        <input type="password" name="password" id="password" v-model="password">
      </div>
      <div>
        <label for="pais">País</label>
        <select name="pais" id="pais" v-model="pais">
          <option value="es">España</option>
          <option value="fr">Francia</option>
          <option value="it">Italia</option>
```

```
          <option value="ge">Alemania</option>
        </select>
      </div>
      <div>
        <label for="estudiante">Eres estudiante?</label>
        <input type="radio" name="estudiante" id="estudiante" value="si" v-model="estudiante">Si
        <input type="radio" name="estudiante" id="estudiante" value="no" v-model="estudiante">No
      </div>
      <div>
        <label for="hobbies">Hobbies</label>
        <input type="checkbox" name="hobbies" id="hobbies" value="tenis" v-model="hobbies">Tenis
        <input type="checkbox" name="hobbies" id="hobbies" value="futbol" v-model="hobbies">Futbol
        <input type="checkbox" name="hobbies" id="hobbies" value="natacion" v-model="hobbies">Natación
        <input type="checkbox" name="hobbies" id="hobbies" value="teatro" v-model="hobbies">Teatro
        <input type="checkbox" name="hobbies" id="hobbies" value="series" v-model="hobbies">Series
        <input type="checkbox" name="hobbies" id="hobbies" value="cine" v-model="hobbies">Cine
      </div>
      <button type="submit">Registrarme</button>
    </form>
    <hr>
    <pre>{{datosForm}}</pre>
  </div>
</template>

<script>
export default {
  data() {
    return {
      nombre: 'Charly Falco',
      email: 'cfalco@gmail.com',
      password: 'cfalco123',
      pais: '',
      estudiante: 'si',
      hobbies: ['futbol', 'series']
    }
  },
  computed: {
    datosForm() {
      return JSON.stringify(this.$data, null, 4)
    }
  }
}
</script>

<style>

</style>
```

Y hasta aquí ya tenemos la forma de ir controlando los valores del formulario. Una vez que el formulario se ha rellenado, tendriamos que detectar el evento **submit** sobre la etiqueta **form** y ejecutar el código necesario para realizar el envio del formulario.

No hay que olvidar que si no llamamos a **event.preventDefault**, se enviará una petición que hará que la página se refresque perdiendo el estado actual de la aplicación.

/vuejs-formularios-lab/src/components/CmpFormulario.vue

```
<template>
  <div>
    <form @submit.prevent="guardarDatos">
      <div>
        <label for="nombre">Nombre</label>
        <input type="text" name="nombre" id="nombre" v-model="nombre">
      </div>
      <div>
        <label for="email">Email</label>
        <input type="email" name="email" id="email" v-model="email">
      </div>
      <div>
        <label for="password">Contraseña</label>
        <input type="password" name="password" id="password" v-model="password">
      </div>
      <div>
        <label for="pais">País</label>
        <select name="pais" id="pais" v-model="pais">
          <option value="es">España</option>
          <option value="fr">Francia</option>
          <option value="it">Italia</option>
          <option value="ge">Alemania</option>
        </select>
      </div>
      <div>
        <label for="estudiante">Eres estudiante?</label>
        <input type="radio" name="estudiante" id="estudiante" value="si" v-model="estudiante">Si
        <input type="radio" name="estudiante" id="estudiante" value="no" v-model="estudiante">No
      </div>
      <div>
        <label for="hobbies">Hobbies</label>
        <input type="checkbox" name="hobbies" id="hobbies" value="tenis" v-model="hobbies">Tenis
        <input type="checkbox" name="hobbies" id="hobbies" value="futbol" v-model="hobbies">Futbol
        <input type="checkbox" name="hobbies" id="hobbies" value="natacion" v-model="hobbies">Natación
        <input type="checkbox" name="hobbies" id="hobbies" value="teatro" v-model="hobbies">Teatro
        <input type="checkbox" name="hobbies" id="hobbies" value="series" v-model="hobbies">Series
        <input type="checkbox" name="hobbies" id="hobbies" value="cine" v-model="hobbies">Cine
      </div>
      <button type="submit">Registrarme</button>
    </form>
    <hr>
    <pre>{{datosForm}}</pre>
  </div>
</template>

<script>
export default {
  data() {
    return {
      nombre: 'Charly Falco',
      email: 'cfalco@gmail.com',
      password: 'cfalco123',
      pais: '',
      estudiante: 'si',
      hobbies: ['futbol', 'series']
    }
```

```
  },
  computed: {
    datosForm() {
      return JSON.stringify(this.$data, null, 4)
    }
  },
  methods: {
    guardarDatos() {
      // Aquí habría que añadir la petición que manda los datos al servidor para guardarlos
      console.log(this.$data);
    }
  }
}
</script>

<style>

</style>
```

Una vez hecho esto, ahora al darle al botón, tenemos que ver en la consola del navegador los datos que habíamos rellenado en el formulario.

Capítulo 21. Modificadores

Los **modificadores** nos van a permitir realizar alguna acción predefinida cuando ocurra algún evento.

Podemos diferenciar los modificadores en varios tipos:

- De eventos
- De teclas
- De inputs

21.1. Modificadores de eventos

Estos modificadores se ponen detrás del evento que queremos detectar sobre algún elemento.

Aquí nos podemos encontrar con los modificadores:

- `prevent`: nos va a permitir es evitar el comportamiento por defecto que realizan algunos elementos de HTML ante algún evento.
- `stop`: podemos detener la propagación del evento hacia los elementos superiores.
- `native`: permite detectar un evento que no se ha definido en un componente sobre la etiqueta de dicho componente.

21.2. Modificadores de teclas

A veces necesitamos ejecutar una función solo en caso de pulsar una tecla concreta, y para ello tenemos que comprobar el código asignado a esa tecla.

Vue nos proporciona los modificadores de teclas que nos permiten lanzar los eventos asociados a unas teclas. Algunos de estos modificadores son: `enter`, `tab`, `delete`, `space`, `up`, `right`...

Estos modificadores se pueden concatenar poniendo uno detrás de otro en caso de que todos ellos tengan que ejecutar el mismo código.

21.3. Modificadores de inputs

Los modificadores de inputs, se aplican a elementos HTML `input` a los que se le van a realizar una modificación sobre el valor que tiene el campo. Estos modificadores se ponen después de la directiva `v-model`.

Podemos encontrarnos con los modificadores:

- `trim`: que va a eliminar espacios en blanco sobrantes al inicio y final del valor del elemento `input`.
- `number`: que va a mandarnos el valor de un `input` de tipo númerico como número y no como string que es como se manda el valor por defecto.

- `lazy`: el cual va a hacer que el `v-model` en lugar de realizar el evento `input`, realice el evento `change`, es decir, que no va a mandar el valor hasta que el campo pierda el foco, pulsemos la tecla `enter`...

21.4. Lab: Modificadores de eventos

En este laboratorio vamos a ver como usar los modificadores de eventos `prevent`, `stop` y `native`.

Empezamos por crear el proyecto lanzando el siguiente comando y seleccionamos la configuración por defecto.

```
$ vue create vuejs-modificadores-de-eventos-lab
```

Una vez creado el proyecto, eliminamos el componente `HelloWorld` y dejamos vacío el componente `App`.

Vamos a crear el componente `CmpModificadores` dentro de la carpeta `components` y lo vamos a añadir dentro del `App`.

/vuejs-modificadores-de-eventos-lab/src/App.vue

```
<template>
  <div>
    <CmpModificadores />
  </div>
</template>

<script>
import CmpModificadores from './components/CmpModificadores';

export default {
  components: {
    CmpModificadores
  }
}
</script>

<style>

</style>
```

21.4.1. `prevent`

Dentro del componente de los modificadores, vamos a añadir un enlace que nos mande a https://vuejs.org/ en una pestaña nueva del navegador.

/vuejs-modificadores-de-eventos-lab/src/components/CmpModificadores.vue

```
<template>
  <div>
    <a href="https://vuejs.org" target="_blank">Ir a la página oficial de Vue</a>
  </div>
</template>

<script>
export default {

}
</script>

<style>

</style>
```

Si pulsamos sobre este enlace, vemos que abre la página en una nueva pestaña, pero vamos a usar el modificador `prevent` para evitar que habrá la página. Normalmente haremos esto porque no queremos que las etiquetas realicen las acciones que tienen definidas por defecto cuando ocurre algún evento sobre ellas, como por ejemplo:

- Evento `submit` sobre un formulario: realiza una petición con los datos del formulario.
- Evento `click` sobre un link: abre la url asignada al link.

En este caso, pondremos el evento seguido del modificador separado por un punto, y le asignaremos la función que queremos ejecutar en el lugar de que realice su acción predefinida.

/vuejs-modificadores-de-eventos-lab/src/components/CmpModificadores.vue

```
<template>
  <div>
    <a href="https://vuejs.org" target="_blank" @click.prevent="mostrarEnlace">Ir a la página oficial de Vue</a>
  </div>
</template>

<script>
export default {
  methods: {
    mostrarEnlace(e) {
      alert('Interceptada navegación a ' + e.target.href);
    }
  }
}
</script>

<style>

</style>
```

Ahora si pulsamos el link, en lugar de abrir la página, nos mostrará el popup del navegador con el mensaje que hemos puesto.

21.4.2. `stop`

Para ver este modificador, vamos a añadir dos cajas en la plantilla, una dentro de otra, y vamos a hacer que muestren un mensaje cada vez que se pulsen sobre ellas, indicando en el mensaje sobre que caja se ha pulsado.

/vuejs-modificadores-de-eventos-lab/src/components/CmpModificadores.vue

```
<template>
  <div>
    <a href="https://vuejs.org" target="_blank" @click.prevent="mostrarEnlace">Ir a la página oficial de Vue</a>
    <div id="caja1" @click="mostrarCajaPulsada('Caja 1')">
      <div id="caja2" @click="mostrarCajaPulsada('Caja 2')">
      </div>
    </div>
  </div>
</template>

<script>
export default {
  methods: {
    mostrarEnlace(e) {
      alert('Interceptada navegación a ' + e.target.href);
    },
    mostrarCajaPulsada(textoCaja) {
      alert('Has pulsado sobre ' + textoCaja)
    }
  }
}
</script>

<style>

</style>
```

Ahora vamos a añadir unos estilos a las cajas para diferenciarlas.

/vuejs-modificadores-de-eventos-lab/src/components/CmpModificadores.vue

```
<template>
  <div>
    <a href="https://vuejs.org" target="_blank" @click.prevent="mostrarEnlace">Ir a la página oficial de Vue</a>
    <div id="caja1" @click="mostrarCajaPulsada('Caja 1')">
      <div id="caja2" @click="mostrarCajaPulsada('Caja 2')">
      </div>
    </div>
  </div>
</template>

<script>
export default {
  methods: {
    mostrarEnlace(e) {
      alert('Interceptada navegación a ' + e.target.href);
    },
    mostrarCajaPulsada(textoCaja) {
      alert('Has pulsado sobre ' + textoCaja)
    }
  }
}
</script>

<style scoped>
  #caja1 {
    width: 400px;
    height: 400px;
    border: 1px solid orange;
  }

  #caja2 {
    width: 200px;
    height: 200px;
    border: 1px solid blue;
  }
</style>
```

Si pulsamos sobre la caja de dentro, vemos que se muestra que se ha pulsado tanto en la caja de dentro como la caja de fuera, y esto se debe a la propagación del evento hacía los elementos superiores, es decir, al pulsar sobre la caja de dentro, se va a ejecutar la función asociada a dicho evento, pero el evento se irá propagando hacia los elementos superiores, llegando a ejecutar también la función de la caja superior también.

Esto podemos evitarlo añadiendo el modificador `stop` que se encarga de parar la propagación del evento ocurrido en la caja de dentro.

/vuejs-modificadores-de-eventos-lab/src/components/CmpModificadores.vue

```
<template>
  <div>
    <a href="https://vuejs.org" target="_blank" @click.prevent="mostrarEnlace">Ir a la página oficial de Vue</a>
    <div id="caja1" @click="mostrarCajaPulsada('Caja 1')">
      <div id="caja2" @click.stop="mostrarCajaPulsada('Caja 2')">
      </div>
    </div>
  </div>
</template>

<script>
export default {
  methods: {
    mostrarEnlace(e) {
      alert('Interceptada navegación a ' + e.target.href);
    },
    mostrarCajaPulsada(textoCaja) {
      alert('Has pulsado sobre ' + textoCaja)
    }
  }
}
</script>

<style scoped>
  #caja1 {
    width: 400px;
    height: 400px;
    border: 1px solid orange;
  }

  #caja2 {
    width: 200px;
    height: 200px;
    border: 1px solid blue;
  }
</style>
```

Ahora al pulsar la caja de fuera se muestra un mensaje, y al pulsar la caja de dentro se muestra otro y no dos como ocurría antes.

21.4.3. `native`

Ahora vamos a crear un componente `Texto` que va a mostra un texto cualquiera y que pondremos dentro del componente `CmpModificadores`.

/vuejs-modificadores-de-eventos-lab/src/components/Texto.vue

```
<template>
  <div>
    <p>Ir por leña y volver caliente, le ocurre a mucha gente.</p>
  </div>
</template>

<script>
export default {

}
</script>

<style scoped>
  div {
    height: 50px;
    border: 1px solid black;
    text-align: center;
    font-style: italic;
  }
</style>
```

Lo que vamos a hacer es añadir un evento, no dentro del componente `Texto`, sino en la etiqueta del componente `Texto` que hemos añadido dentro del componente `CmpModificadores`, lo que nos va a permitir añadir eventos sobre un componente entero, y no dentro de este.

Para ello, añadimos un evento con el modificador `native` sobre dicha etiqueta y vamos mostrar el texto que tiene dentro dicho componente.

/vuejs-modificadores-de-eventos-lab/src/components/CmpModificadores.vue

```
<template>
  <div>
    <a href="https://vuejs.org" target="_blank" @click.prevent="mostrarEnlace">Ir a la página oficial de Vue</a>
    <div id="caja1" @click="mostrarCajaPulsada('Caja 1')">
      <div id="caja2" @click.stop="mostrarCajaPulsada('Caja 2')">
      </div>
    </div>
    <Texto @mouseover.native="mostrarTexto" />
  </div>
</template>

<script>
import Texto from './Texto';

export default {
  components: {
    Texto
  },
  methods: {
    mostrarEnlace(e) {
      alert('Interceptada navegación a ' + e.target.href);
    },
    mostrarCajaPulsada(textoCaja) {
      alert('Has pulsado sobre ' + textoCaja)
    },
    mostrarTexto(e) {
      alert(e.target.textContent)
    }
  }
}
</script>

<style scoped>
  #caja1 {
    width: 400px;
    height: 400px;
    border: 1px solid orange;
  }

  #caja2 {
    width: 200px;
    height: 200px;
    border: 1px solid blue;
  }
</style>
```

Ahora al pasar el ratón por encima del componente nos mostrará el texto en un popup del navegador.

21.5. Lab: Modificadores de teclas

En este laboratorio vamos a ver como usar los modificadores de teclas para ejecutar ciertas funciones cuando se pulsa sobre ciertas teclas como `space`, `enter`, `up`, `right`, `delete`...

Empezamos por crear el proyecto lanzando el siguiente comando y seleccionamos la configuración por defecto.

```
$ vue create vuejs-modificadores-de-teclas-lab
```

Una vez creado el proyecto, eliminamos el componente `HelloWorld` y dejamos vacío el componente `App`.

Vamos a crear el componente `CmpModificadores` dentro de la carpeta `components` y vamos a añadir un input en el que vamos a detectar cuando se ha pulsado sobre la tecla de espacio y de intro para mostrar un mensaje con las teclas pulsadas.

Lo único que hay que hacer es añadir un modificador con el nombre de la tecla que queremos detectar y llamar a una función que ejecute el código necesario para mostrar el mensaje.

/vuejs-modificadores-de-teclas-lab/src/components/CmpModificadores.vue

```
<template>
  <div>
    <input type="text" @keyup.space="mostrarTecla('Espacio')" @keypress.enter="mostrarTecla('Intro')">
  </div>
</template>

<script>
export default {
  methods: {
    mostrarTecla(tecla) {
      alert('Has pulsado la tecla ' + tecla);
    }
  }
}
</script>

<style>

</style>
```

Ahora vamos a añadir este componente dentro del componente `App`.

/vuejs-modificadores-de-teclas-lab/src/App.vue

```
<template>
  <div>
    <CmpModificadores />
  </div>
</template>

<script>
import CmpModificadores from './components/CmpModificadores';

export default {
  components: {
    CmpModificadores
  }
}
</script>

<style>

</style>
```

Ahora si pulsamos las teclas para las que hemos puesto los modificadores, debería de mostrarse el mensaje con la tecla pulsada.

Además, podemos concatenar estos modificadores en el caso de que necesitemos ejecutar la misma función para distintas teclas. Lo único que hay que hacer es poner un modificador detrás de otro.

Vamos a añadir otro input en el que vamos a concatenar los modificadores de las teclas de las flechas de dirección, y vamos a usar `$event.code` para mostrar el nombre de las teclas pulsadas.

/vuejs-modificadores-de-teclas-lab/src/components/CmpModificadores.vue

```
<template>
  <div>
    <input type="text" @keyup.space="mostrarTecla('Espacio')" @keypress.enter="mostrarTecla('Intro')">
    <input type="text" @keyup.right.down.left.up="mostrarTecla($event.code)">
  </div>
</template>

<script>
export default {
  methods: {
    mostrarTecla(tecla) {
      alert('Has pulsado la tecla ' + tecla);
    }
  }
}
</script>

<style>

</style>
```

Ahora debería de mostrarnos el nombre de las flechas cuando las pulsamos sobre el segundo input que hemos añadido.

21.6. Lab: Modificadores de inputs

En este laboratorio vamos a ver como usar los modificadores de inputs (de la directiva `v-model`) como `trim`, `number` y `lazy`.

Empezamos por crear el proyecto lanzando el siguiente comando y seleccionamos la configuración por defecto.

```
$ vue create vuejs-modificadores-de-inputs-lab
```

Una vez creado el proyecto, eliminamos el componente `HelloWorld` y dejamos vacío el componente `App`.

Vamos a crear el componente `CmpModificadores` dentro de la carpeta `components` y lo vamos a añadir dentro del `App`.

/vuejs-modificadores-de-inputs-lab/src/App.vue

```
<template>
  <div>
    <CmpModificadores />
  </div>
</template>

<script>
import CmpModificadores from './components/CmpModificadores';

export default {
  components: {
    CmpModificadores
  }
}
</script>

<style>

</style>
```

Ahora vamos a ver algunos modificadores de inputs o aquellos que vamos a ponerle a la directiva `v-model`.

21.6.1. `trim`

En nuestro componente, vamos a añadir un input que va a usarse para modificar el valor de un dato del componente a través del *two way data binding*.

/vuejs-modificadores-de-inputs-lab/src/components/CmpModificadores.vue

```
<template>
  <div>
    <input type="text" v-model="nombre">
  </div>
</template>

<script>
export default {
  data() {
    return {
      nombre: ''
    }
  }
}
</script>

<style>

</style>
```

Ahora vamos a añadir el modificador `trim` a la directiva para que cuando se vaya a modificar el dato que estamos modificando, se aplique una función de *trim* y elimine los espacios al inicio y final del texto, de esta forma nos aseguramos de que los usuarios no pueden meter en los campos de texto espacios en blanco.

/vuejs-modificadores-de-inputs-lab/src/components/CmpModificadores.vue

```
<template>
  <div>
    <input type="text" v-model.trim="nombre">
  </div>
</template>

<script>
export default {
  data() {
    return {
      nombre: ''
    }
  }
}
</script>

<style>

</style>
```

Para comprobar que este modificador se encarga de realizar lo dicho anteriormente, vamos a

mostrar el valor de la longitud del dato usando una computed prop para ello.

/vuejs-modificadores-de-inputs-lab/src/components/CmpModificadores.vue

```
<template>
  <div>
    <p>Longitud del nombre: {{longitudDelNombre}}</p>
    <input type="text" v-model.trim="nombre">
  </div>
</template>

<script>
export default {
  data() {
    return {
      nombre: ''
    }
  },
  computed: {
    longitudDelNombre() {
      return this.nombre.length;
    }
  }
}
</script>

<style>

</style>
```

Si probamos a meter espacios, la longitud debería de ser 0.

21.6.2. `number`

Ahora vamos a añadir otro input de tipo numérico con la directiva `v-model` para cambiar el valor del input, además de añadir una computed prop que nos indique el tipo del dato que estamos guardando.

/vuejs-modificadores-de-inputs-lab/src/components/CmpModificadores.vue

```
<template>
  <div>
    <p>Longitud del nombre: {{longitudDelNombre}}</p>
    <input type="text" v-model.trim="nombre">
    <p>Tipo del precio: {{tipoDelPrecio}}</p>
    <input type="number" v-model="precio">
  </div>
</template>

<script>
export default {
  data() {
    return {
      nombre: '',
      precio: 12
    }
  },
  computed: {
    longitudDelNombre() {
      return this.nombre.length;
    },
    tipoDelPrecio() {
      return typeof(this.precio);
    },
  }
}
</script>

<style>

</style>
```

Si cambiamos el valor del input, deberíamos de ver que el tipo que se muestra es un **string**, aunque nosotros le hayamos puesto al campo de texto que el tipo es numérico. Esto se debe a que este tipo del campo hace que solo podamos introducir números y no podamos introducir otro tipo de carácteres, pero el valor que hayamos introducido lo va a tratar como un string.

El modificador `number` que vamos a añadirle a la directiva, se encarga de parsear a un número el valor que hayamos escrito en el campo.

/vuejs-modificadores-de-inputs-lab/src/components/CmpModificadores.vue

```
<template>
  <div>
    <p>Longitud del nombre: {{longitudDelNombre}}</p>
    <input type="text" v-model.trim="nombre">
    <p>Tipo del precio: {{tipoDelPrecio}}</p>
    <input type="number" v-model.number="precio">
  </div>
</template>

<script>
export default {
  data() {
    return {
      nombre: '',
      precio: 12
    }
  },
  computed: {
    longitudDelNombre() {
      return this.nombre.length;
    },
    tipoDelPrecio() {
      return typeof(this.precio);
    },
  }
}
</script>

<style>

</style>
```

Ahora al escribir un nuevo valor en el campo, veremos que siempre mostrará que tipo del dato es un **number**.

21.6.3. `lazy`

Por último, tenemos el modificador `lazy` el cual va a hacer que el `v-model` en lugar de realizar el evento `input`, realice el evento `change`, es decir, que no va a mandar el valor hasta que el campo pierda el foco, pulsemos la tecla `enter`...

Vamos a añadir otro campo de texto además de mostrar el valor del dato que vamos a manejar, y le añadiremos el modificador indicado a la directiva.

/vuejs-modificadores-de-inputs-lab/src/components/CmpModificadores.vue

```
<template>
  <div>
    <p>Longitud del nombre: {{longitudDelNombre}}</p>
    <input type="text" v-model.trim="nombre">
    <p>Tipo del precio: {{tipoDelPrecio}}</p>
    <input type="number" v-model.number="precio">
    <p>Texto lazy: {{texto}}</p>
    <input type="text" v-model.lazy="texto">
  </div>
</template>

<script>
export default {
  data() {
    return {
      nombre: '',
      precio: 12,
      texto: ''
    }
  },
  computed: {
    longitudDelNombre() {
      return this.nombre.length;
    },
    tipoDelPrecio() {
      return typeof(this.precio);
    },
  }
}
</script>

<style>

</style>
```

Ahora en lugar de ir cambiando el valor al mismo tiempo que lo escribimos, solo se reflejará el cambio cuando se pulse la tecla *ENTER* o se pulse con el ratón fuera del campo de texto.

Capítulo 22. Referencias

Las **referencias** en Vue nos van a permitir acceder a los elementos del DOM para interactuar con ellos. Equivaldría a usar métodos como `getElementById`, `querySelector`... del objeto *document*, pero como aquí estamos trabajando con un Virtual DOM, entonces esos métodos pueden llegar a dar problemas.

Para crear una referencia solo hay que asignarle un atributo `ref` a la etiqueta con la que queramos interactuar y asignarle un nombre a ese atributo. Y luego podemos acceder a las referencias creadas con `this.$refs`.

22.1. Lab: referencia a un audio

En este laboratorio vamos a ver como usar las referencias de Vue para acceder a una etiqueta audio y poder reproducir y pausar dicho audio.

Empezamos creando un proyecto con **vue-cli**.

```
$ vue create vuejs-referencias-lab
```

Seleccionamos la configuración por defecto.

```
? Please pick a preset: (Use arrow keys)
❯ default (babel, eslint)
  Manually select features
```

Ahora vamos a eliminar el componente `HelloWorld` y quitar las referencias a este componente del `App` dejando este último de la siguiente forma:

/vuejs-referencias-lab/src/App.vue

```
<template>
  <div>

  </div>
</template>

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

<style>
</style>
```

Ahora vamos a crearnos un componente nuevo `CmpReferencias` dentro de la carpeta `components`. Además vamos a importarlo y mostrarlo en el `App`.

/vuejs-referencias-lab/src/App.vue

```
<template>
  <div>
    <CmpReferencias />
  </div>
</template>

<script>
import CmpReferencias from './components/CmpReferencias';

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

<style>
</style>
```

Una vez importado, vamos a rellenar el nuevo componente y vamos a añadir en la parte del HTML una etiqueta `audio` con dos botones, uno para reproducir la canción, y otro para pausarla.

/vuejs-referencias-lab/src/components/CmpReferencias.vue

```
<template>
  <div>
    <h2>Referencias</h2>
    <audio src="../assets/sonido-piolin.mp3"></audio>
    <button type="button">Reproducir</button>
    <button type="button">Pausar</button>
  </div>
</template>

<script>
export default {

}
</script>

<style>
</style>
```

Una vez que tenemos el HTML, vamos a añadir dos métodos a los botones donde meteremos la lógica para trabajar con el audio.

/vuejs-referencias-lab/src/components/CmpReferencias.vue

```
<template>
  <div>
    <h2>Referencias</h2>
    <audio src="../assets/sonido-piolin.mp3"></audio>
    <button type="button" @click="reproducir">Reproducir</button>
    <button type="button" @click="pausar">Pausar</button>
  </div>
</template>

<script>
export default {
  methods: {
    reproducir() {

    },
    pausar() {

    }
  }
}
</script>

<style>
</style>
```

Una vez que tenemos todas las partes del componente, nos falta ver como acceder a los métodos de la etiqueta audio, y para hacer esto tenemos que añadirle una referencia a dicha etiqueta.

Para crear una referencia solo hay que asignarle un atributo `ref` a la etiqueta con la que queramos interactuar y asignarle un nombre a ese atributo.

/vuejs-referencias-lab/src/components/CmpReferencias.vue

```
<template>
  <div>
    <h2>Referencias</h2>
    <audio ref="audioRef" src="../assets/sonido-piolin.mp3"></audio>
    <button type="button" @click="reproducir">Reproducir</button>
    <button type="button" @click="pausar">Pausar</button>
  </div>
</template>

<script>
export default {
  methods: {
    reproducir() {

    },
    pausar() {

    }
  }
}
</script>

<style>
</style>
```

Una vez que tenemos la referencia, si usamos `this.$refs` podemos acceder a las referencias que se hayan creado, y interactuar con esos elementos.

/vuejs-referencias-lab/src/components/CmpReferencias.vue

```
<template>
  <div>
    <h2>Referencias</h2>
    <audio ref="audioRef" src="../assets/sonido-piolin.mp3"></audio>
    <button type="button" @click="reproducir">Reproducir</button>
    <button type="button" @click="pausar">Pausar</button>
  </div>
</template>

<script>
export default {
  methods: {
    reproducir() {
      this.$refs.audioRef.play();
    },
    pausar() {
      this.$refs.audioRef.pause();
    }
  }
}
</script>

<style>
</style>
```

Y con esto, si pulsamos los botones, el sonido debería de empezar a escucharse y pausarse.

Capítulo 23. Componentes dinámicos

Los **componentes dinámicos** nos permiten cambiar el componente que se está mostrando cambiando el valor de una propiedad del componente en el que se van a mostrar. A esta propiedad le daremos el nombre del componente que queremos mostrar.

Para usar los componentes dinámicos, tenemos que usar la etiqueta `component` a la cual le pasamos un atributo `is` cuyo valor será el nombre del componente que se tiene que mostrar. Según vaya cambiando el valor de la propiedad asignada a `is`, se irán creando y destruyendo los componentes.

En este caso vamos a crear un formulario *multistep*, y para ello empezamos por crear los siguientes componentes:

/src/components/ComponentesDinamicos/Form1.vue

```
<template>
  <fieldset>
    <legend>Login: 1/2</legend>
    <div>
      <label for="email">Email:</label>
      <input type="email" id="email" name="email" ref="emailInput" :value="email" >
    </div>
  </fieldset>
</template>

<script>
export default {
  props: ['email'],
  beforeDestroy() {
    this.$emit('saveState', ['Form1', this.$refs.emailInput.value])
  }
}
</script>
```

/src/components/ComponentesDinamicos/Form2.vue

```
<template>
  <fieldset>
    <legend>Login: 2/2</legend>
    <div>
      <label for="password">Contraseña:</label>
      <input type="password" id="password" name="password" ref="pwInput"
:value="password" @change="cambiarValor($event.target.value)" >
    </div>
  </fieldset>
</template>


<script>
export default {
  props: ['password'],
  methods: {
    cambiarValor(nuevoValor) {
      this.$emit('saveState', ['Form2', nuevoValor])
    }
  },
  beforeDestroy() {
    this.cambiarValor(this.$refs.pwInput.value)
  },
}
</script>
```

Una vez que tenemos los distintos pasos del formulario, toca crear el componente que irá mostrando estos otros usando la etiqueta `component`.

/src/components/ComponentesDinamicos/ComponentesDinamicos.vue

```
<template>
  <div>
    <h1>Componentes dinámicos</h1>
    <component :is="stepNumber" v-bind:[getPropKey]="getPropVal"
@saveState="changeData"></component>
    <button type="button" v-if="currentStep > 1" @click="changeStep(-
1)">Anterior</button>
    <button type="button" v-if="currentStep < totalSteps"
@click="changeStep(1)">Siguiente</button>
    <button type="button" v-if="currentStep == totalSteps"
@click="login">Login</button>
  </div>
</template>

<script>
import Form1 from './Form1.vue';
import Form2 from './Form2.vue';
```

```
export default {
  components: {
    Form1,
    Form2
  },
  data() {
    return {
      currentStep: 1,
      totalSteps: 2,
      Form1: {
        key: 'email',
        val: 'angel',
      },
      Form2: {
        key: 'password',
        val: '1234',
      }
    }
  },
  computed: {
    stepNumber() {
      return 'Form' + this.currentStep;
    },
    getPropKey() {
      return this[this.stepNumber].key;
    },
    getPropVal() {
      return this[this.stepNumber].val;
    }
  },
  methods: {
    changeStep(num) {
      this.currentStep += num;
    },
    login() {
      console.log(this.Form1.val)
      console.log(this.Form2.val)
    },
    changeData(event) {
      const [cmpName, newVal] = event;
      this[cmpName].val = newVal;
    }
  }
}
</script>
```

Como estos componentes pierden su estado cuando se destruyen entonces tenemos que actualizar el valor en el componente que los va a ir mostrando justo antes de eliminarlos por si acaso necesitamos volver unos pasos atrás para cambiar alguno de los campos que ya se habían

rellenado.

23.1. keep-alive

Vue tiene una etiqueta que funciona con los componentes dinámicos, y que va a evitar que se destruyan y se vuelvan a crear cada vez que se van cambiando. Esta etiqueta es `keep-alive` y tiene que envolver a la etiqueta `component` que hemos usado.

/src/components/ComponentesDinamicos/ComponentesDinamicos.vue

```
<template>
  <div>
    <h1>Componentes dinámicos (keep-alive)</h1>
    <button type="button" v-if="currentStep > 1" @click="changeStep(-
1)">Anterior</button>
    <button type="button" v-if="currentStep < totalSteps"
@click="changeStep(1)">Siguiente</button>
    <keep-alive>
      <component :is="stepNumber" :prop="stepNumber"></component>
    </keep-alive>
  </div>
</template>

<script>
import Form1 from './Form1.vue';
import Form2 from './Form2.vue';

export default {
  components: {
    Form1,
    Form2
  },
  data() {
    return {
      currentStep: 1,
      totalSteps: 2,
    }
  },
  computed: {
    stepNumber() {
      return 'Form' + this.currentStep;
    }
  },
  methods: {
    changeStep(num) {
      this.currentStep += num;
    }
  }
}
</script>
```

Ahora estos componentes se van a quedar guardados en memoria por lo tanto no perderemos el estado al ir cambiando de componente dinámicamente. Ahora si necesitamos volver hacía atrás para editar algún dato que ya habíamos rellenado, no tendremos ningún problema ni será necesario ir guardando el estado de estos componentes.

Como no se van a crear y destruir, entonces no se van a ejecutar los métodos `created` y `destroyed` del ciclo de vida, sino que ahora entran en juego otros dos métodos que son `activated` y `deactivated`.

/src/components/ComponentesDinamicos/Form1.vue

```
<template>
  <fieldset>
    <legend>Login: 1/2</legend>
    <div>
      <label for="email">Email:</label>
      <input type="email" id="email" name="email" ref="emailInput" >
    </div>
    <button type="button" @click="guardarCampo">Guardar</button>
  </fieldset>
</template>

<script>
export default {
  methods: {
    guardarCampo() {
      console.log(this.$refs.emailInput.value)
    }
  }
}
</script>
```

/src/components/ComponentesDinamicos/Form2.vue

```
<template>
  <fieldset>
    <legend>Login: 2/2</legend>
    <div>
      <label for="password">Contraseña:</label>
      <input type="password" id="password" name="password" ref="pwInput" >
    </div>
    <button type="button" @click="guardarCampo">Guardar</button>
  </fieldset>
</template>

<script>
export default {
  methods: {
    guardarCampo() {
      console.log(this.$refs.pwInput.value)
    }
  }
}
</script>
```

Capítulo 24. Componentes asíncronos (lazy loading)

Muchas veces, con aplicaciones que son muy grandes, no vamos a querer cargar todos los componentes de una vez al entrar en la página por ejemplo porque la mayoría de los usuarios no llegan a entrar en estos componentes, o porque queremos que la primera carga no dure demasiado para no hacer de esperar al usuario y para ello solo bajaremos los componentes con funcionalidades principales. El resto de componentes solo se van a descargar según se vayan necesitando.

Para conseguir que los componentes se carguen cuando se vayan a mostrar usaremos la **importación dinámica**. En lugar de importar un componente como se ha hecho hasta ahora, vamos a usar una función en la que se realizará la importación con `import('ruta-al-componente')`. De esta forma cuando se cargue el componente padre, como no se ha importado directamente como el resto de componentes que hemos creado hasta ahora, entonces no se lo descargará en la descarga inicial de la aplicación.

24.1. Lab: Componente asíncrono

En este laboratorio vamos a ver como cargar un componente de forma diferida, en lugar de hacerlo como hasta ahora al descargar la aplicación entera.

Empezamos por crear el proyecto lanzando el siguiente comando y seleccionamos la configuración por defecto.

```
$ vue create vuejs-componentes-asincronos-lab
```

Una vez creado el proyecto, eliminamos el componente `HelloWorld` y dejamos vacío el componente `App`.

Vamos a crear un componente `CmpPerezoso` que va a ser el vamos a cargar de forma asíncrona.

/vuejs-componentes-asincronos-lab/src/components/CmpPerezoso.vue

```
<template>
  <h1>Componente cargado de forma diferida</h1>
</template>

<script>
export default {

}
</script>

<style>

</style>
```

Una vez que lo tenemos, nos vamos a nuestro componente raíz en el que vamos a añadir un botón que cambie un dato booleano de valor cada vez que se pulsa para mostrar/ocultar el componente. Inicialmente tiene que aparecer oculto, por tanto le daremos como valor `false`.

/vuejs-componentes-asincronos-lab/src/App.vue

```
<template>
  <div>
    <button type="button" @click="mostrar = !mostrar">{{ mostrar ? 'Ocultar' : 'Mostrar' }}</button>
  </div>
</template>

<script>
export default {
  data() {
    return {
      mostrar: false
    }
  }
}
</script>

<style>

</style>
```

Ahora vamos a importar el componente perezoso dentro de la propiedad donde definimos los componentes locales, y para ello, le vamos a pasar una función en la que se va a llamar a `import('ruta-del-componente')` para que solo se importe cuando se vaya a cargar el componente en la vista.

/vuejs-componentes-asincronos-lab/src/App.vue

```
<template>
  <div>
    <button type="button" @click="mostrar = !mostrar">{{ mostrar ? 'Ocultar' : 'Mostrar' }}</button>
  </div>
</template>

<script>
export default {
  data() {
    return {
      mostrar: false
    }
  },
  components: {
    CmpPerezoso: () => import('./components/CmpPerezoso')
  }
}
</script>

<style>

</style>
```

Para terminar, vamos a poner este componente con la directiva `v-if` para que solo se cargue cuando la condición pasada a esta directiva sea `true`.

/vuejs-componentes-asincronos-lab/src/App.vue

```
<template>
  <div>
    <button type="button" @click="mostrar = !mostrar">{{ mostrar ? 'Ocultar' : 'Mostrar' }}</button>
    <CmpPerezoso v-if="mostrar" />
  </div>
</template>

<script>
export default {
  data() {
    return {
      mostrar: false
    }
  },
  components: {
    CmpPerezoso: () => import('./components/CmpPerezoso')
  }
}
</script>

<style>

</style>
```

Ahora cuando entremos en la aplicación, al pulsar el botón de **Mostrar** será cuando se va a intentar

mostrar el componente, pero como todavía no se había descargado, se descargará y una vez que haya terminado la descarga se mostrará.

Para comprobar que se descarga este componente, en la pestaña de **Network** de las herramientas del desarrollador podemos ver que se descarga un nuevo archivo de javascript `0.js` cuando le damos al botón de *Mostrar* por primera vez.

O también podemos ver como aparece ese mismo archivo en la pestaña de `sources > top > localhost:8080`.

Capítulo 25. Routing: vue-router

Las rutas permiten cambiar entre las distintas secciones de la aplicación, y dependiendo de la ruta se cargan unos componentes u otros. El Router de Vue parsea la URL e intenta identificar el componente que tiene que cargar teniendo en cuenta los datos que van en la ruta en caso de que los tenga.

Para poder usar el routing con Vue, necesitamos instalar un módulo aparte ya que Vue no tiene rutas.

```
$ npm install --save vue-router
```

Una vez instalado el módulo, ya podemos crearnos una instancia del Router en el `main.js`.

/main.js

```
import Vue from 'vue'
import App from './App.vue'
import Router from 'vue-router';

Vue.use(Router);

const router = new Router({
  // aquí van nuestras rutas
});

new Vue({
  el: '#app',
  router,
  render: h => h(App)
})
```

Nos vamos a crear un archivo en el que vamos a tener las rutas, y estas rutas estarán guardadas en un array.

Cada ruta será un objeto en el que tendremos unas propiedades entre las cuales necesitamos el `path` donde pondremos la ruta, y el `component` donde vamos a poner que componente se tiene que renderizar cuando entremos en la URL que coincide con el `path`.

/routes.js

```
import Home from './components/Home.vue';
import NuevoUsuario from './components/NuevoUsuario.vue';

export const routes = [
  { path: '/', component: Home },
  { path: '/nuevo-usuario', component: NuevoUsuario },
];
```

Una vez que hemos definido las rutas, las añadimos en el objeto de opciones de la instancia del *Router*.

/main.js

```
import Vue from 'vue'
import App from './App.vue'
import Router from 'vue-router';
import { routes } from './routes';

Vue.use(Router);

const router = new Router({
  routes
});

new Vue({
  el: '#app',
  router,
  render: h => h(App)
})
```

Ahora que tenemos vamos a crear los componentes que hemos puesto en las rutas, y unos enlaces en el componente donde se irán renderizando los componentes según las rutas en las que vayamos entrando.

/components/Home.vue

```
<template>
  <div>
    <h2>Inicio</h2>
  </div>
</template>
```

/components/NuevoUsuario.vue

```
<template>
  <div>
    <h2>Nuevo Usuario</h2>
  </div>
</template>
```

/App.vue

```
<template>
  <div>
    <h1>Routing</h1>
    <a href="/">Home</a>
    <a href="/nuevo-usuario">NuevoUsuario</a>
  </div>
</template>

<script>
import NuevoUsuario from './components/NuevoUsuario.vue';
import Home from './components/Home.vue';

export default {
  components: {
    appHome: Home,
    appNuevoUsuario: NuevoUsuario
  }
}
</script>
```

Una vez que tenemos los componentes, podemos ver que no se muestran en ningún lado cuando pulsamos sobre los enlaces, esto es debido a que Vue no sabe donde los tiene que pintar. Para indicarle a Vue el lugar donde tiene que renderizar los componentes vamos a usar la etiqueta `router-view`.

/App.vue

```
<template>
  <div>
    <h1>Routing</h1>
    <a href="/">Home</a>
    <a href="/nuevo-usuario">NuevoUsuario</a>
    <hr>
    <router-view></router-view>
  </div>
</template>

<script>
import NuevoUsuario from './components/NuevoUsuario.vue';
import Home from './components/Home.vue';

export default {
  components: {
    appHome: Home,
    appNuevoUsuario: NuevoUsuario
  }
}
</script>
```

Sin embargo, siguen sin funcionar bien las rutas, solo se muestra el componente asociado a la ruta raíz, y esto se debe a que no se está usando el Router en el modo correcto.

Se esta usando el modo `Hash`, y por eso aparece el signo `#` en la URL de la página.

En lugar del modo `Hash`, vamos a usar el modo `History` que es el modo al que estamos acostumbrados. Para cambiar el modo, solo hay que indicarselo al *Router* en el objeto de opciones donde hemos puesto nuestras rutas.

/main.js

```
import Vue from 'vue'
import App from './App.vue'
import Router from 'vue-router';
import { routes } from './routes';

Vue.use(Router);

const router = new Router({
  routes,
  mode: 'history'
});

new Vue({
  el: '#app',
  router,
  render: h => h(App)
})
```

Con esto ya podemos navegar entre nuestras rutas, sin problema, pero al cambiar de ruta, se refresca la página entera con lo que perderiamos el estado de la aplicación, por tanto en lugar de usar las etiquetas `a`, vamos a usar la etiqueta `router-link` que nos proporciona el Router. A esta etiqueta le pasamos la ruta a la que queremos ir en la propiedad `to` en lugar del `href`.

/App.vue

```
<template>
  <div>
    <h1>Routing</h1>
    <router-link to="/">Home</router-link>
    <router-link to="/nuevo-usuario">Nuevo Usuario</router-link>
    <hr>
    <router-view></router-view>
  </div>
</template>

<script>
import NuevoUsuario from './components/NuevoUsuario.vue';
import Home from './components/Home.vue';

export default {
  components: {
    appHome: Home,
    appNuevoUsuario: NuevoUsuario
  }
}
</script>
```

Con esto ya podemos navegar por la aplicación sin problemas.

25.1. Navegación por código

Hay veces en las que necesitamos cambiar de ruta, pero no a través de la interfaz, sino que necesitamos hacerlo a través del código. Por ejemplo cuando guardamos un nuevo dato en la BBDD y en lugar de quedarnos en la página queremos ir a la página donde se muestra la información del elemento que hemos guardado.

Desde los componentes tenemos acceso al router, y lo único que hay que hacer, es añadirle la ruta a la pila de rutas por las que hemos ido pasando.

/components/NuevoUsuario.vue

```
<template>
  <div>
    <h2>Nuevo Usuario</h2>
    <button type="button" @click="navegarAInicio">Guardar</button>
  </div>
</template>

<script>
export default {
  methods: {
    navegarAInicio() {
      this.$router.push('/');
    }
  }
}
</script>
```

25.2. Redireccionar rutas

En las aplicaciones hay veces que cuando entramos en la ruta raíz donde se muestra una lista de los datos que manejamos, necesitamos que en la ruta ponga el nombre de los items que se están mostrando para saber que datos se van a ver en el componente.

Lo que podemos hacer es redireccionar a la ruta que queremos, cuando entremos en la ruta raíz, y esto lo vamos a hacer añadiendo una ruta más y en lugar de poner la propiedad `component` esta vez vamos a poner la propiedad `redirect` a la que le vamos a pasar el `path` de la ruta a la que queremos ir.

/routes.js

```
import Home from './components/Home.vue';
import NuevoUsuario from './components/NuevoUsuario.vue';

export const routes = [
  { path: '/', component: Home },
  { path: '/nuevo-usuario', component: NuevoUsuario },
  { path: '/to-home', redirect: '/' },
];
```

Ahora cambiamos los enlaces para que apunten a la ruta correcta, y nos ahorramos la redirección a través de los enlaces.

/App.vue

```
<template>
  <div>
    <h1>Routing</h1>
    <!-- <a href="/">Home</a>
    <a href="/nuevo-usuario">NuevoUsuario</a> -->
    <router-link to="/usuarios">Home</router-link>
    <router-link to="/nuevo-usuario">Nuevo Usuario</router-link>
    <hr>
    <router-view></router-view>
  </div>
</template>

<script>
import NuevoUsuario from './components/NuevoUsuario.vue';
import Home from './components/Home.vue';

export default {
  components: {
    appHome: Home,
    appNuevoUsuario: NuevoUsuario
  }
}
</script>
```

/components/NuevoUsuario.vue

```
<template>
  <div>
    <h2>Nuevo Usuario</h2>
    <button type="button" @click="navegarAInicio">Guardar</button>
  </div>
</template>

<script>
export default {
  methods: {
    navegarAInicio() {
      this.$router.push('/');
    }
  }
}
</script>
```

25.3. Rutas con parámetros

Las rutas las podemos personalizar un poco haciendolas dinámicas. De momento en la ruta `/usuarios` nos muestra todos los usuarios que tenemos, pero en la mayoria de las aplicaciones vamos a necesitar ver la información de uno en concreto, y para ello le tendremos que pasar un parámetro a la URL que será el *id*.

Estos parámetros se definen en el propio path de la ruta poniendole `:nombreParam` donde vayamos a necesitar un valor que puede cambiar.

/routes.js

```
import Home from './components/Home.vue';
import NuevoUsuario from './components/NuevoUsuario.vue';

export const routes = [
  { path: '/usuarios/:id', component: Home },
  { path: '/nuevo-usuario', component: NuevoUsuario },
  { path: '/', redirect: '/usuarios/2' },
];
```

Vamos a añadir unos enlaces para ver como obtener el valor de los parámetros en los componentes.

/App.vue

```
<template>
  <div>
    <h1>Routing</h1>
    <router-link to="/usuarios">Home</router-link>
    <router-link to="/nuevo-usuario">Nuevo Usuario</router-link>
    <router-link to="/usuarios/1">Usuario 1</router-link>
    <router-link to="/usuarios/2">Usuario 2</router-link>
    <hr>
    <router-view></router-view>
  </div>
</template>

<script>
import NuevoUsuario from './components/NuevoUsuario.vue';
import Home from './components/Home.vue';

export default {
  components: {
    appHome: Home,
    appNuevoUsuario: NuevoUsuario
  }
}
</script>
```

Si le damos a los enlaces vemos como cambia la URL cambiando el id. Para obtener ese parámetro, lo que hay que hacer es acceder a `this.$route.params` que es donde nos llegan los distintos parámetros que les pasamos a las rutas.

/components/Home.vue

```
<template>
  <div>
    <h2>Inicio</h2>
    <p>Usuario {{idUsuario}}</p>
  </div>
</template>

<script>
export default {
  data: function() {
    return {
      idUsuario: this.$route.params.id
    };
  }
}
</script>
```

Ahora ocurre un problema y es que cada vez que cambiamos entre el *usuario 1* y el *usuario 2*, la

URL cambia pero el *id* que estamos mostrando no lo hace.

Esto se debe a que Vue detecta que estamos en el mismo componente y no lo destruye para volver a crearlo, por lo que no vuelve a crearlo y no coge el nuevo *id* que le debería de llegar.

Para solucionar esto, podemos añadir un **watcher** al objeto `$route` de esta forma, cada vez que se detecte un cambio en ese objeto, se va a ejecutar una función en la que asignaremos otra vez el valor del *id*. En la función del *watcher* vamos a recibir como parámetros dos objetos, el primero es la ruta destino, y el segundo es la ruta origen.

/components/Home.vue

```
<template>
  <div>
    <h2>Inicio</h2>
    <p>Usuario {{idUsuario}}</p>
  </div>
</template>

<script>
export default {
  data: function() {
    return {
      idUsuario: this.$route.params.id
    };
  },
  watch: {
    '$route'(to, from) {
      this.idUsuario = to.params.id;
    }
  }
}
</script>
```

Otra forma posible de arreglar este problema es indicar en las rutas que queremos que los parámetros le lleguen en forma de propiedades al componente. Para ello solo hay que añadir una propiedad `props: true` en la ruta correspondiente.

/routes.js

```
import Home from './components/Home.vue';
import NuevoUsuario from './components/NuevoUsuario.vue';

export const routes = [
  { path: '/usuarios/:id', component: Home, props: true },
  { path: '/nuevo-usuario', component: NuevoUsuario },
  { path: '/', redirect: '/usuarios/3' },
];
```

También hay que indicarle al componente que va a recibir unas propiedades.

/components/Home.vue

```
<template>
  <div>
    <h2>Inicio</h2>
    <p>Usuario {{idUsuario}}</p>
  </div>
</template>

<script>
export default {
  props: ['id'],
  data: function() {
    return {
      idUsuario: this.$route.params.id
    };
  },
  watch: {
    '$route'(to, from) {
      this.idUsuario = to.params.id;
    }
  }
}
</script>
```

25.4. Rutas hijas

Las rutas hijas o anidadas son aquellas que tienen una parte del `path` en común, y se van a anidar para evitar poner esa parte en las del hijo. Además nos permitirá mostrar los componentes hijos, dentro del componente padre.

Para crear estas rutas vamos a añadir una propiedad `children` en el componente al que le vamos a poner las rutas anidadas. El valor de esta propiedad va a ser un array en el que se van a definir las rutas. En estas rutas no hay que poner la parte del `path` que ya contiene el padre, porque el `path` de hijo lo concatenara al del padre.

/routes.js

```
import Home from './components/Home.vue';
import NuevoUsuario from './components/NuevoUsuario.vue';
import InfoUsuario from './components/InfoUsuario.vue';
import EditUsuario from './components/EditUsuario.vue';

const rutasUsuarios = [
  { path: ':id/info', component: InfoUsuario, props: true },
  { path: ':id/edit', component: EditUsuario },
];

export const routes = [
  { path: '/usuarios', component: Home, children: rutasUsuarios },
  { path: '/nuevo-usuario', component: NuevoUsuario },
  { path: '/', redirect: '/usuarios' },
];
```

Ahora vamos a añadir los enlaces correspondientes a estas rutas, y vamos a crear los componentes correspondientes.

/App.vue

```
<template>
  <div>
    <h1>Routing</h1>
    <router-link to="/usuarios">Home</router-link>
    <router-link to="/nuevo-usuario">Nuevo Usuario</router-link>
    <hr>
    <router-view></router-view>
  </div>
</template>

<script>
import NuevoUsuario from './components/NuevoUsuario.vue';
import Home from './components/Home.vue';

export default {
  components: {
    appHome: Home,
    appNuevoUsuario: NuevoUsuario
  }
}
</script>
```

/components/Home.vue

```
<template>
  <div>
    <h2>Inicio</h2>
    <ul>
      <li>Usuario 1
        <router-link to="/usuarios/1/info">Información</router-link>
        <router-link to="/usuarios/1/edit">Editar</router-link>
      </li>
      <li>Usuario 2
        <router-link to="/usuarios/2/info">Información</router-link>
        <router-link to="/usuarios/2/edit">Editar</router-link>
      </li>
    </ul>
    <hr>
    <router-view></router-view>
  </div>
</template>

<script>
export default { }
</script>
```

/components/EditUsuario.vue

```
<template>
  <div>
    <h3>Editar</h3>
  </div>
</template>
```

/components/InfoUsuario.vue

```
<template>
  <div>
    <h3>Info Usuario: {{id}}</h3>
  </div>
</template>

<script>
export default {
  props: ['id']
}
</script>
```

Con esto ya podríamos navegar entre los distintos componentes, y si entramos al componente de Información de cualquiera de los usuarios que aparecen, se puede ver que no desaparece la lista, porque al estar anidadas las rutas, podemos hacer que se muestre el componente padre al mismo

tiempo que uno de los componentes hijos.

25.5. Rutas con nombres

A nuestras rutas les podemos poner un nombre y de esta forma cada vez que vayamos a indicar que tenemos que ir a una ruta concreta podemos usar este nombre que le hemos puesto en lugar de formar el path.

Para ponerles nombres a las rutas, solo hay que poner el atributo `name` con el nombre como valor en las rutas.

/routes.js

```
import Home from './components/Home.vue';
import NuevoUsuario from './components/NuevoUsuario.vue';
import InfoUsuario from './components/InfoUsuario.vue';
import EditUsuario from './components/EditUsuario.vue';

const rutasUsuarios = [
  { path: ':id/info', component: InfoUsuario, props: true },
  { path: ':id/edit', component: EditUsuario },
];

export const routes = [
  { path: '/usuarios', component: Home, children: rutasUsuarios, name: 'home' },
  { path: '/nuevo-usuario', component: NuevoUsuario, name: 'nuevoUsuario' },
  { path: '/', redirect: '/usuarios' },
];
```

Una vez que se han puesto los nombres, ya no hace falta formar a mano el `path` que hay que pasar como valor en los enlaces o en la navegación por código, sino que le pasamos un objeto con el nombre de la ruta a la que queremos ir.

/App.vue

```
<template>
  <div>
    <h1>Routing</h1>
    <router-link to="/usuarios">Home</router-link>
    <router-link to="/nuevo-usuario">Nuevo Usuario</router-link>
    <router-link :to="{ name: 'nuevoUsuario' }">Nuevo Usuario (ruta con
nombre)</router-link>
    <hr>
    <router-view></router-view>
  </div>
</template>

<script>
import NuevoUsuario from './components/NuevoUsuario.vue';
import Home from './components/Home.vue';

export default {
  components: {
    appHome: Home,
    appNuevoUsuario: NuevoUsuario
  }
}
</script>
```

/components/NuevoUsuario.vue

```
<template>
  <div>
    <h2>Nuevo Usuario</h2>
    <button type="button" @click="navegarAInicio">Guardar</button>
  </div>
</template>

<script>
export default {
  methods: {
    navegarAInicio() {
      this.$router.push({ name: 'home' });
    }
  }
}
</script>
```

Si una ruta necesita unos parámetros y queremos usar su nombre para navegar hasta ella, solo hay que pasarle un objeto `params` con los parámetros necesarios.

/routes.js

```
import Home from './components/Home.vue';
import NuevoUsuario from './components/NuevoUsuario.vue';
import InfoUsuario from './components/InfoUsuario.vue';
import EditUsuario from './components/EditUsuario.vue';

const rutasUsuarios = [
  { path: ':id/info', component: InfoUsuario, props: true, name: 'info' },
  { path: ':id/edit', component: EditUsuario },
];

export const routes = [
  { path: '/usuarios', component: Home, children: rutasUsuarios, name: 'home' },
  { path: '/nuevo-usuario', component: NuevoUsuario, name: 'nuevoUsuario' },
  { path: '/', redirect: '/usuarios' },
];
```

/components/Home.vue

```
<template>
  <div>
    <h2>Inicio</h2>
    <ul>
      <li>Usuario 1
        <router-link to="/usuarios/1/info">Información</router-link>
        <router-link :to="{ name: 'info', params: {id: 1}}">Información (ruta con
nombre)</router-link>
        <router-link to="/usuarios/1/edit">Editar</router-link>
      </li>
      <li>Usuario 2
        <router-link to="/usuarios/2/info">Información</router-link>
        <router-link :to="{ name: 'info', params: {id: 2}}">Información (ruta con
nombre)</router-link>
        <router-link to="/usuarios/2/edit">Editar</router-link>
      </li>
    </ul>
    <hr>
    <router-view></router-view>
  </div>
</template>

<script>
export default { }
</script>
```

También nos facilita la vida en caso de necesitar cambiar el `path` en la ruta, ya que no tendremos que ir cambiandolo en todos los enlaces que tengamos por la aplicación.

25.6. Guards

Las guards se usan para controlar a que partes de la aplicación puede navegar un usuario. Por ejemplo, es posible que algunas de las rutas de la aplicación las queramos restringir solo a usuarios logueados, y podemos usar estas guards para controlar las condiciones en las que pueden o no entrar o salir de una ruta.

Tenemos tres tipos de guards:

- beforeRouteEnter
- beforeRouteLeave
- beforeRouteUpdate

25.6.1. BeforeRouteEnter

Esta guarda se va a ejecutar antes de entrar a la ruta, y desde esta no vamos a tener acceso a los datos que hay en el componente porque todavía no se ha montado.

Este método o guard lo ponemos en el componente como si fuera un método del ciclo de vida, y le llegan tres parámetros:

- to: ruta destino.
- from: ruta origen.
- next: función que hay que ejecutar para continuar con el montaje del componente (no le pasamos ningún valor, o le pasamos un `true`) o para indicar que no podemos entrar (le pasamos como parámetro `false`).

/components/InfoUsuario.vue

```
<template>
  <div>
    <h3>Info Usuario: {{id}}</h3>
  </div>
</template>

<script>
export default {
  props: ['id'],
  beforeRouteEnter(to, from, next) {
    next(confirm('Quiere entrar?'));
  }
}
</script>
```

25.6.2. BeforeRouteLeave

En esta guard si que tenemos acceso a los datos que hay en el componente, ya que el componente está montado y esta función se va a ejecutar cuando vayamos a salir de el.

Al igual que la anterior, esta función se pone como un método del ciclo de vida, y le llegan los mismos parámetros.

/components/EditUsuario.vue

```
<template>
  <div>
    <h2>Editar</h2>
    <button type="button" @click="guardados = !guardados">Guardar</button>
    <p>Datos guardados: {{guardados}}</p>
    <button type="button" @click="irAHome">Salir</button>
  </div>
</template>

<script>
export default {
  data: function() {
    return {
      guardados: false
    }
  },
  methods: {
    irAHome() {
      this.$router.push({ name: 'home' });
    }
  },
  beforeRouteLeave(to, from, next) {
    if (this.guardados) {
      next();
    } else {
      next(confirm('Quieres salir?'));
    }
  }
}
</script>
```

25.6.3. BeforeRouteUpdate

Esta última guard funciona como las otras dos anteriores solo que se va a ejecutar cuando los parámetros de una ruta se modifiquen. Por ejemplo podemos usarla para cuando cambiamos de una ruta a la misma pero con distintos parámetros y queremos que se vuelva a comprobar la condición.

/components/InfoUsuario.vue

```
<template>
  <div>
    <h3>Info Usuario: {{id}}</h3>
  </div>
</template>

<script>
export default {
  props: ['id'],
  beforeRouteEnter(to, from, next) {
    next(confirm('Quiere entrar?'));
  },
  beforeRouteUpdate(to, from, next) {
    next(confirm('Quiere entrar?'));
  }
}
</script>
```

25.7. Ruta comodín

La ruta comodín es una ruta que nos permite mostrar un componente en caso de que ninguna de las rutas definidas coincida con la URL.

Esta ruta se suele usar para mostrar la página de error *404* o para redireccionar a una ruta en concreto. El path de esta ruta es *.

/routes.js

```
import Home from './components/Home.vue';
import NuevoUsuario from './components/NuevoUsuario.vue';
import InfoUsuario from './components/InfoUsuario.vue';
import EditUsuario from './components/EditUsuario.vue';
import Error from './components/Error.vue';

const rutasUsuarios = [
  { path: ':id/info', component: InfoUsuario, props: true, name: 'info' },
  { path: ':id/edit', component: EditUsuario },
];

export const routes = [
  { path: '/usuarios', component: Home, children: rutasUsuarios, name: 'home' },
  { path: '/nuevo-usuario', component: NuevoUsuario, name: 'nuevoUsuario' },
  { path: '/', redirect: '/usuarios' },
  { path: '*', component: Error }
];
```

Y creamos el componente de error que mostraremos.

/components/Error.vue

```
<template>
  <h2 style="color: red">404: Page not found...</h2>
</template>
```

25.8. Query params

A veces necesitamos incluir en la URL unos parámetros opcionales para proporcionar más información, como por ejemplo para indicar a la paginación en que página nos encontramos. Los **query params** se componen de una clave y un valor, y si hay múltiples de ellos entonces van separados por `&`.

Estos parámetros los podemos concatenar directamente a la ruta pero también podemos pasarlos como un objeto, entonces será el Router quien los una y los concatene a la ruta.

/components/Home.vue

```
<template>
  <div>
    <h2>Inicio</h2>
    <ul>
      <li>Usuario 1
        <router-link to="/usuarios/1/info">Información</router-link>
        <router-link :to="{ name: 'info', params: {id: 1}}">Información (ruta con
nombre)</router-link>
        <router-link to="/usuarios/1/edit">Editar</router-link>
        <router-link :to="{ name: 'editar', params: {id: 1}, query: {guardados:
true}}">Editar (con query)</router-link>
      </li>
      <li>Usuario 2
        <router-link to="/usuarios/2/info">Información</router-link>
        <router-link :to="{ name: 'info', params: {id: 2}}">Información (ruta con
nombre)</router-link>
        <router-link to="/usuarios/2/edit">Editar</router-link>
        <router-link :to="{ name: 'editar', params: {id: 2}, query: {guardados:
true}}">Editar (con query)</router-link>
      </li>
    </ul>
    <hr>
    <router-view></router-view>
  </div>
</template>

<script>
export default { }
</script>
```

A la hora de extraer estos parámetros hacemos igual que con los parámetros de la ruta, solo que en

lugar de acceder a `params` ahora vamos a acceder a `query`.

/components/EditUsuario.vue

```
<template>
  <div>
    <h2>Editar</h2>
    <button type="button" @click="guardados = !guardados">Guardar</button>
    <p>Datos guardados: {{guardados}}</p>
    <button type="button" @click="irAHome">Salir</button>
  </div>
</template>

<script>
export default {
  data: function() {
    console.log(this.$route.query)
    return {
      guardados: this.$route.query.guardados
    }
  },
  methods: {
    irAHome() {
      this.$router.push({ name: 'home' });
    }
  },
  beforeRouteLeave(to, from, next) {
    if (this.guardados) {
      next();
    } else {
      next(confirm('Quieres salir?'));
    }
  }
}
</script>
```

Capítulo 26. Variables de entorno

Cuando trabajamos en algún proyecto nos podemos encontrar con que algunas variables deben de tener unos valores en algunos entornos, y otros valores distintos en otros entornos. Para ello podemos crear unos archivos donde pondremos estas variables de entorno y se cargarán aquellas que corresponden al entorno en el que se está ejecutando la aplicación.

Para que las variables se carguen automáticamente tenemos que ponerlas en un archivo `.env` (en la raíz del proyecto) precedidas de `VUE_APP_`. Según el nombre que le demos a este archivo de variables se cargará uno u otro dependiendo del modo en el que se este ejecutando la aplicación. Aquellos valores que queramos cargar cuando estamos en **desarrollo** tendremos que ponerlos en un archivo con el nombre de `.env.development`, pero si queremos cargarlo en **producción**, entonces el archivo debería de llamarse `.env.production`.

Una vez definidas las variables en estos archivos, dentro de la aplicación podremos acceder a ellas usando `process.env.VARIABLE`.

26.1. Lab: Variables de entorno

En este laboratorio vamos a ver como cargar unas variables para el entorno de desarrollo.

Empezamos por crear el proyecto lanzando el siguiente comando y seleccionamos la configuración por defecto.

```
$ vue create vuejs-variables-de-entorno-lab
```

Una vez creado el proyecto, eliminamos el componente `HelloWorld` y dejamos vacío el componente `App`.

Vamos a empezar por crear dos archivos con los valores para estas variables de entorno, un `.env.development` y un `.env.production` en la raíz del proyecto.

/vuejs-variables-de-entorno-lab/.env.development

```
VUE_APP_URL=http://localhost:8080
VUE_APP_API_KEY=ksjid0384-43903nde-23ds-f393fhdjk
```

/vuejs-variables-de-entorno-lab/.env.production

```
VUE_APP_URL=http://localhost:9045
VUE_APP_API_KEY=ui79wjdw-j6d6772-chsir220sd
```

Una vez que tenemos estos archivos con los valores para estas variables, vamos a crear un archivo `config.js` en el que vamos a leer estas variables y las vamos a exportar para poder usarlas en la aplicación.

/vuejs-variables-de-entorno-lab/src/config.js

```
export default {
  URL: process.env.VUE_APP_URL,
  API_KEY: process.env.VUE_APP_API_KEY
}
```

Una vez que tenemos este archivo creado, vamos a ir al componente de la aplicación para mostrar estos valores.

Dentro de este componente, tenemos que importar el objeto que hemos exportado desde el archivo de configuración que acabamos de crear, y vamos a crear unas computed props que se encarguen de mostrar los datos de las variables de entorno que se han cargado.

/vuejs-variables-de-entorno-lab/src/App.vue

```
<template>
  <div>
    <p>URL: {{url}}</p>
    <p>API_KEY: {{apiKey}}</p>
  </div>
</template>

<script>
import config from '@/config';

export default {
  computed: {
    url() {
      return config.URL;
    },
    apiKey() {
      return config.API_KEY;
    }
  }
}
</script>

<style>

</style>
```

Ahora si abrimos la aplicación, deberíamos de ver los valores que se encuentran en los archivos de entorno.

Si modificamos los archivos de las variables de entorno, o se ha levantado el servidor antes de crear dichos archivos, no se habrán cargado los valores de estas variables, y por lo tanto no podremos usarlos.

Para solucionarlo, hay que volver a levantar el servidor con el comando `npm run serve`.

En el caso de querer mostrar los valores del entorno de producción, tenemos que levantar el servidor añadiendo la opción `--mode production`, por lo tanto vamos a ir al archivo `package.json` para añadir este nuevo script.

/vuejs-variables-de-entorno-lab/package.json

```
{
  "name": "vuejs-variables-de-entorno-lab",
  "version": "0.1.0",
  "private": true,
  "scripts": {
    "serve": "vue-cli-service serve",
    "serve:prod": "vue-cli-service serve --mode production",
    "build": "vue-cli-service build",
    "lint": "vue-cli-service lint"
  },
  "dependencies": {
    "core-js": "^3.6.4",
    "vue": "^2.6.11"
  },
  "devDependencies": {
    "@vue/cli-plugin-babel": "^4.3.0",
    "@vue/cli-plugin-eslint": "^4.3.0",
    "@vue/cli-service": "^4.3.0",
    "babel-eslint": "^10.1.0",
    "eslint": "^6.7.2",
    "eslint-plugin-vue": "^6.2.2",
    "vue-template-compiler": "^2.6.11"
  },
  "eslintConfig": {
    "root": true,
    "env": {
      "node": true
    },
    "extends": [
      "plugin:vue/essential",
      "eslint:recommended"
    ],
    "parserOptions": {
      "parser": "babel-eslint"
    },
    "rules": {}
  },
  "browserslist": [
    "> 1%",
    "last 2 versions",
    "not dead"
  ]
}
```

Ahora solo tenemos que lanzar el comando siguiente y deberíamos de ver los valores de las variables de entorno añadidas para producción.

```
$ npm run serve:prod
```

Capítulo 27. Guia de estilo

Vue ha publicado una guía de estilo para el código de nuestras aplicaciones de Vue en la que se exponen distintos escenarios en los que los desarrolladores suelen cometer errores, y como sería la forma correcta de hacerlos.

Esta guía la podemos encontrar en https://es.vuejs.org/v2/style-guide.

Algunos de los casos que se comentan son:

- Definición correcta de las propiedades.
- Uso de `key` con la directiva `v-for`.
- Cuando usar `scoped` para los estilos.
- Cuando usar las abreviaciones de las directivas.
- Como comunicar componentes hijos con los componentes padres.
- ...