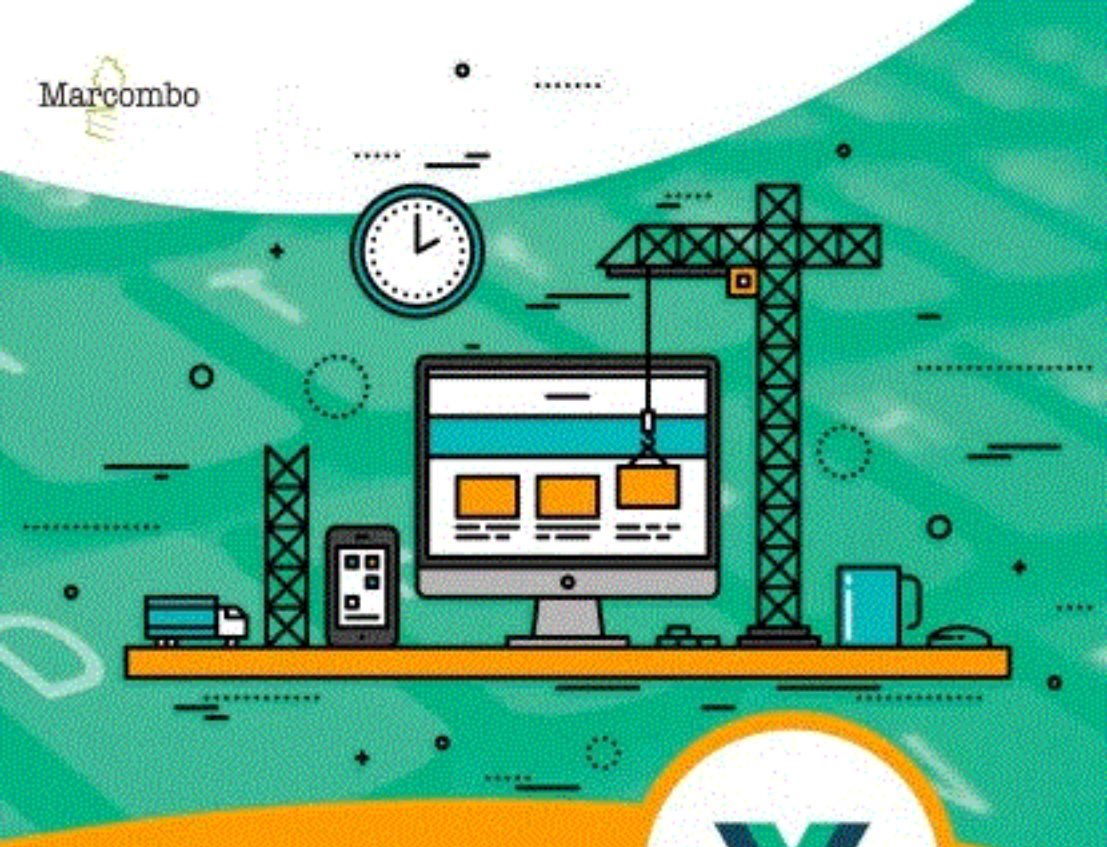
Marcombo



Aprender

VueJS

con 100 ejercicios prácticos

Ángel Vázquez Vázquez y Ramón Serrano Valero

CONTENIDOS WEB

Aprender

VueJS

con 100 ejercicios prácticos

Acceda a www.marcombo.info
para descargar gratis
el contenido adicional
complemento imprescindible de este libro

Código: **VUEJS1**

Aprender

VueJS

con 100 ejercicios prácticos

Marcombo

Aprender VueJS con 100 ejercicios prácticos

© 2020 Ángel Vazquez Vazquez y Ramón Serrano Valero

© 2020 MARCOMBO, S.L.
www.marcombo.com

Diseño de la cubierta: Giancarlo Salinas

Director de colección: Pablo Martínez Izurzu

Directora de producción: M.ª Rosa Castillo Hidalgo

Maquetación: María Paz Mora Encinas

Correctora: Laura Seoane

«Cualquier forma de reproducción, distribución, comunicación pública o transformación de esta obra sólo puede ser realizada con la autorización de sus titulares, salvo excepción prevista por la ley. Diríjase a CEDRO (Centro Español de Derechos Reprográficos, www.cedro.org) si necesita fotocopiar o escanear algún fragmento de esta obra.»

ISBN: 978-84-267-2802-9

D.L.: B-22250-2019

Impreso en Ulzama Digital

Printed in Spain

Presentación

APRENDER VUEJS CON 100 EJERCICIOS PRÁCTICOS

Los 100 capítulos que contiene este libro realizan un recorrido por los principios de desarrollo utilizando el framework de Vue. Es recomendable realizar los ejercicios en orden, sobre todo a partir del ejercicio 40, donde empieza la aplicación SPA mediante el CLI. Puede apoyarse en los ejercicios, ya implementados subidos en la plataforma Marcombo, plenamente funcionales.

Una vez finalizado este libro, el lector habrá adquirido el conocimiento para crear su propia aplicación SPA, habilitándole para afrontar proyectos en esta tecnología. Tendrá capacidad de crear aplicaciones rápidamente, mejorar su pensamiento crítico a la hora de abordar un nuevo desarrollo, así como la habilidad para tomar como base la arquitectura propuesta y poder mejorarla conforme nuevos conocimientos sean incorporados.

A QUIÉN VA DIRIGIDO

A todo aquel que busque iniciarse en el desarrollo web, la programación o adentrarse en el mundo de este nuevo framework Vue.

También está dirigido a aquellos que ya hayan realizado sus primeros pasos con Vue, pues en este libro encontrarán las respuestas a las dudas que les surjan y aquellas que no hayan podido resolver hasta ahora.

LA FORMA DE APRENDER

Nuestra experiencia en el ámbito de la enseñanza nos ha llevado a diseñar este tipo de manual, en el que cada una de las funciones se ejercita mediante la realización de un ejercicio práctico. Dicho ejercicio se halla explicado paso a paso y pulsación a pulsación, a fin de no dejar ninguna duda en su proceso de ejecución. Además, lo hemos ilustrado con imágenes descriptivas de los pasos más importantes o de los resultados que deberían obtenerse y con recuadros IMPORTANTE que ofrecen información complementaria sobre los temas tratados en los ejercicios.

LOS ARCHIVOS NECESARIOS

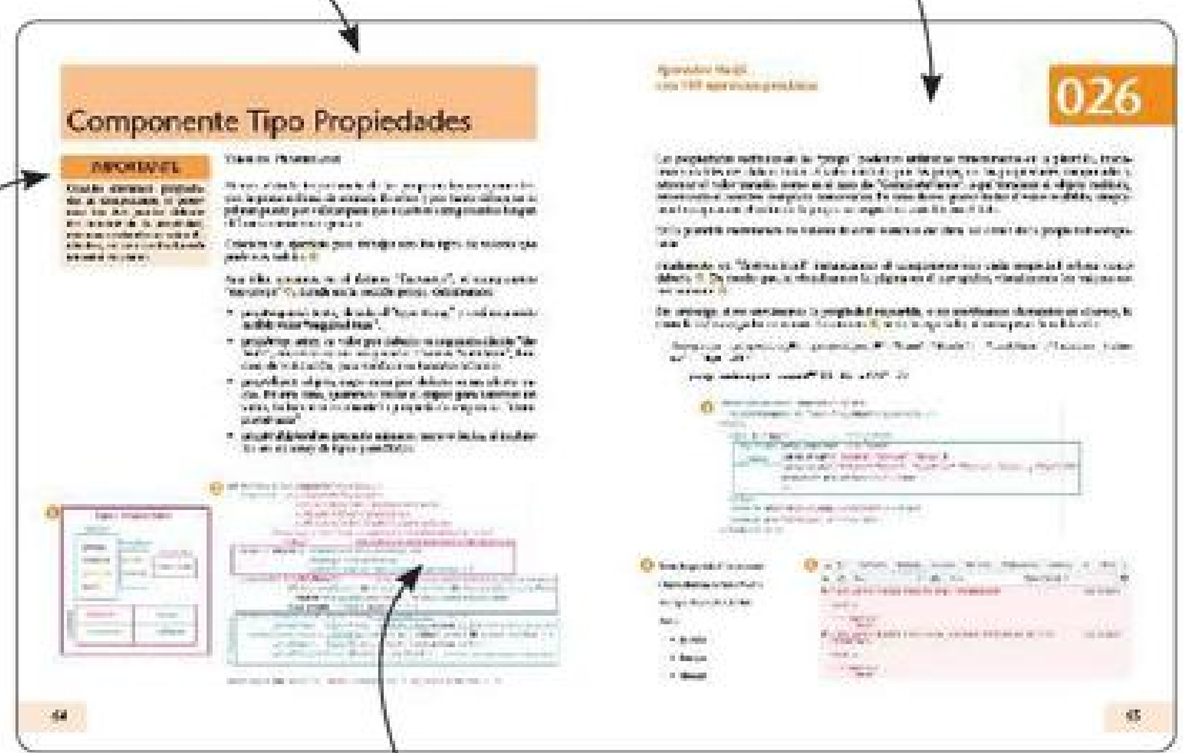
En la parte inferior de la primera página del libro encontrará el código de acceso que le permitirá descargar de forma gratuita los contenidos adicionales del libro en *www.marcombo.info.*

Cómo leer los libros "**Aprender...**"

El título de cada ejercicio expresa, sin lugar a dudas, en qué consiste este. De esta forma, si le interesa, puede acceder directamente a la acción que desea aprender o refrescar.

Puede seguir el ejercicio de forma gráfica y paso a paso. Los números colocados en las fotos le remiten a entradas en el cuerpo de texto.

El número a la derecha de la página le indica claramente en qué ejercicio se encuentra en todo momento.



Los recuadros Importante incluyen acciones que deben hacerse para asegurarse de que realiza el ejercicio correctamente. También contienen información interesante a aprender, porque le facilitará su trabajo con el programa.

Los ejercicios se han escrito sistemáticamente, paso a paso, para que nunca se pierda durante su realización.

Ramón: *Ha sido duro compaginar el mundo laboral con el Máster en Big Data y la escritura de este libro, sin embargo, gracias al apoyo de mi mujer Sara de la V., he podido sacar fuerzas para poder finalizar este libro, ¡eres mi fuerza, siempre sacas lo mejor de mi! Agradezco a mis padres por inculcarme el espíritu de esfuerzo. A Xavi y Martín, mis sobrinos, que, algún día, cuando comprendan este libro, aprendan este framework y los nuevos que aparezcan. A mi amigo Ángel V., una mente siempre activa, preparado para cualquier reto, un placer escribir junto a ti. A Gabriel G. por años compartiendo conocimientos que nos permitieron innovar. Así como a mis compañeros Miguel N. y Alonso V. por habernos aventurado juntos en este mundo Vuetizado.*

Ángel: *Este libro es para las personas que me rodean: mis padres que siempre me apoyan, mis amigos y compañeros, mis gatas que tanta compañía me hicieron a la hora de trabajar en este libro, ¡y para tí, Lucía!*
Mención especial para mi compañero Ramón con el que colaboro ya por tercera vez en este tipo de aventuras. ¡Gracias por hacer tan fácil trabajar contigo, nano!

No pudisteis estar presentes Juanto G. y Pablo F. en este libro, pero formáis parte de este equipo, esperamos abordar nuevos retos juntos de nuevo.

Índice

001 ¡Hola, Vue! 14
002 Configurar el entorno 16
003 Instancia Vue 18
004 Instancias Vue 20
005 Ciclo de vida 22
006 Data binding 24
007 Watch 26
008 Computed properties 28
009 Métodos 30
010 Componentes de la instancia 32
011 Eventos 34
012 Renderizar HTML 36
013 Vincular propiedades 38
014 Condicionales 40
015 Listas 42
016 Enumeración de propiedades 44
017 Filtros 46
018 Renderizar solo una vez 48
019 Mouse move 50
020 Directivas personalizadas 52
021 Directivas personalizadas II 54
022 Directivas personalizadas III 56
023 Directivas personalizadas IV 58
024 Componente 60
025 Componente parametrizables 62

026 Componente Tipo Propiedades 64
027 Cambiar valor de props..... 66
028 Reactividad Data 68
029 Reactividad NextTick 70
030 Directiva v-show vs v-if 72
031 Bucle v-for y key..... 74
032 Bucle atributos y rango 76
033 Peticiones Fetch 78
034 Contexto This 80
035 Promesas 82
036 Conjunto de Promesas..... 84
037 Inline templates 86
038 Slots..... 88
039 Transiciones/Animaciones I..... 90
040 Transiciones/Animaciones II..... 92
041 Instalación NodeJs 94
042 Nuevo proyecto con Vue CLI 3..... 96
043 Estructura del proyecto 98
044 Compilar en desarrollo y producción..... 100
045 Variables de entorno y ejecución..... 102
046 CLI Service scripts y servidor http 104
047 Componente – App, Librería y WC 106
048 Despliegue en Servidor de Aplicaciones 108
049 Instalación Vue Router y Modos..... 110
050 Carga óptima de Ruta y navegación 112

Índice

051 Rutas dinámicas 114
052 Rutas anidadas 116
053 Navegación desde código..... 118
054 Protección global de rutas..... 120
055 Protección en ruta y componente 122
056 Vuetify Diseño menú lateral 124
057 V-Toolbar y rutas de navegación..... 126
058 Crear Tablas..... 128
059 Selección de filas en tabla 130
060 Comunicación Hijo a Padre y Snackbar 132
061 Diseñar y visualizar ficha 134
062 Vuex, primeros pasos 136
063 Vuex – Getters 138
064 Vuex – Mutaciones..... 140
065 Vuex – Acciones 142
066 Vuex – Módulos 144
067 Vuex – Plugin Logger 146
068 Vuex – Persistencia estados al refrescar..... 148
069 Vuex – Persistencia en Cookies 150
070 Vuex – Subscriptores 152
071 Axios – Comunicación con Servidor 154
072 Axios – Cancelar solicitud..... 156
073 Axios – Interceptor de solicitudes 158
074 Axios – Interceptor Respuestas..... 160
075 Axios – Post 162

076 Axios – PUT 164
077 Axios – Delete 166
078 Comunicación – Padre – Hijo 168
079 Comunicación – Hijo – Padre 170
080 Comunicación – Hijo – Padre por Callback 172
081 Comunicación – Entre hermanos 174
082 Comunicación – Bus de eventos 176
083 Comunicación – Mensajería Global 178
084 Comunicación bidireccional 180
085 Formulario – Campo Texto 182
086 Formularios – Email con validación 184
087 Formularios – Fechas moment 186
088 Formularios – Opciones Checkbox 188
089 Formularios – Opción única Radiobutton 190
090 Formularios – Combobox 192
091 Formularios – Campo Personalizado 194
092 Formularios – Validación Vuelidate 196
093 Formularios – Validación personalizada 198
094 Pruebas Unitarias – Jest 200
095 Pruebas Unitarias – Creación 202
096 Pruebas Unitarias – Props 204
097 Pruebas Unitarias – Aislar componentes 206
098 Pruebas Unitarias – Mock de Vuex 208
099 Pruebas Unitarias – Router 210
100 Pruebas desde UI 212

¡Hola, Vue!

Los comienzos

¡Bienvenidos a Vue, queridos lectores! En este ejercicio veremos cómo empezar a trabajar con Vue con el mínimo esfuerzo, pero en primer lugar daremos algunas pinceladas sobre Vue.

¿Qué es Vue? Vue es un framework progresivo para construir interfaces de usuario ❶.

¿Qué significa progresivo? Simplemente que está modularizado de manera que su librería principal está enfocada solo a la parte visual y, usándola en conjunto con otras librerías o proyectos, puede construir fantásticas aplicaciones web SPA. Esto significa que Vue no es un todo o nada sino que puede escoger las partes que más le interesen e incluso usarlas conjuntamente con otros frameworks o librerías web.

Para ilustrar el concepto de progresivo vamos a importar lo mínimo necesario para crear nuestra primera aplicación.

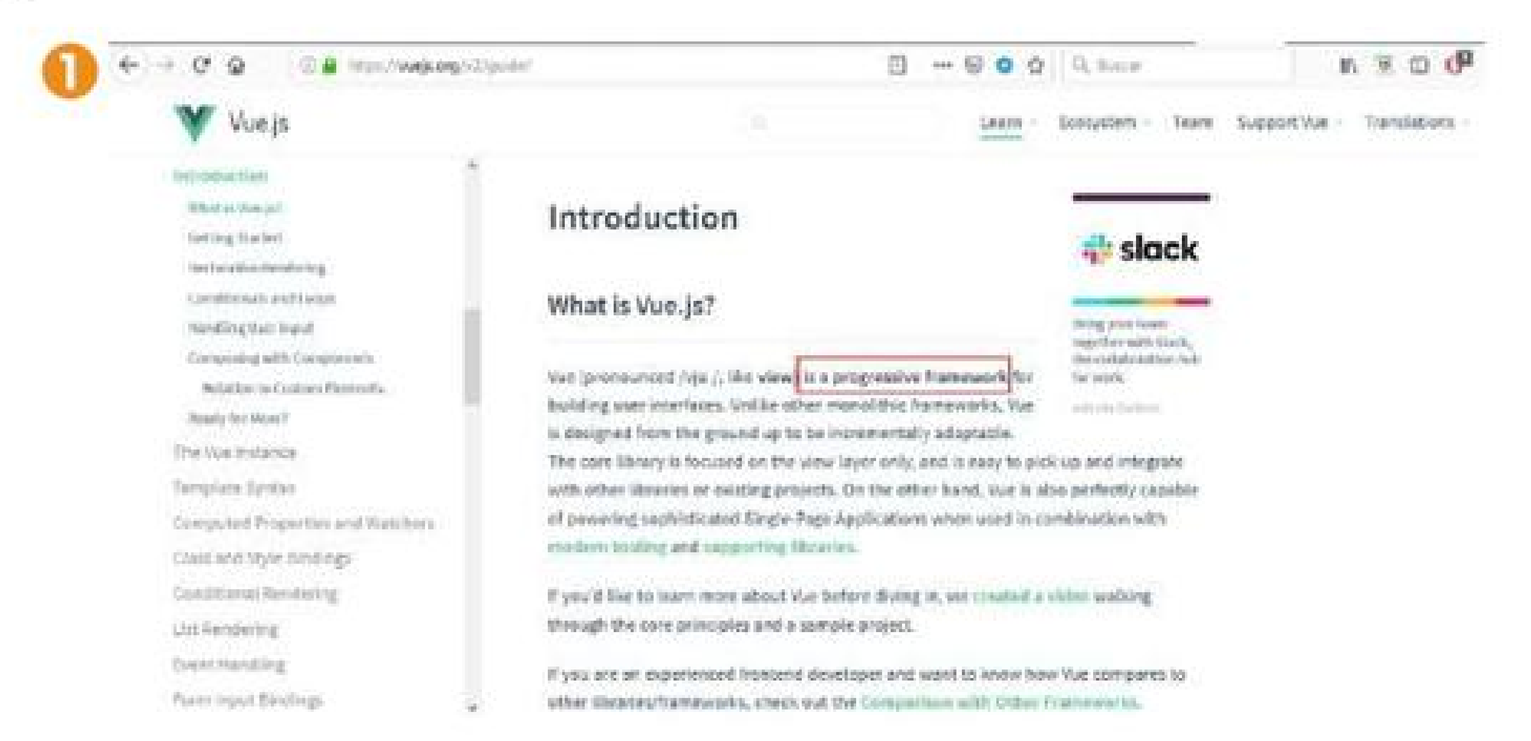
En primer lugar, vamos a utilizar un editor online como JSFiddle (https://jsfiddle.net/) y en la ventana de HTML añadiremos el siguiente contenido ❷.

Tenemos nuestro "Hola Vue" que renderiza una página de forma estática. Para trabajar de forma dinámica añadiremos los scripts necesarios para importar la librería de Vue ❸.

En la ventana del editor para el código Javascript añadimos la instancia de Vue, la ligamos al elemento con id #app y añadimos una variable en el bloque **data** denominada message❹.

Volviendo al HTML asociamos a uno de los elementos el id que hemos declarado en la parte de Javascript ❺. ¡Dentro de este elemento añadimos un título (h1) con la variable message y ya tenemos nuestro "¡Hola Vue!"

¡Así de fácil ha sido crear nuestra primera aplicación y de forma online! En siguientes ejercicios configuraremos nuestro entorno en local para trabajar cómodamente y con la ayuda de múltiples herramientas para el desarrollo web.



2

Run Save Tidy Collaborate Update New and updated JS/CSS linters

Fiddle meta

Untitled fiddle

No description

Add title to make the fiddle visible on your profile page

Resources URL cdnjs

Async requests

Other (links, license)

HTML

```
<html>

  <head>
    <meta charset="utf-8">
    <title>My Vue</title>
  </head>

  <body>
    <div>
      <h1>Hola Vue!</h1>
    </div>
  </body>

</html>
```

CSS

Hola Vue!

3

Untitled fiddle

No description

Add title to make the fiddle visible on your profile page

Resources URL cdnjs

Async requests

Other (links, license)

```
<html>

  <head>
    <meta charset="utf-8">
    <title>My Vue</title>
  </head>

  <body>
    <div id="app">
      <h1>{{message}}</h1>
    </div>
    <script src="https://unpkg.com/vue"></script>
  </body>

</html>
```

JavaScript + No-Library (pure JS)

```
const app = new Vue({
  el: '#app',
  data: {
    message: 'Hello Vue!'
  }
});
```

Hello Vue!

4

```
    <meta charset="utf-8">
    <title>My Vue</title>
  </head>

  <body>
    <div id="app">
      <h1>{{message}}</h1>
    </div>
    <script src="https://unpkg.com/vue"></sc
  </body>

</html>
```

JavaScript + No-Library (pure JS)

```
const app = new Vue({
  el: '#app',
  data: {
    message: 'Hello Vue!'
  }
});
```

5

Run Save Tidy Collaborate Update New and upd

ddle meta

HTML

```
<html>

  <head>
    <meta charset="utf-8">
    <title>My Vue</title>
  </head>

  <body>
    <div id="app">
      <h1>{{message}}</h1>
    </div>
    <script src="https://unpkg.com/vue">
  </body>

</html>
```

Configurar el entorno

Primeros pasos con atom

Comenzaremos este viaje de aprendizaje configurando un entorno que nos facilite el desarrollo de nuestras aplicaciones. Para este libro como editor hemos elegido Atom pero podría utilizar cualquier otro con el que se sienta cómodo.

En primer lugar, vamos a la página de Atom (https://atom.io/)❶ y una vez descargado instalamos ❷ y lo iniciamos ❸ y a partir de este momento ya tenemos un editor para nuestro código.

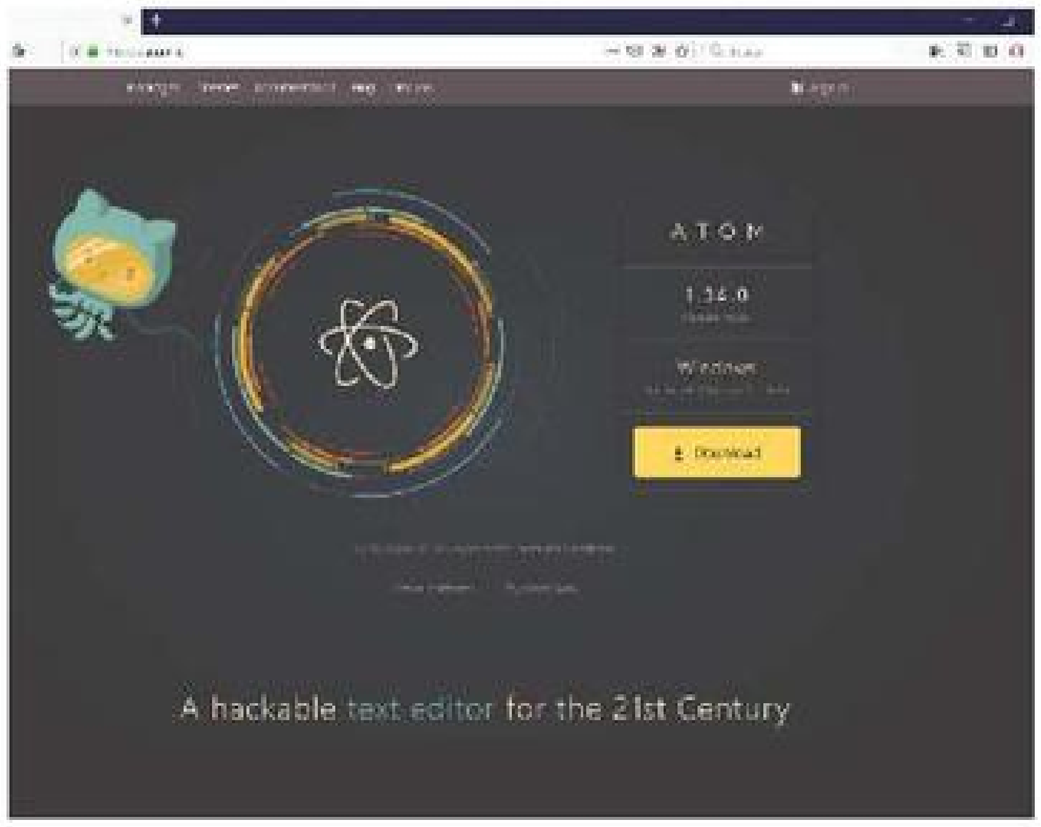
Esto no se acaba aquí, querremos tener también a nuestro alcance las herramientas que nos hagan desarrollar nuestras aplicaciones más cómoda y rápidamente, con lo que vamos a añadir algunos plugins.

Para ello iremos a la vista de ajustes en la que podremos buscar y seleccionar plugins para su instalación ❹. Entre la ingente cantidad de herramientas que podremos añadir a nuestro editor nosotros hemos elegido algunas que pueden ser útiles. Sin embargo, utilice tantas como quiera y necesite para sentirse cómodo ya que esta es una elección muy personal.

Entre las que hemos seleccionado están language-vue ❺ que resalta la sintaxis de vue, vue-format ❻, atom-beautify ❼ que dan formato a nuestro código y atom-html-preview ❽ que integra dentro de nuestro editor una ventana para renderizar el contenido automático de código en vivo.

De esta forma, si retomamos los archivos de nuestra primera aplicación, podremos trabajar cómodamente en nuestro editor local con todos los plugins para la ayuda al desarrollo y el renderizado HTML integrado ❾.

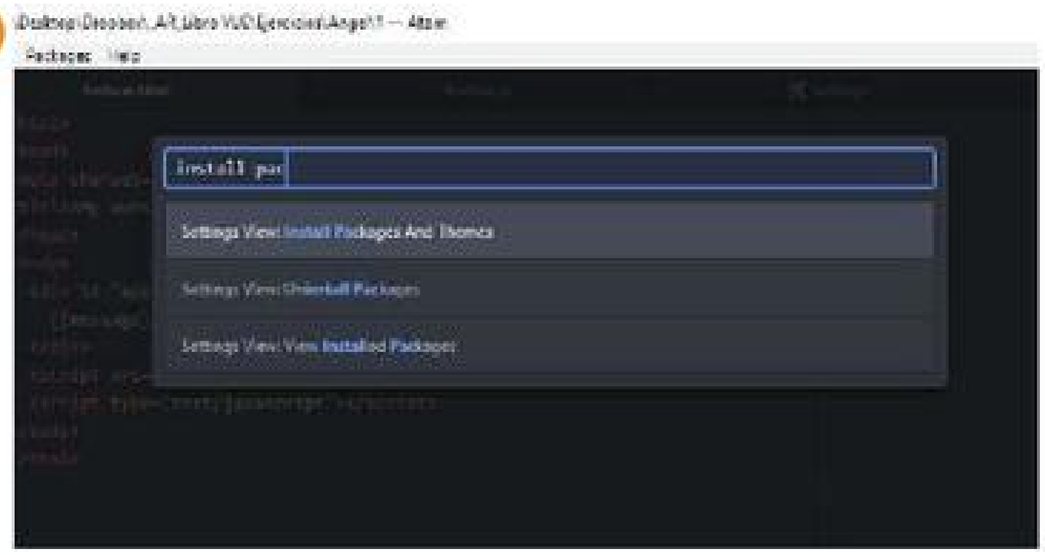
❶



❷

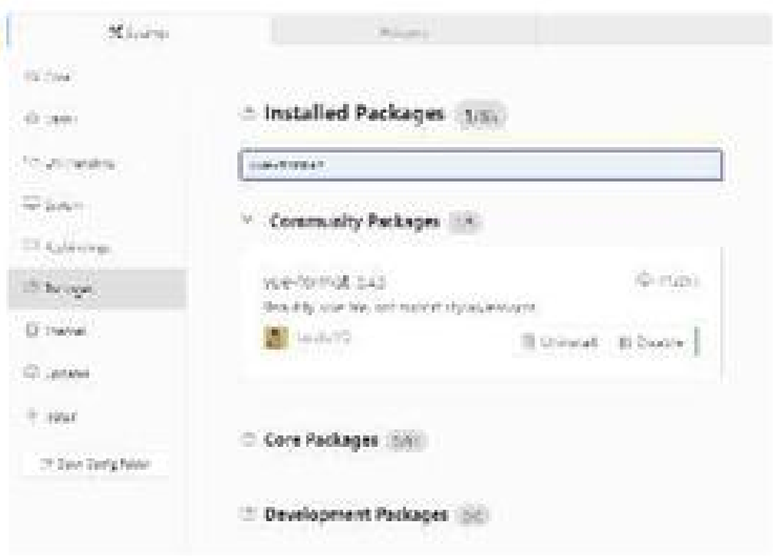
3

4

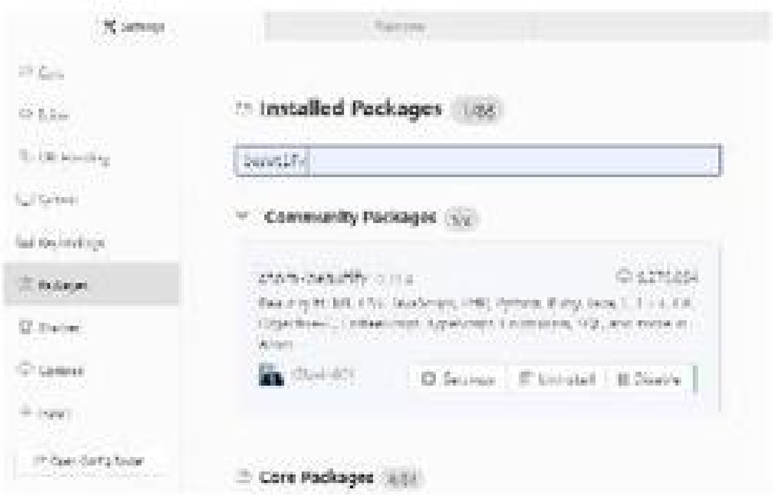


5

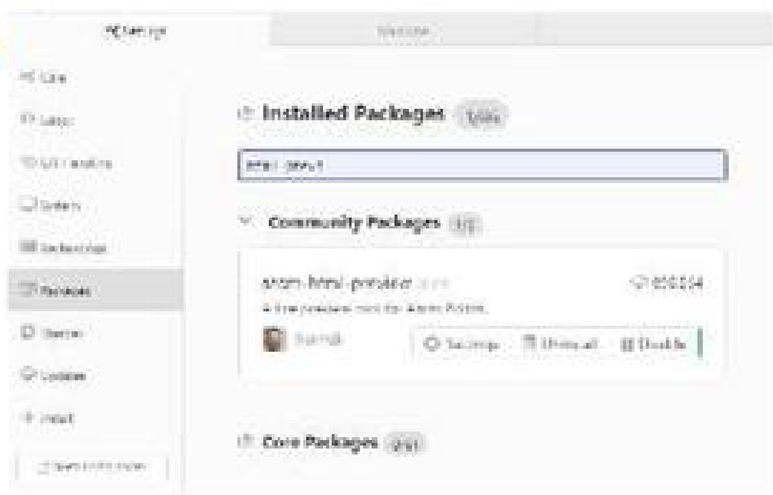
6



7



8



9

```html
<html>

<head>
  <meta charset="utf-8">
  <title>My Vue</title>
</head>

<body>
  <div id="app">
    <h1>{{message}}</h1>
  </div>
  <script src="https://unpkg.com/vue"></script>
  <script src="firstvue.js"></script>
</body>

</html>
```

Hello Vue!

Instancia Vue

Manejo de variables

Vamos a construir un pequeño blog de viajes y, para ello, como hemos visto en anteriores ejercicios, creamos la instancia Vue que manejará el contexto del elemento con id #app asociado ❶.

Dentro de este elemento crearemos un título general y una serie de artículos con un pequeño titular y contenido ❷. De esta forma definimos una plantilla para nuestro blog y para la que nos faltaría añadir contenido. Este contenido variará, con lo que como primera aproximación crearemos unas cuantas variables dentro de la instancia Vue que representarán los textos que hayamos escrito sobre nuestros últimos viajes.

Tenemos nuestro pequeño blog de viajes listo para añadir contenido, este contenido podría cargarse en las variables desde alguna fuente externa tanto para imágenes como para texto, pero como primer paso simplemente representamos el valor contenido dentro de las variables de la instancia Vue ❸.

Esto Vue lo realiza así de fácil, podemos considerar el data como el lugar donde definimos las variables del modelo con las que trabaja nuestra instancia de Vue. Hasta ahora la definición de estos valores se realiza de forma estática, ya veremos más adelante cómo podemos ir modificando estos valores dentro de la instancia, donde su valor se reflejaría automáticamente en la plantilla, al haber sufrido un cambio, esto es propio de la reactividad de Vue. ¡Genial!

❶

firstvue.html

```
8   <body>
9     <h1>My Vue Blog</h1>
10    <div class="flex-container">
11      <div id="app" class="flex-item flex-item-content">
12        <h2>Content</h2>
13
14        <h3>{{title1}}</h3>
15        {{post1}}
16
17        <h3>{{title2}}</h3>
18        {{post2}}
19
20        <h3>{{title3}}</h3>
21        {{post3}}
22      </div>
23    </div>
24    <script src="https://unpkg.com/vue"></script>
25    <script src="firstvue.js"></script>
26  </body>
```

firstvue.js

```
1  const app = new Vue({
2    el: '#app',
3    data: {
4      title1:'Trip 1',
5      title2:'Trip 2',
6      title3:'Trip 3',
7      post1: 'Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiu
```

firstvue.html Preview

My Vue Blog

Content

Trip 1

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Trip 2

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Trip 3

2

firstvue.html

```
8   <body>
9     <h1>My Vue Blog</h1>
10    <div class="flex-container">
11      <div id="app" class="flex-item flex-item-content">
12        <h2>Content</h2>
13
14        <h3>{{title1}}</h3>
15        {{post1}}
16
17        <h3>{{title2}}</h3>
18        {{post2}}
19
20        <h3>{{title3}}</h3>
21        {{post3}}
22      </div>
23    </div>
24    <script src="https://unpkg.com/vue"></script>
25    <script src="firstvue.js"></script>
26  </body>
```

firstvue.js

```
1  const app = new Vue({
2    el: '#app',
3    data: {
4      title1:'Trip 1',
5      title2:'Trip 2',
6      title3:'Trip 3',
7      post1: 'Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do elu
8      post2: 'Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do elu
```

firstvue.html Preview

My Vue Blog

Content

Trip 1

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Trip 2

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Trip 3

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed

3

firstvue.js

```
1  const app = new Vue({
2    el: '#app',
3    data: {
4      title1:'Trip 1',
5      title2:'Trip 2',
6      title3:'Trip 3',
7      post1: 'Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt
8      post2: 'Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt
9      post3: 'Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt
10   }
11 });
12
```

firstvue.html Preview

My Vue Blog

Content

Trip 1

Lorem ipsum dolor sit amet, consectetur adipisicing elit, s
do eiusmod tempor incididunt ut labore et dolore magna
aliqua. Ut enim ad minim veniam, quis nostrud exercitatio
ullamco laboris nisi ut aliquip ex ea commodo consequat.
Duis aute irure dolor in reprehenderit in voluptate velit es
cillum dolore eu fugiat nulla pariatur. Excepteur sint occa
cupidatat non proident, sunt in culpa qui officia deserunt
mollit anim id est laborum.

Trip 2

Lorem ipsum dolor sit amet, consectetur adipisicing elit, s
do eiusmod tempor incididunt ut labore et dolore magna
aliqua. Ut enim ad minim veniam, quis nostrud exercitatio
ullamco laboris nisi ut aliquip ex ea commodo consequat.
Duis aute irure dolor in reprehenderit in voluptate velit es
cillum dolore eu fugiat nulla pariatur. Excepteur sint occa
cupidatat non proident, sunt in culpa qui officia deserunt
mollit anim id est laborum.

Trip 3

Lorem ipsum dolor sit amet, consectetur adipisicing elit, s

Instancias Vue

Gestionando más de una instancia

En nuestro blog de viajes además de una colección de artículos queremos añadir un pequeño índice con información sobre los títulos y en qué fecha se publicaron. Con este fin podemos añadir una nueva instancia de Vue que controle este componente índice.

Partimos de nuestro ejemplo anterior ❶ creando una nueva columna en la parte derecha con nuestro nuevo componente. Este nuevo elemento contendrá una lista no ordenada de títulos con fecha de publicación, y para ello crearemos el HTML adecuado y lo identificamos con el id app2.

La segunda parte del ejercicio será crear una nueva instancia Vue en el código Javascript con la que manejar nuestro nuevo componente. Hasta este momento tenemos declarada una lista con elementos en HTML con una variable por ítem de la lista y ahora declararemos y daremos valores a todas y cada una de ellas ❷ ❸.

Hemos creado nuestro blog de artículos de viaje y lo hemos completado con la creación de un pequeño índice resumen con títulos y fechas. Comprobamos así que pueden convivir varias instancias Vue, una para cada componente, sin ningún problema.

❶

firstvue.html

```
8   <body>
9     <h1>My Vue Blog</h1>
10    <div class="flex-container">
11      <div id="app" class="flex-item flex-item-content">
12        <h2>Content</h2>
13
14        <h3>{{title1}}</h3>
15        {{post1}}
16
17        <h3>{{title2}}</h3>
18        {{post2}}
19
20        <h3>{{title3}}</h3>
21        {{post3}}
22      </div>
23    </div>
24    <script src="https://unpkg.com/vue"></script>
25    <script src="firstvue.js"></script>
26  </body>
```

firstvue.js

```
1  const app = new Vue({
2    el: '#app',
3    data: {
4      title1:'Trip 1',
5      title2:'Trip 2',
6      title3:'Trip 3',
7      post1: 'Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiu
```

firstvue.html Preview

My Vue Blog

Content

Trip 1

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Trip 2

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Trip 3

2

firstvue.html

```
    <h3>{{title2}}</h3>
    {{post2}}

    <h3>{{title3}}</h3>
    {{post3}}
  </div>

  <div id="app2" class="flex-item flex-item-summary">
    <h2>{{message}}</h2>
    <ul class="summary">
      <li class="item">{{title1}}</li>
      <li class="item">{{title2}}</li>
      <li class="item">{{title3}}</li>
    </ul>
  </div>
</div>


<script src="https://unpkg.com/vue"></script>
<script src="firstvue.js"></script>
</body>

</html>
```

firstvue.js

```
    title2:'Trip 2',
    title3:'Trip 3',
    post1: 'lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt
```

firstvue.html Preview

My Vue Blog

Content

Trip 1

Lorem ipsum dolor sit amet, consectetur adipisicing elit
do eiusmod tempor incididunt ut labore et dolore magna
aliqua. Ut enim ad minim veniam, quis nostrud exercita
ullamco laboris nisi ut aliquip ex ea commodo conseque
Duis aute irure dolor in reprehenderit in voluptate velit
cillum dolore eu fugiat nulla pariatur. Excepteur sint occ
cupidatat non proident, sunt in culpa qui officia deserun
mollit anim id est laborum.

Trip 2

Lorem ipsum dolor sit amet, consectetur adipisicing elit
do eiusmod tempor incididunt ut labore et dolore magna
aliqua. Ut enim ad minim veniam, quis nostrud exercita
ullamco laboris nisi ut aliquip ex ea commodo conseque
Duis aute irure dolor in reprehenderit in voluptate velit
cillum dolore eu fugiat nulla pariatur. Excepteur sint occ
cupidatat non proident, sunt in culpa qui officia deserun
mollit anim id est laborum.

Trip 3

Lorem ipsum dolor sit amet, consectetur adipisicing elit

3

firstvue.html | firstvue.js

```
const app = new Vue({
  el: '#app',
  data: {
    title1:'Trip 1',
    title2:'Trip 2',
    title3:'Trip 3',
    post1: 'lorem ipsum dolor sit amet, consectetur adipisicing
    post2: 'lorem ipsum dolor sit amet, consectetur adipisicing
    post3: 'lorem ipsum dolor sit amet, consectetur adipisicing
  }
});

const app2 = new Vue({
  el: '#app2',
  data: {
    message: 'Summary',
    title1: '16/06/2019 Trip 1',
    title2: '05/12/2018 Trip 2',
    title3: '04/10/2018 Trip 3'
  }
});
```

firstvue.html Preview

My Vue Blog

Content

Trip 1

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Trip 2

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Trip 3

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed

Summary

16/06/2019 Trip 1

05/12/2018 Trip 2

04/10/2018 Trip 3

Ciclo de vida

1

firstvue.js

```
const app = new Vue({
  el: '#app',
  data: {
    message: 'Hola Vue!',
  },
  beforeCreate: function() {
    console.log("beforeCreate");
  },
  created: function() {
    console.log("created");
  },
  beforeMount: function() {
    console.log("beforeMount");
  },
  mounted: function() {
    console.log("mounted");
  },
  beforeUpdate: function() {
    console.log("beforeUpdate");
  },
  updated: function() {
    console.log("updated");
  },
  beforeDestroy: function() {
    console.log("beforeDestroy");
  },
  destroyed: function() {
    console.log("destroyed");
  },
  methods: {
    handleClick: function(){
      app.$destroy();
    }
  }
});
```

Fases del ciclo de vida

Se denomina ciclo de vida de una instancia Vue a una serie de estados por los que va pasando el componente. Estos estados son básicamente cuatro: created, mounted, updated y destroyed.

Vue nos permitirá definir acciones anteriores y posteriores a la transición desde o hacia cada estado interno del componente. Los métodos en cuestión para implementar estas acciones son:

- beforeCreate: evento lanzado antes de tener el componente cargado, implica no poder acceder al componente a nivel de Dom.
- created: evento donde se verifica si el componente tiene plantilla, entonces se compila y se observan las propiedades computadas, data, métodos y eventos. Pero no podemos acceder al $el.
- beforeMount: evento que ocurre antes de representar el componente.
- mounted: evento que implica que el componente está cargado por completo, se añade al DOM, quedando el componente accesible a través de $el.
- beforeUpdate: evento que se ejecuta cuando el valor del data del componente cambia.
- updated: evento invocado tras finalizar la modificación de valor del data.
- beforeDestroy: evento que elimina eventos que estuvieran activos en el componente, antes de eliminar la instancia.
- destroyed: evento lanzado tras desacoplar el componente.

2

firstvue.html

```
  <title>My Vue</title>
  <style src="firstvue.css"></style>
  <link rel="stylesheet" href="firstvue.css">
</head>

<body>
  <div id="app">
    <h1>{{ message }}</h1>

    <input type="text" v-model="message"/>
     <button type="button" v-on:click="handleClick">Destroy</button>
  </div>
```

firstvue.js

```
const app = new Vue({
  el: '#app',
  data: {
```

firstvue.html Preview

Hola Vue!

Hola Vue! | Destroy

1. beforeCreate
2. created
3. beforeMount
4. mounted
5. Download the Vue Devtools extension for a better development experience:
   https://github.com/vuejs/vue-devtools
6. You are running Vue in development mode.
   Make sure to turn on production mode when deploying for production.
   See more tips at https://vuejs.org/guide/deployment.html

En estos métodos podremos reservar o liberar recursos, realizar cálculos, hacer llamadas http, etc.

Para ilustrar mejor cómo funciona el ciclo de vida implementaremos cada uno de estos métodos escribiendo en una consola del navegador un mensaje por cada vez que entre en los métodos mencionados ❶.

Una vez que carga la página vemos que la instancia es creada, cargada en el DOM y está disponible cuando pasa al estado mount ❷.

Para comprobar qué pasa cuando actualizamos el componente podemos editar el texto "Hola Vue!" ❸ y añadir alguna modificación para ver que pasa por los métodos beforeUpdate y updated ❹.

En el caso de que eliminemos el componente podremos forzar su destrucción a través del botón Destroy comprobar que la instancia pasa por los métodos beforeDestroy y destroyed ❺.

❸

firstvue.html

```
3  <head>
4    <meta charset="utf-8">
5    <title>My Vue</title>
6    <style src="firstvue.css"></style>
7    <link rel="stylesheet" href="firstvue.css">
8  </head>
9
10 <body>
11   <div id="app">
12     <h1>{{ message }}</h1>
13
14     <input type="text" v-model="message"/>
15      <button type="button" v-on:click="handleClick">Destroy</button>
16   </div>
17
18   <ul id="myULContainer" class="console"></ul>
19
20   <script src="https://unpkg.com/vue"></script>
21   <script type="application/javascript" src="https://cdn.rawgit.com/Alorel/co
22
```

firstvue.html Preview

Hola Vue!

Hola Vue! | Destroy

1. beforeCreate
2. created
3. beforeMount
4. mounted
5. Download the Vue Devtools extension for a bett
   https://github.com/vuejs/vue-devtools
6. You are running Vue in development mode.
   Make sure to turn on production mode when dep
   See more tips at https://vuejs.org/guide/deploym

❹

firstvue.html Preview

Hola Vue!!!

Hola Vue!!! | Destroy

1. beforeCreate
2. created
3. beforeMount
4. mounted
5. Download the Vue Devtools extension for a better development experience
   https://github.com/vuejs/vue-devtools
6. You are running Vue in development mode.
   Make sure to turn on production mode when deploying for production.
   See more tips at https://vuejs.org/guide/deployment.html
7. beforeUpdate
8. updated
9. beforeUpdate
10. updated

❺

Hola Vue!!! | Destroy

1. beforeCreate
2. created
3. beforeMount
4. mounted
5. Download the Vue Devtools extension for a better development experience
   https://github.com/vuejs/vue-devtools
6. You are running Vue in development mode.
   Make sure to turn on production mode when deploying for production.
   See more tips at https://vuejs.org/guide/deployment.html
7. beforeUpdate
8. updated
9. beforeUpdate
10. updated
11. beforeDestroy
12. destroyed

Data binding

Valores y variables

Volvemos de nuevo con nuestro blog de viajes, en esta ocasión además de listar los artículos escritos y el índice querremos añadir un pequeño apartado para editar nuevos artículos.

En primer término, crearemos un input para escribir un título para nuestro artículo y una caja de texto más amplia para poder escribir todo nuestro artículo en la misma ❶.

Además de crear el marcado en lenguaje HTML añadiremos a cada input un nuevo atributo con el siguiente formato v-model="nombreVariable" para enlazar la variable con el contenido que escribamos. Del mismo modo tendremos que declarar esas variables con los nombres que habremos elegido en el código Javascript de la instancia Vue ❷.

Posteriormente añadiremos el título y el contenido en la página para que se renderice a medida que se escribe y podamos ir viendo cómo queda ❸. Esta es la "magia" del enlazado de variables con los inputs ya que somos capaces de darle un valor a una variable dentro de la instancia e ir mostrando lo que guarda esa variable dentro del contexto de nuestra instancia. ❹

Simplemente con estos pasos tendremos una forma fácil de añadir un nuevo artículo con su título y contenido y poder ver el renderizado en la página inmediatamente a medida que vamos generando contenido❺.

❶

```
<body>
  <h1>My Vue Blog</h1>
  <div id="app"><h2>Nueva entrada</h2>
    <h3>Titulo</h3><input v-model="title" placeholder="Write a title">
    <h3>Entrada</h3>
    <textarea cols="50" v-model="post" placeholder="Write a post"></textarea>
    <h2>{{ title }}</h2>
    <p>{{post}}</p>
    <div class="flex-container">
      <div class="flex-item flex-item-content"><h2>Content</h2>
        <h3>{{title1}}</h3> {{post1}}
        <h3>{{title2}}</h3> {{post2}}
        <h3>{{title3}}</h3> {{post3}}
      </div>
      <div class="flex-item flex-item-summary"><h2>{{message}}</h2>
        <ul class="summary"><li class="item">{{title1}}</li>
                            <li class="item">{{title2}}</li>
                            <li class="item">{{title3}}</li></ul></div>
    </div>
  </div>
  <script src="https://unpkg.com/vue"></script>
  <script src="firstvue.js"></script>
</body>
```

❷

```
const app = new Vue({ el: '#app',data: { title: '', post: '',
    title1: 'Trip 1', title2: 'Trip 2', title3: 'Trip 3',
    post1: 'Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempo
    post2: 'Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempo
    post3: 'Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempo
    message: 'Summary',
    title1: '16/06/2019 Trip 1', title2: '05/12/2018 Trip 2',title3: '04/10/2018 Trip 3' }
```

❸

```
<h2>{{ title }}</h2>
<p>{{post}}</p>
```

4

```
10 <body>
11   <h1>My Vue Blog</h1>
12   <div id="app">
13     <h2>Nueva entrada</h2>
14     <h3>Título</h3>
15     <input v-model="title" placeholder="Write a title">
16     <h3>Entrada</h3>
17     <textarea cols="50" v-model="post" placeholder="Write a post">
18     </textarea>
19     <h2>{{ title }}</h2>
20     <p>{{post}}</p>
21     <div class="flex-container">
```

firstvue.js

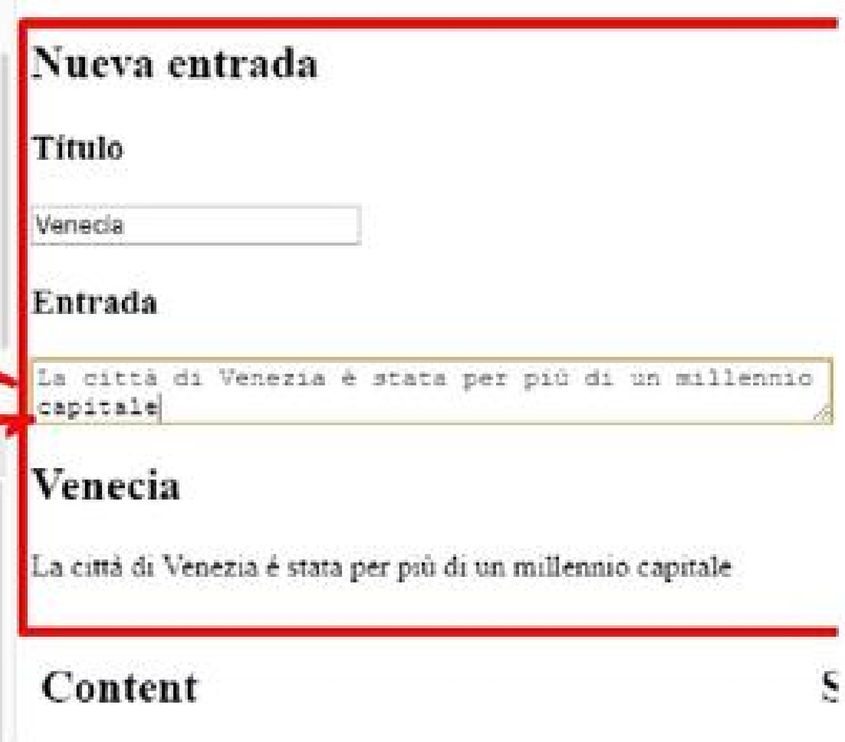
```
1 const app = new Vue({
2   el: '#app',
3   data: {
4     title: '',
5     post: '',
6     title1: 'Trip 1',
7     title2: 'Trip 2',
8     title3: 'Trip 3',
9     post1: 'Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eius
```

5

```
10 <body>
11   <h1>My Vue Blog</h1>
12   <div id="app">
13     <h2>Nueva entrada</h2>
14     <h3>Título</h3>
15     <input v-model="title" placeholder="Write a title">
16     <h3>Entrada</h3>
17     <textarea cols="50" v-model="post" placeholder="Write a post">
18     </textarea>
19     <h2>{{ title }}</h2>
20     <p>{{post}}</p>
21     <div class="flex-container">
```

firstvue.js

```
1 const app = new Vue({
2   el: '#app',
3   data: {
4     title: '',
5     post: '',
6     title1: 'Trip 1',
7     title2: 'Trip 2',
```



Watch

Propiedades observadas

Hasta ahora hemos visto cómo hacer el enlazado entre las variables y su contenido y la renderización en la web de los cambios. Hemos hablado de las variables de la instancia de Vue: cómo declararlas, manejarlas y mostrar su contenido.

Para observar una propiedad lo único que tendremos que hacer es declarar dentro en la instancia una variable dentro del bloque watch. El bloque watch se define como un objeto, y cada variable dentro del mismo es una función con el mismo nombre, con lo que cada vez que la variable cambie se ejecutará una función con dos argumentos: un valor antes del cambio y el nuevo valor.

Para ilustrar este concepto en este ejercicio veremos cómo observar variables de la instancia y mostrar su resultado. Compondremos una nueva página con un input y veremos cómo cambia la propiedad observada visualizándolo en una consola dentro de la propia página.

En el apartado del HTML tenemos nuestro título de página y un input para meter texto libre que enlazamos con el atributo v-model ❶, y en el apartado del código Javascript creamos un bloque watch en el que observamos la variable message además de escribir la función que se ejecutará cuando cambie su valor ❷.

Vemos que cada vez que hacemos cambios en el texto ❸ se ejecuta la función con el valor antes del cambio y el nuevo valor ❹.

Watch permite detectar cambios en los valores que se les proporciona a componentes hijos a través de las "props", detectando cambios en el valor proporcionado en sus props, para finalmente refrescar el valor del componente, aunque este uso lo veremos con más detalle en otros ejercicios.

❶

```
8   </head>
9
10  <body>
11    <div id="app">
12      <h1>{{ message }}</h1>
13      <input type="text" v-model="message" />
14    </div>
15
16
17    <ul id="myULContainer" class="console"></ul>
18
```

❷

firstvue.js

```
1  const app = new Vue({
2    el: '#app',
3    data: {
4      message: 'Hola Vue!',
5    },
6    watch: {
7      message: function(newValue, oldValue) {
8        console.log('oldValue=> ',oldValue, ' newValue=>', newValue);
9      }
10   }
11 });
```

3

```
<head>
  <meta charset="utf-8">
  <title>My Vue</title>
  <style src="firstvue.css"></style>
  <link rel="stylesheet" href="firstvue.css">
</head>

<body>
  <div id="app">
    <h1>{{ message }}</h1>
    <input type="text" v-model="message" />
  </div>


  <ul id="myULContainer" class="console"></ul>

  <script src="https://unpkg.com/vue"></script>
  <script type="application/javascript" src="https://cdn.rawgit.com/Alorel/consol
```

firstvue.js

```
const app = new Vue({
  el: '#app',
  data: {
    message: 'Hola Vue!',
  },
  watch: {
    message: function(newValue, oldValue) {
      console.log('oldValue=> ',oldValue, ' newValue=>', newValue);
    }
```

Hola Vue!

Hola Vue!

1. You are running Vue in development mode.
   Make sure to turn on production mode when deploying for prod
   See more tips at https://vuejs.org/guide/deployment.html
2. Download the Vue Devtools extension for a better development
   https://github.com/vuejs/vue-devtools

4

```
<head>
  <meta charset="utf-8">
  <title>My Vue</title>
  <style src="firstvue.css"></style>
  <link rel="stylesheet" href="firstvue.css">
</head>

<body>
  <div id="app">
    <h1>{{ message }}</h1>
    <input type="text" v-model="message" />
  </div>


  <ul id="myULContainer" class="console"></ul>

  <script src="https://unpkg.com/vue"></script>
  <script type="application/javascript" src="https://cdn.rawgit.com/Alorel/consol
```

firstvue.js

```
const app = new Vue({
  el: '#app',
  data: {
    message: 'Hola Vue!',
  },
  watch: {
    message: function(newValue, oldValue) {
      console.log('oldValue=> ',oldValue, ' newValue=>', newValue);
    }
  }
});
```

Hola Vue! Que tal?

Hola Vue! Que tal?

1. oldValue=> Hola Vue! newValue=> Hola Vue! Que tal?
2. You are running Vue in development mode.
   Make sure to turn on production mode when deploying for prod
   See more tips at https://vuejs.org/guide/deployment.html
3. Download the Vue Devtools extension for a better development
   https://github.com/vuejs/vue-devtools

Computed properties

Evitando el recómputo

Hasta ahora hemos manejado propiedades y métodos de la instancia. Sabemos cómo declarar propiedades, cómo modificarlas y con ello vemos cómo se renderizan automáticamente de nuevo en la página.

En este ejercicio veremos cómo usar las computed properties que nos dan un punto extra de eficiencia ya que solo se evaluarán en el momento que afecte a alguna parte de su cálculo.

Imaginemos que tenemos una pequeña calculadora de sumas y restas en la que tenemos cuatro inputs que se corresponden con los diferentes operandos involucrados ❶.

Contamos con cuatro propiedades en el apartado data y dos métodos diferentes encargados de hacer el cómputo correspondiente y mostrarnos el resultado ❷.

Ahora mismo si editamos alguno de los operandos vemos que se efectúa el cálculo y se muestra el resultado después del signo igual. Hasta aquí es lo que se espera de nuestra calculadora personal, ¿no? Sin embargo, si vamos al detalle de cómo se efectúan los cálculos revisando la consola, podemos comprobar que cada vez que modificamos un solo operando se hacen los cálculos de todas las operaciones que tengamos declaradas.

Para nuestro caso quizá no es algo muy crítico desde el punto de vista de la eficiencia, pero nos vale para ver que, si se hiciera algún cálculo costoso, este repetiría fuese cual fuese el cambio en la instancia ❸.

Para evitar este recálculo haremos uso de las computed properties simplemente sustituyendo el bloque de methods por un bloque computed y cambiar las llamadas para el cálculo de los resultados como si fueran propiedades de la instancia ❹.

❶

```
const app = new Vue({ el: '#app',
  data: { title: 'Calculator', operator1: '0'
          ,operator2: '0', operator3: '0', operator4: '0' },
  methods: {
    resultSum: function() {=},
    resultSub: function() {=} } });
```

```
<html><head><meta charset="utf-8">=
<body>
  <div id="app"><h1>{{ title }}</h1>
    <input type="text" v-model="operator1" /> +
    <input type="text" v-model="operator2" /> = {{ resultSum }}
    <input type="text" v-model="operator3" /> -
    <input type="text" v-model="operator4" /> = {{ resultSub }}
  </div>
```

❷

```
const app = new Vue({ el: '#app',
  data: { title: 'Calculator', operator1: '0'
          ,operator2: '0', operator3: '0', operator4: '0' },
  methods: {
    resultSum: function() {
      let op1 = parseInt(this.operator1===''?'0':this.operator1);
      let op2 = parseInt(this.operator2===''?'0':this.operator2);
      console.log( op1 + ' + ' + op2 + ' = ' + (op1+op2));
      return op1 + op2;
    },
    resultSub: function() {
      let op3 = parseInt(this.operator3===''?'0':this.operator3);
      let op4 = parseInt(this.operator4===''?'0':this.operator4);
      console.log( op3 + ' - ' + op4 + ' = ' + (op3-op4));
      return op3 - op4;
    } } });
```

Con esta modificación podemos comprobar que si modificamos alguno de los operandos solo se volverán a calcular aquellas operaciones en las que ese operando esté involucrado, mientras que el resto no se recalcularán 5.

Para comprobar cómo se hace el recálculo utilizaremos el operando 2 en ambas propiedades computadas, con lo que podremos jugar y ver qué es lo que se recalcula en función del operando que modifiquemos 6.

3

Calculator

2 + 0 = 2

3 - 0 = 3

1. 3 - 0 = 3
2. 2 + 0 = 2
3. 0 - 0 = 0
4. 2 + 0 = 2
5. You are running Vue in development mode.
   Make sure to turn on production mode when deploying for production.
   See more tips at
   https://vuejs.org/guide/deployment.html
6. Download the Vue Devtools extension for a better development experience:
   https://github.com/vuejs/vue-devtools
7. 0 - 0 = 0
8. 0 + 0 = 0

4

firstvue.js

```
const app = new Vue({ el: '#app',
  data: { title: 'Calculator', operator1: '0'
          ,operator2: '0', operator3: '0', operator4: '0' },
  computed: {
    resultSum: function() {
      let op1 = parseInt(this.operator1===''?'0':this.operator1);
      let op2 = parseInt(this.operator2===''?'0':this.operator2);
      console.log( op1 + ' + ' + op2 + ' = ' + (op1+op2));
      return op1 + op2;
    },
    resultSub: function() {
      let op3 = parseInt(this.operator3===''?'0':this.operator3);
      let op4 = parseInt(this.operator4===''?'0':this.operator4);
      console.log( op3 + ' - ' + op4 + ' = ' + (op3-op4));
      return op3 - op4;
    }
  }
});
```

firstvue.h

Calculat

0

0

1. You are run
   Make sure t
   production.
   See more ti
2. Download t
   experience:
   https://gith
3. 0 - 0 = 0
4. 0 + 0 = 0

5

firstvue.html Preview

Calculator

1 + 0 = 1

2 - 0 = 2

1. 2 - 0 = 2
2. 1 + 0 = 1
3. You are running Vue in development mode.
   Make sure to turn on production mode when deploying for production.
   See more tips at https://vuejs.org/guide/deployment.html
4. Download the Vue Devtools extension for a better development experience:
   https://github.com/vuejs/vue-devtools
5. 0 - 0 = 0
6. 0 + 0 = 0

6

```
  computed: {
    resultSum: function() {
      let op1 = parseInt(this.operator1===''?'0':this.operator1);
      let op2 = parseInt(this.operator2===''?'0':this.operator2);
      console.log( op1 + ' + ' + op2 + ' = ' + (op1+op2));
      return op1 + op2;
    },
    resultSub: function() {
      let op3 = parseInt(this.operator3===''?'0':this.operator3);
      let op4 = parseInt(this.operator4===''?'0':this.operator4);
      console.log( op3 + ' - ' + op4 + ' = ' + (op3-op4));
      return op3 - op4;
    }
  }
});
```

Métodos

Operando con las propiedades

Dentro de una instancia Vue además de las propiedades se exponen métodos que operan sobre los datos o ejecutan acciones. Los métodos se ejecutan cada vez que las invocamos, cosa que no ocurre con las propiedades computadas, ejecutadas cada vez que alguna de las variables utilizadas en la obtención de un resultado en las propiedades computadas cambia su valor; en dicho caso se ejecutaría la propiedad, para recalcular.

Estos métodos se declaran dentro de la instancia dentro del bloque methods, y se pueden asociar a los elementos HTML de la instancia.

En este ejercicio vamos a relacionar un componente de tipo botón con una acción que se corresponderá con un método declarado en la instancia Vue. Para indicar esta relación usamos la directiva v-on que se coloca dentro del HTML del botón y el evento (en este caso el evento de click).

```
<button type="button" v-on:click="showMessage">Mostrar Titulo</button>
```

Una vez dado este paso creamos un bloque methods en la instancia y dentro de este creamos la función showMessage que concatenará la propiedad title a un texto.

```
methods: {
    showMessage: function() {
        console.log('Showing message: '+this.title + ' Aprendamos!');
        return this.title + ' Aprendamos!';
    }
}
```

Posteriormente declaramos dentro de una etiqueta h1 el método para renderizar el resultado de la función ❶.

```
<h1>{{ showMessage() }}</h1>
```

Para volver a renderizar el resultado de la función accionamos el botón, si nos fijamos se calcula cada vez que salta el evento asociado al botón ❷.

1

firstvue.html

```
  <link rel="stylesheet" href="firstvue.css">
</head>

<body>
  <div id="app">
    <h1>{{ showMessage() }}</h1>
    <button type="button" v-on:click="showMessage">Mostrar Título</button>
  </div>


  <ul id="myULContainer" class="console"></ul>
```

firstvue.js

```
const app = new Vue({
  el: '#app',
  data: {
    title: 'Hola Vue!'
  },
  methods: {
    showMessage: function() {
      console.log('Showing message: '+this.title + ' Aprendamos!');
      return this.title + ' Aprendamos!';
    }
  }
```

firstvue.html Preview

Hola Vue! Aprendamos!

Mostrar Título

1. Showing message: Hola Vue! Aprendamos!
2. Download the Vue Devtools extension for a better developm
   https://github.com/vuejs/vue-devtools
3. You are running Vue in development mode.
   Make sure to turn on production mode when deploying for p
   See more tips at https://vuejs.org/guide/deployment.html

2

firstvue.html

```
  <link rel="stylesheet" href="firstvue.css">
</head>

<body>
  <div id="app">
    <h1>{{ showMessage() }}</h1>
    <button type="button" v-on:click="showMessage">Mostrar Título</button>
  </div>


  <ul id="myULContainer" class="console"></ul>
```

firstvue.js

```
const app = new Vue({
  el: '#app',
  data: {
    title: 'Hola Vue!'
  },
  methods: {
    showMessage: function() {
      console.log('Showing message: '+this.title + ' Aprendamos!');
      return this.title + ' Aprendamos!';
    }
  }
});
```

firstvue.html Preview

Hola Vue! Aprendamos!

Mostrar Título

1. Showing message: Hola Vue! Aprendamos!
2. Download the Vue Devtools extension for a better developm
   https://github.com/vuejs/vue-devtools
3. You are running Vue in development mode.
   Make sure to turn on production mode when deploying for p
   See more tips at https://vuejs.org/guide/deployment.html
4. Showing message: Hola Vue! Aprendamos!
5. Showing message: Hola Vue! Aprendamos!
6. Showing message: Hola Vue! Aprendamos!
7. Showing message: Hola Vue! Aprendamos!

Componentes de la instancia

Instancia Vue

Hasta ahora hemos estado jugando con la instancia de Vue, su ciclo de vida, el renderizado y el doble enlazado de propiedades, los métodos, etc., pero no nos hemos parado con los bloques básicos que tiene.

Una instancia se compone de:

- $el: es el objeto componente HTML al que estará asociado mediante el id correspondiente
- $data: es el objeto que contiene las propiedades de la instancia
- $refs: es el objeto donde se registran los elementos marcados con el atributo ref.

Para ilustrarlo en este ejercicio hemos creado una instancia de Vue con cada uno de los bloques mencionados a la que hemos añadido unas sentencias que imprimen por consola cada uno de estos elementos para que podamos ver su contenido cuando se carga la página.

El contenido de $el es efectivamente el objeto que contiene la definición del HTML de nuestra instancia. Si inspeccionamos el objeto accediendo a la propiedad innerHTML nos devuelve en formato texto el contenido en lenguaje HTML ❶.

El contenido de $data nos devuelve el objecto con las variables como la de nuestro ejercicio la denominada *message*.

Podemos también invocar los métodos de nuestra instancia tal y como vemos en el ejercicio y nos ejecutará la función.

Dentro de $refs tenemos un objeto con los elementos marcados con el atributo ref en el HTML de nuestra instancia, en este caso son dos elementos h2 (mysubtitle y mysubtitle2).

Hemos añadido un botón que tiene asociado el método clickedButton y si lo accionamos se nos mostrarán dos mensajes en los elementos h2 referenciados. En este método accedemos a las propiedades de los componentes para añadirles un mensaje y un estilo que se hará visible en nuestra página una vez se ejecute ❷.

1

firstvue.html

```
<html><head><meta charset="utf-8">
  <title>My Vue</title><style src="firstvue.css"></style>
  <link rel="stylesheet" href="firstvue.css"></head>
<body> <div id="app">
      <h1>{{ message }}</h1> <h2 ref="mysubtitle"></h2>
      <h2 ref="mysubtitle2"></h2>
      <button @click="clickedButton">Click Me!</button>
  </div>
  <ul id="myULContainer" class="console"></ul>
```

firstvue.js

```
const app = new Vue({ el: '#app',
  data: { message: 'Elementos de la instancia Vue'},
  methods: { sayHi: function() { return 'Hi!!!'; },
             clickedButton: function() {
                this.$refs.mysubtitle.innerText = "100 ejercicios";
                this.$refs.mysubtitle.style.color = "red";
                this.$refs.mysubtitle2.innerText = "para aprender Vue";
                this.$refs.mysubtitle2.style.color = "blue";
                console.log([app.$refs.mysubtitle.innerText,
                      app.$refs.mysubtitle2.innerText].join(" ")); } }
});
console.log(app.$el.innerHTML); console.log(app.$data);
console.log(app.sayHi());       console.log(app.$refs);
```

Elementos de la instancia Vue

Click Me!

1. <h1>Elementos de la instancia Vue</h1> <h2></h2> <h2></h2> <button>Click Me! <\button>
2. Object {"message": "Elementos de la instancia Vue"}
3. Hi!!!
4. Object {"mysubtitle": {},"mysubtitle2": {}}
5. Download the Vue Devtools extension for a better development experience: https://github.com/vuejs/vue-devtools
6. You are running Vue in development mode. Make sure to turn on production mode when deploying for production. See more tips at https://vuejs.org/guide/deployment.html

2

firstvue.html Preview

Elementos de la instancia Vue

100 ejercicios

para aprender Vue

Click Me!

1. <h1>Elementos de la instancia Vue</h1> <h2></h2> <h2></h2> <button>Click Me! <\button>
2. Object {"message": "Elementos de la instancia Vue"}
3. Hi!!!
4. Object {"mysubtitle": {},"mysubtitle2": {}}
5. Download the Vue Devtools extension for a better development experience: https://github.com/vuejs/vue-devtools
6. You are running Vue in development mode. Make sure to turn on production mode when deploying for production. See more tips at https://vuejs.org/guide/deployment.html
7. 100 ejercicios para aprender Vue

Eventos

Escuchando eventos

Podemos empezar a escuchar eventos añadiendo al input la directiva v-on seguido de que evento queremos escuchar para reaccionar invocando a un método concreto.

Hasta ahora hemos visto el doble enlazado de las propiedades de nuestra instancia, dentro del bloque data. Podemos declarar una propiedad que se renderiza en nuestra interfaz de usuario y declaramos un input que pueda modificarla mediante la directiva v-model ❶.

De esta forma, cada vez que modifiquemos el contenido del input se modificará la propiedad y se renderizará inmediatamente. Sin embargo, ¿cómo haríamos si queremos que se modifique solamente cuando se finalice su edición? ¿Cuándo detectamos que se ha acabado la edición?

En este caso vamos a interpretar que se acaba de editar el valor cuando se pulsa la tecla Enter de nuestro teclado. Para ello tendremos que detectar esta pulsación y esto lo podemos conseguir con la directiva v-on diciendo que ejecutaremos una acción justo cuando se deje de presionar (modificador keyup) la tecla Enter (modificador .enter)

```
v-on:keyup.enter="onEnterPressed"
```

❶ firstvue.html

```
<html><head><meta charset="utf-8">
  <title>My Vue</title><style src="firstvue.css"></style>
  <link rel="stylesheet" href="firstvue.css"></head>
<body>
  <div id="app">
    <h1>{{ message }}</h1>
    <input type="text" v-on:keyup.enter="onEnterPressed" v-model="inputText"/>
    <p>{{ enterMessage }}</p>
  </div>
```

firstvue.html Preview

Escuchando evento teclado [Enter]

firstvue.js

```
const app = new Vue({ el: '#app',
  data: { message: 'Escuchando evento teclado [Enter]',
          inputText: '', enterMessage: '' },
  methods: {
    onEnterPressed: function() {
      this.enterMessage = this.inputText;
      console.log(this.enterMessage); } } });
```

De esta forma vemos que cuando dejamos de escribir y cuando soltamos la tecla Enter se ejecuta la acción que hemos definido, que no será otra que la de renderizar el contenido de la variable en la página ❷.

Comprobamos además que solo se ejecuta en este caso como se puede ver en el log de la consola en la que se imprime un mensaje cuando se llama.

Escuchar eventos de teclado suele ser una práctica muy común para interceptar eventos según ocurran. Para invocar determinadas acciones, hemos probado el evento keyup. Además, podemos establecer un modificador concreto a invocar tras la captura del evento "haber pulsado una tecla", por ejemplo, capturar una tecla en concreto, la tecla escape o incluso definir alias para las F del teclado. Esto último se realizaría mediante la configuración en Vue de los códigos de tecla: "Vue.config.keyCodes.f" ❸.

❷

firstvue.html Preview

Escuchando evento teclado [Enter]

Aprendamos Vue!

Aprendamos Vue!

❸

firstvue.html

```
<html><head><meta charset="utf-8">
<body>
  <div id="app">
  <ul id="myULContainer" class="console"></ul>
```

firstvue.js

```
const app = new Vue({ el: '#app',
  data: { message: 'Escuchando evento teclado [Enter]',
          inputText: '', enterMessage: '' },
  methods: {
    onEnterPressed: function() {
      this.enterMessage = this.inputText;
      console.log(this.enterMessage); } } });
```

Escuchando evento teclado [Enter]

Renderizar HTML

HTML en una propiedad

En este ejercicio veremos cómo añadir código HTML para que sea renderizado dentro de nuestra instancia Vue.

En principio podríamos pensar que es algo muy sencillo, ¿no? Creamos una nueva instancia dentro de un elemento, una propiedad myhtml dentro del bloque **data** a la que asignamos una cadena con el formato de marcado HTML y la mostramos dentro de un bloque {{ myhtml }} ❶.

¿Esto funcionaría? Bueno, la respuesta en este caso sería negativa ya que dentro de la instancia no se renderizaría nuestro código HTML si no como un simple texto. Esto es así para evitar una vulnerabilidad crítica en la web (XSS: cross site scripting) que pudiera permitir una modificación maliciosa del código HTML de la variable.

¿Y no existe ninguna forma de añadir código HTML? Sí que la hay, Vue nos da una directiva específica, que igualmente deberemos utilizar siempre con cuidado, denominada v-html.

En el ejercicio declaramos la variable html con el contenido a renderizar que simplemente es un título y un link a otra página y dentro de un elemento div indicamos mediante la directiva v-html que dentro se renderizará el código HTML. Fácil ¿verdad? ❷

1

firstvue.html

```
<meta charset="utf-8">
<title>My Vue</title>
<style src="firstvue.css"></style>
<link rel="stylesheet" href="firstvue.css">
</head>

<body>
<div id="app">
  <div v-html="html"></div>
</div>
```

firstvue.js

```
const app = new Vue({
  el: '#app',
  data: {
    html: `
    <div>
       <h1>Ejemplo de renderizado HTML</h1>
       Este en lace nos lleva a <a href="https://www.google.com">Google</a>
    </div>`,
  },
  methods: {
  }
});
```

firstvue.html Preview

Ejemplo de renderizado HTML

Este en lace nos lleva a Google

2

firstvue.html

```
<meta charset="utf-8">
<title>My Vue</title>
<style src="firstvue.css"></style>
<link rel="stylesheet" href="firstvue.css">
</head>

<body>
<div id="app">
  <div v-html="html"></div>
</div>
```

firstvue.js

```
const app = new Vue({
  el: '#app',
  data: {
    html: `
    <div>
       <h1>Ejemplo de renderizado HTML</h1>
       Este en lace nos lleva a <a href="https://www.google.com">Google</a>
    </div>`,
  },
  methods: {
  }
});
```

firstvue.html Preview

Vincular propiedades

Uso de v-bind

Con la directiva v-bind podemos vincular un atributo de algún elemento html a una propiedad de nuestra instancia ❶.

Así contado parece algo abstracto pero para ilustrarlo en este ejercicio tendremos simplemente dos elementos: un área en la cual se van a mostrar imágenes y un botón que, cuando lo accionemos, modificará esta imagen.

Para la imagen tendremos un elemento HTML como img que se encarga de mostrar una imagen que le indiquemos dentro de la propiedad src.

```
<img src="placeholder.jpg" alt="placeholder">
```

Por otro lado en nuestra instancia tenemos una propiedad llamada image que tendrá la ruta a la imagen que queremos mostrar.

Ahora simplemente tendremos que vincular o enlazar la propiedad image con el atributo src del elemento img mediante la directiva v-bind.

```
<img v-bind:src="image"></img>
```

Una alternativa al v-bind sería utilizar los dos puntos, es decir, en lugar de "v-bind:src=...", podríamos utilizar ":src=...", muy útil a la hora de trabajar, por ahorrarnos tener que escribir más de la cuenta. Por lo que podríamos trabajar indistintamente con cualquiera de las dos alternativas.

Finalmente, añadimos el botón con una directiva v-on, que escuchará el evento click, que podríamos también haber simplificado con un @click, en lugar de v-on:click , para ejecutar el método responsable de cambiar el valor de la propiedad y cargar una nueva ruta de imagen al azar ❷ y ❸.

❶

firstvue.html

```
8   </head>
9
10  <body>
11    <div id="app">
12      <h1>v-bind image</h1>
13      <div class="button">
14        <button type="button" v-on:click="handleClick">Change image!</button>
15      </div>
16      <div class="image">
17        <img v-bind:src="image"></img>
18      </div>
19    </div>
```

firstvue.js

```
1  const app = new Vue({
2    el: '#app',
3    data: {
4      image: 'https://via.placeholder.com/320x240.png',
5      index: 0
6    },
7    methods: {
```

firstvue.html Preview

v-bind image

Change image!

320 x 240

Powered by HTML.COM

De esta forma tan sencilla hemos ejemplificado la definición que hablábamos al principio. Hemos enlazado un atributo de un elemento HTML con una propiedad de nuestra instancia.

2

firstvue.html

```
</head>

<body>
  <div id="app">
    <h1>v-bind image</h1>
    <div class="button">
      <button type="button" v-on:click="handleClick">Change image!</button>
    </div>
    <div class="image">
      <img v-bind:src="image"></img>
    </div>
  </div>
```

firstvue.js

```
const app = new Vue({
  el: '#app',
  data: {
    image: 'https://via.placeholder.com/320x240.png',
    index: 0
  },
  methods: {
    handleClick: function() {
      this.image = `https://loremflickr.com/320/240?random=${this.index++}`;
    }
  }
});
```

firstvue.html Preview

v-bind image

Change image!

3

firstvue.html

```
</head>

<body>
  <div id="app">
    <h1>v-bind image</h1>
    <div class="button">
      <button type="button" v-on:click="handleClick">Change image!</button>
    </div>
    <div class="image">
      <img v-bind:src="image"></img>
    </div>
  </div>
```

firstvue.js

```
const app = new Vue({
  el: '#app',
  data: {
    image: 'https://via.placeholder.com/320x240.png',
    index: 0
  },
  methods: {
    handleClick: function() {
      this.image = `https://loremflickr.com/320/240?random=${this.index++}`;
    }
  }
});
```

firstvue.html Preview

v-bind image

Change image!

Condicionales

Uso v-if y v-else

Durante el desarrollo de una aplicación es seguro que se nos presentarán ocasiones en las que queramos renderizar un bloque u otro de nuestra interfaz de usuario en función de alguna determinada condición.

En este caso haremos uso de las directivas v-if y v-else y, como ejemplo de las mismas, crearemos un nuevo ejercicio en el que en base a un selector mostraremos uno u otro bloque ❶.

En este pequeño ejercicio se pregunta al usuario qué le gusta más: los gatos o los perros; y en función de su elección mostraremos un mensaje y una foto de su animal favorito.

La directiva v-if siempre contiene una condición en su interior, del estilo v-if="miCondicion", si quisiéramos añadir una segunda alternativa al bloque anterior, proseguiríamos con un "sino si…", es decir "v-else-if", también acompañada por una condición en su interior como la anterior. En caso de añadir la condición de, si no se han cumplido todas las anteriores, entonces habilita este bloque, utilizaríamos la directiva v-else, sin condición.

Para llevarlo a cabo necesitaremos una propiedad de la instancia Vue que denominaremos *pet*, que ligaremos con la directiva v-model al selector. Al seleccionar uno de los valores posibles se dispara un cambio en las diferentes propiedades de la instancia. Posteriormente crearemos los dos bloques

❶

```
10 <body>
11   <div id="app">
12     <h1>Vue v-if v-else</h1>
13     <h2>Eres de perros o gatos?</h2>
14
15     <select v-model="pet" v-on:change="choosePet">
16       <option></option>
17       <option>dog</option>
18       <option>cat</option>
19     </select>
20     <div v-if="this.pet!=''">
21       <div v-if="this.pet === 'dog'">
22         <h1> Woof!</h1>
23         <img v-bind:src="image"></img>
24       </div>
25       <div v-else="this.pet !== 'dog'">
26         <h1> Miau!</h1>
27         <img v-bind:src="image"></img>
28       </div>
29     </div>
30   </div>
```

firstvue.js

```
1 const app = new Vue({
2   el: '#app',
3   data: {
4     pet: '',
5     image: ''
6   },
7   methods: {
8     choosePet: function(){
9       this.image = `https://loremflickr.com/320/240/${this.pet}`;
10    }
11  }
```

Vue v-if v-else

Eres de perros o gatos?

con el contenido opcional y los marcaremos con las directivas v-if y v-else. Estas directivas evalúan la condición y, en función de lo escogido, en el selector se mostrará el bloque del perro ❷ o del gato ❸.

❷

```
10  <body>
11    <div id="app">
12      <h1>Vue v-if v-else</h1>
13      <h2>Eres de perros o gatos?</h2>
14
15      <select v-model="pet" v-on:change="choosePet">
16        <option></option>
17        <option>dog</option>
18        <option>cat</option>
19      </select>
20      <div v-if="this.pet!=''">
21        <div v-if="this.pet === 'dog'">
22          <h1> Woof!</h1>
23          <img v-bind:src="image"></img>
24        </div>
25        <div v-else="this.pet !== 'dog'">
26          <h1> Miau!</h1>
27          <img v-bind:src="image"></img>
28        </div>
29      </div>
30    </div>
```

firstvue.js

```
1  const app = new Vue({
2    el: '#app',
3    data: {
4      pet: '',
5      image: ''
6    },
7    methods: {
```

❸

```
10  <body>
11    <div id="app">
12      <h1>Vue v-if v-else</h1>
13      <h2>Eres de perros o gatos?</h2>
14
15      <select v-model="pet" v-on:change="choosePet">
16        <option></option>
17        <option>dog</option>
18        <option>cat</option>
19      </select>
20      <div v-if="this.pet!=''">
21        <div v-if="this.pet === 'dog'">
22          <h1> Woof!</h1>
23          <img v-bind:src="image"></img>
24        </div>
25        <div v-else="this.pet !== 'dog'">
26          <h1> Miau!</h1>
27          <img v-bind:src="image"></img>
28        </div>
29      </div>
30    </div>
```

firstvue.js

```
1  const app = new Vue({
2    el: '#app',
3    data: {
4      pet: '',
5      image: ''
6    },
7    methods: {
```

Listas

Renderizado de listas

Otra situación típica en la creación de nuevas interfaces de usuario es la enumeración de elementos dentro de una lista. Para ello en Vue se utiliza la directiva v-for a la que se le pasa una colección de elementos que se representarán dentro del bloque.

Como ejercicio plantearemos una clásica lista de la compra. Simplemente crearemos dentro de la instancia Vue una propiedad que denominaremos *shoppingList* que es un array de cadenas de texto que representan cada uno de los productos de la lista.

Con la propiedad definida solo nos resta indicar mediante la directiva v-for que iteraremos por cada producto de nuestra lista de la compra y renderizaremos cada elemento de la lista <li>.

```
<ul>
   <li v-for="product in shoppingList">  {{ product }} </li>
   </ul>
```

Con esta simple definición ya tenemos nuestra lista de la compra. 1

1

firstvue.html

```
<head>
  <meta charset="utf-8">
  <title>My Vue</title>
  <style src="firstvue.css"></style>
  <link rel="stylesheet" href="firstvue.css">
</head>

<body>
  <div id="app">
    <h1>v-for</h1>
    <h2>Lista de la compra</h2>
    <ul>
      <li v-for="product in shoppingList">
        {{ product }}
      </li>
    </ul>
  </div>
```

firstvue.js

```
const app = new Vue({
  el: '#app',
  data: {
    shoppingList: ['manzanas', 'leche', 'hamburguesa', 'patatas fritas'],
  },
  methods: {
  }
});
```

firstvue.html Preview

v-for

Lista de la compra

- manzanas
- leche
- hamburguesa
- patatas fritas

Modificando ligeramente el código podemos añadir un nuevo input que añada elementos a mi compra con algunos mecanismos que hemos aprendido en anteriores ejercicios. De esta forma vemos cómo si añadimos otro nuevo elemento a nuestra lista de productos se actualiza nuestra compra ❷.

❷ firstvue.html

```
11  <div id="app">
12    <h1>v-for</h1>
13    <h2>Lista de la compra</h2>
14    <input v-on:keyup.enter="onEnterPressed" v-model="newProduct"/>
15    <ul>
16      <li v-for="product in shoppingList">
17        {{ product }}
18      </li>
19    </ul>
20  </div>
```

firstvue.js

```
1  const app = new Vue({
2    el: '#app',
3    data: {
4      newProduct: '',
5      shoppingList: ['manzanas', 'leche', 'hamburguesa', 'patatas fritas'],
6    },
7    methods: {
8      onEnterPressed: function() {
9        this.shoppingList.push(this.newProduct);
10       this.newProduct = '';
11     }
12   }
13 });
14
```

firstvue.html Preview

v-for

Lista de la compra

cerezas

- manzanas
- leche
- hamburguesa
- patatas fritas
- pan

Enumeración de propiedades

V-FOR PARA OBJETOS

Hemos visto cómo mostrar una enumeración de productos de nuestra lista de la compra con la directiva v-for y en este ejercicio veremos cómo mostrar un listado de las propiedades de un objeto de nuestra instancia Vue.

Tenemos una propiedad dentro de la instancia Vue que es un objeto que representa la información de uno de los posts de nuestro blog y que queremos renderizar.

```
post: {
      title: 'Nuevo artículo',
      date: '24/03/2019',
      author: 'Ramón',
}
```

Ahora lo que tenemos que hacer es usar la directiva v-for con el siguiente formato, descomponiendo el objeto en sus propiedades en pares clave-valor.

```
<ul>
   <li v-for="(value, key)  in post" v-blue>
      {{ key }}: {{ value }}
    </li>
 </ul>
```

También podríamos usar el índice en esta descomposición, si lo necesitáramos, de la siguiente manera:

```
<ul>
   <li v-for="(value, key, index)  in post"
      style= "list-style-type:none;" v-blue>
      {{ index + 1 }} => {{ key }}: {{ value }}
   </li>
</ul>
```

Podemos ver el resultado de la inspección del objeto en la imagen ❶ y para seguir jugando con nuestro ejemplo meter un nuevo atributo y ver cómo se renderiza perfectamente cuando guardamos. ❷

1

firstvue.html

```
11  <div id="app">
12    <h1>v-for object</h1>
13    <h2></h2>
14    <ul>
15      <li v-for="(value, key, index)  in post"
16          style= "list-style-type:none;">
17        {{ index + 1 }} => {{ key }}: {{ value }}
18      </li>
19    </ul>
20  </div>
```

firstvue.js

```
1  const app = new Vue({
2    el: '#app',
3    data: {
4      post: {
5        title: 'Nuevo articulo',
6        date: '24/03/2019',
7        author: 'Ramón',
8      },
9    }
10 });
```

firstvue.html Preview

v-for object

1 => title: Nuevo articulo
2 => date: 24/03/2019
3 => author: Ramón

2

firstvue.html

```
11  <div id="app">
12    <h1>v-for object</h1>
13    <h2></h2>
14    <ul>
15      <li v-for="(value, key, index)  in post"
16          style= "list-style-type:none;">
17        {{ index + 1 }} => {{ key }}: {{ value }}
18      </li>
19    </ul>
20  </div>
```

firstvue.js

```
1  const app = new Vue({
2    el: '#app',
3    data: {
4      post: {
5        title: 'Nuevo articulo',
6        date: '24/03/2019',
7        author: 'Ramón',
8        content: 'Este es el contenido'
9      },
10   }
11 });
```

firstvue.html Preview

v-for object

1 => title: Nuevo articulo
2 => date: 24/03/2019
3 => author: Ramón
4 => content: Este es el contenido

Filtros

Modificadores de valor

Vamos a imaginarnos que tenemos nuestra interfaz que renderiza la lista de la compra que hemos realizado en anteriores ejercicios. Hemos sacado la aplicación que muestra la lista de la compra pero los usuarios se quejan de que se muestran en letras minúsculas y ellos quieren que se vea bien grande y se muestre en letras mayúsculas aun cuando ellos los introduzcan en minúsculas.❶

Podremos modificar este comportamiento con una herramienta que pone a nuestra disposición Vue: los filtros.

Definiremos un filtro que pase a mayúsculas cada valor que se le pase en este caso de la siguiente forma:

```
Vue.filter('productName', function(value){
   if(!value) return ''
    value = value.toString();
    return value.toUpperCase();
});
```

Y en la declaración HTML lo haremos de la siguiente forma

```
<ul>
   <li v-for="product in shoppingList">
   {{ product | productName }}
   /li>
</ul>
```

Esta notación es el típico pipe que es muy común y que en este caso se podría leer algo así como "a este valor la aplico esta función o transformación".❷

1

firstvue.html

```
11  <div id="app">
12    <h1>Vue filters</h1>
13    <h2>Lista de la compra</h2>
14    <ul>
15      <li v-for="product in shoppingList">
16        {{ product }}
17      </li>
18    </ul>
19  </div>
```

firstvue.js

```
1  Vue.filter('productName', function(value){
2    if(!value) return ''
3    value = value.toString();
4    return value.toUpperCase();
5  });
6
7  const app = new Vue({
8    el: '#app',
9    data: {
10     shoppingList: ['apple', 'orange', 'banana', 'grapes'],
11   },
12   methods: {
13   }
14 });
```

firstvue.html Preview

Vue filters

Lista de la compra

- apple
- orange
- banana
- grapes

2

firstvue.html

```
11  <div id="app">
12    <h1>Vue filters</h1>
13    <h2>Lista de la compra</h2>
14    <ul>
15      <li v-for="product in shoppingList">
16        {{ product | productName }}
17      </li>
18    </ul>
19  </div>
```

firstvue.js

```
1  Vue.filter('productName', function(value){
2    if(!value) return ''
3    value = value.toString();
4    return value.toUpperCase();
5  });
6
7  const app = new Vue({
8    el: '#app',
9    data: {
10     shoppingList: ['apple', 'orange', 'banana', 'grapes'],
11   },
12   methods: {
13   }
14 });
```

firstvue.html Preview

Vue filters

Lista de la compra

- APPLE
- ORANGE
- BANANA
- GRAPES

Renderizar solo una vez

Directiva v-once

En caso de que necesitemos mostrar un único valor cuando carguemos la página haremos uso de una directiva especial llamada v-once. Esta directiva asegura que un elemento solo tomará un valor durante el renderizado y, en el caso de que la página se vuelva a renderizar, considerará al elemento como un elemento estático que no se tomará en cuenta de nuevo.

Una razón por la que se puede marcar un elemento con la directiva v-once sería optimizar el rendimiento en la actualización de datos. El contenido del elemento que se renderice una sola vez podría ser costoso.

En este ejemplo vemos cómo se renderiza una sola vez aquello que tengamos dentro de la variable incluida en el input ❶.

Si posteriormente a la carga modificamos el contenido del input y, por tanto, el contenido de la variable podemos comprobar cómo no se renderiza de nuevo el contenido de nuestro título ❷.

❶

firstvue.html

```
11  <div id="app">
12    <h1 v-once>{{ message }}</h1>
13
14    <input type="text" v-model="message"/>
15  </div>
```

firstvue.js

```
1  const app = new Vue({
2    el: '#app',
3    data: {
4      message: 'Solo una vez',
5    },
6    methods: {
7    }
8  });
```

firstvue.html Preview

Solo una vez

Solo una vez

2

firstvue.html

```
11  <div id="app">
12    <h1 v-once>{{ message }}</h1>
13
14    <input type="text" v-model="message"/>
15  </div>
```

firstvue.js

```
1  const app = new Vue({
2    el: '#app',
3    data: {
4      message: 'Solo una vez',
5    },
6    methods: {
7    }
8  });
```

firstvue.html Preview

Solo una vez

Solo una vez y nada más

firstvue.html

```
11  <div id="app">
12    <h1 v-once>{{ message }}</h1>
13
14    <input type="text" v-model="message"/>
15  </div>
```

firstvue.js

```
1  const app = new Vue({
2    el: '#app',
3    data: {
4      message: 'Solo una vez',
5    },
6    methods: {
7    }
8  });
```

Mouse move

Directiva v-on:mousemove

En esta ocasión vamos a ver dentro de las directivas para capturar eventos del DOM aquella que nos permite filtrar los generados cuando el ratón se mueve por encima de uno de nuestros componentes.

Estamos hablando de la directiva v-on acompañado del evento mousemove ❶:

```
<div class="mousearea" v-on:mousemove="move"></div>
```

A este div le daremos unas dimensiones mediante la clase CSS mousearea ❷:

```
.mousearea {
   width: 300px;
   border: 15px solid green;
   padding: 50px;
   margin: 20px;
}
```

Enlazaremos el evento con una función de la instancia para ir modificando las coordenadas mientras el ratón está en movimiento. Estas coordenadas las guardaremos y las mostraremos en otro div diferente como resultado.

❶

firstvue.html

```
8   </head>
9
10  <body>
11    <div id="app">
12      <h1>Mouse move</h1>
13      <div class="mousearea" v-on:mousemove="move">
14      </div>
15      <div class="result">{{mouseX}} - {{mouseY}}</div>
16    </div>
17
```

firstvue.js

```
1  const app = new Vue({
2    el: '#app',
3    data: {
4      mouseX:'0',
5      mouseY:'0'
6    },
7    methods: {
8      move:function(event){
9        this.mouseX = event.clientX;
10       this.mouseY = event.clientY;
11     }
12   }
13 });
14
```

firstvue.html Preview

Mouse move

0 - 0

```
methods: {
    move:function(event){
    this.mouseX = event.clientX;
    this.mouseY = event.clientY;
  }
}
<div class="result">{{mouseX}} - {{mouseY}}</div>
```

A medida que se captura el movimiento del ratón se ejecuta la función y vamos refrescando las variables de la instancia que se renderizarán en nuestra página ❸.

❷

firstvue.html | firstvue.css | firstvue.html Preview

```
1  h1 {
2    color: green;
3  }
4  .mousearea {
5    width: 300px;
6    border: 15px solid green;
7    padding: 50px;
8    margin: 20px;
9  }
10 .result {
11   color: green;
12   font-size: 2em;
13   font-weight: bold;
14 }
15
```

firstvue.js

```
1 const app = new Vue({
2   el: '#app',
3   data: {
4     mouseX:'0',
```

Mouse move

0 - 0

❸

firstvue.html | firstvue.html Preview

```
8  </head>
9
10 <body>
11   <div id="app">
12     <h1>Mouse move</h1>
13     <div class="mousearea" v-on:mousemove="move">
14     </div>
15     <div class="result">{{mouseX}} - {{mouseY}}</div>
16   </div>
17
```

firstvue.js

```
1 const app = new Vue({
2   el: '#app',
3   data: {
4     mouseX:'0',
5     mouseY:'0'
6   },
```

Mouse move

124 - 123

Directivas personalizadas

DEFINIENDO UNA DIRECTIVA GLOBAL

Vue permite la creación y registro de nuevas directivas a las que podremos dotar el comportamiento personalizado que queramos.

Las directivas personalizadas en Vue como el resto suelen empezar por v- y el nombre de la directiva. Como primer ejemplo empezaremos definiendo una directiva muy simple que cambie el color del texto renderizado.

De esta forma tendríamos una declaración tal que así:

```
<h1 v-red>Aprendiendo Vue!</h1>
```

En la que modificaremos el color del título a rojo tal y como indica la propia directiva. El código necesario en Javascript para definir la directiva sería el siguiente:

```
Vue.directive("red", function(el, binding, vnode) {
   el.style.color = "red";
});
```

Este código se ejecutará en el bind que es una función que se llama solo una vez cuando se enlaza la primera vez la directiva al elemento. Siendo más estrictos podríamos escribirlo así y con más funciones en función de cuando se quiere ejecutar el código.

```
Vue.directive("red", {
bind: function(el, binding, vnode) {
    el.style.color = "red";
}
});
```

Poniendo todo junto en el ejercicio como resultado tendríamos nuestros textos cambiados a color rojo ya que modificamos la propiedad color del estilo del elemento ❶.

Para seguir jugando con nuestro ejemplo podríamos definir otra directiva con el color azul y aplicarla al otro texto y ver el resultado❷.

1

firstvue.html

```
11  <div id="app">
12    <h1 v-red>Aprendiendo Vue!</h1>
13    <p v-red>Usando la directiva personalizada red!</p>
14  </div>
```

firstvue.js

```
1  Vue.directive("red", function(el, binding, vnode) {
2    el.style.color = "red";
3  });
4
5  const app = new Vue({
6    el: '#app'
7  });
8
```

firstvue.html Preview

Aprendiendo Vue!

Usando la directiva personalizada red!

2

firstvue.html

```
11  <div id="app">
12    <h1 v-red>Aprendiendo Vue!</h1>
13    <p v-blue>
14      Usando la directiva personalizada red!
15    </p>
16  </div>
```

firstvue.js

```
1  Vue.directive("red", function(el, binding, vnode) {
2    el.style.color = "red";
3  });
4
5  Vue.directive("blue", function(el, binding, vnode) {
6    el.style.color = "blue";
7  });
8
9  const app = new Vue({
10   el: '#app'
11 });
```

firstvue.html Preview

Aprendiendo Vue!

Usando la directiva personalizada red!

Directivas personalizadas II

Directivas con argumentos

Hemos visto cómo crear una directiva personalizada muy sencilla y en este nuevo ejercicio haremos una pequeña variación para que sea algo más genérica. Veíamos que para definir nuevos colores teníamos que crear nuevas directivas pero si tenemos que crear una para cada color… vaya lío, ¿no?

Lo ideal es que pudiéramos indicarlo de alguna forma y solo mantener una directiva y, para conseguirlo, sería genial poder tener una única directiva y pasarle un argumento con el color. ¡Esto es lo que vamos a hacer en este ejercicio!

```
<h1 v-color:blue>{{message}}</h1>
<p v-color:red> Usando la directiva personalizada v-color:arg! </p>
```

Declaramos la nueva directiva como v-color pero además le pasamos como argumento el color deseado de forma que colorearemos diferente cada texto según el argumento dado.

Si vamos a la definición podremos hacer algo como esto:

```
Vue.directive("color", function(el, binding, vnode) {
   el.style.color = binding.arg;
   console.log(binding);
});
```

El objeto binding guarda muchas propiedades como el nombre de la directiva, el valor, los modificadores, etc., y entre estas propiedades también podemos obtener el argumento ❶. Es este argumento el que recuperaremos y con el que podremos modificar la propiedad color del estilo del elemento ❷.

1

firstvue.js

```
Vue.directive("color", function(el, binding, vnode) {
    el.style.color = binding.arg;
    console.log(binding);
});
const app = new Vue({
  el: '#app',
  data: {
    message:'Aprendiendo Vue!!!'}
});

```

2

firstvue.html

```
<div id="app">
  <h1 v-color:blue>{{message}}</h1>
  <p v-color:red>
    Usando la directiva personalizada v-color:arg!
  </p>
</div>
```

firstvue.js

```
Vue.directive("color", function(el, binding, vnode) {
    el.style.color = binding.arg;
    console.log(binding);
});
const app = new Vue({
  el: '#app',
  data: {
    message:'Aprendiendo Vue!!!'}
});

```

firstvue.html Preview

Aprendiendo Vue!!!

Usando la directiva personalizada v-color:arg!

Directivas personalizadas III

Utilizando modificadores

En este nuevo ejercicio utilizaremos más propiedades de las directivas, hasta hora hemos visto como crear una directiva que cambia el color de un texto. Una muy simple con un comportamiento estático modificando el estilo del texto y otra un poco más avanzada en la que introducimos un parámetro.

En esta ocasión introduciremos el uso de los modificadores, que son simplemente nuevos decoradores que se le pueden añadir o no a la directiva.

Volviendo a las modificaciones de estilo de los textos, un buen ejemplo sería decidir el formato en función de ciertos modificadores. Para ello crearemos la directiva v-format que podrá recibir diferentes modificadores que cambien el estilo del texto del elemento.

Los modificadores se declaran a partir de un punto y seguido al nombre de la directiva u otro modificador. Este podría ser un ejemplo de declaración:

```
<h1 v-format.bold.underline.highlight>{{message}}</h1>
```

Para el código Javascript tendríamos algo como lo siguiente:

```
Vue.directive("format", function(el, binding, vnode) {
    const modifiers = binding.modifiers;if (modifiers.underline) {
      el.style.textDecoration = "underline";
    }
    if (modifiers.bold) {
      el.style.fontWeight = "bold";
    }
    if (modifiers.highlight) {
      el.style.background = "#eaeaea";
    }
});
```

En el objeto binding tenemos los modificadores ❶ que pueden haberse aplicado y que podremos consultar y aplicar en caso de que existan ❷.

1

firstvue.js

```
Vue.directive("format", function(el, binding, vnode) {
  const modifiers = binding.modifiers;

if (modifiers.underline) {
    el.style.textDecoration = "underline";
  }

if (modifiers.bold) {
    el.style.fontWeight = "bold";
  }

if (modifiers.highlight) {
    el.style.background = "#eaeaea";
  }
});

const app = new Vue({
  el: '#app',
  data: {
    message: "Aprendiendo Vue!"
  }
});

```

2

firstvue.html

```
  <div id="app">
    <h1 v-format.bold.underline.highlight>{{message}}</h
  </div>
```

firstvue.js

```
Vue.directive("format", function(el, binding, vnode) {
  const modifiers = binding.modifiers;

if (modifiers.underline) {
    el.style.textDecoration = "underline";
  }

if (modifiers.bold) {
    el.style.fontWeight = "bold";
  }

if (modifiers.highlight) {
    el.style.background = "#cacaca";
  }
});

const app = new Vue({
  el: '#app',
  data: {
    message: "Aprendiendo Vue!"
  }
});
```

firstvue.html Preview

Aprendiendo Vue!

Directivas personalizadas IV

Directivas en la instancia

Hasta ahora hemos visto cómo crear y declarar nuestras directivas personalizadas de forma global pero en este ejercicio podremos ver cómo se definen de forma local dentro de una instancia o componente Vue.

Para ello simplemente tendremos que ir al código javascript de nuestra instancia y añadir un nuevo objeto *directives* con una sintaxis ligeramente diferente a la que hemos visto hasta ahora.

En este ejemplo condensaremos varias de las directivas de los ejercicios anteriores añadiéndolas en este apartado con sus correspondientes funciones.

En la declaración no cambia nada:

```
<h1 v-color:yellow v-format.bold.underline.highlight>{{message}}</h1>
```

Y el código JS quedaría de la siguiente forma:

```
directives: {
    'white': {
      bind(el, binding, vnode) {
           el.style.color = '#fff';
      }
    },
```

1

firstvue.html

```
11  <div id="app">
12    <h1 v-color:yellow v-format.bold.underline.highlight>
13      {{message}}
14    </h1>
15  </div>
```

firstvue.js

```
1  const app = new Vue({
2    el: '#app',
3    data: {
4      message: "Aprendiendo Vue!"
5    },
6    directives: {
7      'white': {
8        bind(el, binding, vnode) {
9            el.style.color = '#fff';
10       }
11     },
12     'color': {
13       bind(el, binding, vnode) {
14           el.style.color = binding.arg;
15       }
16     },
17     'format': {
18       bind(el, binding, vnode) {
19         const modifiers = binding.modifiers;
20
```

firstvue.html Preview

Aprendiendo Vue!

```
'color': {
  bind(el, binding, vnode) {
        el.style.color = binding.arg;
  }
},
  'format': {
        bind(el, binding, vnode) {
              const modifiers = binding.modifiers;
              if (modifiers.underline) {
              el.style.textDecoration = "underline";
              }
              …
        },
  }
}
});
```

Tendríamos un resultado similar ❶ al de ejercicios anteriores ❷.

Componente

IMPORTANTE

La nomenclatura del componente es importante, debe ser de tipo camel-case "MiComponente" o kebab-case "mi-componente", en definitiva, una forma de unir palabras.

Componente

Los componentes en Vue son unos contenedores donde implementamos lógica para que funcionen de manera aislada y puedan ser reutilizados por la aplicación.

Creemos un componente donde, al escribir una palabra, se muestre su traducción si existe en el diccionario.

Al usar CDN en lugar del CLI, para definir un componente, usamos "Vue.component", cuyo primer parámetro es el nombre del componente.

Creamos la página html, que incluye la librería de vue, el css y el contenedor de la aplicación, donde incluimos la etiqueta del componente a crear, "<my-translator />" ❶.

Además, creamos el fichero "firstvue.css", con un estilo para agrandar los elementos de tipo texto ❷.

Definimos el componente, en el fichero "firstvue.js" ❸. Nombramos al componente como "my-translator".

```
let Translator = Vue.component('my-translator', { })
```

En la propiedad "template", definimos su plantilla, mediante "template literal", comillas "`" usadas a principio y fin del html de la plantilla, para visualizarla en varias líneas. Definimos una

❶
```
<html><head><meta charset="utf-8">
  <title>Componente Traductor</title>
  <link rel="stylesheet" type="text/css" href="firstvue.css"> </head>
<body>                                  Aplicación compuesta por un componente
  <div id="app">                                  Componente Traductor
    <my-translator></my-translator>
  </div>
  <script src="https://unpkg.com/vue"></script>
  <script src="firstvue.js"></script>
```

❷
```
input[type=text]{ width: 20em; height: 3em; }
```

❹ **Traductor**

h | Limpiar

Hola->Hello

❺ **Traductor Castellano-Inglés**

Introduzca Palabra a traducir | Limpiar

caja de texto, que representa la palabra "word", que el usuario escribe para buscar su traducción. Mediante "v-model" vinculamos el input con el modelo.

Incluimos un botón para limpiar el valor del input al pulsarlo, así como las sugerencias que la propiedad computada "AnyMatch", nos retorna.

En la propiedad "data" incluimos las variables, "placeholderWord", texto en el placeholder del input, "word" y "dictionary", que contiene una lista de palabras con sus traducciones entre castellano "ES" e inglés "EN".

En la propiedad "computed" añadimos la propiedad computada "AnyMatch" que recorre el diccionario y compara cada palabra, de modo que si el valor contiene la palabra "word", la añade a un array retornado por la función.

En la propiedad "methods" definimos el método "clear", para vaciar el input al pulsar el botón "limpiar".

Tras definir el componente, lo añadimos a la instancia de la aplicación, en la propiedad "components", mediante kebab-case.

```
const app = new Vue({ el: '#app', components: { 'my-translator': Translator } });
```

Abrimos en el navegador el html, visualizamos el componente, tras introducir letras en la caja de texto, deberían aparecer las sugerencias de traducción ❹. Si pulsamos el botón limpiar, se vacía el contenido del input ❺ y las sugerencias.

```
❸ let Translator = Vue.component('my-translator', {
    template: `<div component="my-translator">
                <h1>Traductor</h1>
                <input type="text" :placeholder="placeholderWord" v-model="word" />
                <input type="button" @click="Clear" value="Limpiar" />
                <div v-for="match in AnyMatch">
                  <span>{{match.ES}}->{{match.EN}}</span>
                </div>
              </div>`
  ,data(){ return { placeholderWord:'Introduzca Palabra a traducir', word:''
                  , dictionary:[{'EN':'Hello', 'ES':'Hola'},{'EN':'Bye', 'ES':'Adios'}
                  ,{'EN':'Good', 'ES':'Bueno'},{'EN':'Bad', 'ES':'Malo'}] } }
  ,computed:{
    AnyMatch(){ let match=false; let words=[];
      this.dictionary.map((w)=>{if( this.word!='' &&
          w.ES.toLowerCase().includes(this.word.toLowerCase()) ) words.push(w) })
      return words }
  }
  ,methods:{ Clear(){ this.word = '' } }
})
const app = new Vue({ el: '#app', components: { 'my-translator': Translator } });
```

Nombre del componente
Plantilla del componente
Data-bind palabra
Evento clic, limpiar palabra
Modelo
Palabra
Array de palabras disponibles
Propiedad Computada
Incluir palabra en lista si coincide
Visualizamos sugerencias de traducciones si contiene la palabra escrita
Método para limpiar el input de palabra
Incluimos los componentes que utiliza la aplicación
Instancia de la aplicación

Componente parametrizables

IMPORTANTE

Los componentes en entorno CDN se declaran como Vue.component, sin embargo su definición varía en entornos CLI, que veremos más adelante.

Reutilizables

Los componentes pueden reutilizarse múltiples veces y parametrizarse para condicionar su comportamiento. Realicemos unas modificaciones en el ejercicio anterior para crear dos instancias del mismo componente y que su comportamiento sea diferente en base a un parámetro recibido en las props.

Para ello, el html incluye dos componentes del mismo tipo "my-translator" ❶, el primero para traducciones de castellano, indicado mediante la propiedad "prop-language" con el valor "ES", y el segundo para traducciones de inglés, con valor "EN". La hoja de estilos "firstvue.css" será idéntica a la anterior ❷.

La definición del componente ❸ es idéntica al anterior, con algunas variaciones. En la sección de la plantilla, comprobamos el valor de la propiedad "propLanguage", proporcionada en las props al componente, de modo que, dependiendo del idioma, mostraremos un texto u otro como título, mediante un condicional "v-if". Este mismo condicional entrará en juego a la hora de mostrar las sugerencias recorridas con el "v-for".

La sección del data no varía. Sin embargo, la sección de las propiedades computadas, en lugar de establecer por defecto el idioma "ES" lo hacemos dinámico mediante la prop.

❶
```
<html><head><meta charset="utf-8">
  <title>Componente Traductor</title>
  <link rel="stylesheet" type="text/css" href="firstvue.css"> </head>
<body>                                  Aplicación compuesta por dos componentes
  <div id="app">                        Componente Traductor. Castellano-Inglés
    <my-translator prop-language="ES"></my-translator>
    <my-translator  prop-language="EN"></my-translator>
  </div>                                Componente Traductor. Inglés-Castellano
  <script src="https://unpkg.com/vue"></script>
  <script src="firstvue.js"></script>
</body></html>
```

❷
```
input[type=text]{ width: 20em; height: 3em; }
```

❹ **Traductor Castellano-Inglés**

Hola->Hello

Traductor Inglés-Castellano

Bye->Adios
Bad->Malo

❺ **Traductor Castellano-Inglés**

```
this.dictionary.map( (w)=> {if(this.word != '' &&
w[this.propLanguage].toLowerCase().includes(this.word.toLowerCase())   )   words.
push(w) })
```

Tanto tiempo hablando de "propLanguage" y no la habíamos definido; podemos ver cómo una propiedad puede influir en el comportamiento de nuestro componente. Definamos esta propiedad, estableciendo su tipo a texto mediante "type" y su valor por defecto "ES", es decir, en caso de no indicar la propiedad, tomará este valor.

```
,props:{ propLanguage: {type:String,default:'ES'} }
```

Si abrimos el html en el navegador aparecen dos componentes, su título condicionado por el valor del idioma proporcionado a cada componente. Si escribimos en el componente de castellano, sugerirá palabras de castellano a inglés ❹, si lo hacemos en el componente de inglés será al revés. Finalmente, una vez queramos borrar el contenido de sugerencias y la palabra, pulsaremos el botón "limpiar" del componente que deseemos vaciar ❺, observando un comportamiento aislado entre componentes.

Componente Tipo Propiedades

IMPORTANTE

Cuando enviamos propiedades al componente, si ponemos los dos puntos delante del nombre de la propiedad, estamos enviando un valor dinámico, en caso contrario será un valor en plano.

Tipos de Propiedades

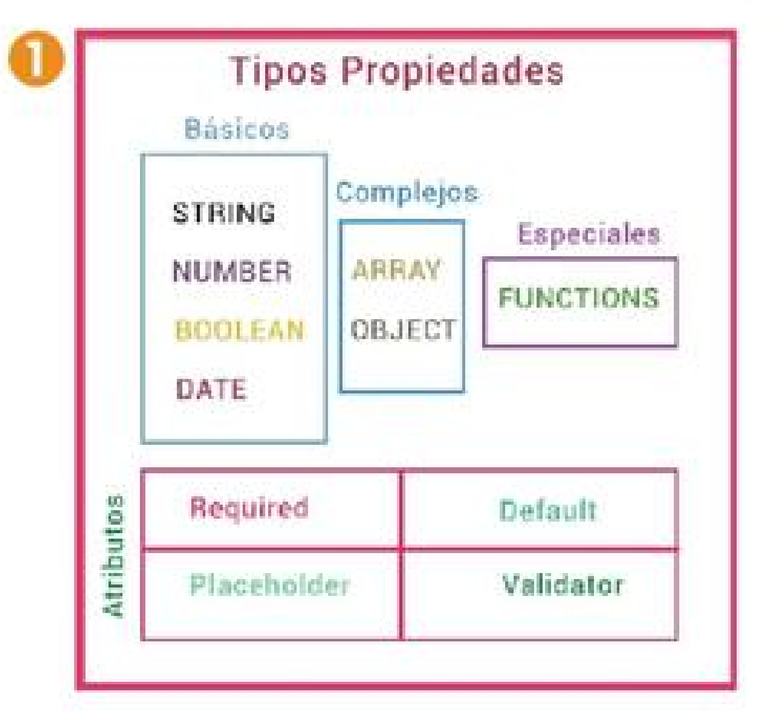
Hemos visto la importancia de las props en los componentes, son la primera línea de entrada de estos y por tanto deben ser el primer punto por validar para que nuestros componentes tengan el funcionamiento esperado.

Creemos un ejercicio para trabajar con los tipos de valores que podemos validar ❶.

Para ello, creamos en el fichero "firstvue.js", el componente "my-props" ❷, donde en la sección props, definiremos:

- propRequired: texto, debido al "type:String" y será requerido recibir valor "required:true".
- propArray: array, su valor por defecto se asigna mediante "default", en este caso un array vacío. Usamos "validator", función de validación, para verificar su tamaño mínimo.
- propObject: objeto, cuyo valor por defecto es un objeto vacío. En este caso, queremos tratar el objeto para retornar un valor, lo haremos mediante la propiedad computada "CompleteName".
- propMultipleValue: permite número, texto o fecha, al incluirlos en un array de tipos permitidos.

❶



❷

```
let MyProps = Vue.component('my-props', {
    template: `<div component="my-props">
                <p>Texto Requerido:{{myRequired}}*</p>
                <p>Objeto:{{CompleteName}}</p>
                <p>Múltiple Valor:{{myMultipleValue}}</p>
        Array:<ul v-for="item in myArray"> <li>{{item}}</li> </ul>
            </div>`                 Inicializamos variables locales con el valor de las props
  ,data(){ return {  myRequired: this.propRequired
                    ,myArray: this.propArray
                    ,myMultipleValue: this.propMultipleValue } }
  ,computed:{ CompleteName(){          Tratar el valor en propiedades computadas, antes de mostrarlo
              if(this.propObject && this.propObject.Name && this.propObject.LastName)
                return this.propObject.Name+' '+this.propObject.LastName
              else return '--Falta Nombre--' } }
  ,props:{ propRequired: {type:String, required:true} Es requerida
           ,propArray:   {type:Array,  default:()=>{ return [] }Valor por defecto, array vacío
     Validar tamaño minimo-> ,validator: values => { return values && values.length>0 } }
           ,propObject:  {type:Object, default:()=>{ return {} } }
           ,propMultipleValue: [Number,String,Date] }    Permite varios tipos para la propiedad
}})

const app = new Vue({ el: '#app', components: { 'my-props': MyProps } });
```

Propiedades

Las propiedades recibidas en las "props" podemos utilizarlas directamente en la plantilla, inicializar variables del data o tratar el valor recibido por las props, en las propiedades computadas y retornar el valor tratado, como es el caso de "CompleteName". Aquí tratamos el objeto recibido, retornando el nombre completo como valor. En caso de no querer tratar el valor recibido, simplemente asignamos el valor de la prop a su respectiva variable en el data.

En la plantilla mostramos los valores de estas variables del data, así como de la propiedad computada.

Finalmente, en "firstvue.html" instanciamos el componente con cada propiedad rellena como debería ❸. De modo que, si visualizamos la página en el navegador, visualizamos los valores correctamente ❹.

Sin embargo, si no enviásemos la propiedad requerida, o no enviásemos elementos en el array, la consola del navegador mostraría los errores ❺, sería lo esperado, al no superar la validación:

```
<my-props :prop-array=[]:prop-object='{"Name":"Ramón", "LastName":"Serrano Vale-
ro", "Age":34}'

   prop-multiple-value="25-10-1984" />
```

❸

```
<html><head><meta charset="utf-8">
  <title>Componente Tipos Propiedad</title></head>
<body>
  <div id="app">
    <my-props prop-required="Hola mundo"
              :prop-array=["España","Europa","Mundo"]
              :prop-object='{"Name":"Ramón", "LastName":"Serrano Valero", "Age":34}'
              prop-multiple-value="25-10-1984"
              />
  </div>
  <script src="https://unpkg.com/vue"></script>
  <script src="firstvue.js"></script>
</body></html>
```

Valor estático

Asignar valor dinámico

❹

Texto Requerido:Hola mundo*

Objeto:Ramón Serrano Valero

Múltiple Valor:25-10-1984

Array:

- España
- Europa
- Mundo

❺

```
Elements  Console  Sources  Network  Performance  Memory  »  2
top  Filter  Default levels
▸[Vue warn]: Missing required prop: "propRequired"                    vue.js:634
  found in
  ---> <MyProps>
         <Root>
▸[Vue warn]: Invalid prop: custom validator check failed for prop     vue.js:634
  "propArray".
  found in
  ---> <MyProps>
         <Root>
```

Cambiar valor de props

IMPORTANTE

Las propiedades recibidas por los componentes se reciben al cargar el componente, cualquier modificación de valor posterior, requiere capturar estos cambios mediante watch.

Cambio Valor

Hasta ahora, enviábamos un valor a un componente, recibiéndolo por las props, sin embargo, no hemos probado que ocurriría si actualizásemos dicho valor. ¿Refrescaría su valor el componente?

Creemos un ejercicio en el que actualicemos el valor de la prop y esta muestre el nuevo valor recibido.

Definimos en "firstvue.html" ❶, dentro de la instancia de la aplicación, una caja de texto cuyo valor realice "data-bind" con la variable "valueData" del data de la aplicación cada vez que modifiquemos el texto de este input. Además, instanciamos el componente "my-props-component" que crearemos, al que proporcionamos por su propiedad "prop-value", el valor de esta variable del data de la aplicación.

En el fichero "firstvue.js", creamos el componente "my-props-component" ❷, en el que definimos una propiedad de tipo texto "propValue", en las props. Esta inicializa el valor de la variable local "myValue", del data del componente.

❶
```
<html><head><meta charset="utf-8">
  <title>Actualizar Props</title></head>
<body>
  <div id="app">
    <input type="text" v-model="valueData" />
      Le pasamos lo que escribamos en el input
    <my-props-component :prop-value="valueData"/>
  </div>                Componente

  <script src="https://unpkg.com/vue"></script>
  <script src="firstvue.js"></script>
</body></html>
```

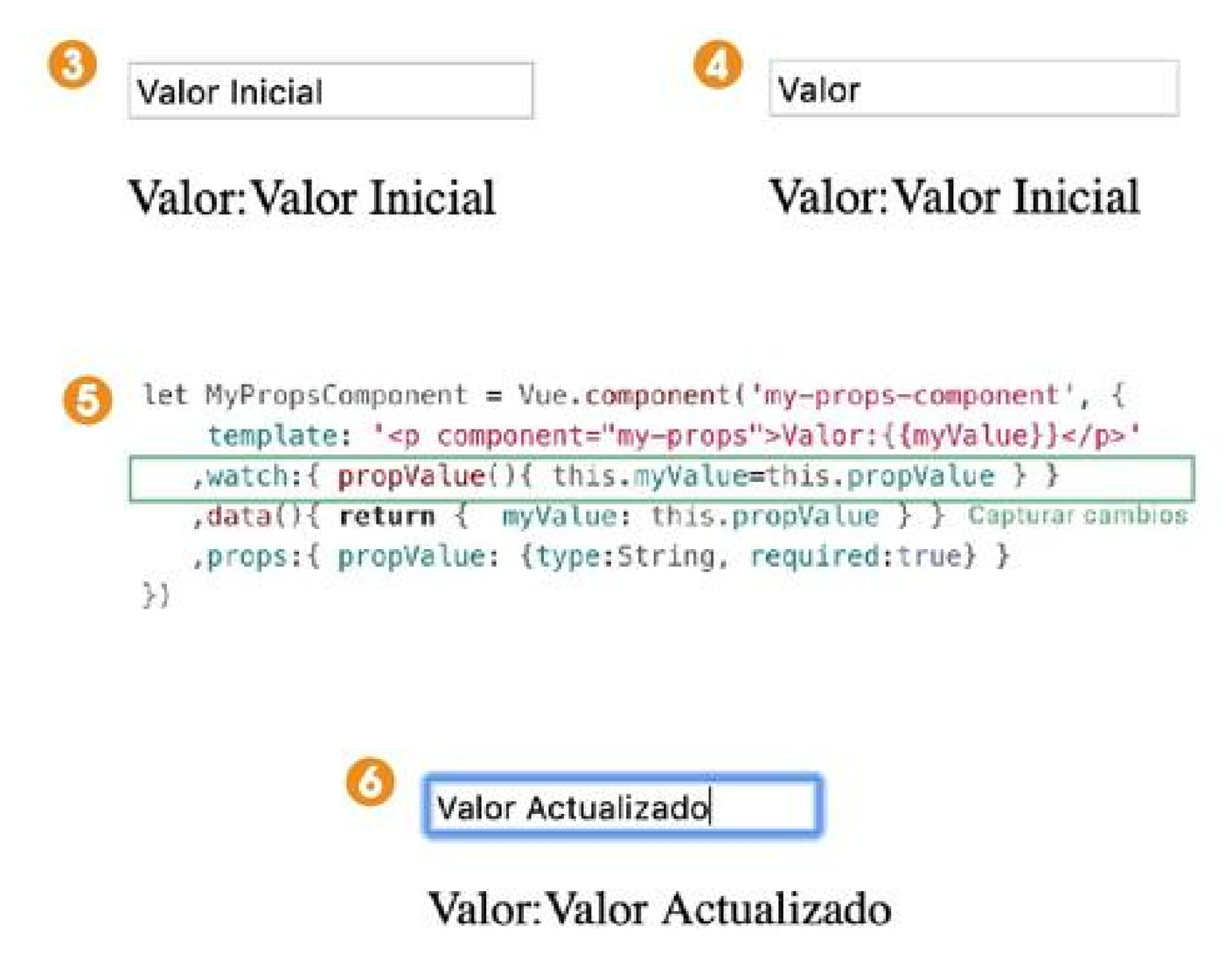
❷
```
let MyPropsComponent = Vue.component('my-props-component', {
    template: '<p component="my-props">Valor:{{myValue}}</p>'
   ,data(){ return {  myValue: this.propValue } }
   ,props:{ propValue: {type:String, required:true} }
})
          Recibe el valor de la prop, por aqui

const app = new Vue{{ el: '#app'                Variable que recibe el valor del input
                    , data(){ return { valueData: "Valor Inicial" } }
                    , components: { 'my-props-component': MyPropsComponent } });
```

Finalmente, agregamos el componente a la aplicación, mediante "components" y agregamos la variable "valueData" al data de la aplicación, con el valor por defecto "Valor Inicial".

Abrimos la página html en el navegador, veremos el valor "Valor Inicial" tanto en el input como en el componente ❸, dado que es el valor con el que inicializamos dicha variable de la aplicación y es recibida correctamente por las props del componente. Si ahora intentamos cambiar el valor de dicho input, vemos que el valor se está pasando por las props al componente, pero el componente no representa dichos cambios ❹.

Esto se debe a que el valor proporcionado en las props inicializa las variables del data del componente la primera vez. De modo que, al recibir actualizaciones, el valor de dicha propiedad no lo controlamos en el componente.

Para controlar estos cambios de valor debemos hacer uso de las "watch", y en este caso observar cambios en la propiedad "propValue", por lo que si añadimos este "watch" al componente ❺, asignamos el nuevo valor recibido al data. Abrimos de nuevo la página en el navegador y al cambiar el valor, el componente actualiza la vista ❻.



Reactividad Data

IMPORTANTE

Cualquier borrado o añadido de un atributo de tipo objeto del data requiere del uso de "Vue.set" y de "Vue.delete" para activar la reactividad al añadir o borrar atributos respectivamente.

Reactividad

La reactividad en Vue es un concepto muy importante: cualquier cambio en una variable del data representada en la vista será detectada por un observador y la vista se refrescará con los cambios ❶.

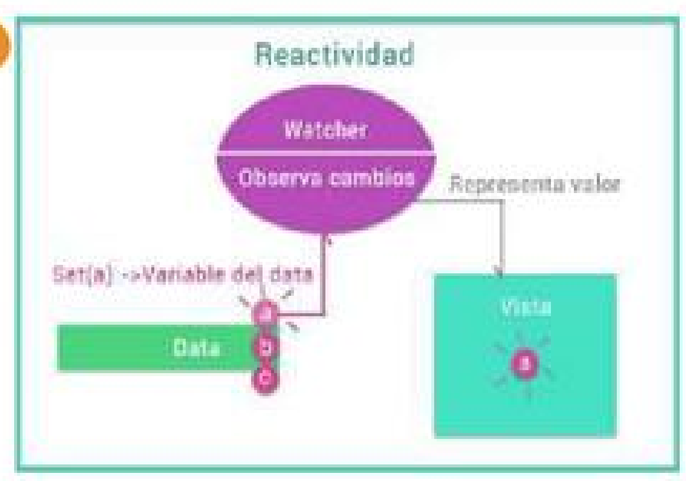
Aunque, como veremos, no siempre ocurre la reactividad. Creemos un ejercicio que nos permita activar reactividad.

Creamos "firstvue.html", con la etiqueta del componente "my-reactivity" ❷, al que le pasamos un objeto persona.

Este valor por defecto es enviado por la propiedad "propPerson" de tipo objeto al componente, por lo que creemos en el fichero "firstvue.js" ❸, el componente.

En la plantilla visualizamos las propiedades del objeto persona, únicamente hemos enviado "FirstName" y "LastName". Declaramos en el data, una propiedad para el input de edad y otra para el del nombre, así como dos botones, uno para borrar la propiedad "FirstName" y otro para añadir la propiedad "Age". Si nos fijamos en los métodos definidos, cada vez que cambie el

❶



❷

```
<html><head><meta charset="utf-8">
  <title>Reactividad</title></head>
<body>
  <div id="app">
    <my-reactivity :prop-person="{ FirstName: 'Ramón', LastName: 'Serrano' }" />
  </div>
```

❸

```
let MyReactivityComponent = Vue.component('my-reactivity', {
  template: `<div component="my-reactivity">
                <p v-if="person"><b>(Reactividad Objeto)Persona:</b>  Visualizar Datos personales
                  <span>{{person.FirstName}} {{person.LastName}} {{person.Age}}</span></p>
Introducir Nombre  <input type="text" placeholder="Nombre"
                        v-model="name" @change="OnChangeName" />

                                                  Visualizar Edad
                <p><b>(Reactividad)Edad:</b>{{edad}}</p>
Introducir Edad  <input type="number" placeholder="Edad" v-model="edad"/>
                  <br />
                <input type="button" @click="DeleteFirstName" value="Borra Nombre" />
                <input type="button" @click="AddAge" value="Añade Edad" />
            </div>`

  ,data(){ return {  person: this.propPerson, name:null, edad:null } }
  ,props:{ propPerson: {type:Object} }
  ,methods:{
     OnChangeName(){ this.person.FirstName=this.name; }
    ,DeleteFirstName() { delete this.person.FirstName; }
    ,AddAge() { this.person.Age=this.edad }
  }
})

const app = new Vue({ el: '#app', components: { 'my-reactivity': MyReactivityComponent } });
```

No causan reactividad

Borrar / Añadir propiedad

valor del input nombre, el método "OnChangeName" se ejecuta, asignando el valor del "name" al atributo "FirstName" del objeto "person".

Como el atributo "FirstName" existe en el objeto persona, en la pantalla ❹ al cambiar el valor al nombre se observa el cambio, de modo que la vista es actualizada mostrando el valor en pantalla ❺.

Si cambiamos el valor en el input de edad, la reactividad es instantánea, al cambiar el valor de una variable del data directamente y no al atributo de un objeto con el que la reactividad tiene problemas ❻.

Ahora bien, si borramos la propiedad "FirstName" del objeto "person", la reactividad no sucede debido a que esta no se activa al añadir o borrar atributos a un objeto ❼. Sucediendo lo mismo al añadir el atributo "Age", tras pulsar el botón "Añade Edad" ❽.

Para forzar la reactividad ante añadir nuevos atributos o asignar valores sobre el atributo de un objeto utilizamos:

```
Vue.set(objeto,"NombreAtributo",valor)
```

Mientras que para activarla en el borrado de un atributo de un objeto usaremos:

```
Vue.delete(objeto,"NombreAtributo")
```

Si modificamos los métodos anteriores por ❾ debería ser reactivo todo cambio sobre el objeto, tanto si borramos atributo ①, como si añadimos atributo ②, como si asignamos valor ③.

❹ (Reactividad Objeto)Persona: Ramón Serrano
Nombre
(Reactividad)Edad:
Edad
Borra Nombre Añade Edad

❼ (Reactividad Objeto)Persona: Juan Serrano
No se ha refrescado
Juan
(Reactividad)Edad:34
34
Borra Nombre Añade Edad

① (Reactividad Objeto)Persona: Serrano
Reactividad activa
Juan
(Reactividad)Edad:34
34
Borra Nombre Añade Edad

❺ (Reactividad Objeto)Persona: Juan Serrano
Juan

❽ (Reactividad Objeto)Persona: Juan Serrano
No refresca vista
Juan
(Reactividad)Edad:34
34
Borra Nombre Añade Edad

② (Reactividad Objeto)Persona: Serrano 34
Reactividad activa
Juan
(Reactividad)Edad:34
34
Borra Nombre Añade Edad

❻ (Reactividad Objeto)Persona: Juan Serrano
Juan
(Reactividad)Edad:34
34
Borra Nombre Añade Edad

❾
```
,methods:{
  OnChangeName(){ //this.person.FirstName=this.name;
    Vue.set(this.person,"FirstName", this.name) }
 ,DeleteFirstName() { //delete this.person.FirstName;
    Vue.delete(this.person,"FirstName") }
 ,AddAge() { //this.person.Age=this.edad
    Vue.set(this.person,"Age", this.edad) }
}
```

③ (Reactividad Objeto)Persona: Juanito Serrano 34
Juanito
Reactividad activa

Reactividad NextTick

IMPORTANTE

Cuando queramos forzar la reactividad, podemos hacer uso de Vue.nextTick, dado que lo que incluyamos dentro de dicho bloque, se ejecutará tras haber finalizado la asignación en el DOM.

NextTick

La reactividad es muy caprichosa, la vista no se refresca a no ser que la variable del data observada cambie de valor, sin embargo nos preguntamos ¿qué ocurriría en caso de querer forzar la reactividad asignando un mismo valor?

Veremos cómo la reactividad ocurre la primera vez que cambia el valor de la variable, pero no en sucesivos y cómo forzar esta reactividad.

Creamos el fichero "firstVue.html" ❶, donde incluimos la etiqueta del componente "my-reactivity".

Tras esto, creamos el fichero "firstVue.js", donde crearemos el componente ❷, con un input sobre el que queremos realizar el "focus" tras pulsar el botón. Incluimos la variable "focus", inicializada a "false", así como un método "DoFocus", donde establecemos el valor de la variable a "true".

Declaramos un watch, de tal forma que cuando ocurran cambios sobre esta variable realice el focus sobre el input, en caso de tener

❶
```
<html><head><meta charset="utf-8">
  <title>Reactividad NextTick</title></head>
<body>
  <div id="app"> <my-reactivity /> </div>

  <script src="https://unpkg.com/vue"></script>
  <script src="firstvue.js"></script>
</body></html>
```

❷
```
let MyReactivityComponent = Vue.component('my-reactivity', {
  template: `<div component="my-reactivity"> <p>Focus:{{focus}}</p>
Input para hacer foco <input type="text" ref="nombreRef" value="Ramón" />
                <input type="button" @click="DoFocus" value="Hacer Foco" />
            </div>` Variable para hacer foco
   ,data(){ return { focus:false } }
   ,watch: {
            focus: function () { console.log('Focus:'+this.focus)
      Hacer foco si la variable es True if (this.focus) this.$refs.nombreRef.focus() }
   }
   ,methods:{ DoFocus(){ this.focus=true } }   Método para hacer foco en input
})
const app = new Vue({ el: '#app'
            , components: { 'my-reactivity': MyReactivityComponent } });
```

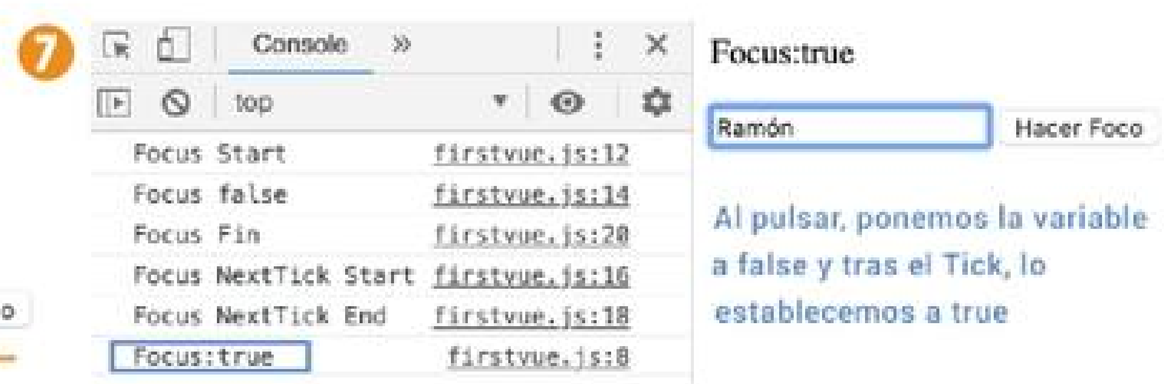
valor "true".

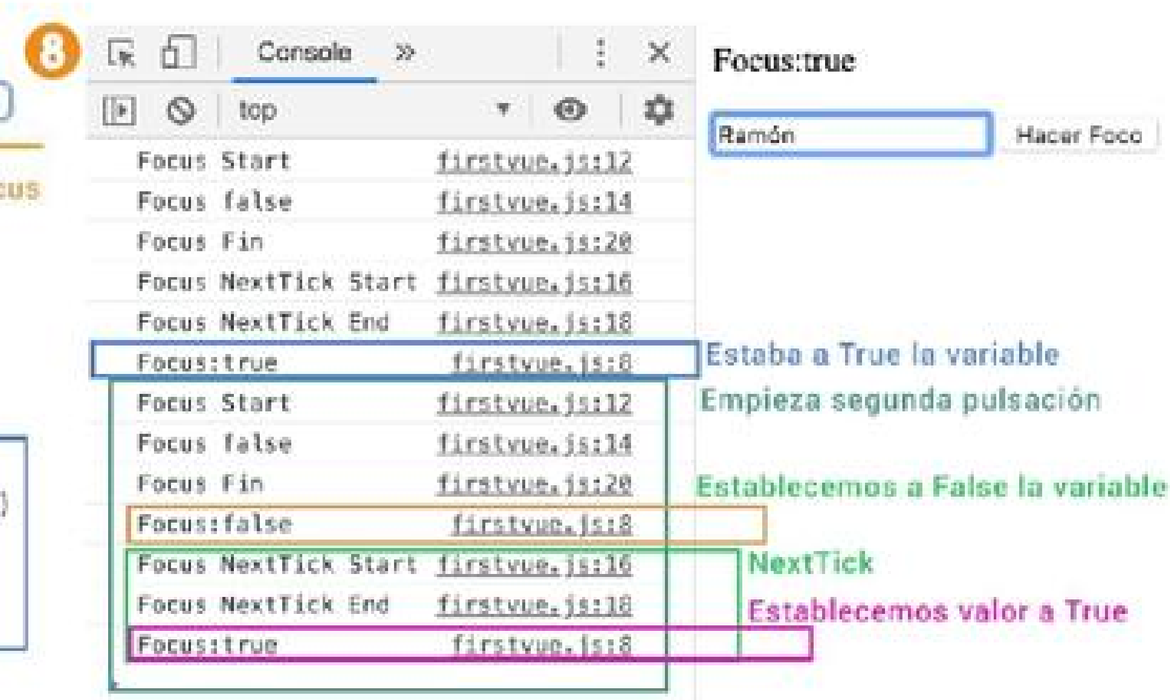
Abrimos la página html ❸, al pulsar el botón, como el valor de la variable era falso, entra en el método y establece el valor a cierto. El "watch" detecta el cambio y realiza el foco ❹. Si lo intentamos de nuevo, no vuelve a realizar el foco porque el valor era cierto y al vuelve a serlo, por lo que el watch no detecta cambios y la reactividad no ocurre ❺.

Modificamos el código del método ❻, en lugar de asignar siempre a cierto, lo inicializamos a falso, dejamos que dicho valor forme parte del DOM y tras esto, se ejecuta el bloque "Vue.nextTick", donde cambiamos el valor a cierto.

Abrimos de nuevo el html, el valor es falso, al pulsar, ponemos su valor a falso y tras asignarlo al DOM, se ejecuta el bloque "nextTick" cambiando su valor a true, por lo que el watch detecta el cambio y ocurre la reactividad ❼.

Ahora el valor es cierto, pulsamos de nuevo, cambia el valor a falso, lo asigna al DOM y tras esto en el "nextTick", cambiamos a cierto, detectamos el cambio y ocurre la reactividad ❽.





6

```
,methods:{ DoFocus(){ //this.focus=true
                console.log('Focus Start')
                this.focus=false
                console.log('Focus false')
                Vue.nextTick(()=>{
                console.log('Focus NextTick Start')
                this.focus=true
                console.log('Focus NextTick End')
                })
                console.log('Focus Fin')
        }
}
```

Establecemos a False la variable

Esperamos al siguiente pulso

Establecemos a True la variable

Directiva v-show vs v-if

IMPORTANTE

Utilice condicional cuando el contenido a mostrar u ocultar sea poco frecuente, ya que invierte recursos en representar de nuevo el contenido, y afecta al rendimiento.

V-SHOW

En ejercicios anteriores hablamos de los condicionales: cuando no cumplen la condición, el contenido en lugar de ocultarse en el DOM, es eliminado. Sin embargo, la directiva "v-show" no elimina del DOM el contenido al no cumplir la condición, sino que la oculta, utilizando el atributo "display" de CSS.

Veamos la diferencia entre ambos, definimos en "firstvue.js", la variable "step" en el data de la instancia vue:

```
const app = new Vue({ el: '#app', data(){ return  {
step:0 } } });
```

Declaremos condicionales en "firstvue.html" ❶, para mostrar un párrafo, dependiendo del valor del "step", así como también varios párrafos con "v-show". Un botón incrementa el valor de "step", para mostrar u ocultar párrafos.

❶

```
<html><head><meta charset="utf-8">
  <title>V-SHOW vs Condicionales</title>
  <link rel="stylesheet" type="text/css" href="firstvue.css"> </head>
<body>
  <div id="app" >
                                                        Bloque condicionales
    <p><b>Condicional</b></p>
    <p v-if="step==0">{{step}}-Si desaparece, este contenido no estará en DOM </p>
    <p v-else-if="step==1">{{step}}-Inspecciona DOM, lo anterior no esta!</p>
    <p v-else>FIN</p>
                                                        Bloque V-SHOW
    <p><b>V-SHOW</b></p>
    <p v-show="step==0">{{step}}-Si desaparece, este contenido permanenterá en DOM</p>
    <p v-show="step==1">{{step}}-Inspecciona DOM, el anterior permanece!</p>

    <input type="button" @click="step++" value="Incrementa pasos" />
  </div>

  <script src="https://unpkg.com/vue"></script>
  <script src="firstvue.js"></script>
</body></html>
```

❷

Si abrimos la página html e inspeccionamos los elementos ❷, dado que el valor de "step" es 0, se muestra en los condicionales el párrafo correspondiente, pero si analizamos los elementos en la consola del navegador, los condicionales únicamente mantienen en DOM el contenido visible en pantalla, cosa que no ocurre con "v-show", donde ambos elementos permanecen en DOM, aunque solo uno sea visible en pantalla.

Si pulsamos el botón, incrementamos el valor del "step", sucede lo mismo, que el condicional en el DOM solo aparece el visible, mientras que en v-show siempre permanecen ❸.

Es recomendable utilizar esta directiva "v-show" en caso de que un contenido se muestre y oculte con relativa frecuencia ya que, al permanecer oculto en el DOM, su coste de representarlo es mínimo y tiene mayor rendimiento.

V-CLOAK

Al refrescar la página, es posible que veamos etiquetas {{step}}, sin valor ❹, para evitar esto haremos uso de la directiva "v-cloak", definiendo unos estilos en la hoja de estilos "firstvue.css" ❺ y añadiendo esta directiva en el div de la app ❻. Si ahora refrescamos la página, en lugar de visualizar las etiquetas veremos el texto "Cargando" ❼.

❹ **Condicional**

{{step}}-Si desaparece, este contenido no estará en DOM

{{step}}-Inspecciona DOM, lo anterior no esta!

FIN

V-SHOW

{{step}}-Si desaparece, este contenido permanenterá en DOM

{{step}}-Inspecciona DOM, el anterior permanece!

Incrementa pasos

❺
```
[v-cloak] > * { display: none; }
[v-cloak]::before { content: "Cargando..."; }
```

❻
```
<div id="app" v-cloak>
```

❼ Cargando...

Bucle v-for y key

IMPORTANTE

El atributo ":key" o "v-bind:key" debe ser una clave única para identificar un elemento en la lista y actualizar solo los elementos modificados, en lugar de todos.

V-FOR

Si el orden de una lista varía al cambiar el contenido de un elemento, Vue copia el contenido a la nueva posición del elemento, permaneciendo a nivel de DOM. Esto es un error, dado que debería mover el contenido a nivel de DOM en lugar de copiarlo. Para ello se necesita asignar una clave única mediante ":key" a cada elemento para identificarlo.

Creamos en el fichero "firstvue.html" un bucle para representar las tareas, añadiendo un "checkbox" en aquellas con estado "done=false", así como una caja de texto para introducir nuevas tareas mediante un botón ❶.

En "firstvue.js" ❷, incluimos en el data una variable para hacer el "bind" del texto del input y otra para la lista de tareas. Hemos añadido una propiedad computada que ordena los elementos de la lista, primero por orden alfabético y posteriormente por estado, no obstante, en este primer ejemplo no haremos uso de esta propiedad. Finalmente añadimos el método "Done" para actualizar el estado de la tarea marcada como finalizada mediante el "checkbox".

Si abrimos la página html en el navegador, aparece la lista sin ordenar, al añadir un elemento aparece en la lista ❸. Si actualizamos el valor del elemento marcando el checkbox, cambiará su estado en la lista ❹.

❶

```
<html><head><meta charset="utf-8">
  <title>V-For y Key</title>
<body>
  <div id="app">                                                    Bucle v-for
    <div v-for="(task,index) in tasks">
      <div v-if="task.done"><h5><b>Tarea Realizada - {{task.title}}</b></h5></div>
      <div v-else >Tarea Pendiente - {{task.title}}
        <input type="checkbox" @click="Done('chk_'+index,task)"
                               :ref="'chk_'+index">Realizar</input> </div>
    </div>
    <input type="text" v-model="text" />
    <input type="button" value="Añadir tarea"              Añadir elementos con nuevo id
        @click="tasks.push({ title:text, done:false, id:new Date().getTime() })" />
  </div>

  <script src="https://unpkg.com/vue"></script>
  <script src="firstvue.js"></script>
</body></html>
```

Hasta aquí no hay problemas, ni siquiera requerimos la clave en el bucle, dado que internamente vue tiene mapeada la lista igual que en la vista, sería equivalente a haber añadido la clave ":key=index" a nivel del bucle "v-for".

Ahora bien, si queremos ver la lista ordenada, lo haremos mediante la propiedad computada "tasksOrdered" en lugar de "tasks":

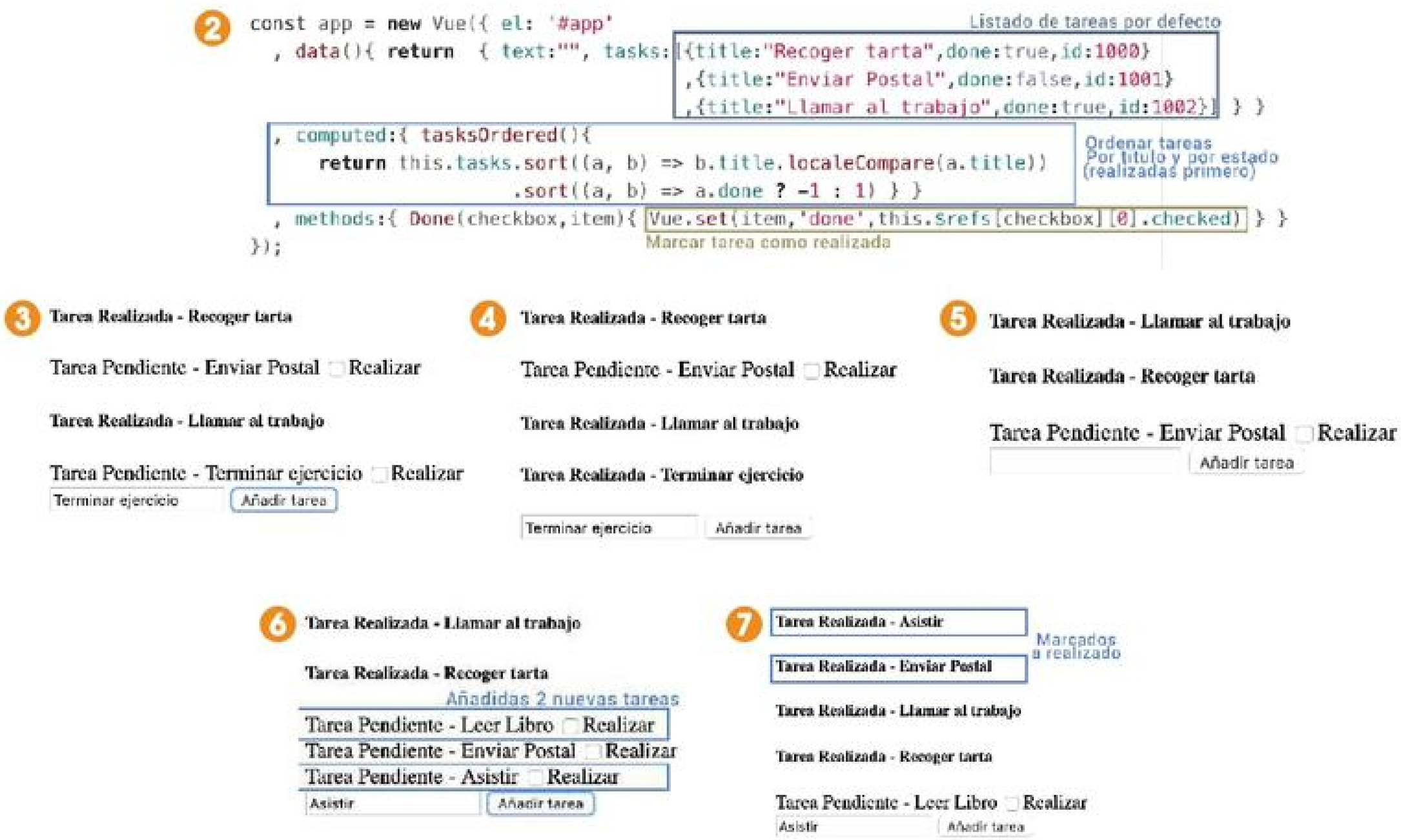
```
<div v-for="(task,index) in tasksOrdered">
```

Al abrir la página, los elementos aparecen ordenados ❺ y añadimos dos elementos nuevos ❻. Si actualizamos el estado de uno de estos a realizado se añade en la lista como marcado, pero en la posición que ocupaba sigue marcado el checkbox, aunque de un elemento erróneo. Se debe a no haber indicado un identificador único a la clave.

Si ahora añadimos la clave única de cada elemento "id":

```
<div v-for="(task,index) in tasksOrdered" :key="task.id">
```

Al abrir la página, observamos como al añadir elementos, cuando cambiemos su estado, se representarán correctamente ❼.



Bucle atributos y rango

V-FOR

Hasta ahora hemos iterado en los bucles sobre elementos de un "array". También existen otros tipos de iteración, tantas como niveles queramos descender. En este ejercicio trataremos de iterar los elementos y, dentro de estos, iteraremos para obtener nombre de atributo y valor de cada elemento.

Comencemos por declarar en "firstvue.html", el bucle "v-for", que iterará por elementos y que, a su vez por cada elemento, iteraremos por otro bucle "v-for", obteniendo clave "key" y valor "value" de cada elemento leído; de esta forma podemos listar cada una de las propiedades que contiene el elemento ❶.

Además, podemos hacer uso de los rangos, establecer un bucle indicando el número máximo por el cual iterar, en nuestro caso, la longitud del listado de tareas, visualizando dicho valor numérico.

Por otra parte, definimos en "firstvue.js" el listado de tareas a iterar con sus propiedades en el data, como el cambio del estado "done" de un elemento iterado, tras pulsar el checkbox ❷.

De modo que, si abrimos la página html, visualizamos la lista de tareas en la que, por cada elemen-

❶
```
<html><head><meta charset="utf-8">
  <title>V-For de los atributos</title>
<body>
  <div id="app">
    <h4>Tareas Atributos</h4>                    Bucle for, de elementos de las tareas
    <div v-for="(task,index) in tasks" :key="task.id">
      {{index}}          Por cada elemento, iteramos en bucle su clave y valor del objeto
      <div v-for="(value,key) in task"> <b>{{key}}</b>:{{value}},</div>
      <input type="checkbox" @click="Done('done_'+index,task)"
                             :checked="task.done"
                             :ref="'done_'+index">Realizado</input>
    </div>
    <h4>Rango de {{tasks.length}}</h4>
    <ul v-for="item in tasks.length"> <li>{{item}}</li> </ul>
  </div>                 Bucle por Rango de valores del 0 a longitud de tareas
  <script src="https://unpkg.com/vue"></script>
  <script src="firstvue.js"></script>
</body></html>
```

❷
```
const app = new Vue({ el: '#app'
  , data(){ return  { text:"", tasks:[{title:"Recoger tarta",done:true,id:1000}
                                     ,{title:"Enviar Postal",done:false,id:1001}
                                     ,{title:"Llamar al trabajo",done:true,id:1002}] } }
  , methods:{ Done(checkbox,item){ Vue.set(item,'done',this.$refs[checkbox][0].checked) } }
});
                                  Cambiar estado de tarea a realizada o no, depende del checkbox
```

to, iteramos a su vez una lista de atributos visualizando la clave y valor ❸. Además, observamos la iteración de rango, de tres elementos, longitud de tareas.

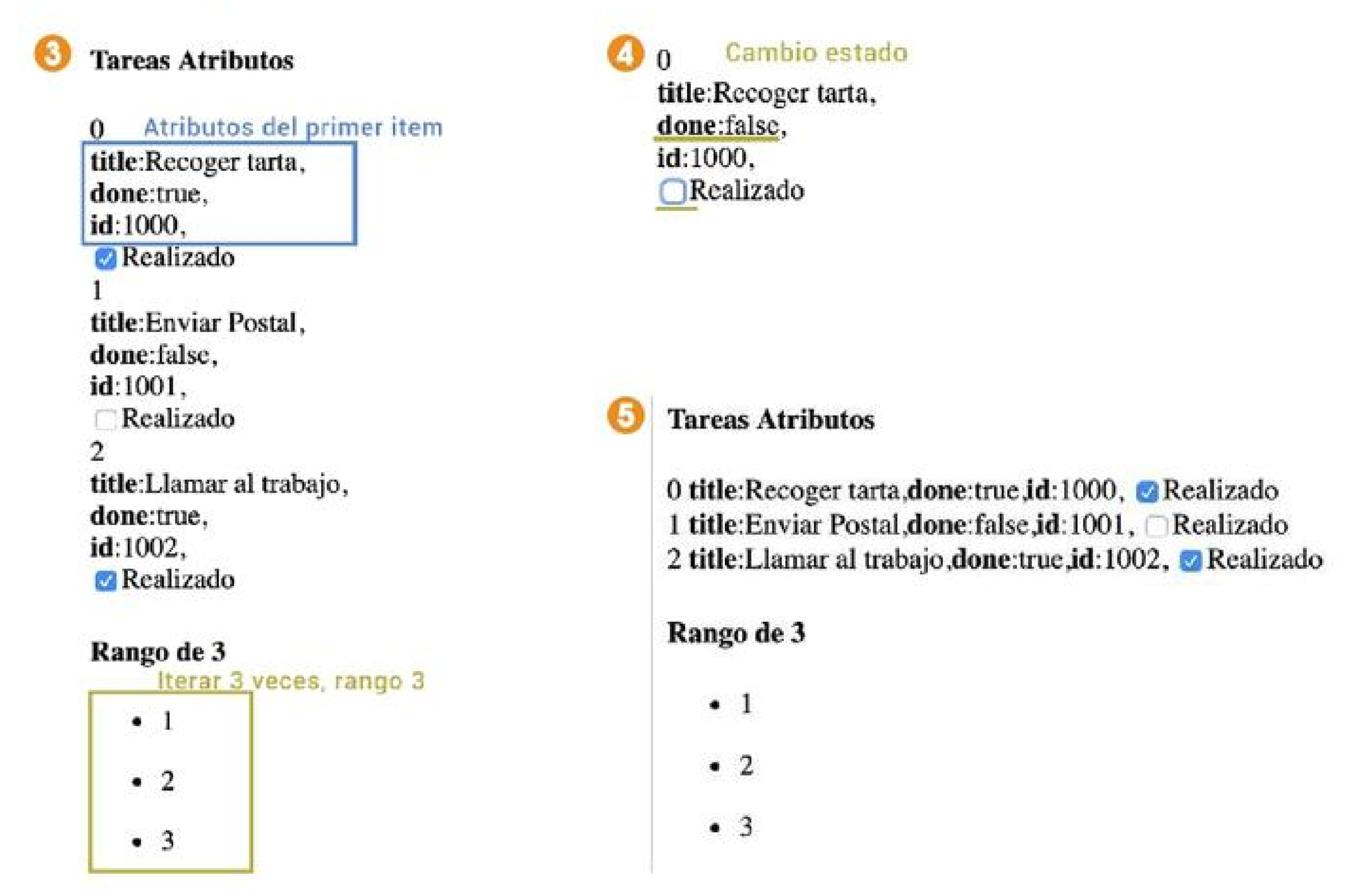
En caso de pulsar el "checkbox" de un elemento de la lista, el valor del atributo cambiará o no a realizado ❹.

Es decir, por cada iteración, el índice indica la posición que ocupa el elemento en el "array", la clave o "key", el nombre del atributo del elemento a visualizar, y el "value" el valor a representar para dicho atributo.

Ahora bien, en caso de querer iterar representando en una misma línea cada atributo, basta con hacer uso de la etiqueta "template", en lugar de los "div":

```
<template v-for="(value,key) in task"> <b>{{key}}</b>:{{value}},</template>
```

De esta forma, si modificamos la vista representamos cada atributo / valor de cada elemento en una misma fila ❺.



Peticiones Fetch

IMPORTANTE

Al recibir la respuesta, trata el tipo de respuesta mediante "response.ok", en caso satisfactorio, transforma la respuesta a objeto JSON mediante el método ".json()".

Fetch

Para obtener datos de un api web remota, una de las alternativas es utilizar el "api fetch", que nos permite tratar peticiones y respuestas HTTP. Tiempo atrás, esta misma funcionalidad la conseguíamos mediante el uso de "XMLHttpRequest"; sin embargo, este api mejora con creces el tratamiento de estas peticiones.

Creemos un ejercicio donde realicemos una petición HTTP GET y obtengamos un array de objetos, que representaremos en pantalla en forma de lista.

Para ello, creamos en el contenedor de la aplicación de "firstvue.html", una lista de elementos, haciendo uso de un bucle "v-for". Esta lista recorrerá la variable del data "series" ❶.

Por otra parte, en el fichero "firstvue.js" ❷, declaramos en el data la variable "series", inicializada a array vacío. Preparemos la solicitud HTTP, en la instancia Vue, en el "mounted", utilizamos "fetch" para realizar una solicitud GET, es compatible con todos los navegadores, por lo que podemos utilizarlo sin temor.

❶

```
<html><head><meta charset="utf-8">
  <title>Fetch</title>
<body>
  <div id="app">
    <h4>Fetch</h4>
    <ul><li v-for="serie in series">{{serie.Titulo}}</li></ul>
  </div>
  <script src="https://unpkg.com/vue"></script>
  <script src="firstvue.js"></script>
</body></html>
```

Recorremos el listado de series recibidas

❷

```
const app = new Vue({ el: '#app'
  ,data(){ return { series:[] } }
  ,mounted(){
    fetch('https://vue-100-ejercicios-cli-series.firebaseio.com/.json')
    .then(response => { if (response.ok) return response.json() })
    .then(response =>{ this.series=response; })
    .catch(error => console.log('Error:', error.statusText));
  }
});
```

Url de inicio de solicitud remota a la api

Tratamos peticion, si es OK retornamos JSON

Retornamos respuesta

Tratamos Error

El primer parámetro de "Fetch" es la url del api para obtener el recurso. Utiliza dos métodos "then", el primero trata si es satisfactoria o no la respuesta.

Esta verificación es posible leyendo la propiedad "ok" de la respuesta recibida, en caso afirmativo, usamos el método ".json()" sobre la respuesta, para transformar y retornarla, como un objeto de tipo JSON, que podamos interpretar.

Esta respuesta es recibida por el segundo método "then()",es aquí donde asignamos el resultado a la variable del data, en este caso un array. Una vez asignado, si abrimos la página html en el navegador, veremos la lista ❸.

En caso de que se hubiera producido un error, la excepción sería tratada en el método "catch" de este mismo.

Async

Podemos realizar esta misma petición mediante una función asíncrona "getAsync", que realice el "fetch" internamente y retorne el resultado, una vez obtenida la respuesta mediante "await" ❹. Esto es posible gracias a las promesas, que veremos en ejercicios posteriores.

❸ **Fetch**

- You
- Black Mirror
- Stranger Things
- The Punisher
- House of Cards
- Dark
- Élite
- Travelers
- Game of Thrones
- WestWorld
- Roma
- The Deuce
- The Left Lovers
- Deadwood
- Big Little Lies
- Silicon Valley

❹

```
const app = new Vue({ el: '#app'
  ,data(){ return { series:[] } }
  ,mounted(){ /* fetch('https://vue-100-ejercicios-cli-series.firebaseio.com/.json'
    this.getAsync('https://vue-100-ejercicios-cli-series.firebaseio.com/.json')
    .then(response =>{ this.series=response; })
    .catch(error => console.log('Error:', error.statusText));
  }
  ,methods:{
    getAsync:async function(url) {
      let response =await fetch(url)
      if(response.ok) return await response.json()
      else throw new Error(response.status) }
  }
```

Método Asíncrono

Esperamos respuesta mediante el await

Una vez recibida, si es OK, trata el resultado como JSON

Contexto This

IMPORTANTE

Intente utilizar funciones regulares en lugar de flecha, si aun así decide usar las de flecha, tenga en cuenta el comportamiento del "this" en cada situación.

This - Funciones Regulares y Arrow

Cuando utilizamos funciones en Vue, es probable que en alguna ocasión suceda que al acceder al valor de la instancia Vue mediante "this" nos encontremos con un valor "undefined" en lugar de la instancia Vue. Esto probablemente ocurra al hacer uso tanto de funciones regulares "function(){}" como funciones flecha "()=>{}".

Realicemos un ejercicio en el que probemos una combinación de casos de uso para estas funciones. Creamos el fichero "firstvue.html" ❶ donde únicamente declaramos el contenedor de la aplicación.

Lo siguiente es crear en "firstvue.js" la implementación del ejercicio ❷. Inicialmente declaramos en el "data" la variable "msg" que contendrá un texto estático, "numbers" será un array de números y "number" es un número para filtrar de la colección del array. Además, en el "mounted" nada más cargar el componente invocamos a cada uno de los métodos que definiremos ahora.

Si declaramos una función regular como la siguiente, tenemos acceso a la instancia vue mediante "this". Esto se debe a que las funciones regulares "this" hacen referencia al propietario de la función, de modo que al definirlo dentro de un componente de Vue, este es su propietario.

```
fRegular() { this.msg }
```

Si declaramos una función flecha, no tendremos acceso al "this", debido a que las funciones flechas el "this" no hacen referencia al propietario de la función flecha, sino que utilizan lo denominado alcance estático o "static scope", por lo que el contexto del "this" se refiere en este caso al bloque de código en el cual está definido.

```
,fArrow:()=> { this.msg }
```

❶
```
<html><head><meta charset="utf-8">
  <title>Contexto This</title>
<body>
  <div id="app"> <h4>Contexto this</h4> </div>
  <script src="https://unpkg.com/vue"></script>
  <script src="firstvue.js"></script>
</body></html>
```

Si utilizamos una función flecha dentro de una función regular, podemos acceder al "this", dado que la función regular establece que "this" es el contexto del componente.

```
,fRegularArrow() { let element=this.numbers.find((num)=>{ return num == this.number
}) }
```

Curiosamente, si utilizamos una función regular dentro de otra regular, "this" no se refiere al componente Vue, debido a que se refiere al propietario de la función que lo contiene.

```
,fRegularReg() { element = this.numbers.find(function(num) { return num == this.
number }) }
```

Para resolver el problema anterior podemos proporcionarle el contexto "this" a esta función regular contenida dentro de la función padre "fRegularRegBind", haciendo uso del método de extensión de función "bind()".

```
,fRegularRegBind() { element = this.numbers.find(function(num) { return num == this.
number }.bind(this)) }
```

Si abrimos la página html y abrimos la consola, veremos cada uno de los resultados de obtener el "this", unos "undefined" y otros cuyo acceso al contexto ha sido satisfactorio ❸.

❷

```
const app = new Vue({ el: '#app'
  ,data(){ return { msg:"Texto del data", numbers:[1,2,3,4,5], number:3 } }
  ,mounted(){ this.fRegular()
              this.fArrow()
              this.fRegularArrow()
              this.fRegularReg()
              this.fRegularRegBind() }
  ,methods:{          Dentro de Función regular, this equivale a la instancia
    fRegular() { console.log('Función Regular:'+this.msg) }
    ,fArrow:()=> { console.log('Función Arrow:'+this.msg) }   Dentro de Función flecha
                                                               , this es "undefined"
                Dentro de función regular, si utilizamos una con arrow, this es instancia Vue
    ,fRegularArrow() {                                          Función flecha
        let element=this.numbers.find((num)=>{ return num == this.number })
        console.log('Función regular con Arrow:'+element) }
    ,fRegularReg() {          Dentro de función regular, usar otra regular, this es undefined
      element=this.numbers.find(function(num) { return num == this.number })
      console.log('Función regular con Otra Regular:'+element) }Función regular
    ,fRegularRegBind() {
      element=this.numbers.find(function(num) { return num == this.number }.bind(this))
      console.log('Función regular con Otra Regular con Bind(this):'+element) }
  }
});
                Dentro de función regular, si usamos otra regular,
                al hacerle bind(this), accedemos a la instancia Vue
```

❸

```
Elements  Console  Sources  Network  »
top  Filter  Default levels
Función Regular:Texto del data                          firstvue.js:12
Función Arrow:undefined   Mal                           firstvue.js:14
Función regular con Arrow:3                             firstvue.js:18
Función regular con Otra Regular:undefined   Mal        firstvue.js:22
Función regular con Otra Regular con Bind(this):3       firstvue.js:26
```

Contexto this

Promesas

IMPORTANTE

Utilice promesas siempre y no olvide retornar "reject" o "resolve" según el tipo de respuesta, o quien espera tratar el resultado de la promesa no recibirá nada.

Promise

El uso de promesas para gestionar llamadas remotas al api está muy extendido. Nos permiten gestionar las respuestas tras enviar una solicitud al servidor y, cuando este responda, se tratará para realizar alguna acción.

Las promesas se componen de dos elementos principales: "resolve" y "reject". El primero se utiliza para gestionar respuestas exitosas, de modo que quien esté a la escucha del resultado de la promesa, a través del método "then", recoja el valor retornado por el "resolve" de la promesa.

El otro elemento es "reject", el caso opuesto, retorna error a quien esté a la escucha de la promesa, por el catch.

Creemos un ejercicio que llame a un api y obtenga el título de la serie del objeto JSON retornado por una promesa.

Para ello creamos el fichero "firstvue.html" ❶, donde mostraremos el valor de la variable título o un texto de carga.

Posteriormente creamos el fichero "firstvue.js" ❷, definiendo la variable "title" en el data, para recoger el valor de la llamada. Definimos el método "getSerie" con parámetro id de la serie a obtener su título. Este método retorna una promesa, que contiene una llamada remota mediante "fetch".

❶
```
<html><head><meta charset="utf-8">
  <title>Promesas</title>
<body>
  <div id="app"> <h4>Promesas</h4>
    <p v-if="title"> {{title}} </p>
    <p v-else> Cargando... </p>
  </div>
  <script src="https://unpkg.com/vue"></script>
  <script src="firstvue.js"></script>
</body></html>
```

En caso de éxito del "fetch", inyectamos el valor devuelto por la api a "resolve()". Si fuera error, retornaríamos "reject", al que inyectaríamos el error.

Será en el "mounted" donde invocamos al método "getSerie" que, como hemos visto, retorna una promesa, de modo que cuando la promesa ejecute el "resolve", se recibirá el valor por el "then" de la invocación del método, asignando el valor a la variable "title" para visualizarlo, mientras que, si ocurriera error, sería capturado por el "catch".

Si abrimos la página html, veremos el mensaje "Cargando" en pantalla, al no tener valor título, en la carga del componente invocamos la promesa. En caso de éxito, la promesa envía mediante "resolve", el valor que queremos que recoja por el "then", quien invocó la promesa, donde se obtendrá el valor y será asignado al título ❸.

❷
```
const app = new Vue({ el: '#app'
  , data() { return { title:null } }
  , mounted() { this.title='La serie 1 es:'
                this.getSerie(1)
                .then((res)=>{ this.title += res.Titulo})
                .catch((error)=>{alert(error.message)}})}
  , methods:{
      getSerie(id){
        return new Promise((resolve, reject) => {
                fetch('https://vue-100-ejercicios-cli-series.firebaseio.com/'+id+'.json')
                .then(response =>{  if(response.ok) return response.json()
                                    else reject(response.statusText)})
                .then(data => { resolve(data) })
                .catch(error=>{ console.log(error)
                                reject(error)   })
        })
      }
  }
});
```

Recibimos datos de respuesta a través del resolve

Recibimos datos de error a través del reject

Inyectamos los datos de la respuesta

Inyectamos el error en reject

❸ **Promesas**

La serie 1 es:Black Mirror

Conjunto de Promesas

IMPORTANTE

Ejecutar promesas al mismo tiempo no excluye poder tratar individualmente cada promesa la respuesta, ya que tienen su "then" individual, sobre todo su "catch" en caso de error.

Promise All

Acabamos de conocer las promesas y su potencial, profundicemos un poco mas. Las promesas controlan de forma asíncrona la resolución de peticiones, sin embargo, es posible que queramos lanzar asíncronamente un conjunto de peticiones a la vez; nos de igual el orden en que acaben, lo que importa es que todas finalicen satisfactoriamente.

Este caso lo resuelve utilizar "Promise.all()", donde le pasamos un conjunto de promesas lanzadas a la vez y, una vez finalicen todas, entonces realizar una acción.

Creemos un ejercicio que simule este caso de uso, para ello reutilizamos el método definido por el ejercicio anterior, que retornaba una promesa y ejecutaba mediante un "fetch", la llamada al api para obtener un objeto serie.

Creamos la página "firstvue.html" ❶, incluyendo un bucle para representar cada elemento obtenido de cada petición, además de un mensaje que mostraremos al final.

Tras esto, crearemos el fichero "firstvue.js", donde definiremos en el data una variable de tipo array, que contendrá un listado de títulos, uno por cada título retornado por cada petición ❷, además de la variable "message" utilizada para retornar un texto tras finalizar con éxito la carga de todas las peticiones.

En el "mounted", definimos una variable de tipo array, por lo que añadiremos al array un par de promesas, de tal forma que cada promesa obtendrá una serie diferente: la primera buscará la serie con identificador 1 y la segunda promesa buscará la serie con identificador 2.

❶
```
<html><head><meta charset="utf-8">
  <title>Promesas All</title>
<body>
  <div id="app"> <h4>Promesas All</h4>
    <p v-for="title in titles"> {{title}} </p>
    <p>{{message}}</p>
  </div>
  <script src="https://unpkg.com/vue"></script>
  <script src="firstvue.js"></script>
</body></html>
```

Por último, invocamos a "Promise.all()" pasándole el array de promesas y se encargará de invocarlas todas a la vez asíncronamente. Una vez toda petición haya concluido, se ejecutará el método "then" en caso de que todas las llamadas fueran satisfactorias, donde asignaremos un mensaje mostrado por pantalla tras cargar todos los datos.

Si abrimos la página html, vemos cómo se han cargado los títulos de las dos peticiones y se ha mostrado un texto indicando el número de elementos que contiene el array, es decir, el número de resultados ❸.

❷
```
const app = new Vue({ el: '#app'
  , data() { return { titles:[], message:null } }
  , mounted() { let promises=[];
    promises.push(this.getSerie(1)
                  .then((res)=>{ this.titles.push(res.Titulo) })
                  .catch((error)=>{alert(error.message)}))
    promises.push(this.getSerie(2)
                  .then((res)=>{ this.titles.push(res.Titulo) })
                  .catch((error)=>{alert(error.message)}))

    Promise.all(promises).then((res)=>{
      this.message="Se han cargado:"+promises.length+" elementos." })
  }
  , methods:{
      getSerie(id){
        return new Promise((resolve, reject) => {
                fetch('https://vue-100-ejercicios-cli-series.firebaseio.com/'+id+'.jsor
                .then(response =>{  if(response.ok) return response.json()
                                    else reject(response.statusText)})
                .then(data => { resolve(data) })
                .catch(error=>{ console.log(error)
                                reject(error)   })
        })
      }
  }
```

❸ **Promesas All**

Stranger Things

Black Mirror

Se han cargado:2 elementos.

Inline templates

Definiendo plantillas de componentes

Dentro de las múltiples formas de definir plantillas de componentes en Vue la siguiente que veremos es la definición *inline*.

Si normalmente definimos el marcado HTML dentro del componente en esta modalidad, si el atributo *inline-template* está presente podremos definirlo dentro de la propia etiqueta del componente. Esto nos aporta mayor flexibilidad y podremos tener un renderizado por defecto definido dentro del componente y la facilidad de sobrescribirlo cuando queramos indicando mediante el uso del atributo inline-template.

Para este ejercicio imaginemos que definimos un componente que representa uno de los artículos de nuestro blog y que querremos mostrar de diferentes maneras en función de la sección en la que esté. Podremos definir nuestro componente *my-post* como un componente con unos atributos propios y una plantilla por defecto ❶.

Posteriormente solo necesitaríamos usar la etiqueta de nuestro componente para renderizarlo de la siguiente forma:

```
<my-post ></my-post>
```

Si queremos definir una miniatura para colocarla en un lateral de nuestro blog y con otra distribución de componentes o mostrando menos variables podríamos hacerlo. Tendríamos que sobrescribir la definición HTML dentro de la etiqueta del componente y añadir el atributo *inline-template* ❷.

El componente tiene los mismos atributos pero renderizados de forma diferente con esta sencilla manera de sobrescribirlos. Cambiamos la composición de nuestro artículo según donde se encuentre dentro de nuestro blog o podemos tener varias vistas para el mismo artículo si le indicamos que lo renderice de una tercera forma.

En la imagen ❸ vemos el ejemplo completo y las diferentes composiciones.

❶
```
Vue.component('my-post', {
  data() {
    return {
      header: 'Article',
      title: 'Title',
      image: 'http://lorempixel.com/400/200/',
      post: 'Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do e
    }
  },
  template: `<div>
    <p>{{title}}</p>
    <h1> {{header}}</h1>
    <p>{{post}}</p>
    <img v-bind:src=image alt="article" width="420" height="200">
  </div>`
});
```

2

```
<div class="row">
<my-post class="column-70 post"></my-post>
<my-post inline-template class="column-30 card  post" v-color="'#eaeaea'">
  <div>
    <p class="subtitle">{{title}}</p>
    <p>{{post}}</p>
    <img v-bind:src=image alt="article" width="210" height="100">
  </div>
</my-post>
</div>
```

3

firstvue.html | firstvue.css

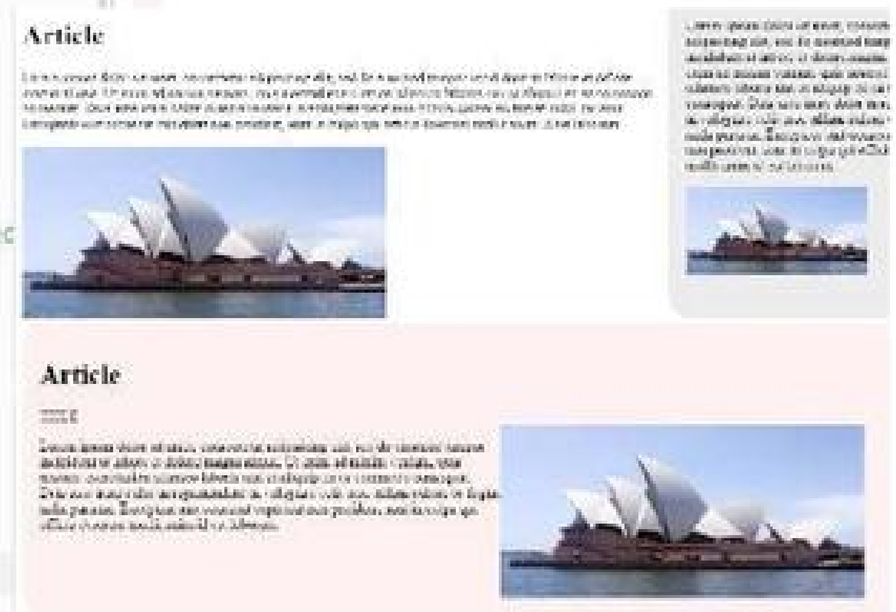
```
11  <div id="posts-demo">
12    <div class="row">
13    <my-post class="column-70 post">x/my-post>
14    <my-post inline-template class="column-30 card  post" v-color="'#eaeaea'">
15      <div>
16        <p class="subtitle">{{title}}</p>
17        <p>{{post}}</p>
18        <img v-bind:src=image alt="article" width="210" height="100">
19      </div>
20    </my-post>
21    </div>
22    <my-post inline-template v-color="'#ffefef'" class="post">
23      <div class="card">
24        <h1> {{header}}</h1>
25        <p class="subtitle">{{title}}</h3>
26        <div class="row">
27          <div class="column">
28            <p>{{post}}</p>
29          </div>
30          <div class="column">
31            <img v-bind:src=image alt="article" width="420" height="200"/>
32          </div>
```

firstvue.js

```
1  Vue.directive("color", function(el, binding, vnode) {
2      el.style.background = binding.value;
3  });
4  Vue.component('my-post', {
5    data() {
6      return {
7        header: 'Article',
8        title: 'Title',
9        image: 'http://lorempixel.com/400/200/',
10       post: 'Lorem ipsum dolor sit amet, consectetur adipisicing elit, sec
11     }
12   },
13   template: `<div>
14     <p>{{title}}</p>
15     <h1> {{header}}</h1>
16     <p>{{post}}</p>
17     <img v-bind:src=image alt="article" width="420" height="200">
18   </div>`
19 });
20 new Vue({ el: '#posts-demo' });
```

firstvue.html

```
1  .subtitle {
2    color:#666;
3    text-transform: uppercase;
4    font: 1em;
5    margin: 0 0 0 0;
6  }
7  .row {
8    display: flex;
9  }
10 .column {
11   flex: 50%;
12 }
13 .column-70 {
14   flex: 70%;
15 }
16 .column-30 {
17   flex: 30%;
18 }
19 .card {
20   background-color: #efefef;
21   border-radius: 25px;
22   padding: 20px;
23 }
24 .post {
25   margin: 2 1 2 1;
26 }
```



Slots

Creando slots

Si volvemos al ejercicio anterior podemos componer con slots algo similar al artículo de blog que lográbamos con las inline-templates.

Crearemos dos componentes para componer nuestro artículo de blog: por un lado tenemos el título y el subtítulo y por el otro el cuerpo del artículo con las imágenes ilustrativas ❶.

Siguiendo las reglas del anterior ejemplo el componente con la cabecera (título y subtítulo) contendrá el cuerpo del mensaje y las propiedades del padre serán inyectadas al hijo donde pongamos el tag <slot>. Podemos ver este ejemplo en la imagen ❷.

❶

firstvue.html

```
11  <div id="slots-demo">
12    <div>
13      <parent></parent>
14    </div>
15  </div>
```

firstvue.js

```
1  Vue.component('child', {
2    data() {
3      return {
4        childVariable: 'Variable del hijo'
5      }
6    },
7    template: `
8    <div>
9      <slot></slot>
10     <p class="childStyle">Este es el contenido del componente hijo</p>
11     <p  class="childStyle">{{childVariable}}</p>
12   </div>`
13 });
14 Vue.component('parent', {
15   data() {
16     return {
17       parentVariable: 'Variable del padre'
18     }
19   },
20   template: '<div>
21     <child>
22       <p class="parentStyle">Este contenido es del padre</p>
23       <p class="parentStyle">
24          Podemos también hacer binding de datos del padre como...
25       </p>
26       <p class="parentStyle">
27          {{parentVariable}}
28       </p>
29     </child>
30   </div>`
31 });
32 new Vue({
33   el: '#slots-demo'
34 });
```

firstvue.html Preview

Este contenido es del padre

Podemos también hacer binding de datos del padre como...

Variable del padre

Este es el contenido del componente hijo

Variable del hijo

2

secvue.html | secvue.css

```
11  <div id="posts-demo">
12    <my-header-post>
13    </my-header-post>
14  </div>
```

secvue.js

```
4  Vue.component('my-header-post', {
5    data() {
6      return {
7        header: 'Article',
8        title: 'Title'
9      }
10   },
11   template: `<div>
12     <my-content-post>
13       <div class="card">
14         <h1> {{header}}</h1>
15         <p class="subtitle">{{title}}</p>
16       </div>
17     </my-content-post>
18   </div>`
19 });
20 Vue.component('my-content-post', {
21   data() {
22     return {
23       image: 'http://lorempixel.com/400/200/',
24       post: 'Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor i
25     }
26   },
27   template: `<div>
28       <slot></slot>
29       <div class="card row">
30         <div class="column">
31           <p>{{post}}</p>
32         </div>
33         <div class="column">
34           <img v-bind:src=image alt="article" width="420" height="200"/>
35         </div>
36       </div>
37     </div>`
38 });
39 new Vue({ el: '#posts-demo' });
40
```

Article

TITLE

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Transiciones/Animaciones I

Usando CSS

Hasta ahora hemos visto cómo hacer cosas básicas con Vue: construir las instancias, componentes, renderizado, interacción con el DOM... pero ha llegado la hora de divertirnos un poco más animando nuestros componentes. Las animaciones o las transiciones se pueden usar para hacer nuestra aplicación más agradable visualmente y para intentar brindar una mejor experiencia de usuario. Conviene no abusar ya que podemos distraer la atención de lo realmente importante para nuestra aplicación.

En Vue hay diferentes formas de proveer estos efectos cuando los elementos son añadidos o actualizados en el DOM y en este capítulo veremos una forma muy sencilla de empezar a animar nuestros componentes.

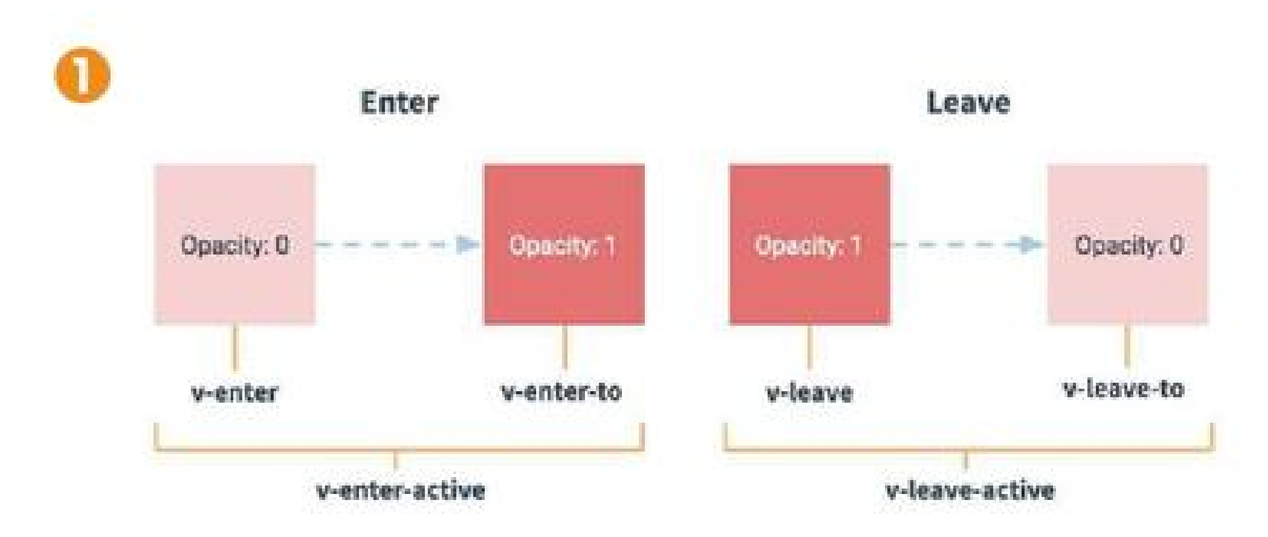
Usaremos la etiqueta <transition/> para envolver una parte de nuestro componente que se renderizará condicionalmente en función de una de sus propiedades. En nuestro caso tendremos una lista con artículos de nuestro blog, cada elemento de la lista será nuestro componente en el que guardamos información de un artículo.

Dentro de nuestro componente mostraremos un título, subtítulo, contenido y una imagen ilustrativa y lo que haremos será animar la entrada/salida de nuestra imagen en el DOM en función de la propiedad show.

Ya tenemos la estructura que hemos ido viendo en ejercicios anteriores, ¿qué hacemos para animarla? En primer lugar, lo primero es envolver nuestra imagen dentro del tag <transition> al que le pondremos un nombre "fade". Le ponemos este nombre porque lo que querremos será que vaya gradualmente apareciendo o desapareciendo en función de la evaluación de la condición de renderizado (fade-in, fade-out).

```
<transition name="fade">
  <img v-if="show" v-bind:src=image alt="article" class="image"/>
</transition>
```

Es importante que dentro de la etiqueta <transition> solo se vaya a renderizar un elemento a la vez y sea condicionado, ya que si no es así no funcionará.



Para cambiar la condición podremos hacerlo pulsando un botón, eligiendo un valor de un desplegable o como se nos ocurra. En este ejercicio hemos optado por usar el evento mouseover de modo que cambiará cada vez que se pase el ratón por encima del título del artículo.

El siguiente paso será usar el nombre que le hemos dado a la etiqueta transition para crear unas clases CSS con las que Vue puede controlar en que punto estamos del renderizado ❶.

```
.fade-enter-active, .fade-leave-active {
   transition: opacity 1s;
}
.fade-enter, .fade-leave-to  {
   opacity: 0;
}
```

Cuando se inicia la transición añadiendo el elemento al DOM y cuando se finaliza con su eliminación queremos que la opacidad sea 0 (no sea visible).

Durante el proceso de añadido ❷ y eliminación ❸ queremos que pase gradualmente a visible para lo que usamos la propiedad CSS transition y un tiempo para que sea gradual.

Simplemente con esto ❹ ya tenemos nuestra primera animación en Vue ❺.

❷

❸

❺

firstvue.js | firstvue.css

```
1  .fade-enter-active, .fade-leave-active {
2    transition: opacity 1s;
3  }
4
5  .fade-enter, .fade-leave-to {
6    opacity: 0;
7  }
```

❹

firstvue.js | firstvue.css

```
1  Vue.directive("color", function(el, binding, vnode) {
2      el.style.background = binding.value;
3  });
4  Vue.component('my-post', {
5    data() {
6      return {
7        show: false,
8        header: 'Article',
9        title: 'Title',
10       image: 'http://lorempixel.com/400/200/',
11       post: 'Lorem ipsum dolor sit amet, consectetur adipisicing elit,
12     }
13   },
14   template: `<div class="card">
15     <h1 @mouseover="show = !show"> {{header}}</h1>
16     <p class="subtitle">{{title}}</h3>
17     <div class="row">
18       <div class="column-70">
19         <p>{{post}}</p>
20       </div>
21       <div class="column-30">
22         <transition name="fade">
23           <img v-if="show" v-bind:src=image alt="article" class="image"
24         </transition>
25       </div>
26     </div>
27   </div>`
28 });
29 new Vue({ el: '#posts-demo' });
```

Transiciones/Animaciones II

Usando Javascript

Hemos realizado nuestra primera animación usando CSS de forma fácil y ahora haremos algo similar solo usando Javascript. Podemos mezclar las dos estrategias usando CSS junto a los métodos hooks de Javascript para aprovechar toda la potencia que nos dan ambos.

Como veremos este código nos quedará quizá un poco más complejo que usando simples clases CSS pero veremos paso a paso cómo hacerlo.

Por simplicidad, la base será similar a la del ejercicio de animaciones con CSS: nuestro componente representará un artículo de nuestro blog con título, subtítulo, texto e imagen. Volvemos a utilizar en este caso también la etiqueta transition con las mismas premisas que en el anterior, solo que esta vez en vez de crear clases CSS lo que haremos será implementar los hooks dentro del aparatado "methods" del componente.

```
<transition v-on:before-enter="beforeEnter" v-on:enter="enter" v-on:leave="leave"
v-bind:css="false">

   <img v-if="show" v-bind:src=image alt="article" class="image"/>

</transition>
```

Marcaremos como falso la propiedad css para asegurar que no se utilicen las variables CSS también en este caso si estas existieran e implementaremos los métodos como podemos ver ❶.

❶

```
16  methods: {
17    beforeEnter: function (el) {
18      this.elementSlide = -200;
19      this.opacity = 0;
20      el.style.opacity = this.opacity;
21      el.style.top = this.elementSlide + 'px';
22      console.log('beforeEnter');
23      console.log(el.style.opacity);
24    },
25    enter: function (el, done) {
26      console.log('enter');
27      let counter = 1;
28      const interval = setInterval(() => {
29        el.style.top = (this.elementSlide + counter * 10) + 'px';
30        el.style.opacity = this.opacity + counter * 0.05;
31        console.log(el.style.opacity);
32        counter++;
33        if(counter > 20) {
34          clearInterval(interval);
35          done();
36        }
37      }, 20);
38    },
```

```
39    leave: function (el, done) {      console.log('leave');
40      console.log(el.style.opacity);
41      el.style.opacity = this.opacity;
42      let counter = 1;
43      const interval = setInterval(() => {
44        el.style.top = (this.elementSlide - counter * 10) + 'px';
45        el.style.opacity = 1 - (this.opacity + (counter * 0.05));
46        console.log(el.style.opacity);
47        counter++;
48        if(counter > 20) {
49          clearInterval(interval);
50          done();
51        }
52      }, 20);
53    }
54  },
```

Estos métodos son bastante intuitivos sobre en qué momento se llevan a cabo y solo tendremos que aplicarles una lógica adecuada, que en este caso será la de que la imagen aparezca o desaparezca lentamente y con una transición desde/hacia arriba.

Usaremos dos propiedades del componente que llamaramos elementSlide y elementOpacity que inicializaremos a 0 pero que modificaremos en el beforeEnter con la situación inicial de partida. Posteriormente, en el método enter crearemos una función para que se ejecute cada cierto tiempo y modificaremos dentro las propiedades para que las transiciones sean más suaves.

```
const interval = setInterval(() => {
 el.style.top = (this.elementSlide + counter * 10) + 'px';
 el.style.opacity = this.opacity + counter * 0.05;
   counter++;
      if(counter > 20) {
           clearInterval(interval);
           done();
      }
   }, 20);
```

De esta forma vamos incrementando las propiedades poco a poco para que la transición sea más natural al ojo humano. Vemos cómo aparece y entra en escena desde arriba ❷ y cómo desaparece de nuevo cuando pasamos el ratón por el título del artículo ❸. Esto podría combinarse como ya dijimos con alguna clase de transición CSS y así podríamos evitar este código que puede ser menos legible.

Hemos visto cómo animar elementos de nuestro componente ❹ solo con código JavaScript. Ahora le toca a usted decidir cómo lo quiere hacer o si prefiere utilizar ambas opciones, puede jugar y experimentar para ir creando nuevas formas de interactuar con su aplicación.

Instalación NodeJs

IMPORTANTE

Si, tras instalar, el comando node no se reconoce, probablemente tengamos que añadir en las variables de entorno $PATH la ruta donde hemos instalado node. En MAC ejecutaríamos "sudo nano /etc/paths", añadiendo al PATH "/usr/local/bin".

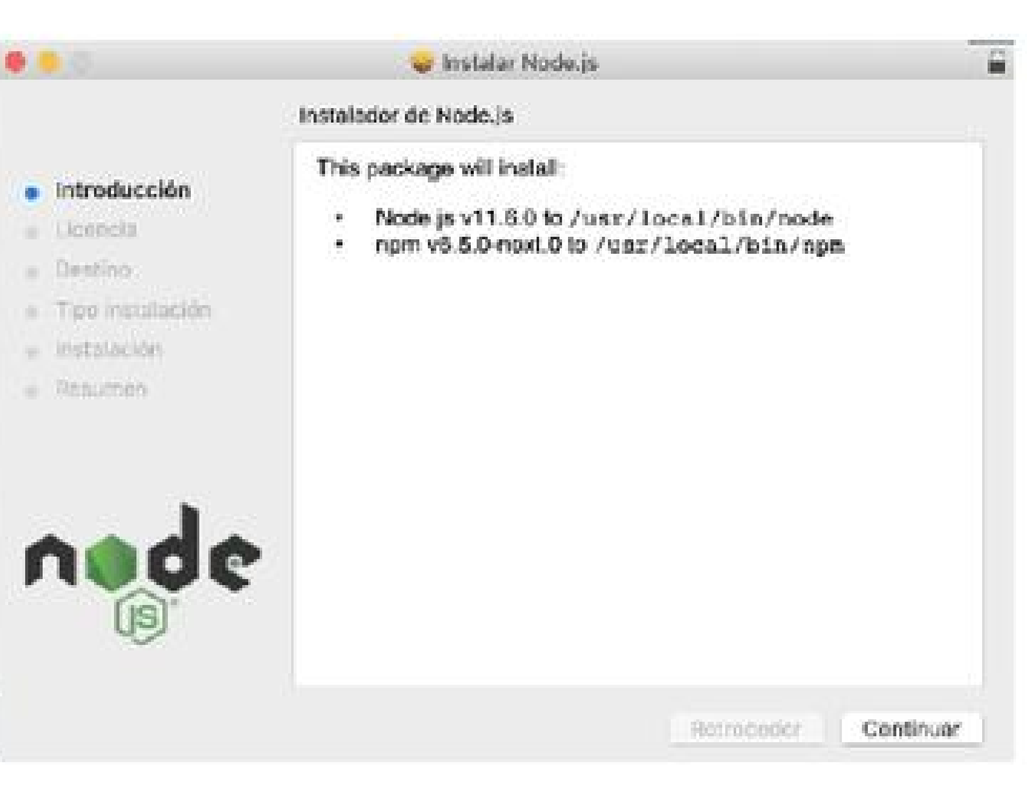
Conociendo el entorno de trabajo

El código escrito de Vue lo tenemos distribuido en diferentes ficheros ".vue". Vue se ejecuta en el navegador cliente, sin embargo, para una aplicación que sirva diferentes páginas Vue se requiere un servidor de desarrollo capaz de ejecutar nuestro código.

Esto se debe a que nuestra aplicación necesita el protocolo http para ejecutarse. Aunque tengamos nuestro entorno de desarrollo alojado localmente, aun así, requerirá un servidor local de desarrollo donde ejecutar nuestra aplicación. No basta el protocolo file://, necesitamos hacer servir diferentes páginas a través del protocolo http.

Los beneficios de ejecutar nuestro código en un servidor local de desarrollo son:

- Compilación en tiempo real por parte del servidor de nuestro proyecto, cada vez que realicemos un cambio en nuestro código.
- Al compilar nuestro código en tiempo real, el servidor nos avisa rápidamente de cualquier error semántico o del framework que hayamos implementado.
- Nos permite utilizar ciertos Plugins como ESLint, analizando si el código escrito por el equipo de desarrollo se asemeja, para mantener una cierta calidad.



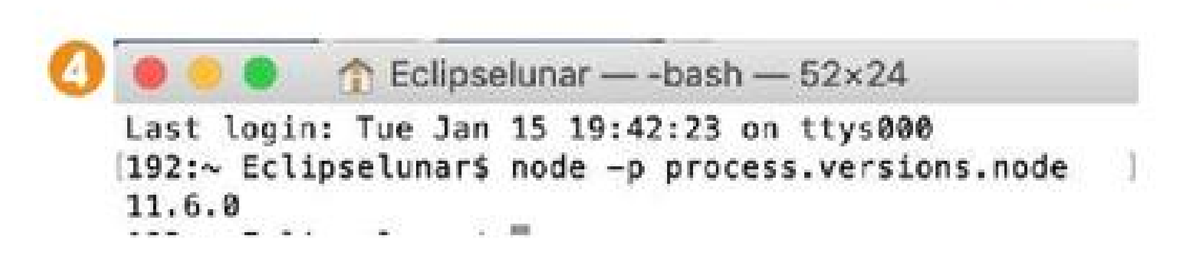
- Podemos trabajar en ES6 (ECMAScript 6), utilizando todo el potencial de las nuevas funciones de JavaScript aunque no todos los navegadores lo soporten hoy en día, gracias a babel, un compilador de código JavaScript capaz de traducirlo a código entendible por cualquier navegador.
- Optimiza nuestro código y los recursos, como imágenes, reduciéndolos para que ocupen menor tamaño y se carguen más rápidamente.
- Carga inteligente de aquellos ficheros que sean necesarios al iniciar la aplicación.

Trabajando con este entorno de desarrollo podremos desarrollar más rápidamente que como habíamos trabajado hasta ahora, simulando su funcionamiento como si la aplicación fuera ejecutada en el servidor de producción.

NodeJS

Uno de los requisitos para empezar a trabajar con el cliente de Vue (@vue/cli) es tener instalado un gestor de paquetes, es decir el Node Package Manager o (comando npm). Para ello necesitaremos tener instalado en nuestro equipo NodeJs; en caso de no tenerlo instalado, podemos descargarlo desde ❶ su página oficial: https://nodejs.org/es/. NodeJS será imprescindible para que nuestro servidor de desarrollo se ejecute y compile nuestra aplicación.

Una vez descargado, lo instalamos ❷ siguiendo todos los pasos con la configuración por defecto, hasta completar los pasos ❸. Una vez instalado, verificaremos su correcta instalación, ejecutando en la consola de comandos el comando ❹, el cual mostrará el número de versión que acabamos de instalar de Node.



Nuevo proyecto con Vue CLI 3

Empezar a trabajar con Vue tiene una curva de aprendizaje corta, hasta ahora habíamos trabajado con el CDN, incluyendo librerías en un fichero. Sin embargo, a medida que se va complicando el tipo de aplicación, necesitamos una arquitectura donde estructurar cada elemento.

Mediante el cliente CLI, desarrollado por el equipo de Vue, podremos localizar plantillas de proyectos Vue, configuradas con una arquitectura base a partir de la cual empezar a trabajar.

Para trabajar con el CLI 3, debemos instalarlo ejecutando el siguiente comando:

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

Abramos la consola, y ejecutemos el comando anterior para instalar vue CLI 3 ❶. Tras finalizar, verificamos su correcta instalación, ejecutando el siguiente comando ❷, donde visualizaríamos la versión 3 del cliente de vue.

Estamos listos para crear nuestro primer proyecto en vue, utilizando la plantilla proporcionada por el CLI 3 de vue. Crearemos un directorio para el proyecto, llamado "VueCLI3", nos posicionamos a través de la consola de comandos en este directorio.

Ejecutamos ❸ "vue create" + "nombre del proyecto", en nuestro caso "series". Tras la ejecución del comando, el CLI de vue solicita elegir el tipo de plantilla para crear el proyecto:

- Default (babel,eslint): plantilla por defecto, contiene los plugins de babel y eslint.
- Manual: permite seleccionar las características que nuestra aplicación deba tener.

Seleccionamos las características de nuestra aplicación ❹ (Babel, Router, Vuex y Linter):

IMPORTANTE

La versión anterior creaba el proyecto desde plantilla, con el comando "vue init" + "Plantilla (simple, web-pack-simple, webpack, browserify-simple, browserify)" + "Nombre proyecto".

IMPORTANTE

En la versión anterior a CLI 3, podíamos instalarlo globalmente en nuestro ordenador, mediante el comando "npm install -g vue-cli".

❶

```
Eclipselunar — -bash — 161×
Last login: Tue Jan 15 20:11:56 on ttys000
192:~ Eclipselunar$ npm install -g @vue/cli
/Users/Eclipselunar/DeveloperApplications/npm-global/bin/vue -> /Users/Eclipselunar/Developer/

> fsevents@1.2.4 install /Users/Eclipselunar/DeveloperApplications/npm-global/lib/node_module:
> node install

node-pre-gyp WARN Tried to download(404): https://fsevents-binaries.s3-us-west-2.amazonaws.co
node-pre-gyp WARN Pre-built binaries not found for fsevents@1.2.4 and node@11.6.0 (node-v67 A
  SOLINK_MODULE(target) Release/.node
  CXX(target) Release/obj.target/fse/fsevents.o
../fsevents.cc:63:6: warning: field 'async_resource' will be initialized after
      field 'lockStarted' [-Wreorder]
   : async_resource("fsevents:FSEvents"), lockStarted(false) {
     ^
1 warning generated.
  SOLINK_MODULE(target) Release/fse.node
  COPY /Users/Eclipselunar/DeveloperApplications/npm-global/lib/node_modules/@vue/cli/node_mo
  TOUCH Release/obj.target/action_after_build.stamp

> protobufjs@6.8.8 postinstall /Users/Eclipselunar/DeveloperApplications/npm-global/lib/node_
> node scripts/postinstall


> nodemon@1.18.9 postinstall /Users/Eclipselunar/DeveloperApplications/npm-global/lib/node_mo
> node bin/postinstall || exit 0

Love nodemon? You can now support the project via the open collective:
 > https://opencollective.com/nodemon/donate

+ @vue/cli@3.3.0
added 739 packages from 515 contributors in 72.87s
```

- Babel: nos permite utilizar todas las características de ES6, traduciendo el código JavaScript ES6, a JavaScript comprensible por todos los navegadores.
- TypeScript: lenguaje de programación.
- Progressive Web App (PWA): permite simular una aplicación móvil nativa desarrollada como web.
- Router: se utiliza en aplicaciones SPA (Single Page Application), para navegar entre vistas de la aplicación, sin cambiar de página.
- Vuex: gestiona los estados de los componentes en aplicaciones Vue.
- CSS Pre-processors: interpreta hojas de estilo CSS.
- Linter/Formatter: define reglas asegurando la calidad del código escrito por los miembros de un equipo, para que todos utilicen una misma sintaxis de desarrollo.
- Unit Testing: creación de pruebas unitarias.
- E2E Testing: creación de pruebas funcionales.

Seguidamente, al haber añadido el router, solicita si deseamos utilizar el modo historia para las URL ❺, seleccionamos afirmativamente. Entraremos más en detalle en otro capítulo.

Como además hemos seleccionado ESLint, debemos indicar la configuración por defecto, en nuestro caso, la básica ❻.

Indicamos que cada configuración de estos plugins, la separe en un fichero dedicado de configuración ❼.

Por último, nos pregunta si deseamos guardar la configuración que hemos introducido, para crear futuros proyectos. Indicamos que Si y le damos un nombre "BabelRouterVuexESLint" ❽.

Tras pulsar Intro, el proyecto se creará con nuestra configuración ❾. Podemos posicionarnos en el proyecto y arrancar en modo desarrollo, aunque esto lo veremos en capítulos posteriores.

❷
```
Eclipselunar — -bash — 49×24
Last login: Tue Jan 15 21:38:25 on ttys000
mbp-de-ramon:~ Eclipselunar$ vue --version
3.3.0
```

❸
```
VueCLI3 — node ~/DeveloperApplications/npm-
mbp-de-ramon:VueCLI3 Eclipselunar$ vue create series

Vue CLI v3.3.0
? Please pick a preset:
  default (babel, eslint)
❯ Manually select features
```

❹
```
? Check the features needed for your project:
>◉ Babel
 ○ TypeScript
 ○ Progressive Web App (PWA) Support
 ◉ Router
 ◉ Vuex
 ○ CSS Pre-processors
 ◉ Linter / Formatter
 ○ Unit Testing
 ○ E2E Testing
```

❺
```
? Use history mode for router? (Requires proper server setup for index fallback in pr
oduction) (Y/n) Y
```

❻
```
? Pick a linter / formatter config:
❯ ESLint with error prevention only
  ESLint + Airbnb config
  ESLint + Standard config
  ESLint + Prettier
```

❼
```
? Where do you prefer placing config for Babel, PostCSS, ESLint, etc.? (Use arrow key
s)
❯ In dedicated config files
  In package.json
```

❽
```
? Save this as a preset for future projects? Ye:
? Save preset as: BabelRouterVuexESLint
```

❾
```
⚠  Running completion hooks...

📄  Generating README.md...

🎉  Successfully created project series.
👉  Get started with the following commands:

 $ cd series
 $ npm run serve

mbp-de-ramon:VueCLI3 Eclipselunar$ cd series
```

Estructura del proyecto

Estructura del directorio creado

Tras haber creado el proyecto Vue se ha generado una estructura base ❶ que el desarrollador puede utilizar. El proyecto está estructurado por:

- Ficheros de configuración: se encuentran en el raíz de la aplicación. Uno de los más importantes es "package.json", que contiene los paquetes instalados y requeridos por nuestro proyecto, además de la definición de scripts, que podemos lanzar en la consola, para compilar en diferentes modos nuestro proyecto.
- Scripts: permite definir scripts a invocar en la consola, para compilar en diferentes modos.
- Dependencies: contiene los paquetes que hemos instalado.
- DevDependencies: contiene los plugins utilizados. "babel", traductor de Javascript comprensible por los navegadores. Uno de los mas importantes, el plugin "service", utiliza de forma interna web-pack, proporcionando los comandos de construcción de proyecto y ejecución en Node.
- public: contiene recursos no optimizados por web-pack, movidos a producción tras compilarse en dicho modo. Aquí encontramos la página inicial de nuestra aplicación SPA (Single Page Application).
- scr: contiene nuestro código estructurado en dos subcarpetas, una para los componentes de la aplicación (components), no navegables desde enrutamiento y otra contendrá las vistas (views), de extensión ".vue" o ".html", navegables desde el enrutador.
- El fichero de entrada de la aplicación es "main.js", aquí crearemos la instancia Vue de la aplicación.
- En "App.vue" encontraremos la vista de nuestro componente raíz.
- Es en "router.js" donde definimos las rutas de las vistas de la aplicación SPA.
- Los estados de los componentes serán gestionados por Vuex en el fichero "store.js".
- Carpeta assets: aquí encontramos las imágenes, iconos y hojas de estilos, que web-pack procesará cada compilación añadiéndole un hash para evitar problemas de la caché del navegador.

Un fichero muy importante, en caso de querer modificar entre otras configuraciones del proyecto la de web-pack, es el fichero "vue.config.js". El CLI de Vue no nos creó este fichero, debido a que la configuración por defecto que proporciona el plugin "cli-service", en la mayoría de los casos es muy completa.

Vamos a crear este fichero en la raíz de la estructura de ficheros ❷, Vue lo reconocerá. Aquí declararemos aquellas opciones ❸ que deseemos sobrescribir.

```
module.exports = {
    configureWebpack: config => { } // Modificar configuración web-pack
    , chainWebpack: config => { } //Permite añadir "loaders", intérpretes de extensiones
como .vue, .css…
}
```

1

Estructura del Proyecto

Módulos Instalados

Contenido Estático

Código Proyecto

Ficheros de Configuración

```
{
  "name": "series",
  "version": "0.1.0",
  "private": true,
  "scripts": {
    "serve": "vue-cli-service serve",
    "build": "vue-cli-service build",
    "lint": "vue-cli-service lint"
  },
  "dependencies": {
    "vue": "^2.5.21",
    "vue-router": "^3.0.2",
    "vuex": "^3.0.1"
  },
  "devDependencies": {
    "@vue/cli-plugin-babel": "^3.3.0",
    "@vue/cli-plugin-eslint": "^3.3.0",
    "@vue/cli-service": "^3.3.0",
    "babel-eslint": "^10.0.1",
    "eslint": "^5.8.0",
    "eslint-plugin-vue": "^5.0.0",
    "vue-template-compiler": "^2.5.21"
  }
}
```

Scripts de Ejecución

Paquetes Instalados

Plugins Instalados

2

Enter the new path for the file.

vue.config.js

3

Project

- series
- .git
- node_modules
- public
- src
- .browserslistrc
- .DS_Store
- .eslintrc.js
- .gitignore
- babel.config.js
- package-lock.json
- package.json
- postcss.config.js
- README.md
- vue.config.js

```
//Configuración del proyecto.
module.exports = {
  // Modificar configuración web-pack
    configureWebpack: config => { }
  //Añadir nuevos Loaders
    ,chainWebpack: config => { }
}
```

Compilar en desarrollo y producción

IMPORTANTE

En modo producción, los recursos como imágenes y hojas de estilo que se encuentren en la carpeta "src/assets", son optimizados y se les añade un hash para evitar problemas de caché.

Modos

En un proyecto profesional, se trabaja con diferentes entornos. Empezamos en un entorno de desarrollo, donde el desarrollador se construye el software, para posteriormente pasarlo a un entorno de pruebas donde deben superarse satisfactoriamente una serie de pruebas unitarias, para poder publicar la solución a un entorno de producción.

Era de esperar que en los proyectos del CLI de Vue existieran estos mismos entornos de ejecución, dependiendo del entorno en el que quisiéramos trabajar, es decir desarrollo, producción y test.

Existe una variable de entorno "NODE_ENV" cuyo valor se asigna según el modo de ejecución en que se encuentre. Esto implica que, en caso de trabajar en modo de desarrollo, esta variable contendría el valor "development", para producción sería "production" y para testing sería "test".

Comandos de Ejecución

Abramos el fichero "packages.config", en la sección "scripts" se definen los comandos que ejecutaremos en la consola a través de Node. Estos comandos ❶ son (serve, build, lint), encapsulan dentro instrucciones que ejecuta el plugin "vue-cli-service", responsable de las funciones de web-pack.

❶
```
"scripts": {
  "serve": "vue-cli-service serve",
  "build": "vue-cli-service build",
  "lint": "vue-cli-service lint"
},
```

❷
```
series — node • npm TERM_PROGRAM=Apple_Te
DONE  Compiled successfully in 5294ms

App running at:
- Local:   http://localhost:8080/
- Network: http://192.168.1.134:8080/

Note that the development build is not optimized.
To create a production build, run npm run build.
```

❸

Para ejecutar el proyecto en modo desarrollo, basta con ejecutar el comando:

```
npm run serve
```

Tras ejecutarlo, arranca el servidor de desarrollo local ❷, en el host "localhost" y el puerto 8080. Copiamos esta URL en el navegador y se abre nuestra aplicación ❸, mostrando el contenido de la vista "Home.vue", al ser esta la ruta por defecto.

Cambiemos el puerto asociado, en el fichero de configuración "vue.config.js", incluyendo la opción "devServer" con un JSON al que modificaremos el puerto a 9000. También podríamos cambiar el host y si necesitamos https.

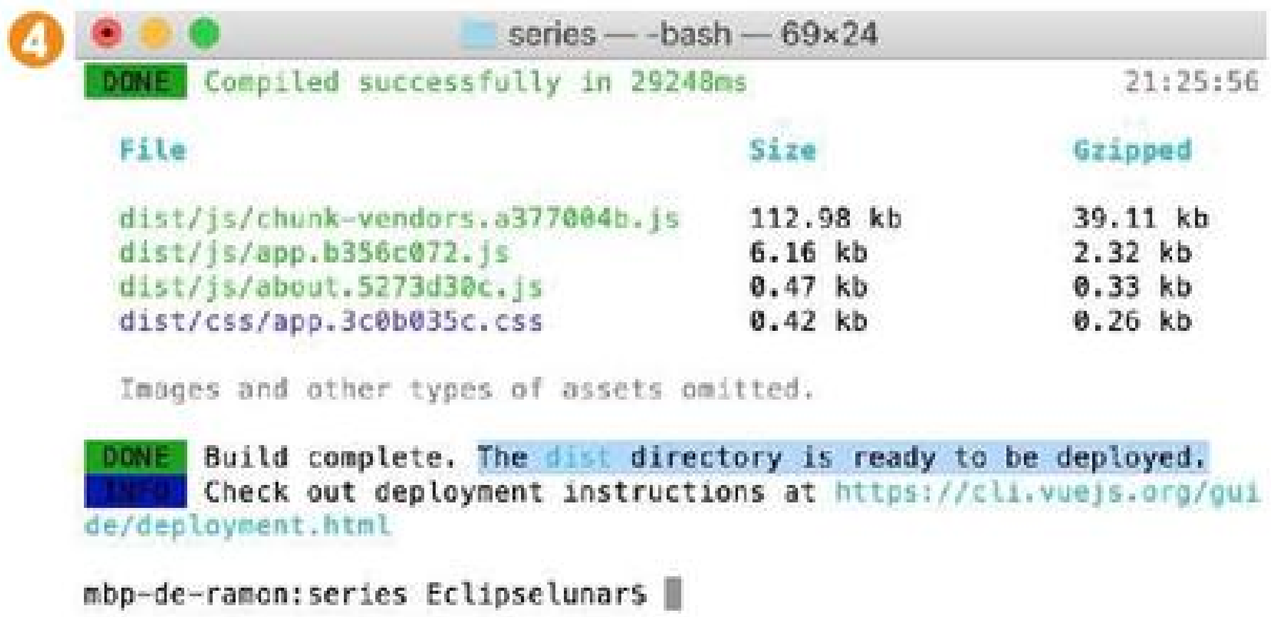
```
devServer: { host: 'localhost', port: 9000, https: false }
```

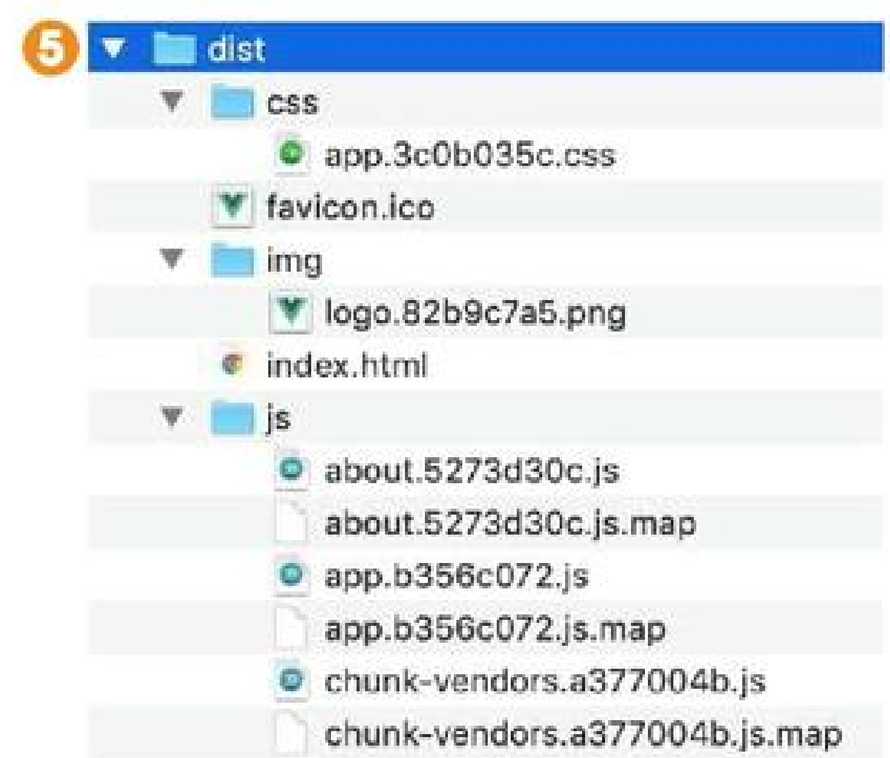
Una vez comprobado que la aplicación funciona, podemos realizar la primera subida de nuestro código, al entorno de producción, para ello ejecutaremos el comando:

```
npm run build
```

Tras ejecutarlo, nos informa ❹ que ha optimizado los recursos de la carpeta "src/assets", así como nuestro código, reduciendo el tamaño y haciendo un "minify", es decir compresión del código sin retornos de carro ni apenas espacios, depositando el resultado de esta compilación en un directorio nuevo, creado en la raíz del proyecto, con el nombre "dist".

Abrimos la nueva carpeta "dist" del directorio del proyecto ❺, generado por el script. El contenido de esta carpeta será el que copiemos en nuestro servidor de aplicaciones del entorno de producción, en caso de disponer de alguno.





Variables de entorno y ejecución

IMPORTANTE

Cada vez que cambiemos el valor de una variable de entorno, deberemos recompilar el proyecto para obtener el valor actualizado.

Variables de entorno

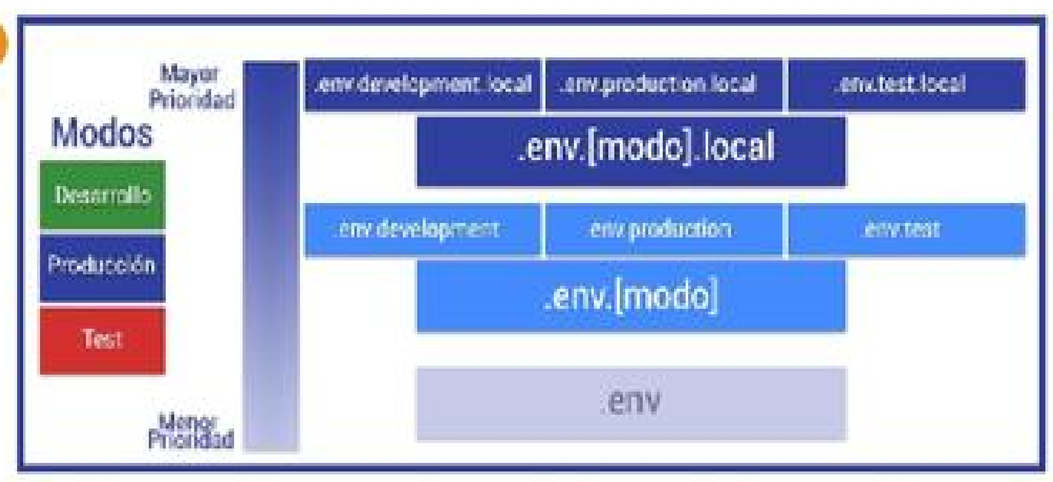
Las variables de entorno son constantes que podemos definir globalmente, en una serie de ficheros, de tal forma que según el entorno de ejecución en el que nos encontremos, utilicen un determinado valor u otro.

Podemos crear estas constantes en ficheros ❶, ubicados en el directorio raíz de la aplicación. Siguen esta nomenclatura:

- ".env": en caso de que únicamente exista este fichero, se toma como fichero global de variables de entorno.
- ".env.[modo]": si existe este fichero, que puede nombrarse según su modo de ejecución, como ".env.development", ".env.production" o ".env.test", al ejecutar un modo en concreto y si existe el fichero del modo, tendrá mayor prioridad que el genérico ".env". En caso contrario, se tomará el fichero genérico ".env".
- ".env.[modo].local": Idéntico que el ".env.[modo]", no obstante si lo hemos definido y trabajamos con GIT, los compañeros del equipo no recibirán la configuración de este fichero, es decir, es ignorado por GIT. Tendrá una prioridad superior a la de ".env.[modo]", permitiéndonos realizar pruebas locales, sin afectar al equipo.

Nuestra aplicación puede alojarse en la raíz de un dominio (http://dominio) o en un subdirectorio dentro del dominio

❶



❷

```
<template>
  <div class="home">
    <h3>Modo:[{{modo}}] - {{titulo}} - Base: {{base}}</h3>
    <img alt="Vue logo" src="../assets/logo.png">
    <HelloWorld msg="Welcome to Your Vue.js App"/>
  </div>
</template>
<script>
import HelloWorld from '@/components/HelloWorld.vue'

export default {
  name: 'home',
  data:()=>{ return { titulo:process.env.VUE_APP_TITULO
                      ,modo:process.env.NODE_ENV
                      ,base:process.env.BASE_URL
                    }
  }
  ,components: { HelloWorld }
}
</script>
```

(http://dominio/web), lo especificamos a través de la opción de configuración "publicPath" en el fichero "vue.config.js". Este valor se asignará a la variable de entorno "BASE_URL", cuyo valor por defecto es la raíz "/".

Las variables de entorno tienen una nomenclatura especial, utilizan el prefijo "VUE_APP_" seguido del nombre que queramos para la constante, es decir, de este tipo "VUE_APP_NOMBRECONSTANTE".

Creemos un fichero con el nombre ".env" en la raíz del proyecto, definiendo una variable de entorno:

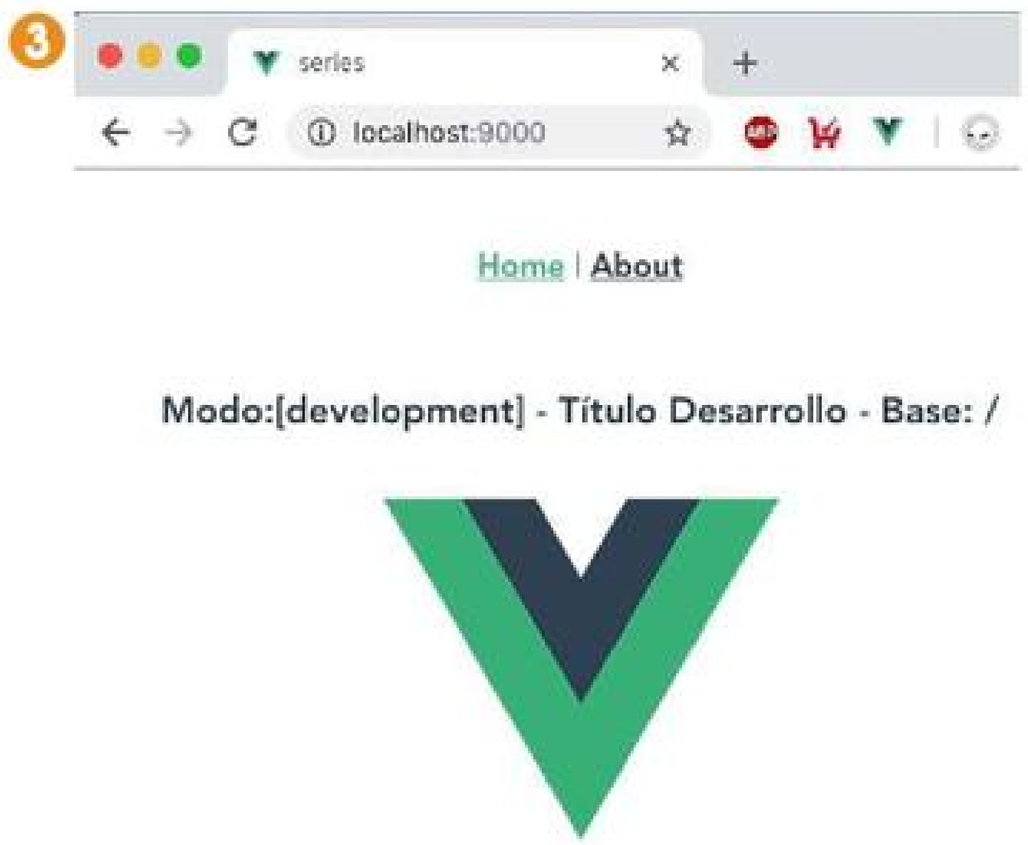
VUE_APP_TITULO=Título Por defecto

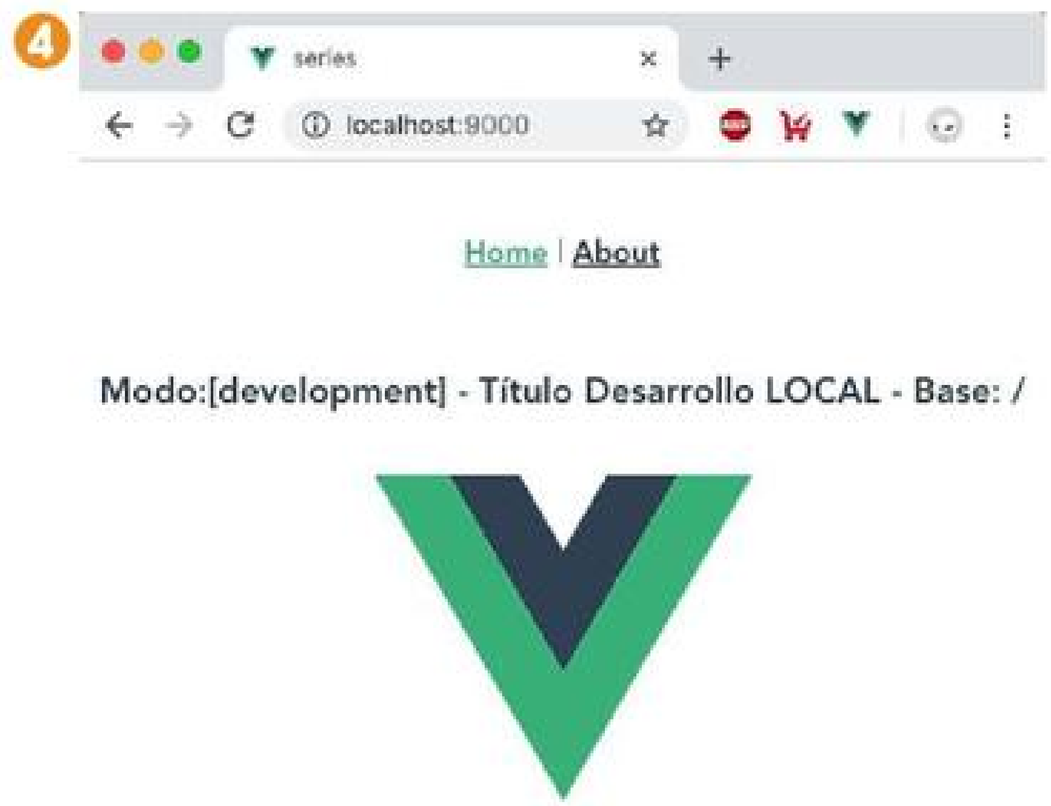
Al no tener otro fichero más prioritario, se toman los valores de este fichero para cualquier modo de ejecución.

Definamos en el data del componente "Home.vue", la propiedad "titulo", para asignarle la anterior variable de entorno, otra "modo" para el entorno de ejecución y "base" para la raíz de la aplicación y preparemos la plantilla, para mostrar estos valores de "data" ❷.

Creemos el fichero ".env.development", en la raíz, definiendo la misma variable de entorno que en ".env", con el valor "Título Desarrollo". Tras ejecutar en modo desarrollo, se mostrará el modo, el título por defecto leído por el fichero ".env.development" y la ruta base ❸, al ser mas prioritario que ".env".

Si además creamos el fichero ".env.development.local", definiendo la misma variable, pero con el valor "Título Desarrollo LOCAL". Tras ejecutar en desarrollo, al ser más prioritario, se toma el valor de la variable del fichero ".env.development.local"❹.





CLI Service scripts y servidor http

Scripts

Exprimamos un poco más el potencial del CLI Service de Vue. Por defecto este servicio nos proporciona unos scripts para ejecutarlos en Node, permitiéndonos también crear los nuestros en la sección "scripts" de "packages.config", especificando opciones para desarrollo ❶ y empaquetado en producción ❷.

Creemos un nuevo comando con el nombre "desarrollo-seguro", que abra el navegador tras compilar el proyecto, por el puerto 44300 para nuestro servidor de desarrollo y que utilice https:

```
"desarrollo-seguro": "vue-cli-service serve --mode development --port 44300 --open
--https",
```

Tras ejecutar el comando "npm run desarrollo-seguro", se abrirá el navegador con la aplicación por el puerto 44300 https ❸.

Para empaquetar el proyecto en modo producción, utilizando el entorno de desarrollo y sin optimizar, definimos el siguiente comando en la sección de scripts:

```
"produccion-modo-desarrollo": "vue-cli-service build --mode developmet",
```

Tras ejecutar el comando "npm run produccion-modo-desarrollo", se generará en la carpeta "dist" del proyecto la aplicación empaquetada en entorno de desarrollo, sin optimizar ni hacer "minify" de ficheros.

Servidor http

Al no poder abrir "index.html" directamente, podemos mover el contenido del directorio "dist" a un servidor de aplicaciones, o lanzar nuestra aplicación en un servidor http de Node. Optamos por la segunda opción. Necesitaremos instalar el paquete "serve" de forma global:

```
npm install -g serve
```

Una vez instalado, ejecutaremos el siguiente comando, donde indicaremos con una "-s", que nuestra aplicación se comporte como SPA con router, además de configurar el servidor a la escucha por el puerto 4000:

```
serve --listen 4000 -s dist
```

Como observamos, hemos levantado un servidor, arrancando el paquete compilado en desarrollo ❹.

Report

Otra función interesante proporcionada por el CLI es mostrar un informe de estructura de la aplicación a primera vista. Para ello declaramos el siguiente comando:

```
"produccion-report": "vue-cli-service build --report",
```

Este generará un fichero en "dist", con el nombre "report.html", donde veremos la estructura del proyecto ❺.

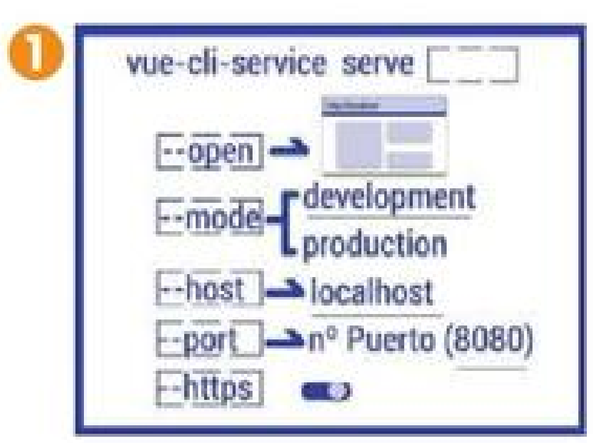
Modern

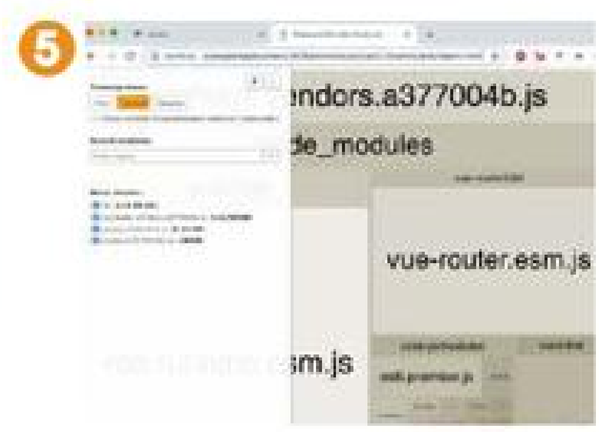
Para considerar los navegadores antiguos, el CLI de Vue dispone de una configuración en el fichero ".browserslistrc", donde se especifica los navegadores soportados por la aplicación ❻. Esta configuración es leída por babel al traducir el JavaScript.

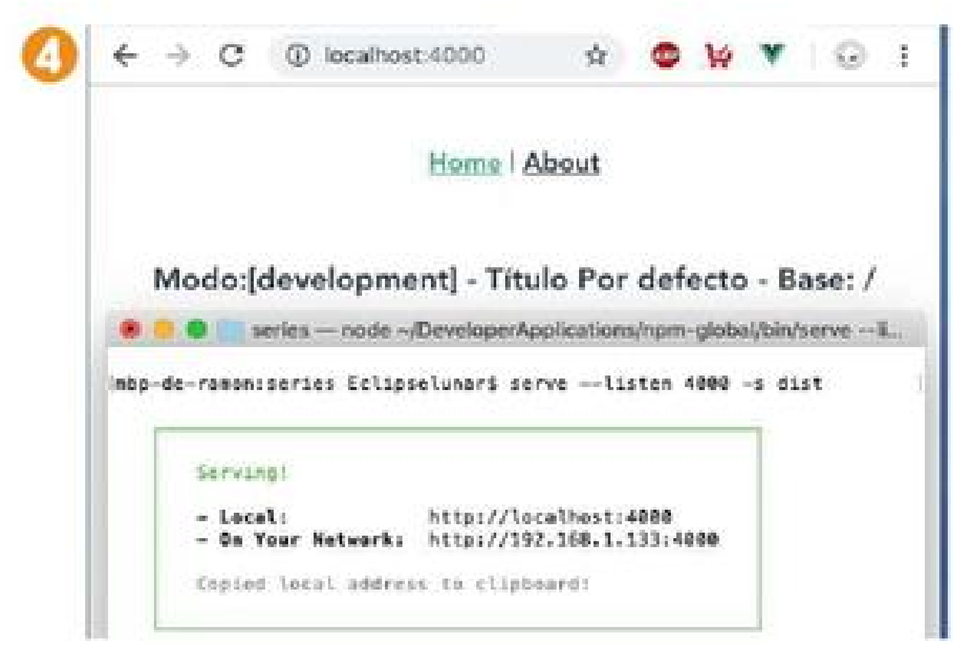
Existe la posibilidad de compilar dos versiones de nuestra aplicación, una para navegadores modernos y otra para los antiguos. De modo que, la opción "--modern", genera dos versiones que el navegador reconocerá en el html, al interpretar <script type="module">, los nuevos navegadores y <script nomodule>, los antiguos.

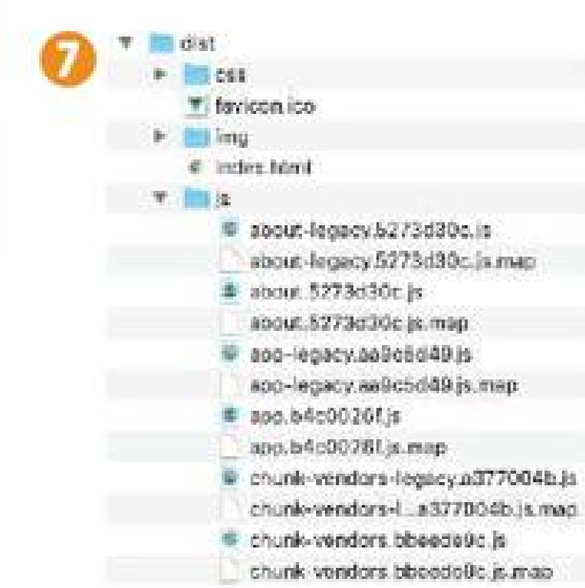
```
"production-modern":  "vue-cli-service build --modern",
```

Podemos ver que hay dos versiones de cada fichero js, uno para moderno y otro "legacy" para el antiguo ❼.









Componente – App, Librería y WC

IMPORTANTE

Es importante probar nuestros componentes antes de distribuirlos, además es importante que el nombre del componente web contenga guiones, para no colisionar con otras etiquetas html.

Tipo de compilación de componentes

Hasta ahora, mediante el comando "build" del servicio CLI de Vue, empaquetábamos de forma optimizada nuestra aplicación Vue por completo, incluyendo todos los componentes, en el directorio "dist" del proyecto. No obstante, es posible que lo que queramos es compilar únicamente un componente.

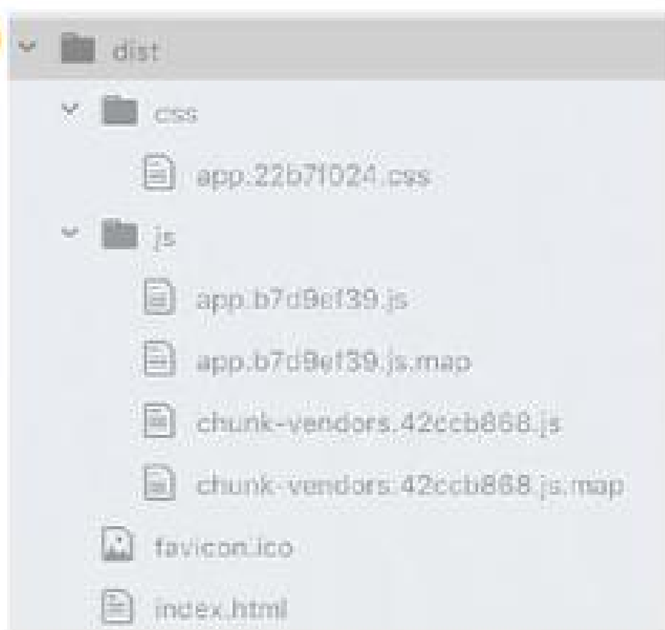
Para ello existen 3 tipos de construcción, conocidos como "build targets" en la documentación oficial de Vue, todos se ejecutan con el comando "vue-cli-service build", seguidos de "--target" y su tipo:

- app: modo utilizado hasta ahora al realizar "vue-cli-service build", sería equivalente a "vue-cli-service build --target app". Compila todo el código incluido de "src", lo optimiza e incluye el Framework de Vue.
- libray: modo utilizado para compartir con la comunidad uno de nuestros componentes, únicamente compilará el componente indicado, generando un paquete redistribuible con únicamente el código del componente, sin el Framework de Vue.
- wc: modo utilizado para crear componentes web reutilizables en otros Frameworks (Angular, React…), compilando el componente indicado, únicamente con el código de nuestro componente.

Antes de comenzar, para ejecutar en consola los comandos de construcción target mediante "vue build" debemos instalar el paquete de Node "@vue/cli-service-global":

```
npm install -g @vue/cli-service-global
```

1



2

3

```
demo.html
1  <meta charset="utf-8">
2  <title>mi-libreria-helloworld demo</title>
3  <script src="https://unpkg.com/vue"></script>
4  <script src="./mi-libreria-helloworld.umd.js"></script>
5  <link rel="stylesheet" href="./mi-libreria-helloworld.css">
6  <div id="app">
7    <demo></demo>
8  </div>
9  <script>
10 new Vue({
11   components: {
12     demo: mi-libreria-helloworld
13   }
14 }).$mount('#app')
15 </script>
```

App

Compilemos el componente ubicado en "src/components/HelloWorld.vue" para construir una aplicación de un solo componente. Esto construirá la aplicación con este componente únicamente.

Ejecutemos el siguiente comando para construir nuestro componente como aplicación, seleccionando el componente a construir y el target de aplicación:

```
vue build src/components/HelloWorld.vue --target app
```

Genera en "dist", el componente listo para despliegue a producción, que incluirá el Framework de Vue ❶. Lancemos el servidor ❷, para visualizar la apertura de la aplicación con este componente.

Lib

Convirtamos nuestro componente en una librería, ejecutando el siguiente comando indicamos el componente a convertir, el target librería y un nombre:

```
vue build src/components/HelloWorld.vue --target lib --name mi-libreria-helloworld
```

Se generará la librería del componente, lista para distribuir como paquete Node, sin incluir el framework de Vue ❸.

Web component

Por último, convirtamos nuestro componente en un componente web reutilizable, ejecutemos el comando, donde indicamos el componente a convertir, el target a componente web y un nombre:

```
vue build src/components/HelloWorld.vue --target wc --name mi-wcomponente-helloworld
```

Generará el componente web ❹ para reutilizar por cualquier Framework, mediante la etiqueta "<mi-wcomponente-helloworld>". Podemos abrir directamente el html generado, visualizando este componente ❺.

Despliegue en Servidor de Aplicaciones

IMPORTANTE

No es necesario crear un sub-site para alojar la aplicación, aunque al ser común consumir nuestros api, es recomendable que api y sitios web cuelguen del mismo dominio.

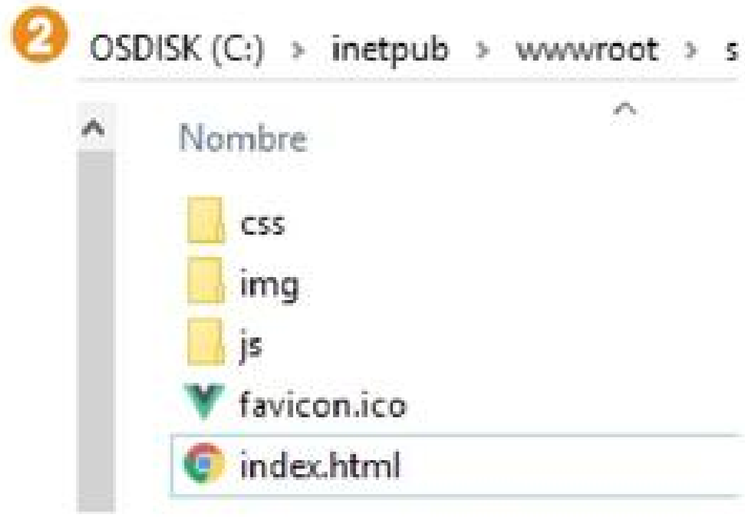
Servidor de aplicaciones

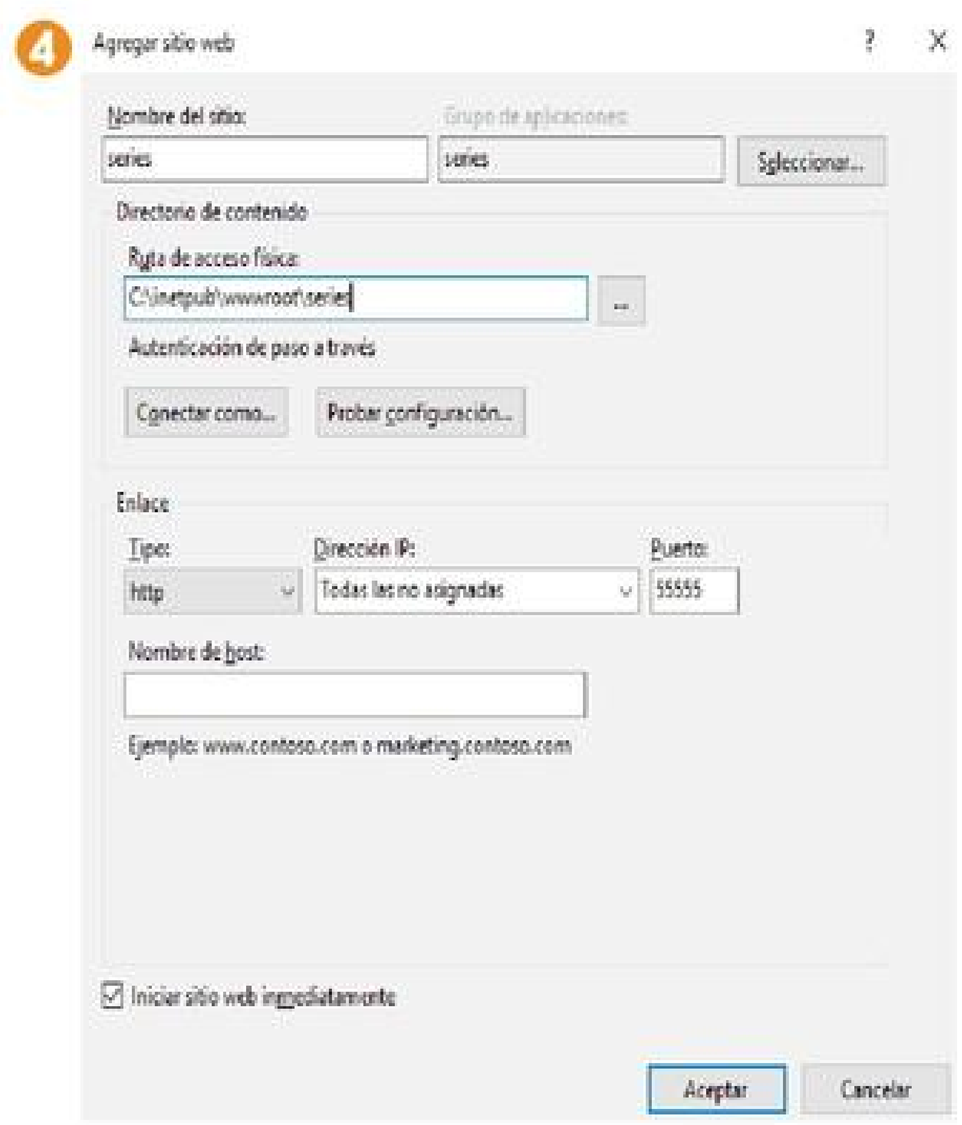
Llegados a este punto, ya dominamos las diversas formas de compilar el proyecto y prepararlo para distintos entornos, tanto para desarrollo, como para producción. Sin embargo, no habíamos probado a desplegar nuestra aplicación en un entorno de producción real, es decir un servidor de aplicaciones.

Para una mayor realidad, establezcamos para el modo desarrollo que todas las rutas de la aplicación se ubiquen en el path "/", es decir, la raíz; y en modo producción se ubiquen en el sub-site "/produccion/" del dominio. Esto lo indicaremos en el fichero "vue.config.js", en la opción "publicPath", donde según el modo de compilación, obtenida por la variable de entorno "NODE_ENV", el valor de la ruta base variará:

```
,publicPath: process.env.NODE_ENV === 'produc-
tion'? '/produccion/': '/'
```

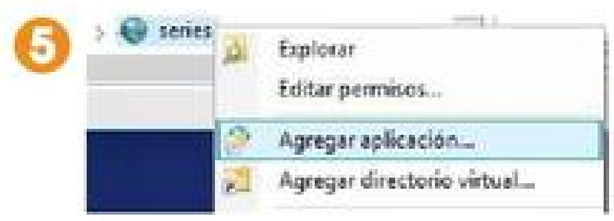
Despleguemos el proyecto Vue sobre el servidor de aplicaciones de Microsoft, Internet Information Services (IIS). Para ello, debemos seguir estos pasos:

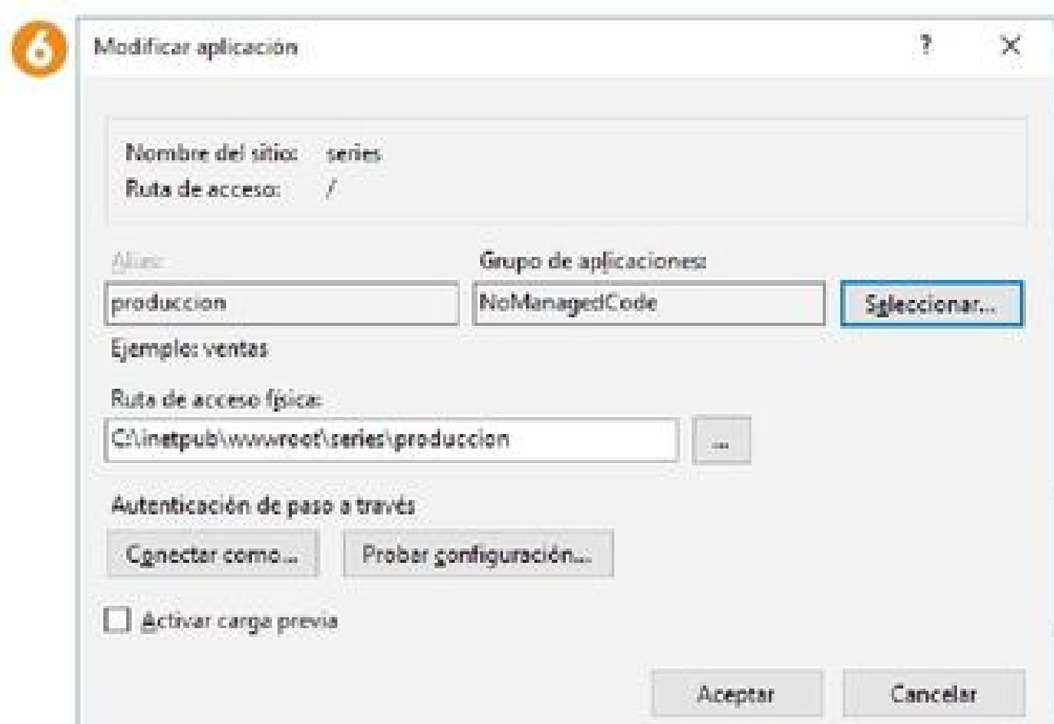




1. Compilamos el proyecto en modo producción, mediante el comando "npm run build". Esto generará en el directorio "dist", la aplicación optimizada y de tamaño reducido, listo para desplegar ❶.
2. Abrimos un Servidor Windows (en caso de disponer de él) o simplemente un ordenador que disponga de Windows, con las características de Windows IIS, instaladas en el equipo.
3. Creamos un directorio donde alojar nuestra aplicación ❷, pegando el contenido del directorio "dist" generado por Vue anteriormente, en dicho directorio.
4. Creamos un nuevo Sitio Web en IIS: botón derecho del ratón sobre "Sitios" -> "Agregar sitio web" ❸.
5. Se abrirá una ventana ❹, donde damos de alta el sitio "series", seleccionamos la ruta donde se encuentra nuestro sitio y le asignamos el puerto 55555.
6. Una vez creado el sitio, aparecerá en el árbol del IIS. Ahora crearemos el sub-site donde alojaremos nuestra aplicación, que será "produccion", para ello botón derecho sobre el sitio "series"-> "Agregar Aplicación" ❺.
7. Se abre una ventana donde seleccionamos como grupo de aplicaciones "NoManagedCode", para que el sitio no esté gestionado y se encargue el propio framework de Vue ❻.
8. Al utilizar vue-router, que explicaremos en próximos capítulos, necesitamos que las rutas las gestione el router de Vue y no el servidor de aplicaciones. Para ello debemos instalar una extensión en el IIS llamada "IIS URL Rewrite", descargable desde https://www.iis.net/downloads/microsoft/url-rewrite.
9. Si el paso 8 es la primera vez que lo realizábamos, reiniciamos el IIS.

Abrimos en el navegador la ruta del sitio del IIS que acabamos de configurar y se abrirá nuestra aplicación ❼.





Instalación Vue Router y Modos

IMPORTANTE

Utilizar el router de vue implica que el servidor de aplicaciones donde se despliegue debe delegar el enrutado de páginas a la aplicación SPA, en lugar de ser gestionadas por el servidor.

Single page application - SPA

Las aplicaciones SPA (Single Page Application) son aplicaciones de una única página, representando distintas vistas, sin recargar el navegador para cambiar de vista.

El punto de entrada de la aplicación es "index.html", cargando cada vista sin recargar navegador. Conseguimos este efecto mediante un sistema de rutas denominado Vue Router ❶.

Vue Router

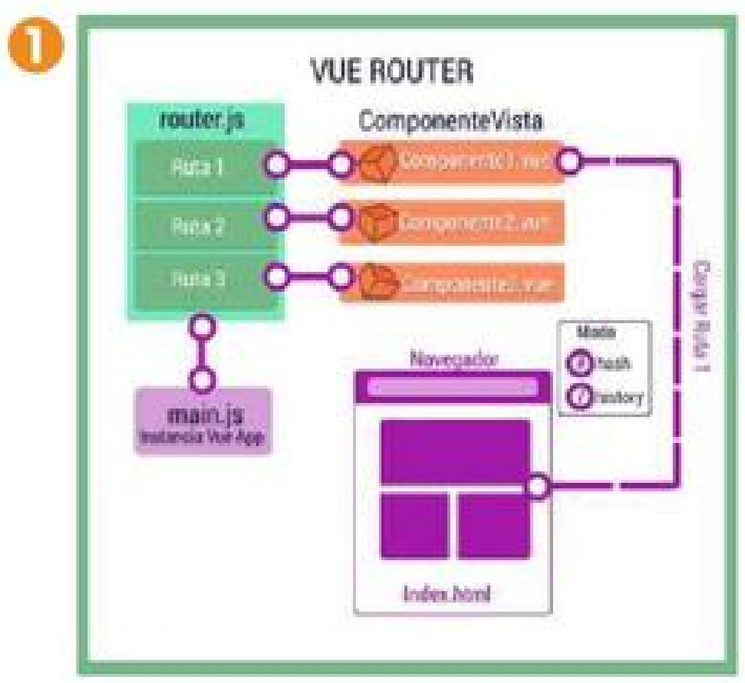
Este router permite definir rutas para cargar vistas, sin perder el foco de la página en que nos encontramos. En el capítulo de creación de proyecto vue mediante CLI 3 creamos el proyecto "series", dotando a nuestra aplicación del router Vue para aplicaciones SPA. Aunque tengamos disponible el router en nuestra aplicación, si quisiéramos a instalar el vue-router utilizaríamos el siguiente comando ❷:

```
npm install vue-router --save
```

En la carpeta de las vistas se encuentran los componentes "About.vue" y "Home.vue".

Tras instalar el router, en caso de no disponer del fichero "router.js", lo creamos. Configuramos en este fichero el router y definimos las rutas de nuestra aplicación. Importamos "vue" y el plugin "vue-router":

```
import Vue from 'vue'
import Router from 'vue-router'
```



Así como los componentes vista de nuestras rutas:

```
import Home from '@/views/Home.vue'
import About from '@/views/About.vue'
```

Para que Vue utilice este plugin debemos indicárselo, evitando utilizar el mismo plugin más de una vez:

```
Vue.use(Router)
```

Exportamos la instancia Router que acabamos de crear ❸, configurando:

- mode: modo de navegación ("hash" o "history").
- base: el Path origen de las rutas. Inicialmente desde "/".
- routes: rutas que interpreta el router (lo veremos en otro capítulo).

Por último, indicaremos en la instancia de Vue de la aplicación, en el fichero "main.js", el router configurado. Importemos el router y asociémoslo a la instancia Vue de la aplicación:

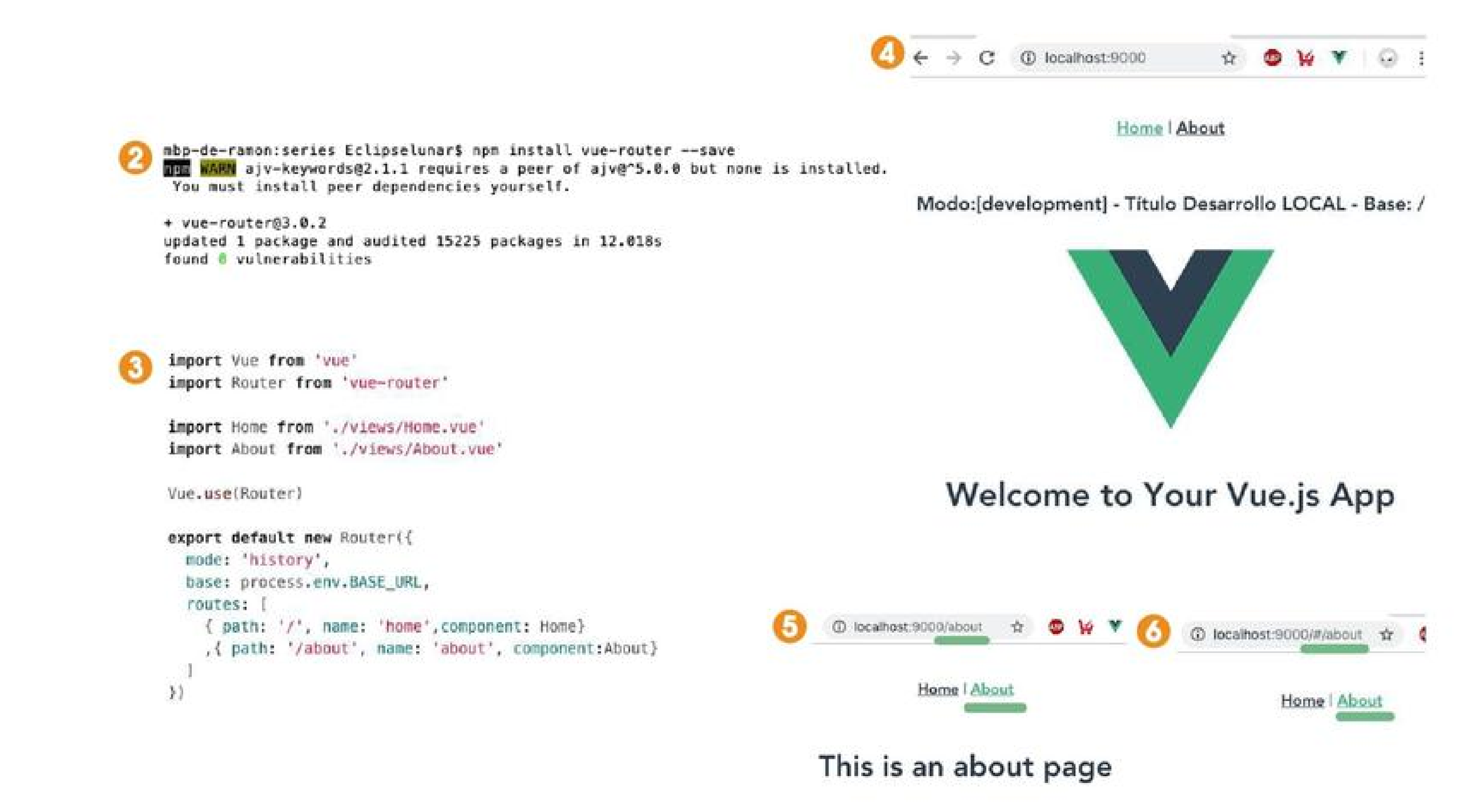
```
new Vue({ router, render: h => h(App) } ).$mount('#app')
```

Añadamos en la plantilla de la vista "App.vue" el contenedor donde mostrar el contenido HTML de la ruta visitada:

```
<router-view/>
```

Ejecutemos el proyecto, visualizando la vista "Home" 4, al ser la ruta por defecto "/"; si pulsamos el enlace "About", la ruta cambia a "/about", mostrando la vista "About" 5.

Si cambiásemos el modo de navegación a "hash", en lugar de "history" aparecería en la ruta "/#/about" 6, en lugar de "/about". Es el modo por defecto, utilizado para evitar la recarga de página, al cambiar de ruta, simulando nuevas URL. La ruta no es demasiado bonita, utilizaremos el modo historia.



Carga óptima de Ruta y navegación

IMPORTANTE

Al definir rutas, es preferible cargar sus vistas únicamente cuando se acceda a su ruta, sobre todo en aplicaciones muy grandes. No necesitamos disponer de todas las vistas cargadas desde inicio.

Rutas

Probemos la navegación entre vistas, definiendo una ruta. Será necesario crear en la carpeta "Views", una vista llamada "Series.vue", conteniendo esta plantilla estática:

```
<template>
   <div class="series"> <h1>Series</h1> </div>
</template>
```

Definamos una ruta en el array de rutas del fichero "router.js", para poder navegar hasta la vista "Series.vue".

```
   routes: [ …,{ path: '/series', name: 'series',
component: () => import('@/views/Series.vue') }
```

Si observamos los atributos de la ruta estática definida anteriormente:

- path: ruta visible en la barra del navegador.
- name: nombre único que identifica la ruta, para poder ser localizada la ruta por nombre.
- component: componente que se visualizará cuando accedamos a la ruta.

Destaquemos este último atributo, en lugar declarar un "import", para posteriormente asociárselo al componente:

```
import Series from '@/views/Series.vue'
```

Hemos optimizado la carga inicial de nuestra aplicación, cargando únicamente la vista "Series.vue", cuando se acceda a la ruta. Otra forma de realizar esta optimización sería de esta forma:

```
const Series = resolve =>
   {      require.ensure(['@/views/Series.vue'],()=>{
resolve(require('@/views/Series.vue')); }); };
```

Y por tanto la ruta sería referenciada de esta forma:

1
```
<template>
  <div id="app">
    <div id="nav">
      <router-link to="/">Home</router-link> |
      <router-link to="/about">About</router-link> |
      <router-link :to="{ name:'series' }">Series</router-link> |
    </div>
    <router-view/>
  </div>
</template>
```

```
{ path: '/series', name: 'series', component: Series }
```

La principal diferencia de cargar mediante "import" global es que, aunque nunca accedamos a la ruta que cargue la vista referenciada, estaría cargada desde el arranque de la aplicación, y eso no es óptimo.

Navegación

La navegación entre rutas de forma declarativa usa la etiqueta "<router-link/>" en lugar de "<a/>", creando enlaces sin recargar páginas. Definamos en la plantilla de "App.vue" tres enlaces ❶ para navegar entre las rutas definidas.

Existen 3 formas de declarar la ruta destino en la propiedad "to" del "router-link":

- Directamente con el nombre de la ruta:

```
<router-link to="/series">Series</router-link>
```

- Utilizando la equivalencia del anterior, mediante un objeto con la propiedad "path":

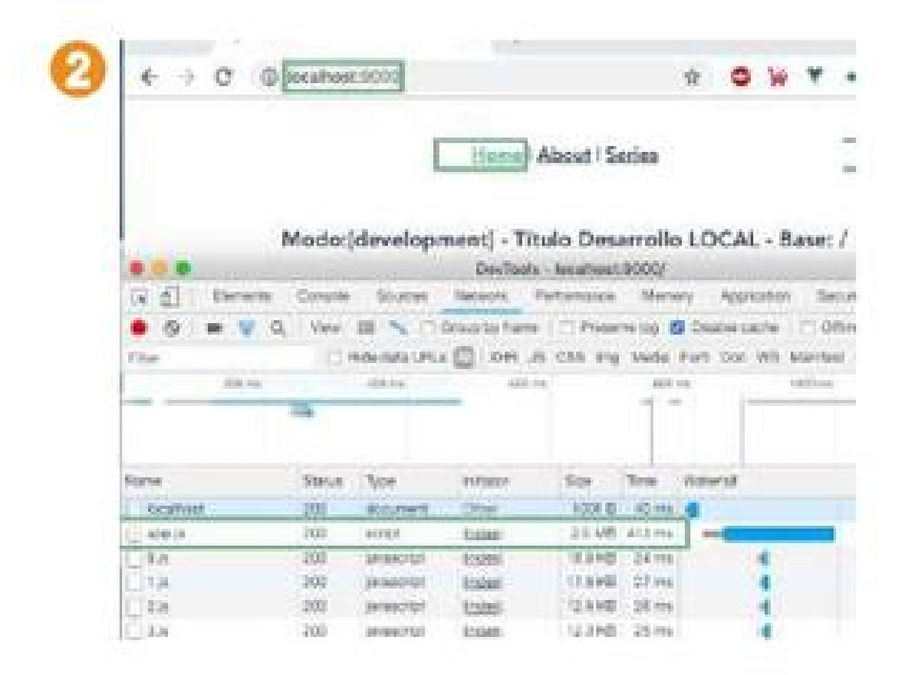
```
<router-link :to="{ path:'/series' }">Series</router-link>
```

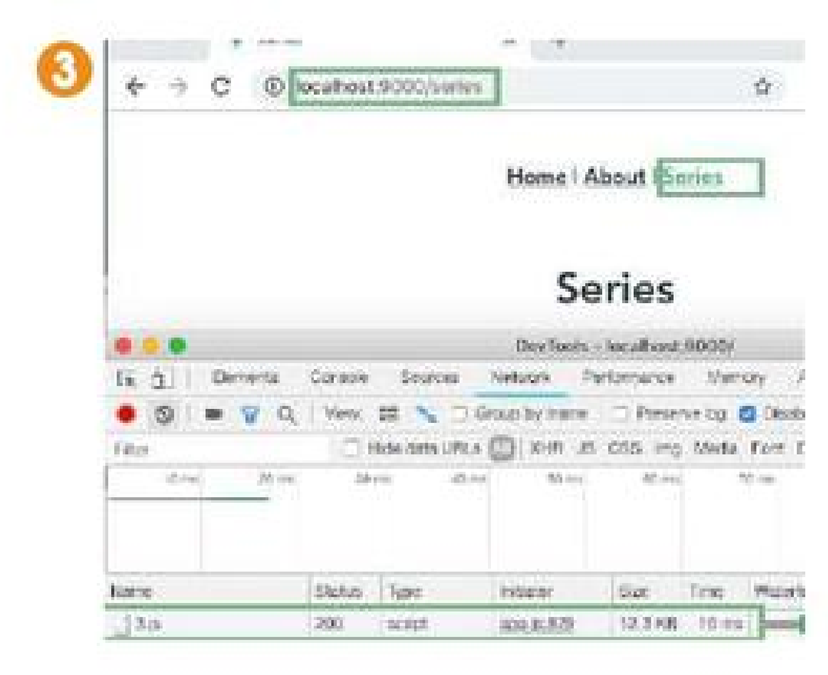
- Mediante el nombre que asignamos a la ruta a través de un objeto con la propiedad "name":

```
<router-link :to="{ name:'series' }">Series </router-link>
```

Ejecutamos la aplicación y abrimos las herramientas para desarrolladores del navegador Chrome, con la pestaña "Network". Veremos que se ha abierto la página, cargando "app.js" con un tamaño de 2MB ❷, si pulsamos el enlace "Series" se abrirá su ruta, cargando únicamente el fichero "3.js" con un peso de 12KB ❸.

Esto significa que inicialmente cargamos en "app.js" lo declarado en los "imports", a excepción de la vista de la ruta "Series", definida para ser cargada únicamente cuando se accediera a su ruta.





Rutas dinámicas

IMPORTANTE

No olvide declarar el "next()" en caso de que quiera controlar el acceso a la ruta en el "beforeRouteUpdate", si no se indicase, el componente no cambiaría de ruta.

Definición

Hasta ahora hemos utilizado las rutas estáticas; sin embargo, el uso de rutas dinámicas es muy común cuando visualizamos datos que pueden cambiar según un identificador proporcionado en la ruta.

Definamos una nueva vista "Serie.vue" en el directorio de las vistas "views":

```
<template>   <div class="serie"> <h1>Visualizar Serie</h1> </div>   </template>
```

Una vez definida, creamos una ruta en el "router.js" para esta vista:

```
   ,{ path: '/series/:id', name: 'serie', component: () => import('@/views/Serie.vue')}
```

El "path" de la ruta anterior contiene un segmento dinámico precedido dos puntos ":", seguido del nombre del parámetro dinámico, en este caso "id".

Definimos de manera declarativa en "App.vue" dos enlaces para acceder a las rutas dinámicas, usando un parámetro. Podemos hacerlo mediante la ruta (primera ruta) o utilizando un objeto (segunda), a través de los atributos "name" o "path" para identificar la ruta y el atributo "params" para el valor del parámetro "id":

```
<router-link to="/series/1">Serie 1</router-link> |

<router-link :to="{ name:'serie', params:{ id:2 } }">Serie 2</router-link>
```

Recepción del parámetro

En una instancia Vue tenemos acceso a las rutas mediante "$route". Modifiquemos la vista anterior para capturar el parámetro enviado por la ruta, añadiendo en la plantilla el acceso directo al

1
```
<template>
  <div class="serie">
    <h1>Visualizar Serie {{id}} - {{ $route.params.id }}
  </div>
</template>
<script>
export default {
  name: 'serie',
  data(){ return { id:this.$route.params.id }  }
}
</script>
```

valor del parámetro "id", de la ruta:

```
{{ $route.params.id }}
```

Así como una variable "id" en el data del componente que tomaría el valor mediante:

```
id: this.$route.params.id
```

Mostrada en la plantilla mediante {{id}} ❶.

Al ejecutar la aplicación, cuando accedemos al enlace "Serie 1", la vista muestra el valor del parámetro id, tanto de la variable del "data", como de la ruta ❷. De manera similar ocurre si pulsamos "Serie 2", salvo que el id obtenido en el data, sigue con el valor anterior ❸.

Se debe a que ambos enlaces referencian al mismo componente y, tras cambiar de ruta, el componente no se repinta, sino que se reutiliza el componente vigente.

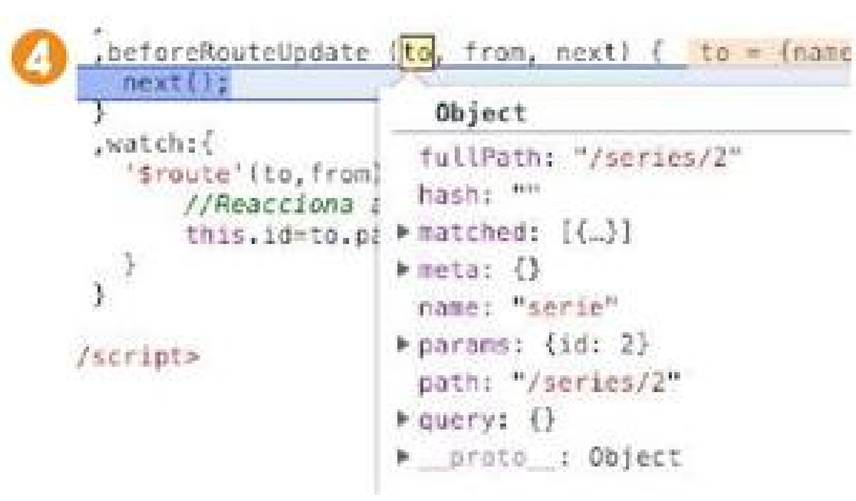
Para solucionar esto, debemos escuchar cambios en el objeto "$route" mediante un "watch", obteniendo valores de la ruta origen (from) y de destino (to), obteniendo el parámetro, que asignaremos a la variable "id" del data:

```
,watch:{   '$route'(to,from){    this.id = to.params.id; }   }
```

Disponemos de un protector de rutas, que permite la apertura de una ruta mediante "next()".

```
,beforeRouteUpdate (to, from, next) {  next();   }
```

Si volvemos a pulsar el enlace "Serie 2" ❹, a través de "beforeRouteUpdate" obtenemos por el "to", información de la ruta destino, permitiendo continuar hasta ser actualizada la vista correctamente ❺.



Rutas anidadas

IMPORTANTE

La propiedad "exact" en el enlace "router-link" nos permite activar el enlace si la ruta del navegador coincide exactamente con la del "path" del enlace.

Rutas

Una ruta puede componerse de varias sub-rutas en la misma página, del mismo modo que un componente puede contener otros anidados. Esta situación puede darse si necesitáramos que una ruta en la que se muestra un componente, tras anexar una sub-ruta, visualizáramos el contenido de ambas. Veámoslo con un ejemplo.

Una situación común para utilizar esta funcionalidad puede darse cuando un componente puede visualizarse de diferentes modos (lectura / escritura) o visualizar un contenido diferente según el área (pública / privada). Creemos este último escenario.

Definimos el componente "Perfil.vue" ❶ de la ruta padre, esta ruta contiene dos sub-rutas, podemos invocarlas desde dos enlaces "router-link", uno para el perfil público y otro para el privado. Además, anidaremos un contenedor "router-view" en el componente padre, contenedor donde representar el componente de la sub-ruta visitada.

Dado que vamos a representar dos sub-rutas, necesitamos crear un componente para cada ruta hija. Por una parte, creamos el componente "PerfilPrivado.vue" ❷, básicamente es una plantilla donde se representa información privada; por otra parte, creamos el componente "PerfilPublico.vue" ❸, conteniendo información poco sensible.

❶
```
<template>
  <div class="perfil">
    <h1>Perfil Usuario</h1>
    <router-link :to="{ name: 'publico' }"
                 active-class="active" exact>Público</router-link>
    <router-link :to="{ name: 'privado' }"
                 active-class="active">Privado</router-link>
    <router-view/>
  </div>
</template>
<style scoped>
  .active { color: #42b983; font-weight: bold; }
</style>
```

❷
```
<template>
  <div class="privado">
    <h1>Información Privada</h1>
    <p><b>Dni:</b> 00000000A</p>
    <p><b>Tel:</b> 555555555</p>
  </div>
</template>
<style scoped>
  .privado {
    padding: 0.25em; background-color: #448AFF; color: #ffffff;
    -webkit-box-shadow: 2px 2px 10px rgba(0,0,0,0.4);
    -moz-box-shadow: 2px 2px 10px rgba(0,0,0,0.4);
  }
</style>
```

❸
```
<template>
  <div class="publico">
    <h1>Información pública</h1>
    <label>Usuario 100 Ejercicios</label>
  </div>
</template>
```

❹
```
,{ path: '/perfil', component: () => import('@/views/Perfil.vue')
 ,children:[
    { path: '',        name: 'publico', component: () => import('@/views/PerfilPublico.vue')}
   ,{ path: 'privado', name: 'privado', component: () => import('@/views/PerfilPrivado.vue')}
 ]
}
```

Por último, creamos en "router.js" ❹, la ruta padre del perfil y las dos rutas hijas que descienden de esta. Para ello creamos la ruta padre, de la misma forma que hasta ahora, salvo que añadiremos en esta ruta la propiedad "children", que contiene un listado de rutas hijas, que serán representadas en el contenedor "router-view" del padre.

En la definición de las rutas hijas no se añade una barra "/" al inicio del "path" del hijo, se presupone que para anexar una ruta hija siempre le precede esta barra. Si ejecutamos la aplicación, al seleccionar el perfil, siempre que la ruta sea "/perfil" o "/perfil/", se representará en esta sub-ruta ❺ tanto el contenido de la ruta padre "perfil", como la ruta hija "público", esto último gracias al "router-view" del padre. Observemos que el enlace "Público" está marcado, debido a que la ruta hija ha detectado que está activa.

Podemos acceder a la ruta hija de la zona privada se a través de "/perfil/privado", por lo que cuando pulsemos el enlace "Privado", se representará tanto la ruta padre, como la ruta hija "privado" ❻.

Observando la última imagen vemos que, estando en la ruta hija "privado", se han marcado tanto el enlace público, como el privado, aun estando en la ruta "/perfil/privado". Esto se debe a que como la ruta pública "/perfil/" es parte de la ruta privada, se está marcando como activa también la pública. Para solucionar esto, debemos de indicar en el enlace "router-link" público, que únicamente tenga la clase activa, en caso de que coincida exactamente con la ruta. Para ello simplemente añadiremos la propiedad "exact" ❼, al primer enlace "router-link" que nos lleva a "/perfil/".

Imaginemos otra situación: si en lugar de visualizar dos zonas, quisiéramos dos sub-rutas, una para mostrar un componente en modo lectura y otra para mostrar un componente en modo edición, se realizaría exactamente igual.

Navegación desde código

Navegación

Tras practicar navegación entre rutas de forma declarativa, probemos desde código. El router trabaja con una pila de rutas LIFO (Last In First Out), la última ruta añadida, se apila al principio y será la primera en salir de la pila si navegamos atrás ❶.

Vamos a declarar un componente "404.vue" ❷, página a visitar en caso de que la ruta buscada no esté definida en el router. Añadiremos al router, la ruta que nos dirigirá a esta página:

```
,{ path: '404', name:'noEncontrada', component:() => import('@/views/404.vue')}
```

Para controlar rutas que no existan en el router declararemos una ruta que lo controle usando en el path "*", solicitando una redirección, mediante la propiedad "redirect", a la ruta de página no encontrada.

```
,{ path: '*', redirect:{ name:'noEncontrada' } }
```

Una vez definidas las rutas, introduzcamos en el navegador, una ruta inexistente, por ejemplo:

```
http://localhost:9000/rutaquenoexiste
```

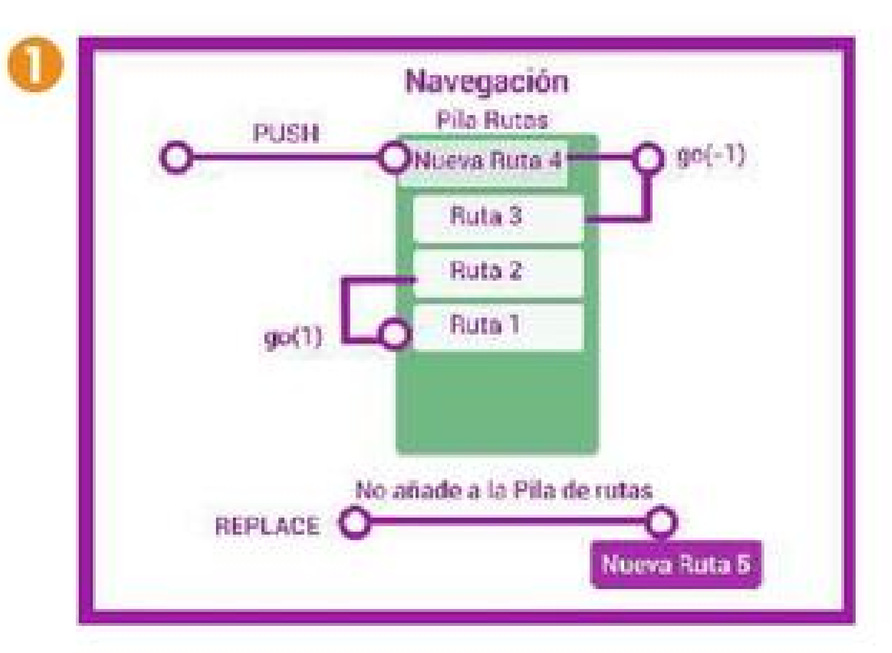
Al no existir en el router, será recogida por el path "*"y , como tiene la acción "redirect" definida, redirigirá a la página no encontrada ❸.

Replace

Una vez en esta página 404, podemos volver a la página principal "home" mediante un método "GoHome", invocado tras pulsar "Volver a Home", utilizando el método "replace" del router, redirigiendo a la ruta especificada sin ser añadida en la pila de rutas, evitando de este modo que la vuelta atrás vuelva a esta ruta.

```
this.$router.replace({name:'home'});
```

❶



❷

```
<template>
  <div class="404">
    <h1>Página No encontrada</h1>
    <span @click="GoHome()">Volver a Home</span>
  </div>
</template>
<script>
export default {
  name: '404'
  ,methods:{
    GoHome(){ this.$router.replace({name:'home'}); }
  }
}
</script>
```

PUSH

Una alternativa al "replace" es el "push" del router; difiere en que se añade la ruta a la pila del router, en primera posición. Probemos a cambiar de ruta en caso de que, al abrir una serie con un identificador no numérico, nos dirija a la pantalla principal. Para ello, definimos en el "mounted" del componente "Serie.vue", cambiar de ruta mediante "push", pasándole un parámetro "error" al cual asignaremos un mensaje.

```
,mounted() { if( isNaN( this.id ) )
  this.$router.push( { name: 'home', params: { error: 'El identificador no pertenece a la colección.' } } ); }
```

Una vez se produzca esta redirección a la ruta "home", mostraremos mediante un "alert" el mensaje de error recibido por parámetro en el "mounted" del componente "Home.vue":

```
,mounted() { if(this.$route.params && this.$route.params.error) alert(this.$route.params.error); }
```

De este modo, si escribimos la ruta http://localhost:9000/series/NoNumerico, debería redirigir la ruta serie a la principal, donde se mostrará el mensaje 4.

GO

Por último, al trabajar con una pila de rutas podemos avanzar n posiciones hacia delante y detrás mediante el método "go" del router. Definamos en "App.vue" dos textos que lancen dos métodos:

```
<span @click="Goback()">&#60;&#60;</span> | <span @click="Goforward()">&#62;&#62;</span>
```

Uno ejecuta el método "Goback" para abrir la ruta anterior mediante "go(-1)" y otro el método "Goforward" para abrir la ruta siguiente en la pila mediante "go(1)" 5.

```
methods:{ Goback() { this.$router.go(-1); } ,Goforward() { this.$router.go(1); } }
```

Protección global de rutas

IMPORTANTE

No olvide definir el "next" en caso de querer que la navegación continúe: no declararlo o asignar "next(false)" detendría la navegación.

Global BeforeEach

Podemos controlar el acceso a cada ruta restringiendo el acceso por diferentes motivos, uno de ellos puede ser no disponer de una sesión de usuario activa o que el usuario no dispone de rol suficiente para acceder a una ruta determinada. Podemos realizarlo globalmente ❶, a nivel de ruta o incluso de componente.

Simulemos el caso global, restringiendo el acceso a la aplicación si no estamos logados. Definamos en el "data" de la instancia de la aplicación del "main.js" la variable "loggedIn", cuyo valor por defecto es falso, simulando no estar logados en la aplicación.

```
data(){ return { loggedIn: false } },
```

Declaramos en este fichero el método "beforeEach" del router, capturando cada intento de cambio de ruta. Obtenemos la ruta origen "from" y la destino "to", habilitando la navegación mediante el método "next()".

En nuestro caso, habilitamos la navegación en caso de que el usuario esté logado, es decir la variable "loggedIn" sea "true", accediendo a la variable del "data" de la aplicación a través del router.

```
router.beforeEach((to,from,next)=>{ next( router.app._data.loggedIn ); })
```

En la plantilla de la aplicación "App.vue" definimos un enlace. Si el valor de esta variable del "data" de la aplicación es afirmativo, implica estar logado, de modo que si pulsamos el enlace, cambiamos su valor a falso mediante el método "DoLogin", o a cierto en caso de realizar login.

```
<span v-if="$router.app._data.loggedIn" @click="DoLogin(false)"> Logueado (Salir) </span>
<span v-else @click="DoLogin(true)"> Hacer Login </span>
```

Siendo el método que asigna el valor a dicha variable:

```
,DoLogin(login){ this.$router.app._data.loggedIn=login; }
```

Si ejecutamos la aplicación, el estado inicial de la variable "loggedIn" es falsa, aparece el enlace "Hacer Login" ❷, de modo que, si pulsamos cualquier ruta, ninguna se abrirá debido a que

"next(false)" nos lo impide.

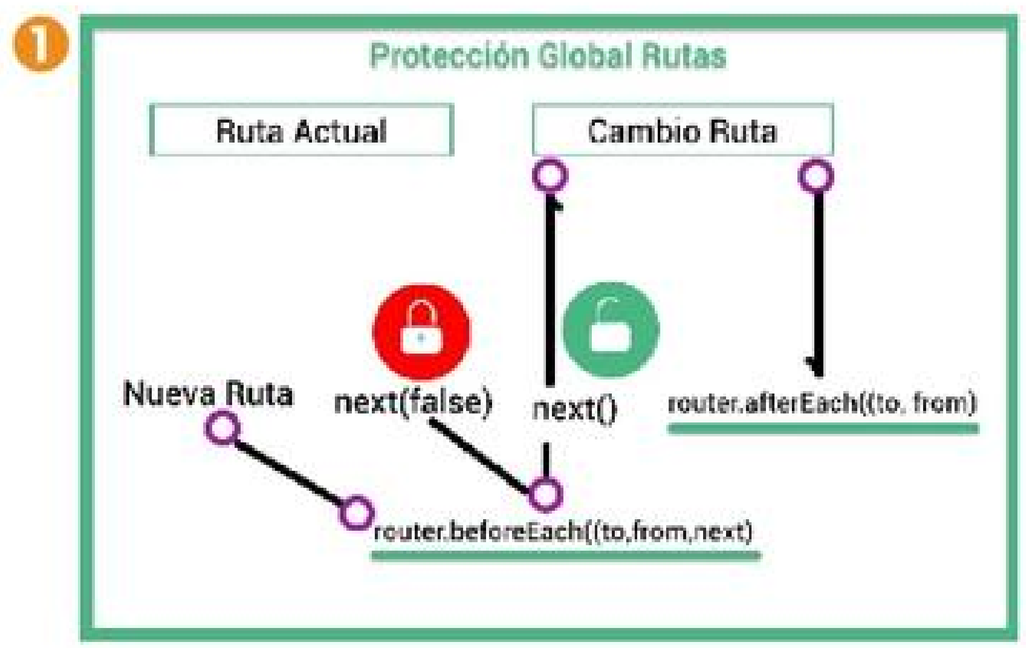
Si pulsamos "Hacer Login", esto cambia el valor de la variable "loggedIn" a "true", si ahora pulsamos cualquier ruta debería permitirnos continuar con la navegación, debido al "next(true)" ❸.

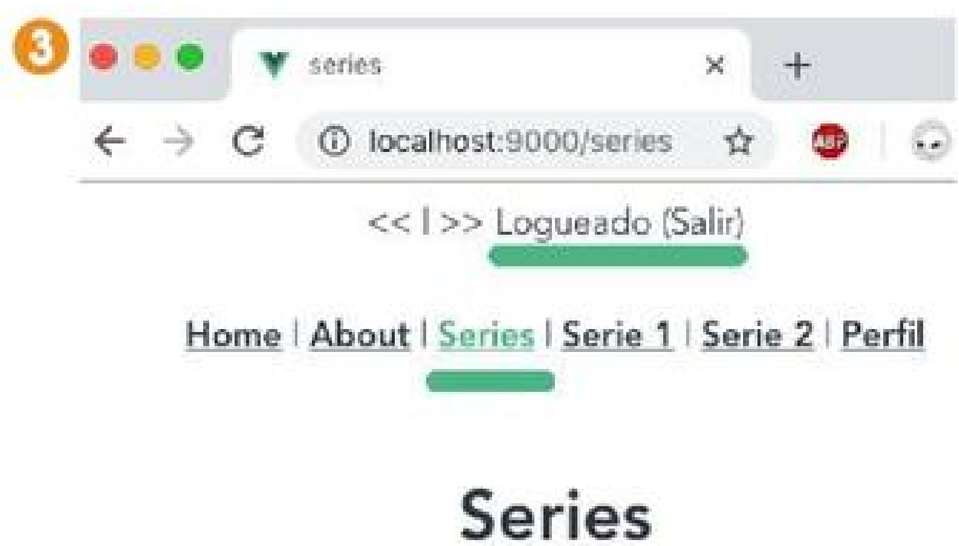
AfterEach

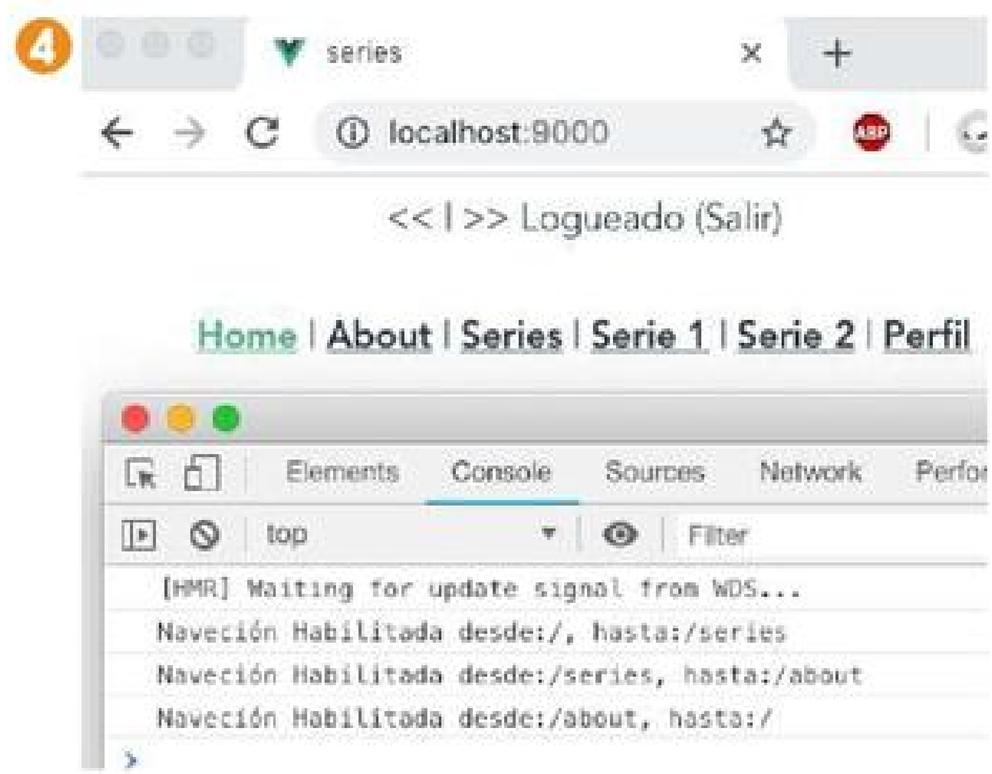
Una vez permitido el acceso, podemos registrar dicho acceso en el servidor o simplemente mostrar en consola desde-hasta qué ruta ha navegado el usuario. Utilizamos la función "afterEach" del router, ejecutada tras el "next" del "beforeEach", es decir, tras acceder a la ruta ❹.

```
router.afterEach( (to, from) => { console.log('Naveción Habilitada desde:'+from.
path+', hasta:'+to.path);  })
```

Esta protección es muy utilizada para este caso, no es el único. Podríamos haber comprobado mediante una llamada al servidor, si el usuario está autorizado o verificar el tipo de rol que tiene el usuario, para acceder a la ruta.







Protección en ruta y componente

Nivel Ruta

Podemos proteger la navegación hacia determinadas rutas a nivel de ruta, en el router. Para probarlo, creamos un componente "Favoritos" ❶, para representar nuestras series favoritas.

Este componente tiene "props" con una propiedad de texto "propCadena", cuyo valor por defecto es "todos". Recibirá del router este valor, asignándolo a la variable "cadena" del "data" del componente para visualizar su valor. Recibiremos actualizaciones de esta propiedad mediante el "watch".

Definimos una ruta para "favoritos", habilitando las "props" con una función para asignar a la propiedad "propCadena" del componente el valor de la ruta, a través del parámetro "cadena", de los parámetros "query" visibles en la barra de navegación "/favoritos?cadena=netflix".

```
,{ path: '/favoritos', name: 'favoritos', component: () => import('@/views/Favo-
ritos.vue')

   ,props:(ruta)=>( { propCadena: ruta.query.cadena } )

   ,beforeEnter: (to, from, next) => { if( to.query.cadena ) next();  else alert(
'Parámetro no permitido.' ); } }
```

Si observamos la ruta anterior, el método "beforeEnter" protege la navegación según un criterio. En nuestro caso, bloquear, si no recibimos por parámetros de navegación, el parámetro "cadena".

Definamos en "App.vue" enlaces para enviar por "query" el parámetro que espera la ruta para asignar las "props":

```
   <router-link :to="{ name:'favoritos', query:{ cadena:'hbo' } }">HBO</router-
link> |

   <router-link :to="{ name:'favoritos', query:{ cadena:'netflix' } }">Netflix</
router-link>
```

Si introducimos un parámetro en la "query", que no sea "cadena", bloqueará el avance ❷. En caso contrario, habilitará la navegación ❸.

❶
```
<template>
  <div class="favoritos">
    <h1>Favoritos {{cadena}}</h1>
  </div>
</template>
<script>
export default {
  name: 'favoritos'
  ,data(){ return { cadena:this.propCadena } }
  ,watch:{ propCadena(){ this.cadena=this.propCadena; }
  ,props:{ propCadena:{ type: String, default: 'todos'}
}
</script>
```

IMPORTANTE

No siempre tenemos acceso a la instancia del componente, sobre todo cuando aún no se ha cargado, como ocurre en "beforeRouteEnter". Podremos tener acceso dentro del "next", una vez cargado el componente.

Nivel componente

La protección puede ser a nivel de componente antes de entrar, durante y antes de salir. Definamos un checkbox que cogerá el valor booleano de la variable "hayCambios", del "data".

```
<input type="checkbox" v-model="hayCambios">Cambios
```

Antes de entrar al componente, asignemos el valor falso a la variable "hayCambios" para indicar que no hay cambios. Capturamos este evento mediante "beforeRouteEnter", al no tener acceso al "data", para asignar a la variable, un valor falso, lo hacemos dentro de "next", donde tenemos acceso a la instancia vue.

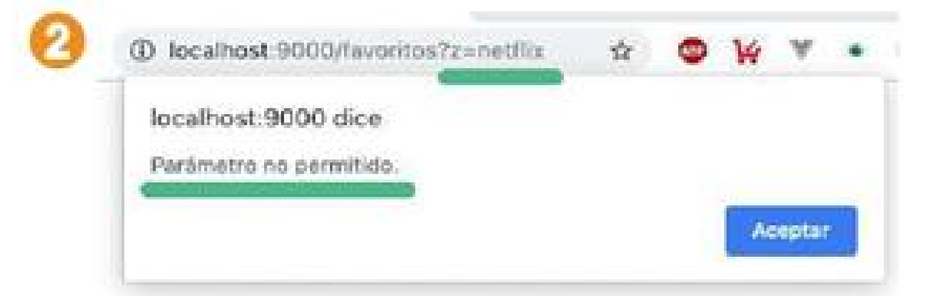
```
,beforeRouteEnter(to, from, next){    next((vm)=>{ vm.hayCambios=false; })   }
```

Si estamos en la misma ruta y cambia "propCadena", mostramos un mensaje de cambio de ruta al detectar este evento mediante "beforeRouteUpdate", donde tenemos acceso al "this" de la instancia ❹.

```
,beforeRouteUpdate (to, from, next){  alert('Cambiando de '+this.cadena+' a '+to.
query.cadena);

   next(); }
```

Finalmente, si pulsamos el checkbox de aplicar cambios e intentamos abandonar la ruta actual, solicita verificar salir sin guardar los cambios, en caso afirmativo habilitará la navegación, en caso contrario, la bloqueará ❺.

```
,beforeRouteLeave(to, from, next) {

   next(!this.hayCambios ||   (this.hayCambios && confirm("No ha guardado, ¿Desea
salir?")) );   }
```



Vuetify Diseño menú lateral

IMPORTANTE

Vuetify no es el único Framework disponible para nuestros proyectos Vue, existen otros como VueMaterial, sin embargo es uno de los más empleados por la comunidad.

Instalación

Para facilitarnos la vida a los desarrolladores, disponemos de un framework de componentes de diseño llamado Vuetify, para dotar a nuestras aplicaciones de un estilo atractivo. Podemos encontrar su documentación en https://vuetifyjs.com. Mejoremos el aspecto del proyecto en el que hemos estado trabajando.

Instalamos Vuetify en nuestro proyecto:

```
npm install vuetify -save
```

Importamos en "main.js" el modulo "vuetify":

```
import Vuetify from 'vuetify'

Vue.use(Vuetify)
```

Este framework utiliza una hoja de estilos, necesitaríamos un interprete de css. Por suerte, "@vue/cli-service" nos proporciona este "css-loader". Importamos la hoja de estilos de vuetify:

```
import 'vuetify/dist/vuetify.min.css'
```

Agregamos en la cabecera de la página de inicio "public/index.html", el uso de los iconos "material-icons" de google.

```
<link href="https://fonts.googleapis.com/
css?family=Material+Icons" rel="stylesheet">
```

Estamos listos para utilizar el framework.

1

```
<template>
  <v-navigation-drawer app permanent absolute class="teal elevation-5" dark
                       v-model="drawer" :mini-variant="menuCollapse">
    <v-list class="pa-0" >
      <v-list-tile >
        <v-list-tile-action>
          <v-toolbar-side-icon @click="menuCollapse=!menuCollapse">
          </v-toolbar-side-icon>
        </v-list-tile-action>
      </v-list-tile>
    </v-list>
    <v-toolbar flat class="transparent" color="teal" dark >

    </v-toolbar>
  </v-navigation-drawer>
</template>
<script>
export default {
  name: 'MenuLateral',
  data () { return { drawer: true, menuCollapse:false } }
}
</script>
```

Propiedad app
Barra Navegación
Toolbar
Propiedad utilizada para mostrar / ocultar barra navegación
Utilizada para mostrar menu reducido o ancho "mini-variant"

2

```
<template>
  <div id="app">
    <v-app>
      <menu-lateral />
      <v-content>
        <v-container>
          <router-view />
        </v-container>
      </v-content>
    </v-app>
  </div>
</template>
```

Aplicación
Navigation Drawer + Toolbar
Vista Router
Contenedor de la aplicación

Menú lateral

Barra de navegación o "Navigation Drawer" es un menú lateral para navegar por las rutas de la aplicación. Creamos una barra de navegación que permita colapsarse y expandirse, mediante un icono.

Creamos un nuevo componente llamado "components/MenuLateral.vue" ❶, cuya plantilla tendrá esta estructura:

```
<v-navigation-drawer>        <v-toolbar>  </v-toolbar>        </v-navigation-drawer>
```

En el "navigation-drawer" indicamos su posición absoluta y fija; además, la propiedad "app" indica vinculación con la aplicación y permitirá al "router-view" redimensionarse junto al menú. Para mostrar u ocultar este menú utilizamos la variable "drawer" del "data" del componente, por defecto siempre visible.

Para reducir su anchura utilizamos la propiedad "mini-variant" del "navigation drawer", cuyo valor será asignado por la variable "menuCollapse", por defecto expandido. Si pulsamos el icono del menú "v-toolbar-side-icon", lo colapsaremos o expandiremos mediante su evento clic:

```
@click="menuCollapse=!menuCollapse"
```

Definimos en la vista principal "App.vue" la estructura global de aplicación, incluyendo en un componente "v-app", el componente "menu-lateral" y dentro del contenedor de la aplicación "v-content" la vista de las rutas "router-view".

Importamos nuestro componente "MenuLateral.vue":

```
import MenuLateral from '@/components/MenuLateral.vue'
```

Se lo asignamos como componente a la aplicación ❷:

```
,components: { menuLateral: MenuLateral }
```

Si ejecutamos la aplicación, se muestra el menú gracias a que la variable "drawer" del componente, por defecto es visible ❸. Si pulsamos el icono superior izquierdo del menú se colapsará el menú, redimensionando la pantalla ❹.

V-Toolbar y rutas de navegación

V-Toolbar

Dotemos a nuestra barra de navegación "Navigation Drawer" del ejercicio anterior, de una "toolbar" que actúa como cabecera de esta. Haremos que en nuestra barra muestre, en caso de estar logado, el avatar del usuario, con su nombre y cierre de sesión. Y en caso de no estar logado, mostramos un enlace para logarnos ❶.

Creamos dentro de la barra de navegación el elemento "v-toolbar" y verificamos si el usuario tiene sesión, comprobando el valor de la variable "loggedIn" del "data" de la aplicación, accesible desde el router:

```
<v-list  v-if="$router.app._data.loggedIn"…
```

En esta lista "v-list" tenemos un icono, "v-icon", para el avatar del usuario; si lo pulsamos abrirá la página del perfil del usuario utilizando un enlace "router-link".

```
<router-link to="/perfil" title="Ir a Perfil">        <v-icon>face</v-icon>     </
router-link>
```

Al estar dentro de un elemento lista "v-list", su contenido es un "v-list-tile-content". Contiene el nombre del usuario, un icono para cerrar sesión que, tras pulsarlo, pondrá a falsa la variable "loggedIn".

```
<v-list-tile-content style="align-items: center;">       <b>Usuario Demo</b>
   <v-icon @click="$router.app._data.loggedIn=false"
      title="Cerrar Sessión">power_settings_new</v-icon>
</v-list-tile-content>
```

Si no estamos logados, visualizamos en lugar de lo anterior el título de la "toolbar" usando "v-toolbar-title". Contiene un icono y un texto "Iniciar Sesión" que, tras pulsarlo, la variable "loggedIn" cambiará a cierta, concediendo sesión.

```
<v-toolbar-title          v-else          @click="$router.app._data.loggedIn=true"
style="cursor:pointer">
   <v-icon title="Iniciar Sesión" >account_circle</v-icon> Iniciar Sesión
</v-toolbar-title>
```

Rutas

Añadimos en nuestro menú enlaces hacia rutas de la aplicación mediante un array definido en el "data" del componente "MenuLateral.vue", que incluye título, icono y ruta del enlace:

```
rutas: [   { titulo: 'Home', icono: 'dashboard', to:'/'},
   { titulo: 'About', icono: 'question_answer', to:'/about'},
```

```
    { titulo: 'Series', icono: 'live_tv', to:{ name:'series' }},
    { titulo: 'Favoritos', icono: 'star', to:{ name:'favoritos', query:{
cadena:'todos' } }} ]
```

Tras el separador del "navigation-drawer", añadimos la lista de enlaces utilizando el array de rutas anterior ❷.

Al abrir la aplicación, aparece la barra expandida sin usuario logado con el inicio de sesión ❸. Si pulsamos, aparecen los enlaces que acabamos de crear así como el icono del usuario logado y el cierre de sesión ❹. Si colapsamos el menú aparecen los iconos de la lista, por ser de tipo acción "list-tile-action" ❺. Por último, si pulsamos una ruta de la lista, se abrirá la ruta ❻.

❶

```
<v-navigation-drawer ...>
  <v-toolbar flat class="transparent" color="teal" dark >
    <v-list class="pa-0" v-if="$router.app._data.loggedIn" >
      <v-list-tile>
        <v-list-tile-avatar >
          <router-link to="/perfil" title="Ir a Perfil">
            <v-icon>face</v-icon>
          </router-link>
        </v-list-tile-avatar>

        <v-list-tile-content style="align-items: center;">
          <b>Usuario Demo</b>
          <v-icon @click="$router.app._data.loggedIn=false"
                  title="Cerrar Sesión">power_settings_new</v-icon>
        </v-list-tile-content>
      </v-list-tile>
    </v-list>
    <v-toolbar-title v-else @click="$router.app._data.loggedIn=true"
                     style="cursor:pointer">
      <v-icon title="Iniciar Sesión" >account_circle</v-icon> Iniciar Sesión
    </v-toolbar-title>
  </v-toolbar>

  <v-divider></v-divider>

</v-navigation-drawer>
```

Toolbar
En caso de estar Logado
Link para ir al Perfil
Icono Cara
Logado
Cerrar Sesión
Hacer Login
No Logado
Título Toolbar
Línea divisoria

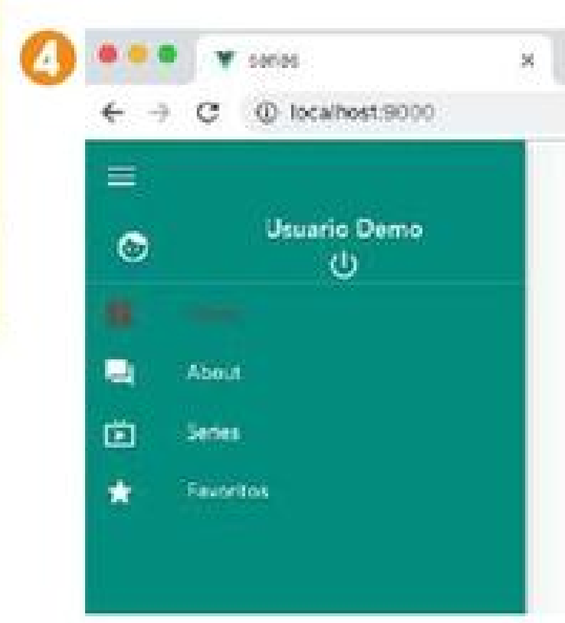
❷

```
  </v-toolbar>

  <v-divider></v-divider>

  <v-list class="pt-0" dense v-if="$router.app._data.loggedIn">
    <v-list-tile v-for="ruta in rutas" :key="ruta.title"
                 :to="ruta.to"   active-class="brown--text">
      <v-list-tile-action>
        <v-icon>{{ ruta.icono }}</v-icon>
      </v-list-tile-action>

      <v-list-tile-content>
        <v-list-tile-title>{{ ruta.titulo }}</v-list-tile-title>
      </v-list-tile-content>
    </v-list-tile>
  </v-list>

</v-navigation-drawer>
```

Listado de rutas
Elemento de la lista de rutas
Icono Ruta
Título Ruta

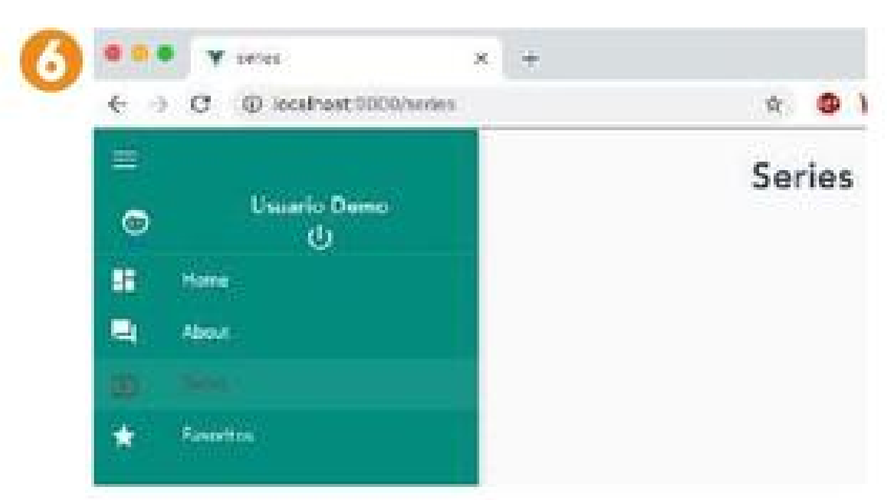
❸

❹



❺

❻



Crear Tablas

V-DATA-TABLE

Vuetify nos proporciona un componente tabla llamado "v-data-table" con un diseño espectacular. Principalmente se compone de dos elementos: las columnas o "headers" y las filas o "items".

En este componente podemos utilizar una serie de plantillas llamadas "slots" y proporcionarlas contenido. Disponemos del "slot=items", donde podemos añadir el contenido para cada fila, su "td". También disponemos del "slot=headers" para el contenido de las columnas "th".

Vamos a crear un nuevo componente con una tabla de Vuetify para representar columnas y filas proporcionadas al componente a través de las "props" ❶.

Para ello creamos el componente "components/Tabla.vue", utilizando un "v-data-table", pasándole columnas a través de "headers" y filas a través de "items".

```
<v-data-table :headers="headers" :items="rows">
```

No utilizaremos el slot de las columnas, pero sí el de las filas para representar el valor de las columnas de cada fila.

```
<template slot="items slot-scope="props">
```

Cada iteración del slot representa una fila, utilizando un bucle "for" por cada columna para representar su valores, utilizando una columna "td" y su valor, que obtenemos mediante "props. item[column.vaue]", siendo "column.value", el nombre de la columna y "props.item" la fila.

A nivel de código, añadimos dos "props":

- propRows: recibe un array de filas del componente padre.
- propColumns: recibe un array de columnas del componente padre.

Para que la tabla interprete nuestras las columnas debemos mapearlas, creando una propiedad computada "headers", donde mapeamos las columnas recibidas, a columnas de la plantilla. Entre otras propiedades, la tabla interpreta que para cada columna "value" es la clave, "text" es el nombre textual, "align" es la alineación del texto y "sortable" indica si la columna es ordenable.

```
headers() { return this.columns.map(col=>{ return { value: col.Key, text:col.Text,
align: 'center', sortable: true} }); }
```

IMPORTANTE

Hay multitud de slots que podemos utilizar en este componente, puede ser interesante definir manualmente el "tr" y sus "th" de la cabecera mediante el slot "headers".

Añadimos dos variables en el "data" del componente, "rows" para las filas y "columns" para las columnas.

Vamos al componente de tipo vista "Series.vue", donde agregamos en la plantilla este componente tabla creado, pasándole por las "props" las columnas y filas que definiremos:

```
<tabla :propColumns="columns" :propRows="rows"></tabla>
```

Importamos el componente tabla:

```
import Tabla from '@/components/Tabla.vue';
```

Por último, agregamos en el "data" de series, el modelo para columnas y filas ❷.

Si abrimos la aplicación, en la ruta "/series", veremos la tabla con dos filas, cinco columnas y paginación. Siendo las columnas ordenables, pulsando en sus cabeceras ❸.

❶

```
<template>                                  Contiene Columnas   Contiene Filas   Clase efecto levitación
  <v-data-table :headers="headers" :items="rows" class="elevation-1">
      <template slot="items" slot-scope="props">
        <td v-for="(column,index) in headers" :key="'td_'+index"
          class="truncar">{{props.item[column.value]}}</td>
      </template>                                   Valor
                                  props.item -> Fila
  </v-data-table>                 column.value-> Nombre columna
</template>
<script>
export default {
  name: 'Tabla'
  ,data () { return { rows:this.propRows, columns:this.propColumns} }
  ,computed:{                     Construimos las columnas, mapeándolas al formato aceptado
    headers() { return this.columns.map(col=>{ return { value: col.Key,
                          text:col.Text, align: 'center', sortable: true}
              }
  },props:{
    propRows:{ type:Array, default() { return [] } },   Recibimos filas por Props
    propColumns:{ type:Array, default() { return [] } } Recibimos columnas por Props
  }
}                                                       Clase para truncar
</script>                                               columnas extensas
<style scoped> .truncar{ white-space: nowrap;     overflow: hidden;
                         text-overflow: ellipsis; max-width: 10em; }
</style>
```

Tabla

Template - filas con td/th

❷

```
,columns:
[{ "Key" : "Titulo"      , "Text" : "Titulo" },
  { "Key" : "Anyo"        , "Text" : "Año" },
  { "Key" : "Cadena"      , "Text" : "Cadena" },
  { "Key" : "Descripcion" , "Text" : "Descripción" },
  { "Key" : "Temporadas"  , "Text" : "Temporadas" }]
,rows:[{ "Anyo":          2018,
         "Cadena":        "Netflix",
         "Descripcion":   "Descripción 1",
         "Id":            1,
         "Temporadas":    10,
         "Titulo":        "Título 1"}
```

❸

Selección de filas en tabla

IMPORTANTE

Cualquier cambio en las "props" de un componente deberá ser capturado por el "watch" y actualizar las variables del modelo del "data" para actualizar la vista del componente.

Actualizar cambios

La selección de filas es una funcionalidad empleada para realizar acciones sobre varios elementos al mismo tiempo, por esta razón consideramos útil ejercitar esta funcionalidad también proporcionada por Vuetify.

Actualicemos la vista de nuestro componente tabla "Tabla.vue", tras recibir modificaciones de filas o columnas desde el componente padre a través del "watch". Esto se debe a que al proporcionar filas y columnas por las "props", estas son leídas en tiempo de creación del componente. Solo con las "props" no detectaríamos los cambios, necesitamos declarar en el "watch", la escucha de cambios de valor en estas propiedades.

```
    ,watch:{ propRows(){ this.rows=this.propRows; },
  propColumns(){ this.columns=this.propColumns; }   }
```

Con esto conseguiríamos que cualquier cambio recibido, que afectara a las filas por "propRows" o a las columnas por "propColumns", el componente lo recibiría y refrescaría su vista.

Selección filas

Dotemos a nuestro componente "Tabla.vue", de la funcionalidad de selección de filas tras pulsar sobre un checkbox que se encontrará en la primera columna de cada fila ❶. Para ello necesitaremos añadir al "data" del componente un array llamado "selected", que contendrá estos elementos seleccionados:

```
,data () { return { rows: this.propRows, columns:
this.propColumns, selected:[] } }
```

❶

```
<template>
  <v-data-table v-model="selected" select-all item-key="Id"
                :headers="headers" :items="rows" class="elevation-1">
      <template slot="items" slot-scope="props">
        <td>
          <v-checkbox v-model="props.selected" primary hide-details></v-checkbox>
        </td>
        <td v-for="(column,index) in headers" :key="'td_'+index"
            class="truncar">{{props.item[column.value]}}</td>
      </template>
  </v-data-table>
</template>
```

Modelo almacenar seleccionados
Checkbox en columna para seleccionar todos
Clave del objeto que identifica la fila
Si está seleccionados, se marca
Checkbox en la columna de cada fila

Este array es el modelo de la tabla declarado en la tabla mediante "v-model", controlando las filas seleccionadas. Mostrando en la cabecera un checkbox de selección de todas las filas mediante la propiedad "select-all":

```
<v-data-table    v-model="selected"    select-all
```

Al abrir la ruta de "/series" vemos la tabla con este checkbox en la cabecera ❷, al marcarlo se seleccionan todas las filas, pero hay un problema: la última columna de cada fila está vacía, debido a haber añadido en la cabecera esta nueva columna para el checkbox de seleccionar todas. Una solución es añadir a cada fila una columna con un checkbox, añadida al "slot=items" de la tabla.

```
<td>  <v-checkbox v-model="props.selected" primary hide-details></v-checkbox>   </td>
```

Acabamos de solucionar el problema ❸. Sin embargo, seguimos teniendo un problema: al seleccionar una fila, se seleccionan todas. Esto se debe a no haber indicado la propiedad que identifica a cada fila; indiquemos que la propiedad "Id" es la clave de cada fila, añadiendo al v-data-table "item-key=Id" ❹.

Esto soluciona los problemas; ahora, al seleccionar una fila se selecciona únicamente dicha fila. Por último, añadamos en el "watch" escuchar cambios en las filas seleccionadas, mostrando por consola del navegador, mediante F12, el número de elementos seleccionados ❺.

```
,watch:{ … , selected() { console.log( this.selected.length + " elementos Selec-
cionados."); }
```

❷

❸

❹

❺

```
1 elementos Seleccionados.
0 elementos Seleccionados.
1 elementos Seleccionados.
2 elementos Seleccionados.
0 elementos Seleccionados.
2 elementos Seleccionados.
0 elementos Seleccionados.
```

Comunicación Hijo a Padre y Snackbar

IMPORTANTE

Recuerde declarar la escucha del evento lanzado por "$emit" en el componente padre, de lo contrario no será capturado el evento. También es posible acceder desde código mediante "this.$emit".

$EMIT

Existen diferentes tipos de comunicación entre componentes, en este capítulo utilizaremos la comunicación entre el componente hijo "Tabla.vue" y su componente padre "Series.vue" ❶ para enviar el objeto de la fila pulsada en el componente tabla hacia el componente series, de tal forma que muestre una notificación.

Empecemos por declarar en la fila del componente "Tabla.vue" un evento clic para que, al pulsar una fila, se lance un evento mediante "$emit". Esta instrucción emite un evento por si mismo, siendo el primer parámetro requerido, contiene el nombre del evento que queremos lanzar, en nuestro caso lo nombraremos "row-clicked", mientras que el segundo parámetro contiene el objeto que lanzamos junto al evento, como parámetro ❷.

```
<td ...  @click="$emit('row-clicked',props.
item)"> >{{props.item[column.value]}}</td>
```

De esta forma, cuando pulsemos una fila lanzamos el evento "row-clicked", que será interceptado por cualquier componente superior, que esté a la escucha mediante un "v-on". Hagamos que el componente padre "Series.vue" esté a la escucha de este evento. Para ello simplemente declaramos dentro de "Series.vue", en la definición del componente tabla, la escucha de este evento, así como el nombre del método a ejecutar "rowClicked" en el componente padre, tras capturar el evento:

```
<tabla  :propColumns="columns"  :propRows="rows"
v-on:row-clicked="rowClicked" ></tabla>
```

❶

❹

Snackbar

Una vez capturado el evento de la tabla, vamos a utilizar un componente de vuetify llamado "Snackbar" para mostrar por pantalla una notificación durante 3 segundos ❸. Necesitamos declarar en el "data" del componente "Series.vue" tres variables; "message" para el texto a mostrar, "snackbar" para mostrar u ocultar la notificación y "timeout" para ocultar automáticamente la notificación pasados 3 segundos.

```
,data(){ return { … ,message:null, snackbar:false, timeout:3000
```

Declaramos el método "rowClicked", que establecerá como texto de la notificación, el título y la cadena seleccionada, datos extraídos del objeto recibido por parámetro del evento que hemos recibido del componente hijo "Tabla.vue".

```
,methods:{
   rowClicked(item){   this.message="Ha seleccionado: "+item.Titulo+"("+item.
Cadena+")";
      this.snackbar=true; } }
```

Abrimos la aplicación y, al pulsar una fila, el hijo lanza el evento "row-clicked" con los datos de la fila seleccionada, el padre "Series.vue" lo captura y muestra una notificación que se cerrará automáticamente a los 3 segundos, también podemos cerrarla manualmente ❹.

❷
```
<template>
  <v-data-table v-model="selected" select-all item-key="Id"
                :headers="headers" :items="rows" class="elevation-1">
     <template slot="items" slot-scope="props">
       <td>
         <v-checkbox v-model="props.selected" primary hide-details></v-checkbox>
       </td>
       <td v-for="(column,index) in headers" :key="'td '+index"
           class="truncar" @click="$emit('row-clicked',props.item)">{{props.item[column.value]}}</td>
     </template>
  </v-data-table>
</template>
```

Evento Semit — Nombre evento — Parámetro

❸
```
<template>
  <div class="series">
    <h1>Series</h1>
    <tabla :propColumns="columns" :propRows="rows"
           v-on:row-clicked="rowClicked"></tabla>
```

Nombre Evento a escuchar — Método a invocar

SNACKBAR
```
<v-snackbar v-model="snackbar" color="teal" :timeout="timeout">
  {{ message }}
  <v-btn dark flat @click="snackbar = false">Cerrar</v-btn>
</v-snackbar>
```

Muestra/Oculta Snackbar — Milisegundos que permanece abierta

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

Diseñar y visualizar ficha

Elementos Vuetify

Utilicemos elementos de Vuetify para representar un elemento seleccionado por la tabla anterior en el componente "Serie.vue".

Continuamos desde el ejercicio anterior en "Series.vue", añadimos en el método "rowClicked", donde capturamos la fila seleccionada, redirigir a la ruta "serie", con el parámetro "id" y el objeto seleccionado "item".

```
      rowClicked(item){  …  this.$router.push({ name:'serie', params:{id:item.
Id, item}})     }
```

Además, añadimos la propiedad "Portada" a los objetos del JSON de la tabla, que contendrá la URL de una imagen.

En el componente "Serie.vue", capturamos por los parámetros de la ruta, el "id" del objeto y el elemento "item" completo. Vayamos a este componente y creemos en el data una variable para almacenar el elemento recibido:

```
  data() {  return {  … ,item: this.$route.params.item }
```

Ahora creemos una plantilla con una miga de pan y la ficha de la serie. Para la miga de pan, utilizamos el elemento tarjeta "<v-card>" de vuetify, contendrá un enlace para volver atrás, hacia el

```
1 <template>
    <div class="serie">
      <v-card>                                   v-card -> contenedor de tarjeta
        <v-layout row wrap>
          <v-flex xs12 style="float:left;margin: 1em;display: flex;">
            <router-link :to="{ name: 'series' }">Series</router-link>/ {{ id }}
          </v-flex>                              Enlace Volver atrás a Series.vue   Id elemento
        </v-layout>
      </v-card>
      <br/>
      <v-card>
        <v-img :src="item.Portada" height="200px">=</v-img>
        <v-container>=</v-container>
        <v-card-actions>=</v-card-actions>
      </v-card>
    </div>
  </template>

2 <v-card>                                       Imagen de portada, con altura fija
    <v-img :src="item.Portada" height="200px">
      <v-container fill-height fluid pa-2 >      Contenido un Titulo
        <v-layout fill-height>
          <v-flex xs12 align-end flexbox>
            <span class="headline white--text" v-text="item.Titulo"></span>
          </v-flex>
        </v-layout>
      </v-container>
    </v-img>
    <v-container>=</v-container>
    <v-card-actions>=</v-card-actions>
  </v-card>
```

Miga Pan / BreadCrum — Cuerpo Tarjeta — Cuerpo Tarjeta

componente "Series.vue", así como el identificador de la serie que estamos representando ❶.

Ahora crearemos otro elemento tarjeta, con una imagen que utilizará la propiedad "Portada" para obtener la ruta de la imagen, se redimensionará manteniendo la altura mediante "fill-height" e incluiremos además su título ❷.

Debajo de esta imagen añadimos un contenedor "v-container" que contiene la ficha de la serie. Una fila compuesta por 3 columnas, una para el año, otra para representar mediante el elemento "<v-chip>", un botón con la cadena que emite la serie y por último un elemento "<v-badge>", para representar en una burbuja el número de temporadas de la serie ❸. Incluimos una barra divisora "<v-divider>" para separar lo anterior de la descripción de la ficha. Por último, añadimos un icono en la barra de acciones de la tarjeta, para indicar los favoritos ❹.

De modo que, si seleccionamos una serie de las filas de la tabla de la aplicación, se abrirá el componente "Serie.vue" donde visualizamos su ficha así como la miga de pan ❺. Podremos volver atrás pulsando la miga de pan.

❸
```
<v-container>
  <v-layout>
    <v-flex xs4 style="margin:1em;"><h3>Año:{{item.Anyo}}</h3></v-flex>
    <v-flex xs4>
      <v-chip v-if="item.Cadena=='Netflix'" color="red" text-color="white">{{item.Cadena}}</v-chip>
      <v-chip v-else-if="item.Cadena=='HBO'" color="black" text-color="white">{{item.Cadena}}</v-chip>
      <v-chip v-else color="primary" text-color="white">{{item.Cadena}}</v-chip>
    </v-flex>
    <v-flex xs4 style="margin:1em;">
      <v-badge color="teal">
       <span slot="badge">{{item.Temporadas}}</span><span>Temporadas</span>
      </v-badge>
    </v-flex>
  </v-layout>
  <v-divider></v-divider>
  <v-layout style="margin:1em;">
    <v-flex xs12><span>{{item.Descripcion}}</span></v-flex>
  </v-layout>
</v-container>
```

v-chip -> Aparencia de botón

v-badge -> Apariencia burbuja

Barra divisora

Descripción

❹
```
<v-card>
  <v-img :src="item.Portada" height="200px">=</v-img>

  <v-container>=</v-container>
  <v-card-actions>
    <v-spacer></v-spacer>
    <v-btn icon> <v-icon>favorite</v-icon></v-btn>
  </v-card-actions>
</v-card>
```

Acciones de la tarjeta

Dejar Espacio

❺

Vuex, primeros pasos

IMPORTANTE

Cuando un componente representa el valor de un estado, al cambiar el estado su vista se actualiza de forma reactiva automáticamente.

Instalación

Vuex ❶ es una librería utilizada por Vue para la gestión los estados de los componentes de la aplicación de forma centralizada, utilizando un patrón de estados de manera que múltiples componentes compartan un estado común.

Por ejemplo, imaginemos una página de Youtube formada por varios componentes, uno para el vídeo, otro para sus comentarios y otro para vídeos recomendados. Todos deberían compartir el mismo estado identificador del védeo.

Para instalar Vuex, si no lo hemos incluido al crear el proyecto, como hicimos en ejercicios anteriores, debemos ejecutar el siguiente comando:

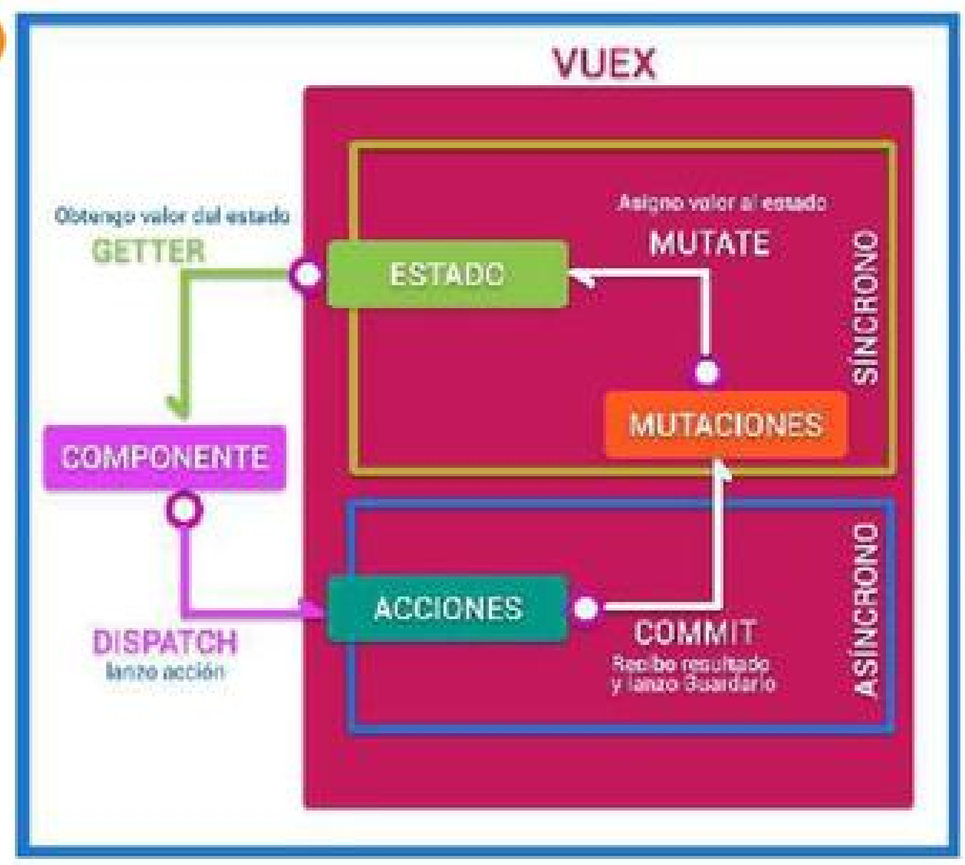
```
npm install vuex -save
```

Tras la instalación, nos habrá creado en "packages.json", la dependencia con vuex:

```
"dependencies": {… "vuex": "^3.1.0" },
```

Creamos el fichero "src/store.js", donde importamos las librerías de Vue y Vuex, ya que trabajaremos con un sistema de módulos y, mediante "Vue.use(Vuex)", lo incluimos en Vue e instanciamos una vez. Además, creamos una instancia "Vuex.Store" con las siguientes secciones ❷:

❶



❷

```
import Vue from 'vue'
import Vuex from 'vuex'

Vue.use(Vuex)

export default new Vuex.Store({
  state: { title:'Aplicación Series Favoritas'}
  , getters:{}
  , mutations:{}
  , actions:{}
  , modules:{}
})
```

- state: almacena el estado de la aplicación, puede ser compartido por diferentes componentes.
- getters: permiten obtener el valor del estado o "state" de forma síncrona.
- mutations: permiten asignar un valor al estado de forma síncrona.
- actions: permiten obtener del backend un valor de forma asíncrona, para posteriormente asignar este valor síncronamente al estado mediante una mutación.
- modules: permiten escalar nuestra aplicación, dividiendo en módulos el "store".

En la sección de estados hemos declarado un nuevo estado, la variable "title", para asignar un título a la aplicación.

Ahora, en el fichero "src/main.js" importamos el "store" que acabamos de crear:

```
import store from '@/store'
```

Obteniendo la instancia del objeto store, exportado por dicho "js", e incluimos en la instancia de Vue de la aplicación el store que acabamos de crear ❸:

```
new Vue({  store, …
```

Si ahora nos vamos a "Home.vue" e incluimos dentro de la plantilla la siguiente variable de estado de Vuex:

```
<h1>{{$store.state.title}}</h1>
```

Observaremos este valor en la aplicación ❹. "$store" es la forma de acceder directamente a una variable de estado global desde plantilla, en este caso la acompaña la sección "state" del "store" definida anteriormente, donde tenemos acceso a estas variables de estado directamente.

❸
```
new Vue({
  data(){
    return {
      loggedIn:true
    }
  },
  router,
  store,
  render: h => h(App)
}).$mount('#app')
```

❹

Vuex – Getters

IMPORTANTE

Podemos acceder a los getters sin "mapGetters", definiendo una función en las propiedades computadas, retornando el estado mediante this.$store.getters.ESTADO.

Getters

Para acceder a estados de los componentes en vuex podemos utilizar "getters". Obtengamos el estado de sesión mediante los getters, en lugar de obtenerlo a nivel de variable "loggedIn" del "data" de la aplicación.

Definimos en "store.js" una nueva variable de estado "loggedIn", para conocer el estado de sesión del usuario:

```
state: { title: 'Aplicación Series Favoritas', loggedIn: true }
```

Para obtener el valor de estas variables del estado de Vuex, usaremos los "getters", se asemejan a las propiedades computadas pero para variables de estado del "store". El número de estos puede crecer, para no correr el riesgo de definir un mismo nombre para distintos "getters" definamos el fichero "src/keyNames.js", donde almacenaremos en constantes los nombres que identificaran a estos.

Definimos en "keyNames.js" la constante "GET_USER_LOGGED", clave del "getter" que obtiene el estado "loggedIn":

```
export const GET_USER_LOGGED = 'global_USER_LOGGED';
```

El valor de esta constante al identificar un "getter" de un estado global en el "store", usamos el prefijo "global_", para diferenciarla de otras claves en otros módulos.

```
1 import Vue from 'vue'
  import Vuex from 'vuex'

  Vue.use(Vuex)
                                  Importamos los getters globales
  //Getters Globales
  import getters from '@/store/global/getters.js'

  export default new Vuex.Store({ Estado global para saber estado sesión
    state: { title:'Aplicación Series Favoritas',loggedIn:true}
    ,getters  Inyectamos los getters en la instancia del Store
    , mutations:{}
    , actions:{}
    , modules:{}
  })
```

Creamos el fichero "src/store/global/getters.js", donde definimos los "getters" globales para la aplicación. Importamos el fichero de claves "keyNames.js" y definiremos un "getter", función flecha que recibe inyectado el estado, retornando el valor de la variable de estado buscada:

```
import * as keyNames from '@/keyNames.js'

export default { [keyNames.GET_USER_LOGGED]: state=>{ return state.loggedIn;}   }
```

Importamos el fichero "getters.js" en "store.js", para asignarselo a la instancia del store "Vuex. Store" ❶.

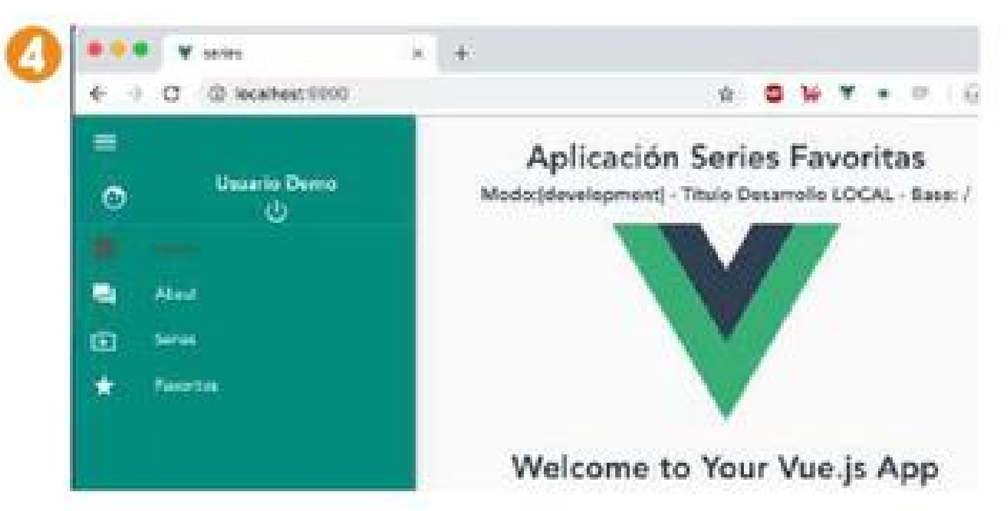
Obtengamos el estado de esta variable en el componente del menú lateral "components/MenuLateral.vue". Lo primero es utilizar el helper "mapGetters" para convertir los getters a propiedades computadas. De modo que importamos de vuex este helper, así como "keyNames.js". Finalmente, en la sección de propiedades computadas, mapeamos la variable del estado "loggedIn" mediante "...mapGetters". Este "helper" puede recibir un array de nombres de getters o un diccionario clave/valor, siendo la clave el nombre de variable que asignamos al getter y el valor el nombre del getter al que referenciamos ❷.

Si ahora en el template de este componente reemplazamos el antiguo acceso a la variable "loggedIn" del data de la aplicación "v-if='$router.app._data.loggedIn'", por esta otra "v-if='loggedIn'" estaremos accediendo a la variable de estado "loggedIn" de vuex en su lugar, mediante el getter que hemos definido ❸, de modo que debería aparecer el menú desplegado al obtener este estado de sesión a true ❹.

❷

```
<script>          mapGetters-> mapea los getters a propiedades computadas
import { mapGetters } from 'vuex'
import * as keyNames from '@/keyNames.js'

export default {
  name: 'MenuLateral'
  ,data() { return {=
  ,computed:{                                  Nombre de la clave del Getter
    ...mapGetters({ loggedIn: keyNames.GET_USER_LOGGED })
  }  Nombre Variable a usar en este componente, que obtiene el valor del estado
}
</script>
```

❹



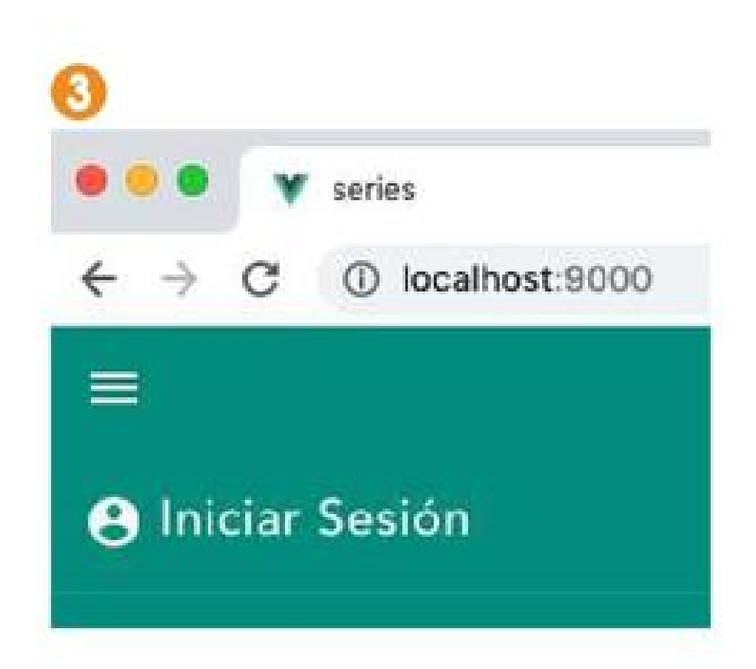
❸

```
<template>                                                    MenuLateral.vue
  <v-navigation-drawer app permanent absolute class="teal elevation-5" dark
                                v-model="drawer" :mini-variant="menuCollapse">
    <v-list class="pa-0">=
    <v-toolbar flat class="transparent" color="teal" dark>
Antes -> <!--<v-list class="pa-0" v-if="$router.app._data.loggedIn">-->
Ahora -> <v-list class="pa-0" v-if="loggedIn">=
      <v-toolbar-title v-else @click="doLogin(true)" style="cursor:pointer">=
    </v-toolbar>
    <v-divider></v-divider>
Antes -><!--<v-list class="pt-0" dense v-if="$router.app._data.loggedIn">-->
Ahora -><v-list class="pt-0" dense v-if="loggedIn">=
  </v-navigation-drawer>
</template>
```

Vuex – Mutaciones

IMPORTANTE

No debemos olvidar que no podemos asignar directamente un valor a un estado de esta forma "this.estado=valor", sino que debemos realizarlo mediante una mutación.



Mutations

Los estados en vuex no pueden modificarse directamente, requieren usar mutaciones, que no es otra cosa que una asignación o "set" de valor para un estado. Los "getters" son la opción inversa, es decir, obtienen el valor del estado.

Tanto los "getters" como las mutaciones trabajan síncronamente, de modo que, al realizar una mutación, cualquier componente que esté observando el valor del estado que mutamos se actualizaría automáticamente, siguiendo los principios de reactividad de Vue.

En el ejercicio anterior obtuvimos el estado de la sesión del usuario mediante los "getters", empleemos ahora las mutaciones para cambiar el valor al estado de sesión, es decir el botón cierre de sesión del componente menú lateral.

Definimos una nueva clave en el fichero "keyNames.js" para esta mutación:

```
export const MUTATE_USER_LOGGED = 'global_MUTATE_USER_LOGGED';
```

Creamos un nuevo fichero "src/store/global/mutations.js", donde definiremos todas las mutaciones globales de la aplicación. Importamos el fichero de constantes y definimos una nueva mutación "MUTATE_USER_LOGGED", con una función compuesta por dos parámetros, el primero inyecta el estado del store de la aplicación y el segundo es el valor que asignaremos a un estado.

```
1  import Vue from 'vue'
   import Vuex from 'vuex'

   Vue.use(Vuex)

   //Getters Globales
   import getters from '@/store/global/getters.js'
   //Mutaciones Globales                Importamos las mutaciones globales
   import mutations from '@/store/global/mutations.js'

   export default new Vuex.Store({
     state: { title:'Aplicación Series Favoritas',loggedIn:true}
     , getters, mutations   Asignamos las mutaciones,
     , actions: {}            a la instancia del Store de Vuex
     , modules:{}
   })
```

```
import * as keyNames from '@/keyNames.js'
export default { [keyNames.MUTATE_USER_LOGGED]: (state,value)=>{ state.
loggedIn=value; } }
```

Importamos el fichero de mutaciones al "store.js", de modo que podamos asignárselo a la instancia de Vuex.Store ❶.

Ahora, en el componente "MenuLateral.vue" importamos el helper "mapMutations" de vuex, que utilizaríamos para mapear la mutación a método en el componente:

```
import { mapGetters, mapMutations } from 'vuex'
```

Definimos este helper en la sección de métodos del componente "MenuLateral.vue", nombrando al mapeador de la mutación como "doLogin":

```
,methods:{ ...mapMutations({ doLogin: keyNames.MUTATE_USER_LOGGED }) }
```

Si no quisiéramos utilizar el helper "mapMutations", la equivalencia a lo anterior sería:

```
,methods:{ doLogin (value) { this.$store.commit(keyNames.MUTATE_USER_
LOGGED,value); } }
```

Es decir, haciendo uso del "commit" del "store" de vuex para asignar un valor a un estado.

Finalmente reemplazamos en la plantilla de este componente el evento clic, que accedía anteriormente a la variable del "data" de la aplicación "@click='$router.app._data.loggedIn=false'", por este otro "@click='doLogin(false)'", que utiliza la mutación "doLogin" ❷. De modo que si en la aplicación cerramos la sesión, se oculta el menú, al haber modificado el valor del estado "loggedIn" del store de vuex ❸.

❷

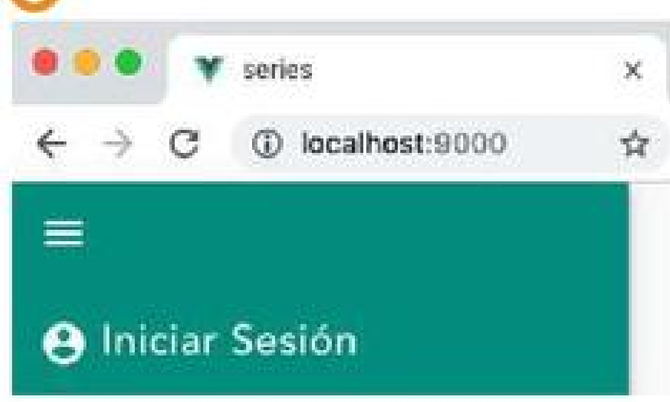
```
<template>
  <v-navigation-drawer app permanent absolute class="teal elevation-5" dark
                       v-model="drawer" :mini-variant="menuCollapse">
    <v-list class="pa-0">
    <v-toolbar flat class="transparent" color="teal" dark>
      <v-list class="pa-0" v-if="loggedIn">
        <v-list-tile>
          <v-list-tile-avatar>
          <v-list-tile-content style="align-items: center;">
            <b>Usuario Demo</b>
            <!--<v-icon @click="$router.app._data.loggedIn=false"
Antes ->          title="Cerrar Sessión">power_settings_new</v-icon>-->
Ahora ->    <v-icon @click="doLogin(false)"
                  title="Cerrar Sessión">power_settings_new</v-icon>
          </v-list-tile-content>
        </v-list-tile></v-list>
Antes -> <!--<v-toolbar-title v-else @click="$router.app._data.loggedIn=true" style="cursor:pointer">
Ahora -> <v-toolbar-title v-else @click="doLogin(true)" style="cursor:pointer">
      <v-divider></v-divider>
      <v-list class="pt-0" dense v-if="loggedIn">
        <v-list-tile v-for="ruta in rutas" :key="ruta.title" :to="ruta.to">
  </v-navigation-drawer>
</template>
```

Vuex – Acciones

IMPORTANTE

Una acción puede disponer en el primer parámetro de { getters, commit, dispatch }, en caso de requerir obtener o asignar el valor de un estado o encadenar otra acción, respectivamente.

3



Actions

Cuando necesitamos invocar al servidor para obtener el valor a asignar al estado, al no conocer el tiempo que demorará la obtención de una respuesta del servidor, no podemos realizar una mutación, porque las mutaciones son síncronas e instantáneas. Es por esta razón que vuex nos proporciona las acciones.

Gracias a las acciones podemos invocar al servidor, de tal forma que cuando recibamos la respuesta dentro de la acción, podremos utilizar una mutación mediante "commit", al disponer del valor para asignar al estado.

Simulemos una acción asíncrona que realice un cierre de sesión, simulando que la respuesta se demora 5 segundos.

Definimos una nueva clave en el fichero "keyNames.js" para esta acción:

```
export const ACTION_DO_LOGIN = 'global_ACTION_DO_
LOGIN';
```

Creamos un nuevo fichero "src/store/global/actions.js" donde definimos las acciones globales. Definimos una nueva acción "ACTION_DO_LOGIN" con una función compuesta por dos parámetros, el primero importa el "commit" de vuex y el segundo, parámetros utilizados en la petición asíncrona, para obtener una respuesta.

```
import * as keyNames from '@/keyNames.js'

export default {  [keyNames.ACTION_DO_LOGIN]: ( {
commit }, parameters ) => {

   setTimeout(()=>{  commit(  keyNames.MUTATE_USER_
```

1

```
import Vue from 'vue'
import Vuex from 'vuex'

Vue.use(Vuex)

//Getters Globales
import getters from '@/store/global/getters.js'
//Mutaciones Globales
import mutations from '@/store/global/mutations.js'
//Acciones Globales
import actions from '@/store/global/actions.js'

export default new Vuex.Store({
  state: { title:'Aplicación Series Favoritas',loggedIn:true}
  , getters, mutations, actions
  ,modules:{ }
})
```

Importamos las acciones globales

Asignamos las acciones a la instancia del Store de Vuex

```
LOGGED, parameters); }, 5000 );      } }
```

Importamos el fichero de acciones al "store.js", para asignárselo a la instancia de Vuex.Store ❶.

Ahora en el componente "MenuLateral.vue", importamos el helper "mapActions" de vuex, para mapear la acción, a método del componente:

```
import { mapGetters, mapMutations, mapActions } from 'vuex'
```

Definimos este helper en la sección de métodos del componente "MenuLateral.vue", nombrando al mapeador de la acción "doAsyncLogin":

```
,methods:{ ...mapActions({ doAsyncLogin: keyNames.ACTION_DO_LOGIN })  }
```

Si no quisiéramos utilizar el helper "mapActions", la equivalencia a lo anterior sería:

```
,methods:{  doAsyncLogin(value)  {  this.$store.dispatch(keyNames.ACTION_DO_LOGIN,
value); }  }
```

Usando el "dispatch" del store de vuex, para invocar al servidor y obtener un valor para asignárselo a un estado.

Finalmente reemplazamos en la plantilla de este componente el evento clic, de cierre de sesión síncrono "@click='doLogin(false)'", por "@click='doAsyncLogin(false)'", que utiliza la acción asíncrona "'doAsyncLogin" ❷.

Si en la aplicación cerramos la sesión, se ocultará el menú a los 5 segundos, al haber simulado una demora mediante "setTimeout" de Javascript. Tras recibir la respuesta, se realizará una mutación mediante "commit", que modificará el valor del estado "loggedIn" del store de vuex ❸.

❷

```
<template>
  <v-navigation-drawer app permanent absolute class="teal elevation-5" dark
                       v-model="drawer" :mini-variant="menuCollapse">
    <v-list class="pa-0">
    <v-toolbar flat class="transparent" color="teal" dark>
      <v-list class="pa-0" v-if="loggedIn">
        <v-list-tile>
          <v-list-tile-avatar>
          <v-list-tile-content style="align-items: center;">
            <b>Usuario Demo</b>
Antes ->    <!--<v-icon @click="doLogin(false)"
Ahora ->    <v-icon @click="doAsyncLogin(false)"
                title="Cerrar Sesión">power_settings_new</v-icon>
          </v-list-tile-content>
        </v-list-tile></v-list>
        <v-toolbar-title v-else @click="doLogin(true)" style="cursor:pointer">
      <v-divider></v-divider>
      <v-list class="pt-0" dense v-if="loggedIn">
        <v-list-tile v-for="ruta in rutas" :key="ruta.title" :to="ruta.to">
  </v-navigation-drawer>
</template>
```

Vuex – Módulos

IMPORTANTE

Un módulo organiza las responsabilidades de cada componente, debemos incluir en él solo lo específico del componente que lo utilice. Puede incluir otros módulos.

Módulos

Sabemos trabajar con estados, acceder, asignarles valor y lanzar una acción asíncrona. Sin embargo, a medida que nuestra aplicación creciera, definiríamos toda esta lógica en un único fichero descomunal "store.js".

Para lidiar con esta problemática, Vuex nos permite crear módulos para segmentar en ficheros los estados, "getters", mutaciones y acciones. Definiremos lo transversal a toda la aplicación, como hasta ahora, en el store, no obstante, todo lo específico, como por ejemplo en series, crearemos un módulo que encapsule estas secciones.

Lo primero será definir las claves que vamos a utilizar para el módulo series en el fichero "keyNames.js":

```
export const ALL_SERIES = 'series_ALL_SERIES';
export const MUTATE_ALL_SERIES = 'series_MUTATE_ALL_SERIES';
export const ACTION_GET_ALL_SERIES = 'series_ACTION_GET_ALL_SERIES';
```

Creamos el directorio donde almacenaremos cada módulo, en este caso el de series en "src/store/modules/series.js". Agregamos cada sección, su estado será una variable "series" para almacenar un listado de series, los "getters", obtendremos el valor de este, las "mutations" le asignaremos valor y tendremos una acción para simular una llamada HTTP y obtener un listado de series

```
1 import * as keyNames from '@/keyNames.js'

  import createLogger from 'vuex/dist/logger'

  const state = { series:[] };  Estado que almacena un array de series

  const getters = { [keyNames.ALL_SERIES]: state => { return state.series; } }; Obtener Series del estado

  const mutations = { [keyNames.MUTATE_ALL_SERIES]: (state,value) => { state.series = value; } };
                                                                  Asignar series al Estado
  const actions = { [keyNames.ACTION_GET_ALL_SERIES]: ({commit},parameters) => {

      setTimeout(()=>{ commit(keyNames.MUTATE_ALL_SERIES,
        [{ "Anyo":          2019,
          "Cadena":        "Netflix",
          "Descripcion":   "Descripción ALL_SERIES",
          "Id":            99,
          "Temporadas":    1,
          "Titulo":        "Título ALL_SERIES"
        }] ); }, 5000 );
    } };

  export default { state, getters, mutations, actions }  Exportamos el módulo con cada una de la secciones
```

Una vez obtenida "respuesta" del servidor
Asignaremos el valor obtenido, al estado series.
Tras 5 segundos... Simulando una peticion HTTP

"mockeada" que, tras 5 segundos, asignará valor al estado con una mutación. Finalmente exportaremos las secciones del módulo ❶.

Una vez creado el módulo, lo importamos en el "store.js" en la sección de módulos o "modules" ❷.

Ya tenemos el módulo cargado en el "store" y accesible en la aplicación, de modo que vamos al componente "Series.vue" y borramos del "data" las filas "rows" que habíamos mockeado, ya que ahora utilizaremos el "getter", responsable de obtener la lista de series:

```
,computed:{ ...mapGetters({ rows: keyNames.ALL_SERIES }) }
```

Por último, en este mismo componente añadiremos el evento "beforeCreate" que, según el ciclo de vida de una instancia Vue, ocurre antes de que el componente se haya creado. Lanzará la petición mediante una acción, que internamente hemos implementado que tras 5 segundos asigne una serie al listado de series del estado ❸.

```
,beforeCreate(){ this.$store.dispatch(keyNames.ACTION_GET_ALL_SERIES); }
```

Si ahora entramos en la aplicación a la ruta de "/series" inicialmente aparece la tabla vacía, pasados 5 segundos, la tabla aparece rellena con la serie obtenida tras la petición lanzada por la acción ❹. Aquí está actuando la reactividad.

❷
```
import moduleSeries from '@/store/modules/series.js'
export default new Vuex.Store({
  state: { title:'Aplicación Series Favoritas',loggedIn:true}
  , getters, mutations, actions
  ,modules:{ moduleSeries }
})
```
Importamos el módulo de series creado

Módulo Series incorporado al Store global

❹

❸
```
<script>
import Tabla from '@/components/Tabla.vue';
import { mapGetters } from 'vuex'
import * as keyNames from '@/keyNames.js'

export default {
  name: 'series'
  ,computed:{ ...mapGetters({ rows: keyNames.ALL_SERIES }) }
  ,data(){
    return { cadena: this.propCadena, hayCambios: false
           , message: null, snackbar: false, timeout: 3000
           , columns:[
          } //,rows:[]
  }
  ,beforeCreate(){ this.$store.dispatch(keyNames.ACTION_GET_ALL_SERIES); }
  ,methods:{
  ,components:{
}
</script>
```
Importamos los getters y las claves

Mapeamos el Getter de obtención del listado de series del estado

Quitamos del data, las filas que teníamos mockeadas

Antes de que se cargue el componente, lanzamos petición al servidor, para obtener el listado de series

Vuex – Plugin Logger

Vuex Logger

La sección "plugins" de vuex permite interceptar mutaciones en los estados, lo cual puede ser interesante para logar estos cambios de estados. La mejor forma de crear este log es empleando el "logger" proporcionado por vuex.

Interceptemos los cambios de estado del módulo "series.js", importando el logger de vuex en el "store.js":

```
import vuexLogger from 'vuex/dist/logger'
```

Implementemos este logger en el "store.js", creando una constante con la instancia del logger con las secciones:

- collapsed: establece si el log aparece colapsado o expandido en consola.
- filter: filtramos las mutaciones que queramos interceptar únicamente.
- mutationTransformer: retornamos el tipo de mutación interceptada y su valor a mutar.
- transformer: retornamos tanto el valor del estado previo, como el posterior a la mutación.
- logger: por donde se van a mostrar los logs de la traza, por defecto la consola del navegador.

Indicamos que el log aparezca colapsado y que utilice la consola del navegador para visualizar el log:

```
const logger = vuexLogger({  collapsed: true, logger: console …});
```

```
1  import vuexLogger from 'vuex/dist/logger'    Librería de vuex para logs

   const logger = vuexLogger({ collapsed: true, logger: console,
     filter (mutation, stateBefore, stateAfter) {
       return process.env.NODE_ENV === 'development'
               && ['series'].some((e)=>{return mutation.type.startsWith(e)})
     },
     mutationTransformer (mutation) {
       const { type, payload } = mutation
       return Array.isArray(payload)?
             `${type} Mutando: ${payload.length} elementos` :`${type}  ${payload}`
     }
     ,transformer (state) { return state }
   })

   export default new Vuex.Store({
     state: { title:'Aplicación Series Favoritas',loggedIn:true}
     , getters, mutations, actions
     ,modules:{ moduleSeries }
     ,plugins:[logger]
   })
```

Añadimos al "vuexLogger", en la sección de filtrado, el método "filter" con los parámetros de mutación interceptada, estado anterior y posterior a la mutación. Permitiendo log en caso de estar en modo desarrollo y si la mutación interceptada, pertenece a alguna del módulo de "series.js".

```
filter(mutation, stateBefore, stateAfter) { return process.env.NODE_ENV ===
'development'
    && ['series'].some( (e)=>{ return mutation.type.startsWith(e) } ) },
```

En el método "mutationTransformer" recibimos la mutación, retornando en el log un texto con el número de elementos mutados en caso de ser un array y, en caso contrario, retornando el valor.

```
mutationTransformer (mutation) {   const { type, payload } = mutation
      return Array.isArray(payload)? `${type} Mutando: ${payload.length} elemen-
tos` :`${type}  ${payload}` },
```

Finalmente, en el método "transformer" recibimos por parámetro el estado previo y posterior mutado, que podemos transformar y retornar en el log; nosotros únicamente retornaremos el estado sin transformar.

```
transformer (state) { return state },
```

Únicamente nos quedaría incluir este plugin en la instancia "Vuex.Store" ❶.

```
export default new Vuex.Store({… , plugins: [logger] })
```

Si abrimos la aplicación en la ruta de "/series" veremos en el log de la consola del navegador Chrome (pulsando F12) la traza con el valor previo y posterior a la mutación, así como el tipo de mutación invocada ❷.

```
❷ ▼ mutation series_MUTATE_ALL_SERIES @ 18:27:47.413
    prev state ▼{title: "Aplicación Series Favoritas", loggedIn: true, moduleSeries: {…}}
                 loggedIn: true
               ▼moduleSeries:
                 ▶series: []
                 ▶__proto__: Object
                 title: "Aplicación Series Favoritas"
               ▶__proto__: Object
    mutation series_MUTATE_ALL_SERIES Mutando: 1 elementos
    next state ▼{title: "Aplicación Series Favoritas", loggedIn: true, moduleSeries: {…}}
                 loggedIn: true
               ▼moduleSeries:
                 ▼series: Array(1)
                   ▶0: {Anyo: 2019, Cadena: "Netflix", Descripcion: "Descripción ALL_SERIES", Id: 99, Temporadas: 1, …}
                    length: 1
                   ▶__proto__: Array(0)
                 ▶__proto__: Object
                 title: "Aplicación Series Favoritas"
               ▶__proto__: Object
```

Vuex – Persistencia estados al refrescar

IMPORTANTE

No almacenar información privada en los tipos de almacenamiento soportados por "vuex-persist"; localStorage, sessionStorage o cookies, ya que serán visibles.

Vuex-Persist

Para persistir los valores de los estados de vuex al refrescar la página podemos utilizar el plugin "vuex-persist".

Instalamos el plugin de persistencia de vuex "vuex-persist":

```
npm install -save vuex-persist
```

Esto añade a las dependencias del "packages.json" la librería instalada: "vuex-persist": "^2.0.0".

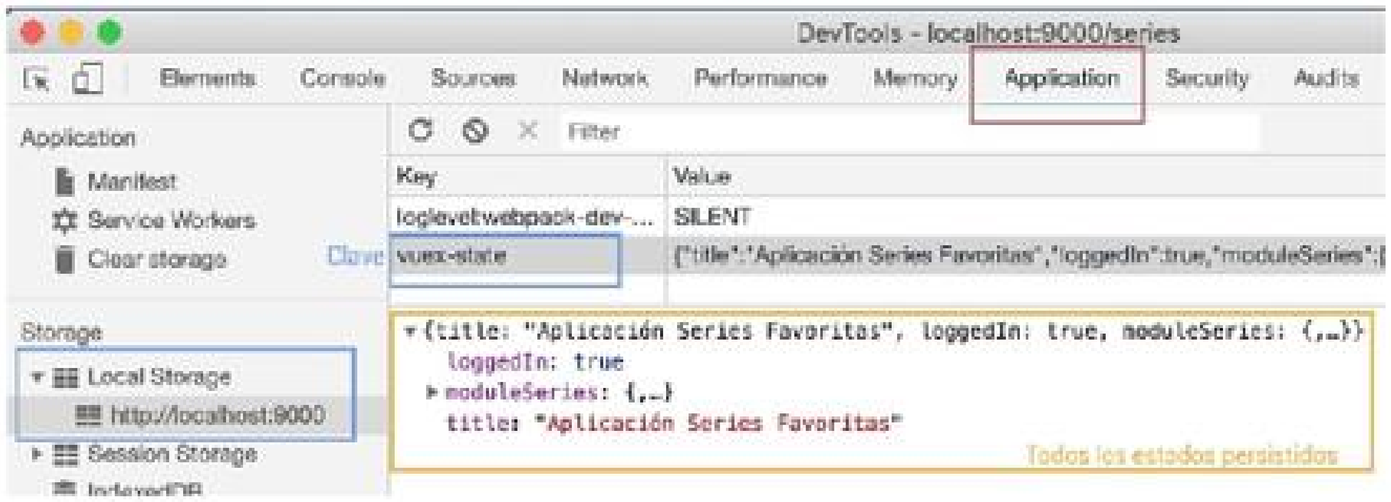
Ahora, en el "store.js" importamos esta librería de persistencia "vuex-persist"

```
import persistence from 'vuex-persist';
```

Implementemos la persistencia local en "store.js", en una constante con la instancia de "vuex-persist" y secciones:

- key: nombre de la clave del objeto a persistir.
- storage: tipo de almacenamiento, localStorage, sessionStorage.
- reducer: indicamos qué módulos queremos almacenar, porque quizás no queramos almacenar todos.
- filter: indicamos si alguna mutación que se repite muy frecuente en milisegundos no debe actualizar el objeto persistido, debido a que penaliza el rendimiento actualizar de forma muy continuada el almacén local.

1



2

```
import persistence from 'vuex-persist';

const vuexSaveLocal = new persistence({
  key: 'vuex-state'
  ,storage: window.localStorage
  ,reducer: state => ({ moduleSeries:state.moduleSeries })
  ,filter: mutation => (['series'].some((e)=>{return mutation.type.startsWith(e)}))
})
```

Instanciamos el plugin de persistencia, indicando la clave del almacén será "vuex-state", así como el tipo de almacenamiento será mediante localStorage de HTML5. Podríamos haber utilizado de igual manera sessionStorage.

```
const vuexSaveLocal = new persistence({ key: 'vuex-state', storage: window.localS-
torage })
```

Agregamos a la sección de plugins de la instancia de Vuex.Store esta instancia de "vuex-persist":

```
export default new Vuex.Store( {…  , plugins: [ logger, vuexSaveLocal.plugin] } )
```

Si ahora abrimos la aplicación y pulsamos en Chrome la tecla F12 accedemos a la pestaña "Aplicación" ❶, donde veremos que en la sección del localStorage tenemos almacenados con la clave "vuex-state" todos los estados.

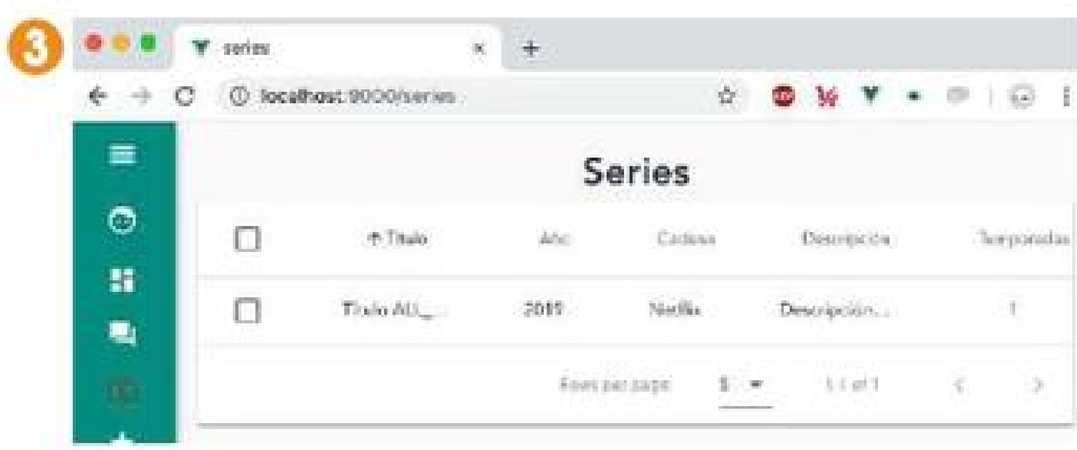
Hagamos ahora que únicamente se almacene el módulo de series. Para ello, en la instancia del objeto vuexSaveLocal añadiremos la sección "reducer", indicando aquellos módulos a persistir, en este caso el de series.

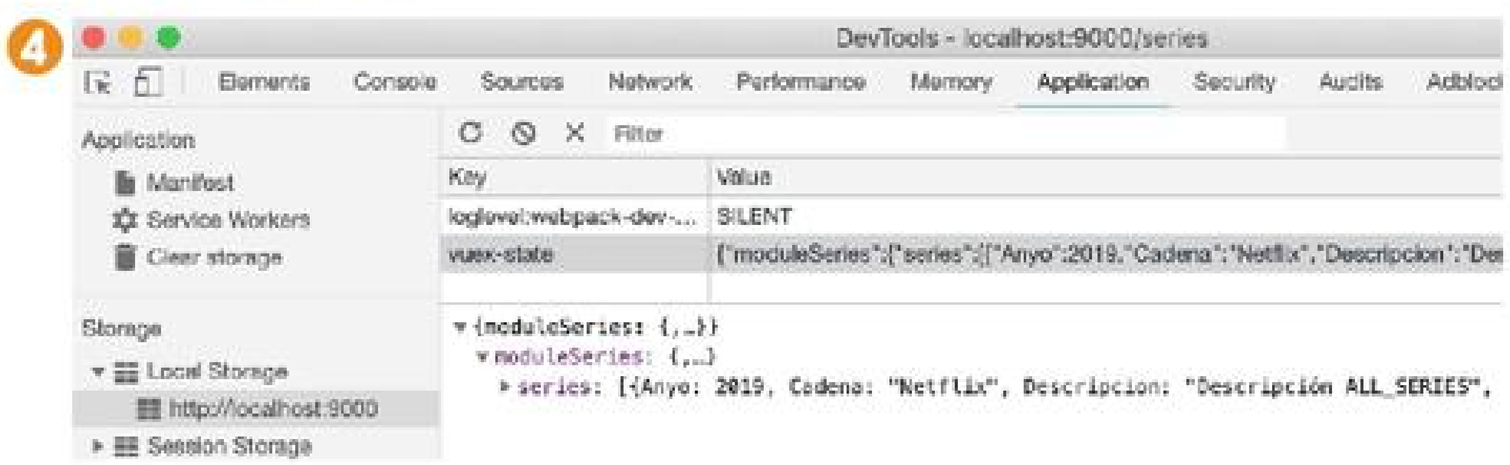
```
… ,reducer: state => ({ moduleSeries: state.moduleSeries })
```

Además, hagamos que únicamente actualicen el modelo persistido, aquellas mutaciones que pertenezcan al módulo series, utilizando para ello la sección "filter" ❷.

```
,filter: mutation => (['series'].some((e)=>{ return mutation.type.startsWith(e) }))
```

Si ahora actualizamos la página en la ruta "/series", veremos que la tabla aparece inicialmente cargada con la fila persistida ❸ y, si visualizamos la pestaña de "Aplicación" del Chrome, únicamente persistimos el módulo series ❹.





Vuex – Persistencia en Cookies

Cookies

Otro método de almacenamiento persistente en navegador son las cookies, surgieron antes que local y sessionStorage de HTML5. Difieren en que su capacidad de almacenamiento es mucho menor, sobre los 5KB respecto a los 5MB de HTML5, además las cookies son enviadas en cada solicitud HTTP, mientras que el de HTML5 no.

No hay uno mejor que otro, tienen funcionalidades combinables. Se suele almacenar en cookies información de sesión del usuario y tiempo de caducidad, útil al enviarse en cada solicitud al servidor al poder controlarlo.

Vamos a utilizar cookies para persistir el estado del mismo módulo de antes, debido a no disponer de un Token de la sesión de usuario, ni tiempo de vida.

Lo primero será instalar el paquete de "js-cookie", para trabajar con cookies:

```
npm install --save js-cookie
```

Esto añade a las dependencias de "packages.json" el paquete "js-cookie": "^2.2.0".

Estamos listos para trabajar con cookies, vamos al "store.js" e importamos este paquete:

```
import Cookies from 'js-cookie'
```

Posteriormente implementamos la persistencia en cookies en la instancia de "vuex-persist" con las secciones:

- restoreState: obtiene del almacenamiento, el estado, mediante una clave.
- saveState: almacena el estado con una clave y le asigna una caducidad.

Asignamos a la instancia de "vuex-persist" el nombre de clave "vuex-state" para la cookie a almacenar:

```
const vuexSaveCookie = new persistence({  key:'vuex-state' …})
```

Mediante "restoreState" obtenemos de la librería "js-cookie" el estado almacenado en la cookie en formato JSON:

```
    ,restoreState: (key, storage) => Cookies.getJSON(key)
```

La función inversa a la anterior, "saveState", recibe por parámetros la clave de la cookie, el estado a almacenar y la instancia de almacenamiento. Almacenando en una cookie que expira en 7 días, el estado:

```
,saveState: (key, state, storage) => Cookies.set(key, state, { expires: 7})
```

Finalmente, indicamos el módulo que queremos almacenar, que será el de series:

```
   ,modules: ['moduleSeries']
```

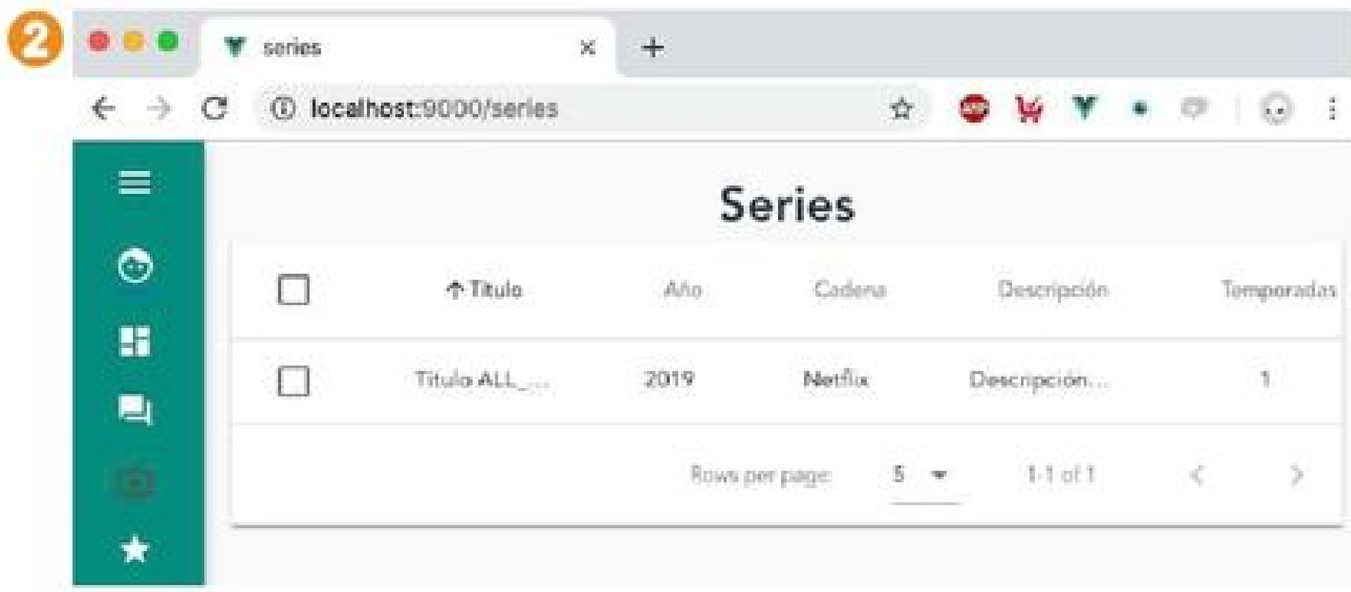
Agregamos a la sección de plugins de la instancia de Vuex.Store, la instancia de "vuex-persist" para cookies ❶:

```
    export default new Vuex.Store( {…  , ,plugins:[ logger, vuexSaveLocal.plugin,
vuexSaveCookie.plugin ] } )
```

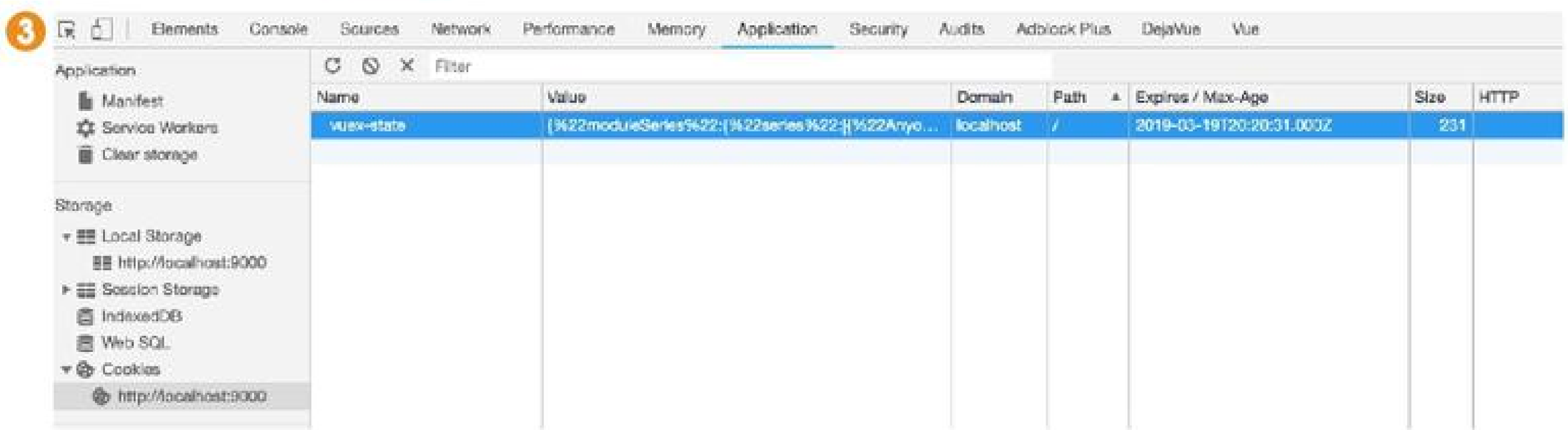
Si ahora actualizamos la página en la ruta "/series", la tabla aparece inicialmente cargada con la fila persistida ❷ y, si visualizamos la pestaña de "Aplicación" del Chrome, únicamente persistimos el módulo series en una cookie ❸.

❶
```
import Cookies from 'js-cookie'

const vuexSaveCookie = new persistence({
  key:'vuex-state'
  ,restoreState: (key, storage) => Cookies.getJSON(key)
  ,saveState: (key, state, storage) => Cookies.set(key, state, { expires: 7})
  ,modules: ['moduleSeries']
  ,filter: (mutation) => (true)
})
```

❷



❸



Vuex – Subscriptores

Subscribe

Toda acción o mutación que invoquemos en Vuex lanza un "hook" capturado por dos tipos de subscriptor del store. Puede ser útil a la hora de centralizar acciones, cada vez que ocurra el evento.

Realicemos una actividad que capture una mutación de cierre de sesión, es decir la "MUTATE_USER_LOGGED", así como su valor enviado. De esta forma capturamos el cierre de sesión antes de cerrarse.

Para ello importamos el fichero de claves y router en el "store.js". Posteriormente creamos un subscriptor para capturar mutaciones, cuyo parámetro es el store con el que podemos trabajar y asignarle subscriptor.

```
const subscriber = store => {
```

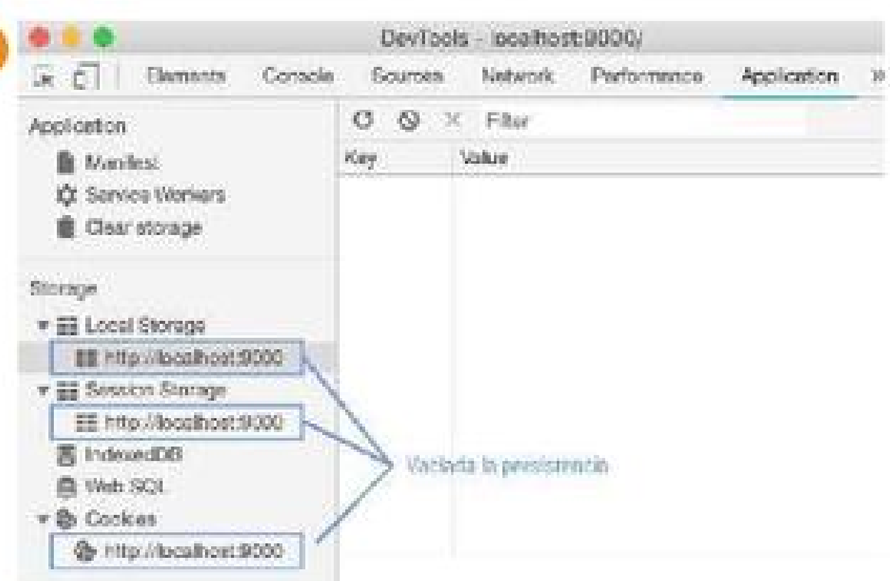
Dentro de este elemento declaramos la subscripción de toda mutación emitida mediante "store.subscribe". De tal forma que cuando capturemos la mutación "MUTATE_USER_LOGGED" y su valor sea falso, borraremos toda persistencia local que tuviéramos, es decir el localStorage, sessionStorage y las cookies, para finalmente mediante el router, redirigir a la página principal ❶:

```
store.subscribe((mutation, state) => {
   if( mutation.type == keyNames.MUTATE_USER_LOGGED && !mutation.payload)
   {  localStorage.clear();    sessionStorage.clear();    Cookies.remove('vuex-
state');
      router.push({name:'home'});  }
}) }
```

❶

```
import router from './router'
import * as keyNames from '@/keyNames.js'

const subscriber = store => { Subscribe a mutaciones del store
  store.subscribe((mutation, state) =>
  {
    if(mutation.type==keyNames.MUTATE_USER_LOGGED && !mutation.payload)
    {  Si el tipo de mutación es Login                Y su valor es FALSE, es decir Logout
      localStorage.clear();     //Limpiamos persistencia localStorage
      sessionStorage.clear(); //Limpiamos persistencia sessionStorage
      Cookies.remove('vuex-state'); //Limpiamos las cookies
      router.push({name:'home'}); //Abrimos la home, sin usuario activo
    }
  })
}
```

❷



Agregamos el subscriptor a la instancia Vuex.Store y ya estaríamos listos para capturar las mutaciones.

```
export default new Vuex.Store( {…,plugins: [ logger, vuexSaveLocal.plugin, vuexSaveCookie.plugin, subscriber ] } )
```

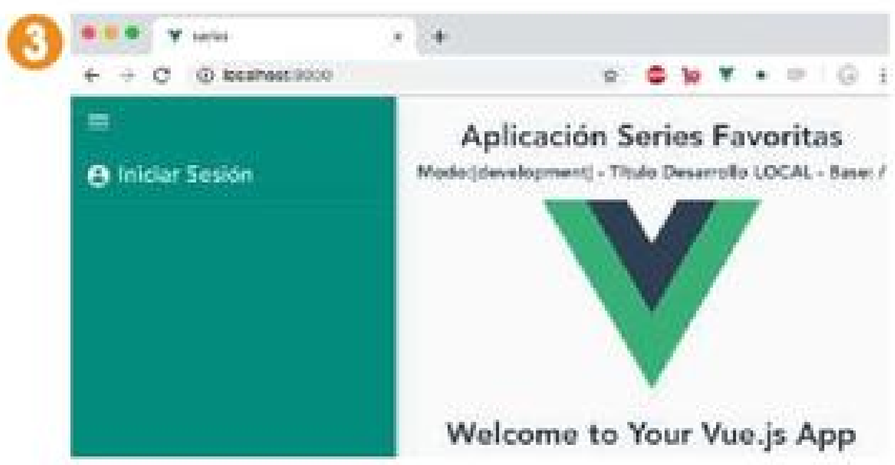
Si pulsamos cerrar sesión, capturaremos el cierre, limpiamos la persistencia local ❷ y redirigirnos al inicio ❸.

SubscribeAction

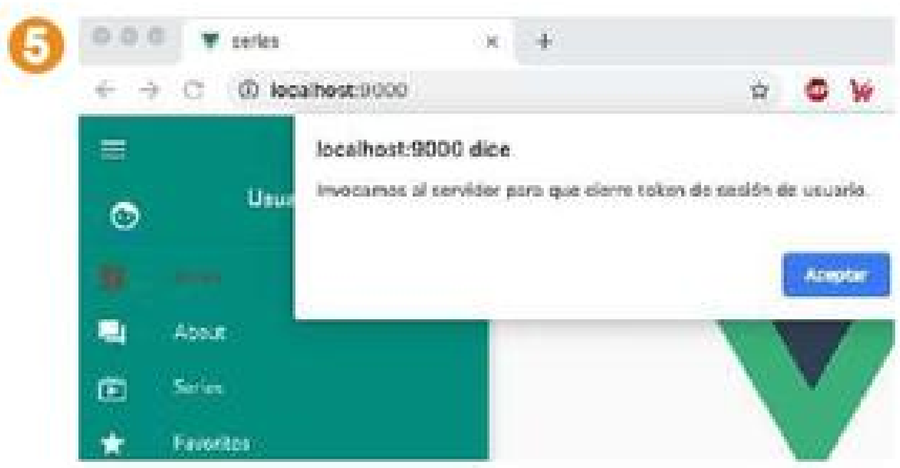
De igual manera que con las mutaciones, podemos subscribirnos a las acciones. Vamos a simular un cierre de sesión en el servidor, para ello creamos una nueva clave "ACTION_CLOSE_SESSION" en el fichero de constantes, y en las acciones globales "actions.js" definimos una nueva acción que simula la llamada de cierre de sesión en el servidor:

```
,[keyNames.ACTION_CLOSE_SESSION]: ({commit},parameters) => { alert('Sesión Cerrada en el servidor'); }
```

En la constante "subscriber" del "store.js" definamos la subscripción de las acciones mediante "store.subscribeAction", donde tenemos la opción de controlar el antes "before" y el después "after" de invocar la acción ❹. Controlaremos que, al cerrar la sesión, se invoque la acción de cierre, mostrando un aviso ❺ y, tras cerrar la sesión el servidor, mostraremos otro aviso por pantalla ❻.




```
const subscriber = store => {
  store.subscribe((mutation, state) =>{})

  store.subscribeAction({
    before:(action,state)=>{ },
    after:(action,state)=>{
      if(action.type==keyNames.ACTION_DO_LOGIN && !action.payload){
        alert('Invocamos al servidor para que cierre token de sesión
        store.dispatch(keyNames.ACTION_CLOSE_SESSION);
      }
    }
  })
}
```



Axios – Comunicación con Servidor

IMPORTANTE

Controle el caso de error "catch" en axios, así como también respuestas nulas, ya que la pantalla puede requerir una nueva acción.

Instalar Axios

Una aplicación web suele comunicarse con el servidor para intercambiar información utilizando peticiones Ajax mediante solicitudes "XMLHttpRequests". Para ello utilizamos la librería basada en JavaScript "Axios", capaz de gestionar peticiones Ajax y promesas de forma independiente.

Para probar una solicitud a un servidor remoto, hemos creado en Firebase una base de datos ❶ para consultar datos de series favoritas mediante una Api Rest que expone Firebase para consumir esta base de datos.

Instalamos Axios, ejecutando el siguiente comando en la consola:

```
npm install axios
```

Esto incluye en el fichero "packages.json" la dependencia de esta librería:

```
"dependencies": { "axios": "^0.18.0",… }
```

Axios GET

Una vez listos para trabajar con axios, agregamos en los ficheros ".env.development", ".env.development.local" y ".env.production" la ruta donde se encuentra la API de nuestro servidor, será la misma en los 3 entornos:

```
VUE_APP_API_URL=https://vue-100-ejercicios-cli-se-
ries.firebaseio.com
```

Abrimos el modulo de las series "store/modules/series.js", donde importamos la librería de axios:

```
import axios from 'axios'
```

En este mismo fichero, en la constante "actions" reemplazamos el código de la acción "ACTION_GET_ALL_SERIES", que realizaba un "setTimeOut" de 5 segundos, por una solicitud de tipo "GET" al servidor de Firebase ❷.

Esta solicitud get la realizamos a través de "axios.get(MI_URL)", indicando según el entorno la ruta de la Api donde se encuentra el recurso. En este caso, la api requiere indicar el tipo de datos de retorno, es decir JSON, indicándolo mediante ".json" al final de la URL.

```
axios.get( process.env.VUE_APP_API_URL + '/.json')
```

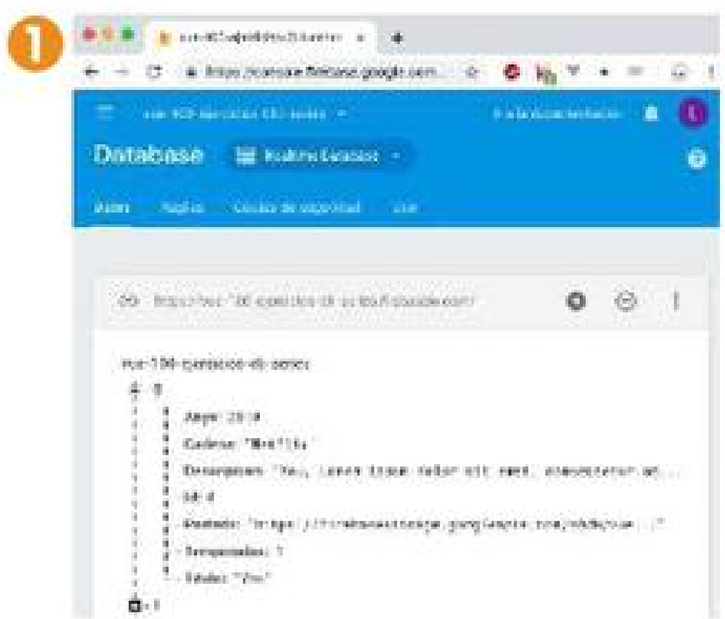
En caso de una respuesta satisfactoria entrará en el método "then", cuyo parámetro de respuesta o "response" contendrá el listado de series a visualizar en la tabla de series. Este listado se lo enviamos a la mutación de series "MUTATE_ALL_SERIES", para que asigne al estado de la tabla de series este listado.

```
.then( (response) => { if(response && response.data)
    commit(keyNames.MUTATE_ALL_SERIES,response.data); } )
```

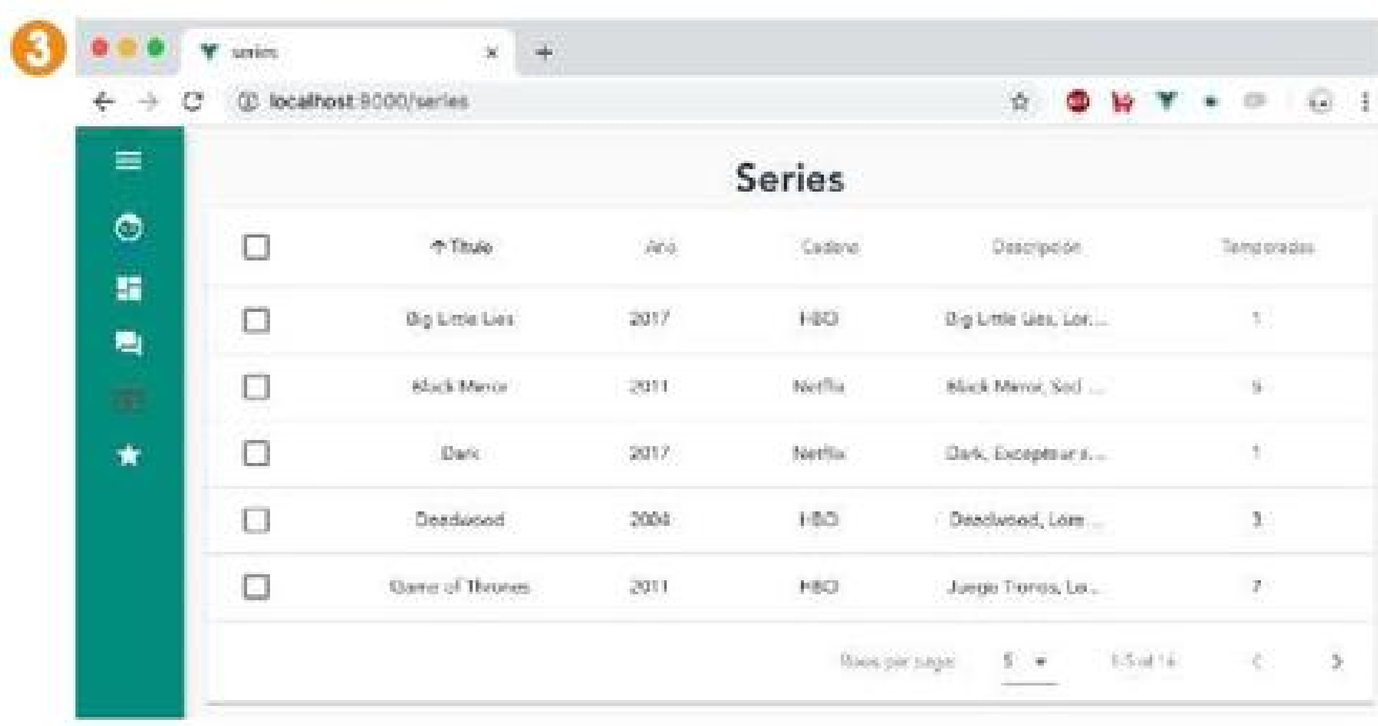
En caso de error en la solicitud, asignaremos una lista vacía a la tabla de series.

```
    catch( (error) => { alert(error.message)      commit(keyNames.MUTATE_ALL_SE-
RIES,[]) })
```

Si abrimos la ruta "/series" veremos cómo esta solicitud obtiene un listado de series ❸, paginadas en la tabla.




```
import axios from 'axios'

const actions = {
  [keyNames.ACTION_GET_ALL_SERIES]: ({commit},parameters) => {
    axios.get(process.env.VUE_APP_API_URL+'/.json'
      .then((response)=>{ if(response && response.data)
                          commit(keyNames.MUTATE_ALL_SERIES,response.data); })
      .catch((error)=>{ alert(error.message)
                        commit(keyNames.MUTATE_ALL_SERIES,[]) })
  }
};
```



Axios – Cancelar solicitud

IMPORTANTE

Cancelar una petición por parte de timeout o por acción directa del usuario es muy útil y necesario, téngalo en cuenta en sus desarrollos. El usuario lo agradecerá.

CancelToken

En ocasiones, por algún tipo de incidencia técnica, bien sea por latencia originada por la red o bien por el servidor que retorna la solicitud o por acción del usuario, queremos cancelar la escucha de respuesta por parte del servidor.

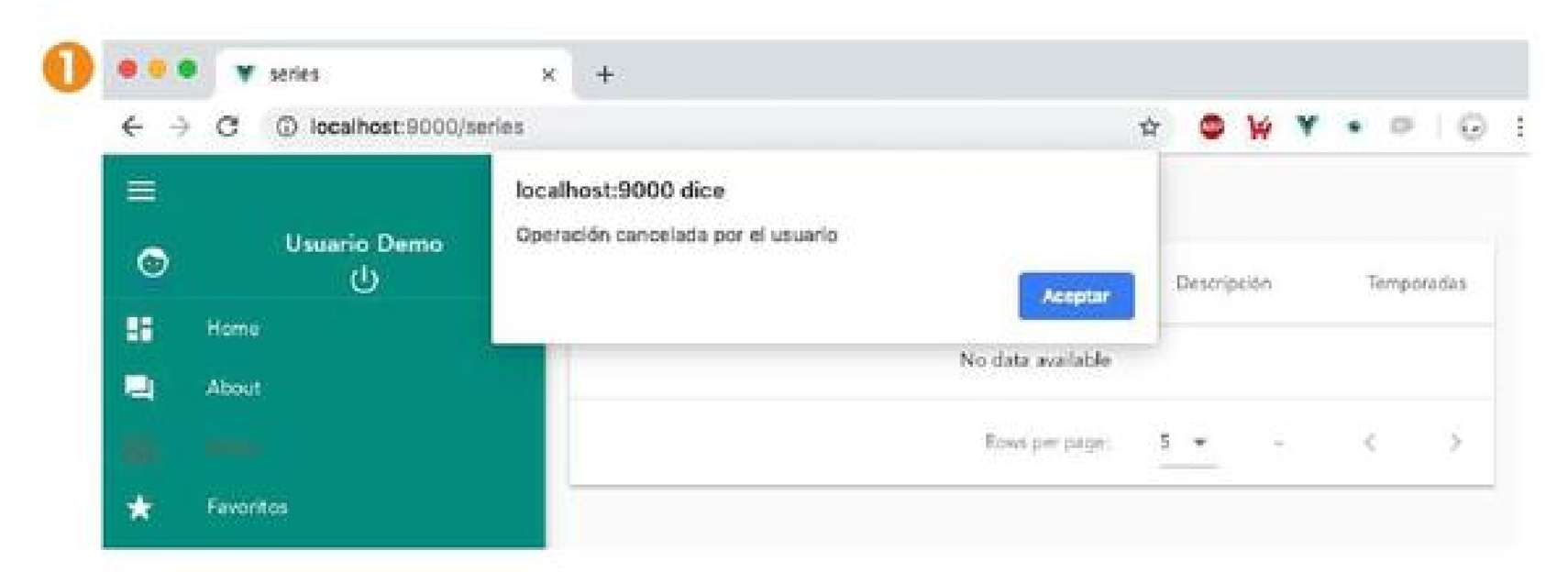
Dado que almacenábamos en cookies en anteriores ejercicios los datos del módulo series, vamos a limpiar el listado de series, antes de realizar el "dispatch" que invoca la acción de obtención de series en el componente "Series.vue":

```
   ,beforeCreate(){    this.$store.commit(  keyNames.
MUTATE_ALL_SERIES, [] )

      this.$store.dispatch(  keyNames.ACTION_GET_ALL_
SERIES) }
```

En la acción que obtiene mediante una petición GET el listado de series "ACTION_GET_ALL_SERIES", vamos a implementar que la solicitud sea cancelada a los 200 milisegundos.

Para ello, en la acción "ACTION_GET_ALL_SERIES" de "store/modules/series.js" necesitamos crear un token que identifique la petición a invocar. Axios se basa en promesas de JavaScript, por lo que disponemos de cancelación de promesas, posible en axios a través de la factoría "source()" de su Api "CancelToken":

```
[keyNames.ACTION_GET_ALL_SERIES]: ( { commit }, pa-
rameters) => {
                let cancelCall = axios.CancelToken.source();
```



Asignamos esta factoría a la constante "cancelCall" para utilizarla en el código. Ahora, en la petición get de axios, utilizamos el parámetro de configuración "cancelToken", que no es más que el token que identifica la solicitud.

```
axios.get( process.env.VUE_APP_API_URL + '/.json', { cancelToken:cancelCall.
token } )
```

Para capturar la excepción que vamos a provocar por cancelar la solicitud lo podemos controlar en el "catch" de axios, mediante el método "isCancel". Donde en nuestro caso escribiremos por consola la acción abortada.

```
.catch( (error) => { if(axios.isCancel(error) ) { console.log( 'Timeout -> ACTION_
GET_ALL_SERIES' ) }  …} )
```

Finalmente, añadimos al final de la acción tras la llamada axios un timeout para abortar la solicitud si tarda más de 200 milisegundos. Se cancelará utilizando el método "cancel()" de la factoría del token.

```
setTimeout(()=>{ cancelCall.cancel( 'Operación cancelada por el usuario' ) }, 200 );
```

Si abrimos la ruta "/series", debido a que 200 milisegundos es un tiempo de respuesta muy bajo, la solicitud será abortada y ningún dato se cargará en la tabla ❶, mostrando un mensaje de operación cancelada por el usuario.

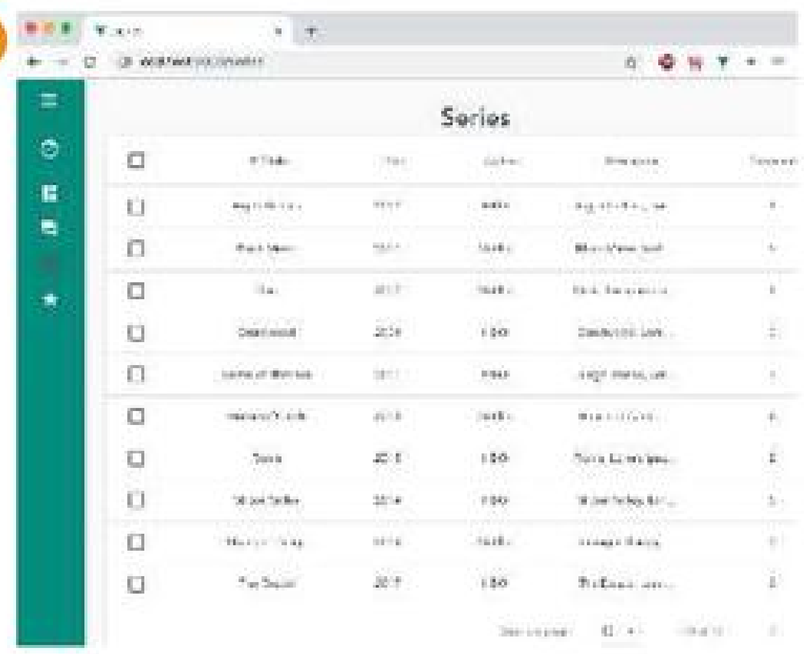
Si ahora le diéramos un tiempo mayor, por ejemplo 2000 milisegundos ❷ en lugar de 200, la solicitud no sería abortada, dado que le da tiempo a recibir una respuesta antes de 2 segundos ❸.

Hemos simulado un timeout, que sería equivalente a declarar en la configuración de axios el parámetro "timeout":

```
axios.get( process.env.VUE_APP_API_URL + '/.json', { timeout:2000 } )
```

❷
```
const actions = {
  [keyNames.ACTION_GET_ALL_SERIES]: ({commit},parameters) => {
                    Api Cancelación de token de axios
    let cancelCall = axios.CancelToken.source();   Factoría de generación y cancelación de tokens

    axios.get(process.env.VUE_APP_API_URL+'/.json',{ cancelToken:cancelCall.token })
        .then((response)=>{ if(response && response.data)
                            commit(keyNames.MUTATE_ALL_SERIES,response.data); })
        .catch((error)=>{
          if(axios.isCancel(error)){ console.log('Timeout ->ACTION_GET_ALL_SERIES')}
          alert(error.message)
          commit(keyNames.MUTATE_ALL_SERIES,[])
        })
                    Invocamos la cancelación de la solicitud
    setTimeout(()=>{cancelCall.cancel('Operación cancelada por el usuario')},2000);
  }
};
                    Parámetro opcional que recibirá mensaje de cancelación
```

❸



Axios – Interceptor de solicitudes

IMPORTANTE

Interceptar el momento previo a lanzar una "request" es una buena práctica para centralizar envío de configuración, como podría ser el Token de sesión.

Solicitudes

Axios nos permite centralizar cada solicitud enviada para poder enviar, por ejemplo, un token de acceso en cada petición o realizar alguna acción, como la que simularemos, de mostrar un spinner o imagen de carga.

Para ello añadimos un gif "loading.gif" en "src/assets/icons" para mostrar por pantalla al iniciar una petición.

Creamos en el fichero "store.js" el estado "loading", utilizado para mostrar o no la imagen de carga.

```
export default new Vuex.Store({ state: {
title:'Aplicación Series Favoritas',loggedIn:true,
loading:false }
```

Tras esto, en el fichero de claves "keyNames.js" crearemos claves para el getter y la mutación del estado "loading".

```
export const GET_LOADING = 'global_GET_LOADING';

export const MUTATE_LOADING = 'global_MUTATE_
LOADING';
```

Esto implica crear el getter de dicho estado en el fichero global "store/global/getters.js":

```
,[keyNames.GET_LOADING]: state=>{ return state.loading;}
```

Así como la mutación para cambiar el valor del estado en cuestión en el fichero global "store/global/mutations.js":

```
,[keyNames.MUTATE_LOADING]: (state,value)=>{ state.
loading=value; }
```

Ahora creamos el interceptor de axios, en un fichero nuevo "in-

```
1 import axios from 'axios'
  import store from '@/store'
  import * as keyNames from '@/keyNames.js'
  Interceptamos todas las solicitudes invocadas
  axios.interceptors.request.use(function (config) {
      store.commit(keyNames.MUTATE_LOADING,true);
      return config;
    }, function (error) {
      return Promise.reject(error);
  });
```

Antes de lanzar la solicitud, mostramos la imagen de carga

Si algo falla en la solicitud invocada, lanzamos error

terceptor.js", ubicado en la raíz "src" ❶. Gracias a "axios.interceptor.request" capturaremos el momento previo a la invocación de la solicitud remota. Cambiamos a verdadero el valor del estado "loading" mediante su mutación "MUTATE_LOADING" para mostrar la imagen.

Tenemos preparado el interceptor; abrimos el fichero "App.vue" e importamos el interceptor, los getters y las claves:

```
import interceptor from '@/interceptor'
import {mapGetters} from 'vuex'
import * as keyNames from '@/keyNames.js'
```

Además, crearemos en este mismo fichero la propiedad computada "loading" para utilizarla en el template:

```
computed: { ...mapGetters({ loading: keyNames.GET_LOADING }) }
```

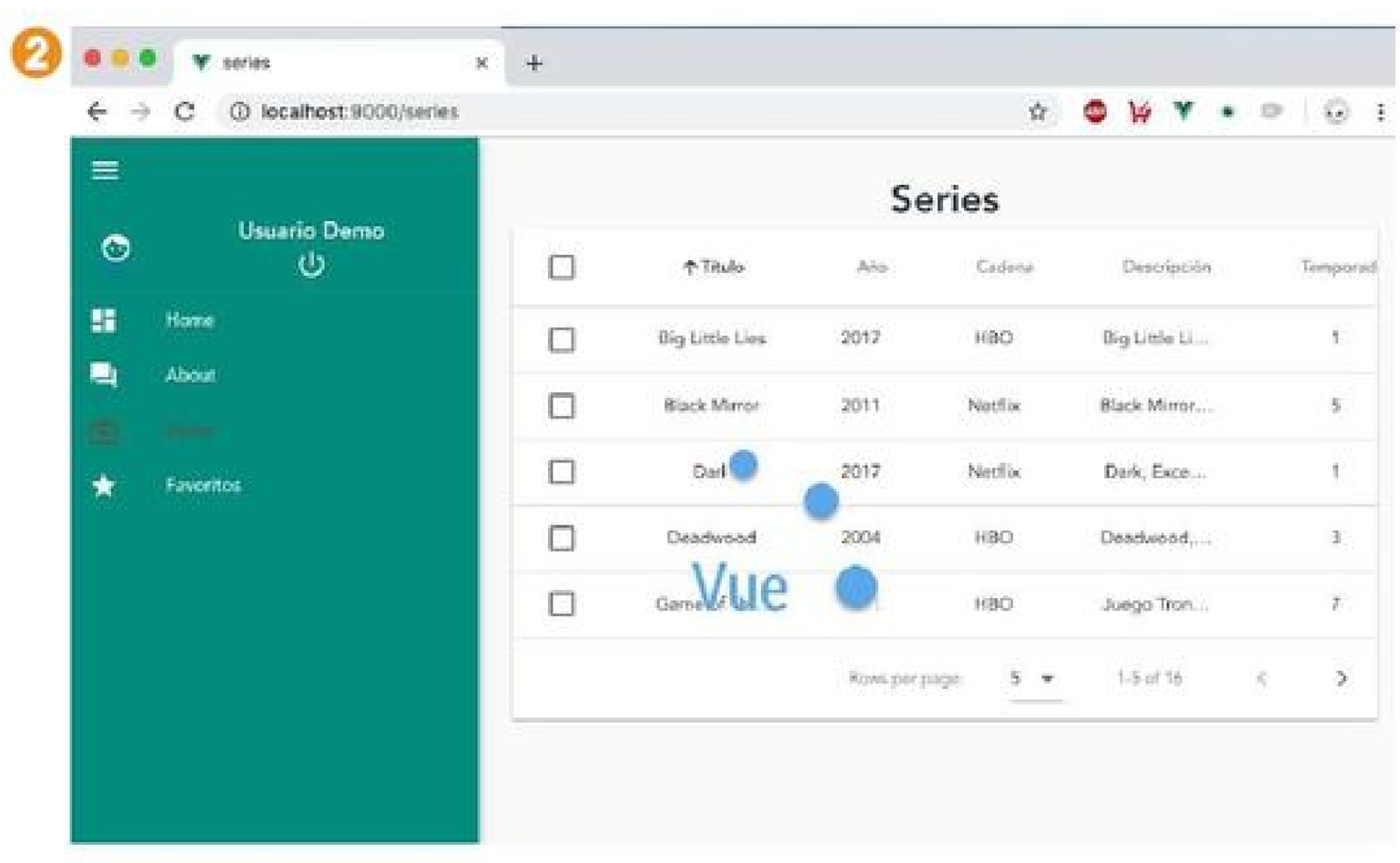
Finalmente, en el template de "App.vue", tras la etiqueta "</v-app>" creamos una etiqueta de imagen con un condicional que mostrará la imagen si el valor del estado "loading" es verdadero.

```
<v-app>…</v-app> <img v-if="loading" class="spinner" src="@/assets/icons/loa-
ding.gif" />
```

Así como definimos la clase de la imagen, para mostrarla en mitad de pantalla:

```
.spinner { position: fixed; margin: auto; top: 0; bottom: 0; right: 0; left: 0;}
```

Si abrimos la ruta "/series", debería mostrarse esta imagen de carga ❷. Sin embargo, observamos un problema, que no se cierra la imagen. Lo resolveremos en el siguiente ejercicio.



Axios – Interceptor Respuestas

IMPORTANTE

Es importante centralizar la gestión de errores de respuestas del servidor para no repetir código innecesario y cumplir el principio DRY de no repetirnos.

Respuestas

Tras invocar una solicitud de datos remotos, toda respuesta recibida cuyo resultado sea satisfactorio o erróneo puede ser interceptado de forma centralizada en un mismo punto, es decir, en el interceptor de axios, de tal forma que podamos actuar de la misma forma ante cualquier situación.

Anteriormente centralizamos en el interceptor las solicitudes, ahora haremos lo mismo pero con la recepción de respuestas. Utilizaremos el interceptor de respuestas de axios "axios.interceptor.response" para capturar el momento en que recibimos la respuesta del servidor.

Será en este interceptor ❶, tanto en respuesta satisfactoria como en la captura de error, donde cambiamos a falso el valor del estado "loading" mediante su mutación "MUTATE_LOADING" para ocultar la imagen:

```
axios.interceptors.response.use(
   function (response) { store.commit(keyNames.MUTA-
TE_LOADING,false);  return response; },
   function (error) {  store.commit(keyNames.MUTATE_
LOADING,false);   return Promise.reject(error);
});
```

Si ahora abrimos la ruta "/series", veremos que se abre la imagen de espera de respuesta ❷ y, cuando la respuesta es capturada, la imagen se oculta ❸.

Errores

Podemos tratar en este interceptor, de forma centralizada, los errores recibidos. En caso de recibir un código de error HTTP 401 implica usuario no autenticado o sesión caducada, debiendo realizar logout y redirigir a la pantalla de login.

Para ello, en el interceptor "axios.interceptors.response" capturamos el error recibido, utilizando el código de estado de la respuesta "error.response.status", donde en caso de ser 401 haremos logout y redirigimos a la pantalla login.

Por otra parte, si el código de error es 500, se ha producido un

error interno en el servidor, dejando de ser accesible. En este caso podremos redirigir a una pantalla de que muestre este código de error. Importemos por tanto el router:

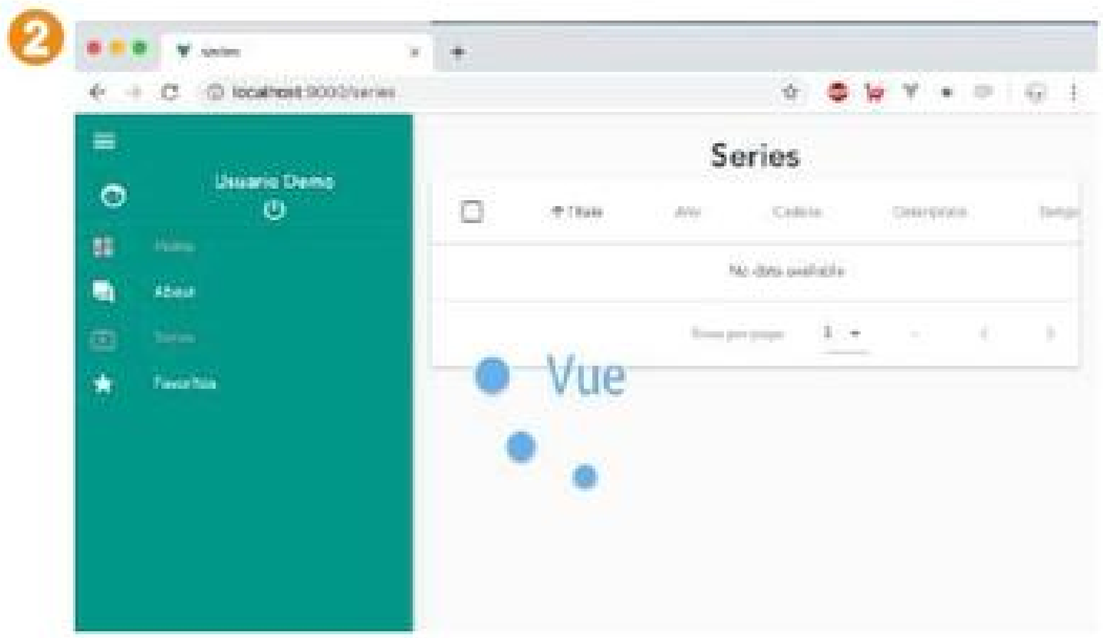
```
import router from './router'
```

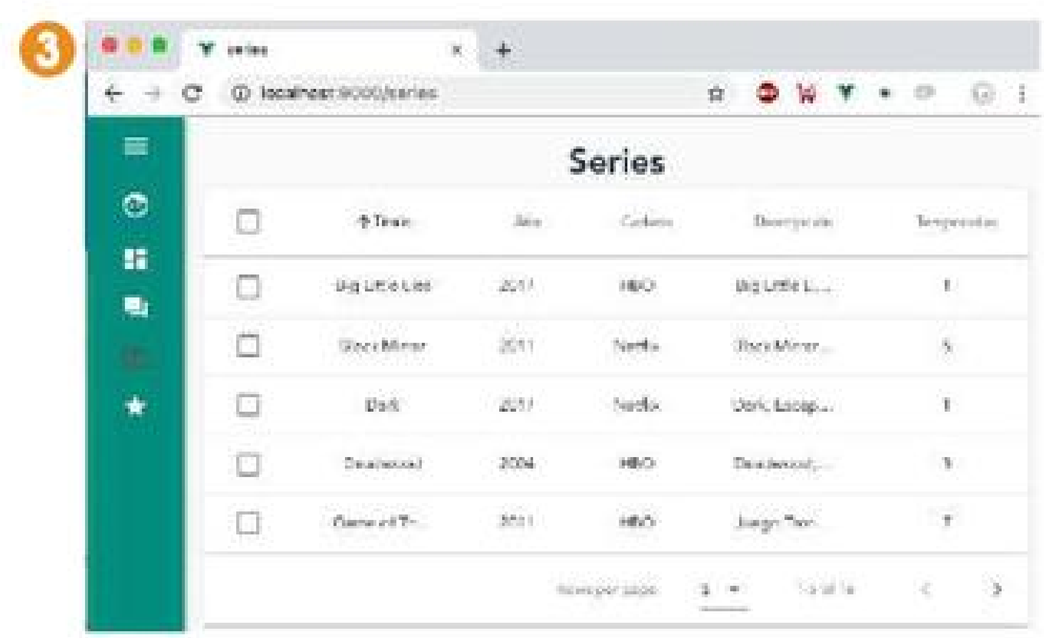
Y en la recepción del error controlemos estos dos errores:

```
function (error) { store.commit(keyNames.MUTATE_LOADING,false);
   switch (error.response.status) {
      case 500: router.replace('/Error500'); break;
      case 401: store.dispatch(keyNames.ACTION_CLOSE_SESSION); router.replace('/
login'); break; }
   return Promise.reject(error); }
```

Por simplificar el ejercicio, no hemos creado las páginas de "Error500.Vue" ni de "Login.vue".

1
```
axios.interceptors.response.use(function (response) {
    store.commit(keyNames.MUTATE_LOADING,false);
    return response;
  }, function (error) {
    store.commit(keyNames.MUTATE_LOADING,false);
    return Promise.reject(error);
});
```

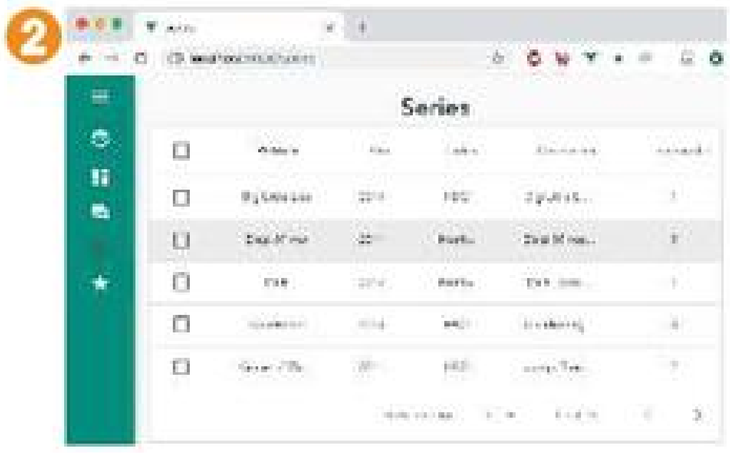
2


3


Axios – Post

IMPORTANTE

En axios, en el método POST el primer parámetro es la URL, el segundo son los datos para enviar y el tercero puede contener la configuración como cabeceras "headers".



POST

El método POST de Axios nos permite enviar datos al servidor. Como ejercicio, enviaremos nuestra serie favorita.

Creamos en cada entorno ".env" la url del api para almacenar en base de datos los favoritos:

```
VUE_APP_API_FAVOURITE_URL=https://vue-100ejercicios-favoritos.firebaseio.com
```

En el fichero keyNames definimos cada nombre de constante a utilizar en el módulo nuevo "favourite.js", que crearemos en "store/modules/favourite.js" e incluye todo lo relativo a favoritos ❶.

La variable de estado "favourite" almacena la serie favorita, mientras que la "favouriteKey" almacena la clave de registro en sessionStorage y es retornada por el api tras la inserción, que será utilizada para obtener dicho registro.

Creamos los "getters" y mutaciones que trabajan con dicho estado e implementamos dos acciones, una para obtener el favorito, identificado por la clave "state.favouriteKey", cuyo resultado mutará la información del favorito.

Otra acción es enviar favorito mediante POST, usando la URL del api favoritos, pasándole en "data" la serie a enviar. Obtendremos la clave de registro del favorito enviado, que almacenaremos localmente y mutaremos dicha serie.

```
axios.post( process.env.VUE_APP_API_FAVOURITE_URL + '/.json', data)
```

Una vez creado el módulo, lo importamos en el "store":

```
import moduleFavourites from '@/store/modules/favourite.js'
```

Y lo añadimos a los módulos:

```
,modules:{ moduleSeries, moduleFavourites }
```

En el componente "Serie.vue" importamos "mapGetters", "keyNames" y definimos las propiedades computadas:

```
,computed:{ ...mapGetters({ favourite: keyNames.FAVOURITE }) }
```

Nada más cargar "Serie.vue", en el "mounted" obtenemos del api, el favorito almacenado remotamente:

```
,mounted() { this.$store.dispatch(keyNames.ACTION_GET_FAVOURITE); …}
```

Definimos un método para marcar una serie como favorita tras pulsar el botón con forma de corazón:

```
,methods:{    MarkAsFavourite(){    this.$store.dispatch(keyNames.ACTION_POST_
FAVOURITE,this.item); }
```

Por último, en el botón de corazón comprobamos si el "Id" de la serie cargada coincide con el de la serie favorita, para cambiar su color. Además, si lo pulsamos, marca la serie como favorita, enviándola remotamente:

```
<v-btn icon :style="(favourite && favourite.Id == item.Id)? 'background-color:
red;color: white' : ''"
   @click="MarkAsFavourite"> <v-icon>favorite</v-icon></v-btn>
```

Si abrimos la pantalla "/series" ❷ y seleccionamos una, aparece su ficha ❸. Al cargar la ficha, preguntará al api, por la serie favorita remota, cambiando el color del botón a rojo, si coincide. Para marcar la serie como favorita, pulsamos en dicho corazón, enviando la serie seleccionada. Tras esto el corazón aparecerá en rojo ❹.

❶

```
import * as keyNames from '@/keyNames.js'
import axios from 'axios'
const state = { favouriteKey:sessionStorage.getItem('favouriteKey'), favourite:null };

const getters = {  [keyNames.FAVOURITE_KEY]: state => { return state.favouriteKey; }
                  ,[keyNames.FAVOURITE]: state => { return state.favourite; }      };

const mutations = { [keyNames.MUTATE_FAVOURITE]: (state,value) => { state.favourite = value; }
                   ,[keyNames.MUTATE_FAVOURITE_KEY]: (state,value) => {
                     state.favouriteKey = value; sessionStorage.setItem('favouriteKey',value);} };
const actions = {
 [keyNames.ACTION_GET_FAVOURITE]: ({commit,state},data) => {
    if(state.favouriteKey)
      axios.get(process.env.VUE_APP_API_FAVOURITE_URL+'/'+state.favouriteKey+'/.json', data)
           .then((response)=>{ if(response) commit(keyNames.MUTATE_FAVOURITE,response.data); })
           .catch((error)=>{ }) }
 ,[keyNames.ACTION_POST_FAVOURITE]: ({commit},data) => {
   axios.post(process.env.VUE_APP_API_FAVOURITE_URL+'/.json', data)
         .then((response)=>{ if(response) { commit(keyNames.FAVOURITE,data);
                                            commit(keyNames.MUTATE_FAVOURITE_KEY,response.data.name); }})
         .catch((error)=>{ }) } };

export default { state, getters, mutations, actions }
```

Axios – PUT

IMPORTANTE

La configuración de Axios se envía en el POST, PATCH y PUT por tercer parámetro. Además, en lugar de "axios.put()" puede usar "axios({ method:'put', url:'URL', data:{} })".

PUT

Otro método ofrecido por Axios ❶ es el PUT. Permite enviar al servidor un registro que queremos actualizar y difiere del método PATCH, en que el PUT actualiza un registro por completo, en lugar de únicamente un atributo. Continuemos con el ejercicio anterior, añadiendo la funcionalidad de cambiar el elemento favorito mediante PUT.

Para ello necesitamos declarar la constante "ACTION_PUT_FAVOURITE" en el fichero "keyNames.js" ❷ junto al resto de constantes del ejercicio anterior, para poder utilizarlas en el módulo existente "store/modules/favourite.js".

Una vez tenemos definidas las constantes necesarias, abrimos el módulo "favourite.js" y añadimos la acción de actualización de registro "ACTION_PUT_FAVOURITE". Esta acción lo primero que comprueba es si disponemos de la clave del favorito, almacenado localmente, como podemos ver en la ventana de desarrolladores F12 ❸. Disponer de esta clave implica haber enviado previamente un favorito al servidor.

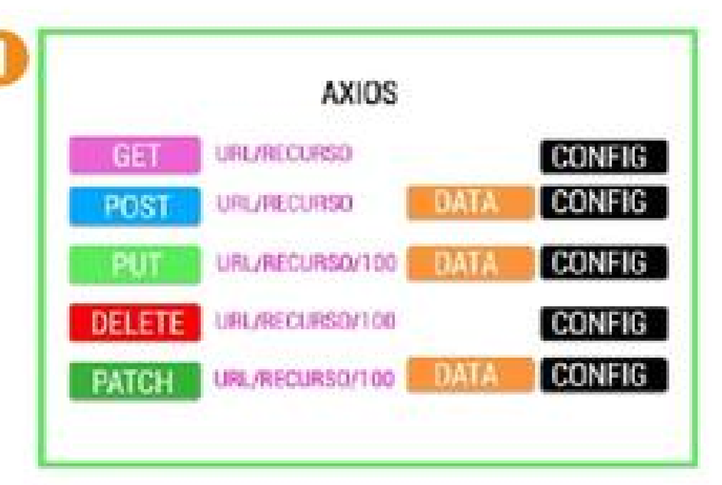
Por tanto, en caso de disponer de esta clave "state.favouriteKey" invocamos el método PUT de axios para actualizar en el servidor la serie favorita enviada, identificada por la clave "favouriteKey". Esta serie que actualizaremos la enviaremos por el parámetro data, al igual que ocurría con el POST, salvo que, en esta ocasión, la url contiene la clave a actualizar. Finalmente, al recibir respuesta, mutamos la variable de estado "favourite":

```
,[keyNames.ACTION_PUT_FAVOURITE]: ({commit},data)
=> {   if(state.favouriteKey)

     axios.put(process.env.VUE_APP_API_FAVOURITE_
URL+'/'+state.favouriteKey+'/.json', data)

          .then((response)=>{          if(response)
commit(keyNames.MUTATE_FAVOURITE,response.data); })

          .catch((error)=>{ })          }
```

Abrimos el componente "Serie.vue", donde añadimos la propiedad computada de la clave del favorito para poder utilizarla en métodos posteriores:

```
,computed:{ ...mapGetters({ favourite: keyNames.
FAVOURITE, favouriteKey: keyNames.FAVOURITE_KEY }) }
```

❶



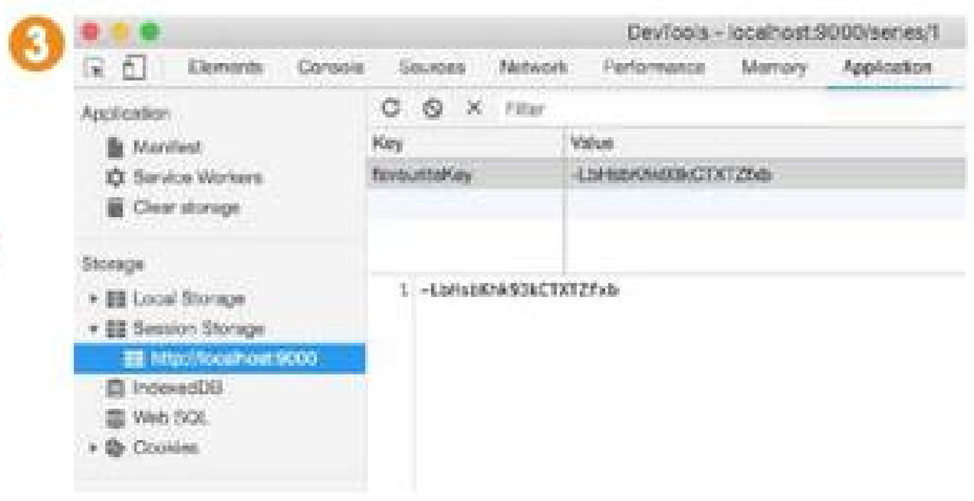
Modificamos el método "MarkAsFavourite" para que, en caso de no disponer de clave, invoque al POST, ello implicaría obtener una nueva clave de registro favorito. En caso contrario invocaríamos al método PUT, para modificar nuestra serie favorita en el servidor, enviando la nueva serie:

```
MarkAsFavourite(){  if(this.favouriteKey == null)
        this.$store.dispatch(keyNames.ACTION_POST_FAVOURITE, this.item);
    else this.$store.dispatch(keyNames.ACTION_PUT_FAVOURITE, this.item);        }
```

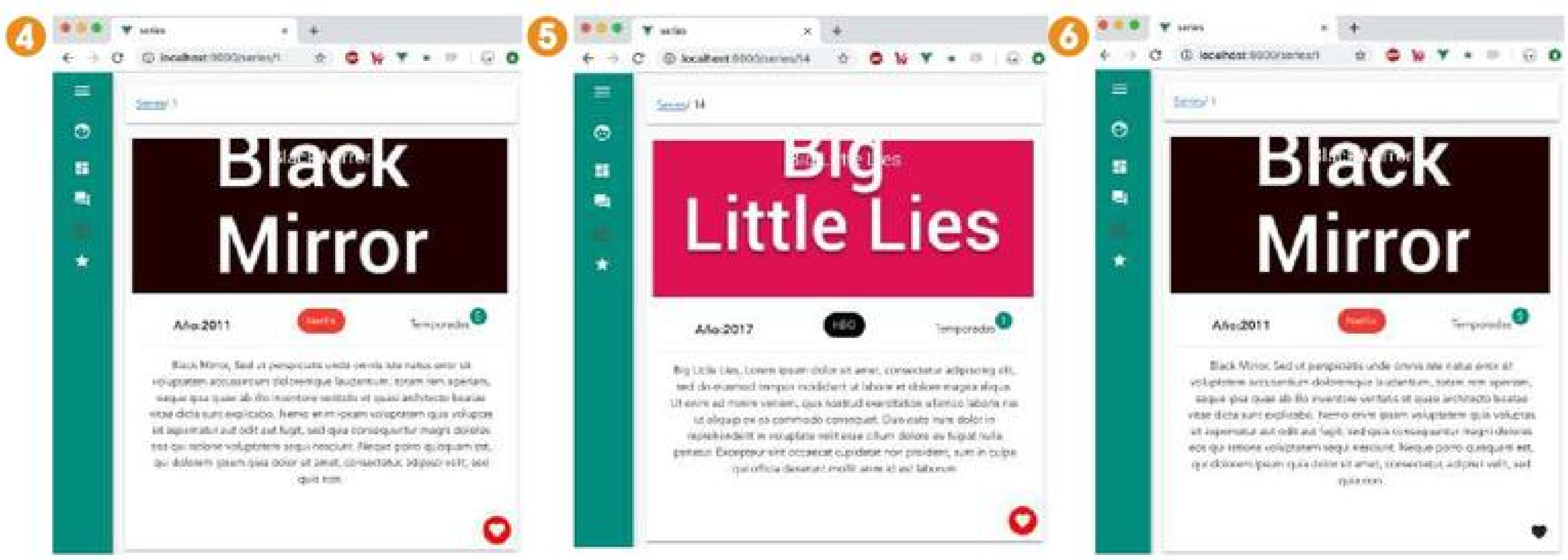
Abrimos la ruta del listado de series "/Series" y abrimos la serie seleccionada anteriormente como favorita, vemos que está marcada con un corazón rojo ❹. Si ahora volvemos al listado de series anterior y seleccionamos otra ficha vemos que no es favorita, de modo que pulsamos el corazón y la marcamos como favorita ❺. Finalmente comprobamos que la anterior ha dejado de ser favorita ❻.

❷
```
export const FAVOURITE = 'favourite_FAVOURITE';
export const MUTATE_FAVOURITE = 'favourite_MUTATE_FAVOURITE';
export const FAVOURITE_KEY = 'favourite_FAVOURITE_KEY';
export const MUTATE_FAVOURITE_KEY = 'favourite_MUTATE_FAVOURITE_KEY';
export const ACTION_GET_FAVOURITE = 'favourite_ACTION_GET_FAVOURITE';
export const ACTION_POST_FAVOURITE = 'favourite_ACTION_POST_FAVOURITE';
export const ACTION_PUT_FAVOURITE = 'favourite_ACTION_PUT_FAVOURITE';
```

❸



❹ ❺ ❻



Axios – Delete

IMPORTANTE

Siempre que realicemos el borrado de información remota desde la interfaz de usuario es conveniente ofrecer una ventana de confirmación que permita al usuario confirmar el borrado.

Delete

Para el borrado de información almacenada en el servidor, utilizamos el método DELETE de axios. Utilizado en el api RESTful, por esta razón practicaremos el borrado de la serie favorita, almacenada en la base de datos remota.

Comencemos por registrar una nueva constante para la acción de borrado de favorito en el fichero "keyNames.js":

```
export const ACTION_DELETE_FAVOURITE = 'favourite_ACTION_DELETE_FAVOURITE';
```

Posteriormente, en el módulo "favourite.js" añadimos la acción de borrado, verificando si existe la clave a eliminar. En caso de existir, invocamos el método DELETE de axios, cuyo primer parámetro es la URL del recurso, seguido del identificador a borrar, la clave del favorito. Tras recibir respuesta, mutamos el valor de la serie favorita a nulo.

```
,[keyNames.ACTION_DELETE_FAVOURITE]: ({commit}) => {
if(state.favouriteKey)

   axios.delete(process.env.VUE_APP_API_FAVOURITE_URL + '/' + state.favouriteKey+'/.json')

   .then((response)=>{ if(response) commit(keyNames.MUTATE_FAVOURITE,null); }) }
```

Ahora, en el componente "Serie.vue" haremos que cuando nos encontremos en una serie marcada como favorita, en lugar de renviar la serie al servidor, mostraremos una ventana modal para confirmar el borrado. Creamos esta ventana con el componente "<v-dialog>" de Vuetify ❶ en la plantilla de "Serie.vue". Definiendo el título de la ventana en "<v-card-title>", el mensaje en "<v-card-text >", así como dos botones en "<v-card-actions>", uno para cancelar la acción de borrado y otro botón para confirmar la acción, ejecutando el método "Delete()". Esta ventana utiliza la variable dialog para aparecer. De modo que añadimos esta variable en el data de este componente:

```
   data(){   return   {   id:this.$route.params.id, item:this.$route.params.item, dialog:false } }
```

Ahora, cuando pulsemos el botón de corazón se invocará al método "MarkAsFavourite()" como hasta ahora, salvo que en esta ocasión verificaremos previamente si el identificador del favorito

y el de la serie en que nos encontramos es la misma. En dicho caso mostraremos la ventana de confirmación modal.

```
MarkAsFavourite(){ if(this.favourite && this.favourite.Id == this.item.Id) this.
dialog=true;
   else { /*El caso antiguo*/ } }
```

Añadimos un nuevo método de borrado "Delete()" que, en caso de pulsar el botón "Eliminar", se ejecutará este método, donde la ventana cerrará y posteriormente se lanzará la acción de borrado remoto:

```
,Delete(){ this.dialog = false; this.$store.dispatch(keyNames.ACTION_DELETE_FA-
VOURITE); }
```

De modo que, en la interfaz de usuario, abrimos el listado de series y seleccionamos la que teníamos marcada como favorita ❷. Aparecerá su botón de corazón activo en rojo, si lo pulsamos aparece una ventana modal de confirmación de borrado ❸. Si cancelamos o pulsamos fuera de la ventana, esta se cierra. En caso contrario, si pulsamos "Eliminar", se elimina el favorito, desmarcando el botón de corazón de la serie en que nos encontramos ❹.

Comunicación – Padre – Hijo

Padre - Hijo

La comunicación entre componentes es unidireccional y esencial. Empecemos por conocer la iniciada desde el componente padre hasta el hijo ❶.

Creamos un componente hijo en "components/ItemSerie.vue" ❷. Tiene definida la propiedad "propSerie" en las props, por donde se recibirán los datos enviados por el padre, que serán representados.

Es el momento de crear el componente padre, en "views/Historial.vue"; su plantilla ❸ utiliza tres botones; al pulsarlos realizan filtros sobre los datos a enviar. Además, contiene una lista de componentes hijos "ItemSerie.vue" para cada serie filtrada. El envío de datos se realiza indicando el nombre de la propiedad que espera recibir el hijo, precedido de dos puntos ":propSerie", pasándole el dato a enviar, en este caso la serie.

```
<item-serie v-for="serie in seriesFiltered" :key="serie.Id"     :propSerie="serie"
> </item-serie>
```

En el código de este componente padre ❹ obtenemos por propiedad computada el listado de series que tenemos de Vuex. Definimos dos variables en el data, "selected" indica si un botón ha sido seleccionado mientras que "seriesFiltered" almacena las series filtradas según el botón pulsado.

De modo que en el método "created()" inicializamos esta variable de series filtradas con todas las series:

```
,created(){ this.seriesFiltered = this.series; }
```

❶

❷

```
<template>
  <v-card :class="propSerie.Cadena">
    <v-layout row wrap>
      <v-flex md12> {{propSerie.Titulo}} - {{propSerie.Cadena}} </v-flex>
    </v-layout>
  </v-card>
</template>
<script>
  export default { name: 'item-serie', props:['propSerie'] }
</script>
<style>
  .v-card  { margin:0.25em; padding:0.5em; font-weight: bold; float:left;
             text-align: left; cursor:pointer; color:white !important; }
  .Netflix { background-color: #f44336!important; }
  .HBO     { background-color: #212121 !important; }
</style>
```

Valores recibidos de los datos enviados por el padre

Propiedad por donde recibe el Hijo datos del padre

❸

```
<template>
  <div class="historial">
    <h1>Historial</h1>
    <v-btn color="primary" :class="{active : selected=='All'}"
           @click="filter()">Todas</v-btn>
    <v-btn dark :class="{active : selected=='HBO'}"
           @click="filter('HBO')">HBO</v-btn>
    <v-btn color="red" :class="{active : selected=='Netflix'}"
           @click="filter('Netflix')">NETFLIX</v-btn>
    <v-divider></v-divider>
    <v-layout row wrap> <v-flex md6>
          <item-serie v-for="serie in seriesFiltered" :key="serie.Id"
                      :propSerie="serie"> </item-serie> </v-flex>
    </v-layout>
  </div>
</template>
```

La clase del botón aparecerá como activa, si "selected" es "All"

Al pulsar el botón, filtra por dicha cadena

Componente Hijo

Propiedad enviada al Hijo

Incluimos en el componente padre el componente hijo:

```
,components:{ itemSerie:ItemSerie }
```

Definimos el método "filter" del componente padre, que recibe por parámetro el valor de la cadena, por la cual se realizará el filtro de datos obteniendo, en base al filtro, las series filtradas. Dichas series las asignamos a la variable "seriesFiltered" de modo que, al cambiar su valor, se envían al hijo por las props.

```
filter(value) { this.seriesFiltered = this.series.filter( (serie) => (!value) || se-
rie.Cadena == value);
    this.selected = (!value) ? 'All' : value; } }
```

Añadimos en el router la siguiente ruta hasta la vista del historial:

```
,{ path: '/historial', name: 'historial', component: () => import('@/views/Histo-
rial.vue')}
```

Añadimos en el componente "Components/MenuLateral.vue" el enlace en el menú para abrir la ruta historial:

```
{ titulo: 'Historial', icono: 'history', to: { name: 'historial' } }
```

En la aplicación aparece el elemento historial en el menú, al pulsarlo se abre la ruta mostrando las series sin filtro previo, así como el botón "TODAS" seleccionado ❺. Si pulsamos otro botón, por ejemplo "HBO", filtraremos los datos por dicha cadena, por lo que será enviado a los hijos los nuevos datos ❻.

❹

```
<script>
import { mapGetters } from 'vuex'
import * as keyNames from '@/keyNames.js'
import ItemSerie from '@/components/ItemSerie.vue';
```

Componente hijo

```
export default {
  name: 'historial'
  ,computed:{ ...mapGetters({ series: keyNames.ALL_SERIES }) }
  ,data(){ return { selected:'All', seriesFiltered:[]} }
  ,created() { this.seriesFiltered = this.series; }
```

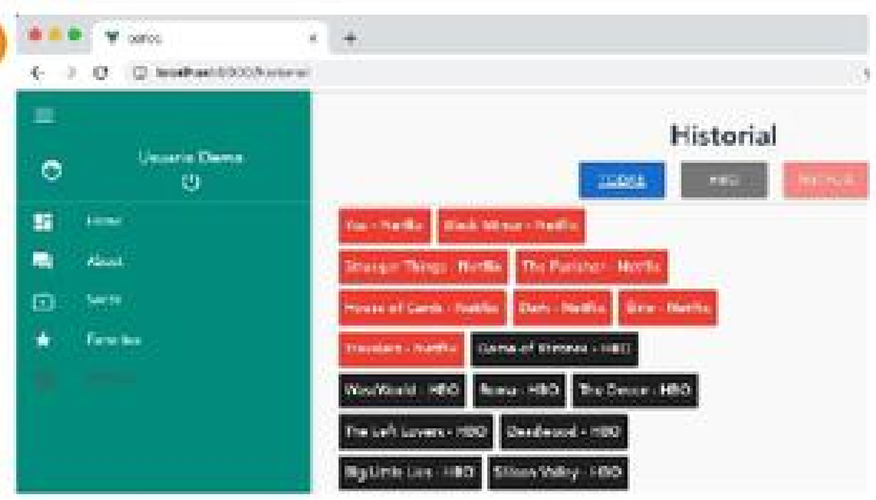
Inicializamos el filtro de series, con todas las series

```
  ,components:{ itemSerie:ItemSerie}
  ,methods:{ filter(value) {
        this.seriesFiltered=this.series.filter((serie)=>(!value) || serie.Cadena==value);
        this.selected=(!value)?'All':value; } }
}
</script>
```

Método filtro, que realiza un filtro según el botón pulsado, filtrando las series por cadena elegida.
Además marca el botón pulsado como el seleccionado, para cambiarle su clase a activa.

```
<style scoped>
  .v-btn { opacity:0.6; color:white; }
  .active { opacity:1; text-decoration:underline;}
</style>
```

❺



❻

Comunicación – Hijo – Padre

IMPORTANTE

Los eventos emitidos desde el hijo únicamente pueden ser capturados por el padre que lo contiene, siempre que el padre haya declarado el "v-on".

Hijo - Padre

La comunicación entre componente hijo y padre consiste en emitir un evento el hijo, capturándolo el padre ❶.

Comuniquemos el componente hijo "ItemSerie.vue" y su padre "Historial.vue" de modo que, al pulsar el hijo, se emita un evento al padre, enviándole la serie seleccionada, que el padre mostrará por pantalla. Además, al pulsar en el hijo la cadena a la que pertenece, el padre filtrará utilizando dicha cadena.

Abramos el componente hijo "ItemSerie.vue" ❷ y añadamos el evento click en su plantilla. En el click, emitiremos el evento de nombre "selected-item", mediante "$emit", enviando por valor el objeto serie "propSerie":

```
<v-flex md12 @click="$emit('selected-item', propSerie)">
```

En esta misma plantilla añadimos el click sobre el nombre de cadena, ejecutando el método "changeFilter" del hijo:

```
   {{propSerie.Titulo}} - <span @click="changeFilter">
{{propSerie.Cadena}} </span>
```

Este método "changeFilter" emite el evento "change-filter", con el nombre de cadena como valor:

```
   changeFilter(){   this.$emit('change-filter',this.
propSerie.Cadena); }
```

Ambos eventos son emitidos hacia el componente que esté a la escucha, en este caso el padre. Vayamos al componente padre "Historial.vue" y definamos en el data la variable "selectedItem", para recoger el elemento seleccionado por el hijo:

```
,data(){ return { selected:'All', seriesFiltered:[],
selectedItem:null } }
```

En la plantilla del padre visualizamos los datos del título del hijo seleccionado:

```
<h2 v-if="selectedItem">Elemento seleccionado: {{ selectedItem.Titulo }}</h2>
```

Capturamos en la plantilla del padre mediante "v-on:nombreEvento" los eventos emitidos por el hijo ❸:

```
<item-serie … v-on:selected-item="selectedItem = $event" v-on:change-filter="filter($event)"> </item-serie>
```

Por un lado capturamos el evento "selected-item" mediante "v-on" de modo que, al capturar este evento, recogemos mediante "$event" el valor emitido desde el hijo, que asignaremos a la variable "selectedItem".

Por otra parte capturamos el evento "change-filter", de modo que invocamos al método del padre "filter", proporcionándole como valor de filtro la cadena de la serie seleccionada.

Si entramos a la pantalla del historial y pulsamos una serie, aparecerá en el padre el título de la serie seleccionada ❹. Por otro lado, si pulsamos el nombre de cadena de un hijo este emitirá un evento con el nombre de cadena seleccionada, que el padre capturará, ejecutando el método "filter" del padre para filtrar por dicha cadena ❺.

❷

```
<template>
  <v-card :class="propSerie.Cadena">
    <v-layout row wrap>
      <v-flex md12 @click="$emit('selected-item',propSerie)">
        {{propSerie.Titulo}} - <span @click="changeFilter"> {{propSerie.Cadena}} </span>
      </v-flex>
    </v-layout>
  </v-card>
</template>
<script>
  export default { name: 'item-serie', props:['propSerie']
                 , methods:{
                     changeFilter(){ this.$emit('change-filter',this.propSerie.Cadena); }
                 } }
</script>
```

Enviamos la serie hacia el padre, al pulsar el hijo

Enviamos al padre un evento para cambiar el filtro

Enviamos la cadena del hijo al padre, para que el padre actualice el filtrado

❹

❸

```
<template>
  <div class="historial">
    <h1>Historial</h1>
    <h2 v-if="selectedItem">Elemento seleccionado: {{ selectedItem.Titulo }}</h2>
    <v-btn color="primary" :class="{active : selected=='All'}"
           @click="filter()">Todas</v-btn>
    <v-btn dark :class="{active : selected=='HBO'}"
           @click="filter('HBO')">HBO</v-btn>
    <v-btn color="red" :class="{active : selected=='Netflix'}"
           @click="filter('Netflix')">NETFLIX</v-btn>
    <v-divider></v-divider>
    <v-layout row wrap>
        <v-flex md6> <item-serie v-for="serie in seriesFiltered" :key="serie.Id"
                     :propSerie="serie"
                     v-on:selected-item="selectedItem = $event"
                     v-on:change-filter="filter($event)">
                     </item-serie> </v-flex>
    </v-layout>
  </div>
</template>
```

El padre a la escucha del evento "selected-item"

Padre a la escucha del evento "change-filter"

❺

Comunicación – Hijo – Padre por Callback

IMPORTANTE

Enviar un método como "prop" al hijo no es una buena práctica, por acoplamiento del hijo al método del padre. Si cambiase el padre, se debería enviar un nuevo método.

Función Callback

Una alternativa al uso de eventos para la comunicación entre el componente hijo y el componente padre es empleando funciones de retorno o "callback" ❶.

Para ello preparamos el componente hijo "ItemSerie.vue" para que, en lugar de emitir un evento, reciba en las "props" la función a invocar del padre. Indicaremos que la propiedad "propSerie" es de tipo objeto, mientras que la nueva propiedad "propFilter" es de tipo función. Con esto estaremos recibiendo la referencia a la función del padre:

```
, props: { propSerie: Object, propFilter: Function }
```

Una vez tenemos las "props", en la plantilla del componente hijo en lugar de emitir un evento al hacer click sobre el nombre de la cadena de la serie, ejecutaremos el método del padre recibido por las props, simplemente invocándolo como si fuera un método del mismo componente hijo, pasándole como parámetro la cadena por la cual filtrar ❷:

```
<span @click="propFilter(propSerie.Cadena)">
{{propSerie.Cadena}} </span>
```

Abrimos ahora el componente padre "Historial.vue" y, en lugar de capturar mediante "v-on:change-filter" e invocar el método "filter" del padre tras recibir el evento, enviamos a través de la propiedad del hijo "propFilter" ❸ el método del padre, esto creará una referencia fuerte a dicho método que tras ser recibido por las "props" del hijo, podrá ser ejecutado cómodamente por parte del componente hijo:

```
<item-serie … :propFilter="filter" v-on:selected-
item="selectedItem = $event"> </item-serie>
```

Si ahora abrimos la página del historial y pulsamos el nombre de una cadena veremos que sigue funcionando de la misma forma que cuando emitíamos un evento, es decir, se ejecuta el mismo método del padre que en el ejercicio anterior, con la salvedad de que antes lo hacíamos capturando el evento y ahora simplemente al enviar la referencia de la función del padre al hijo, el hijo la invoca directamente 4.

2

```
<template>
  <v-card :class="propSerie.Cadena">
    <v-layout row wrap>
      <v-flex md12 @click="$emit('selected-item',propSerie)">
        {{propSerie.Titulo}} -          Invocamos la función del padre, recibida por props
          <span @click="propFilter(propSerie.Cadena)"> {{propSerie.Cadena}} </span>
        <!--<span @click="changeFilter"> {{propSerie.Cadena}} </span>-->
      </v-flex>
    </v-layout>
  </v-card>                                   Sustituimos emisión de evento anterior
</template>
<script>
  export default { name: 'item-serie'
                  //,props:['propSerie']                    El hijo recibe propiedad de tipo función
                  , props:{ propSerie: Object, propFilter: Function }
                  , methods:{
                      changeFilter(){ this.$emit('change-filter',this.propSerie.Cadena); }
                  } }
</script>
```

3

```
<template>
  <div class="historial">
    <h1>Historial</h1>
    <h2 v-if="selectedItem">Elemento seleccionado: {{ selectedItem.Titulo }}</h2>
    <v-btn color="primary" :class="{active : selected=='All'}"
           @click="filter()">Todas</v-btn>
    <v-btn dark :class="{active : selected=='HBO'}"
           @click="filter('HBO')">HBO</v-btn>
    <v-btn color="red" :class="{active : selected=='Netflix'}"
           @click="filter('Netflix')">NETFLIX</v-btn>
    <v-divider></v-divider>
    <v-layout row wrap> <v-flex md6>
           <item-serie v-for="serie in seriesFiltered" :key="serie.Id"
                       :propSerie="serie"  Enviamos al hijo la función "filter" del padre - referencia
                       :propFilter="filter"
                       v-on:selected-item="selectedItem = $event">
                       </item-serie> </v-flex>
                       <!--v-on:change-filter="filter($event)"-->
    </v-layout>                 Dejamos de capturar el evento del hijo para la ejecución del filtro
  </div>
</template>
```

4

Comunicación – Entre hermanos

IMPORTANTE

La comunicación entre hermanos requiere utilizar al componente padre como puente de comunicación con otros componentes hermanos.

Entre Hermanos

Existen diversas formas de comunicar componentes hermanos dentro de un mismo componente padre ❶. Una de estas requiere pasar a través del padre para poder comunicarse con otros componentes hermanos.

Realicemos un ejercicio donde el componente padre "Historial.vue" contenga dos componentes hijos. De modo que, al seleccionar un "ItemSerie.vue", este se comunique con su componente hermano "DetailSerie.vue" para mostrar información del elemento seleccionado.

Comencemos por crear el componente hermano "DetailSerie.vue", que define una propiedad de tipo objeto, la serie seleccionada por el hermano "ItemSerie.vue". El valor de esta propiedad será representado en la plantilla ❷:

```
props:{ propSerie:Object }
```

Ahora nos vamos al padre "Historial.vue" y, dado que ya tenemos disponible en el "data" del padre la variable "selectedItem", de ejercicios anteriores, no la volvemos a definir.

Incluimos en el padre el componente hijo que acabamos de crear, "DetailSerie.vue":

```
import DetailSerie from '@/components/DetailSerie.vue';
```

Y lo agregamos al listado de componentes del padre:

❶

```
,components: { itemSerie: ItemSerie, detailSerie: DetailSerie }
```

En la plantilla del padre teníamos un listado de componentes hijo de tipo "ItemSerie.vue", que ocupaban 6 columnas. Agregamos en otras 6 columnas este segundo componente hijo "DetailSerie.vue" ❸ y le asignamos a la propiedad "propSerie" de este hijo el valor de la variable "selectedItem" del "data" del padre:

```
<v-flex md6> <detail-serie :propSerie="selectedItem"></detail-serie> </v-flex>
```

De modo que abrimos la ruta del historial y observamos dos columnas en el padre, a la izquierda los hijos de tipo "ItemSerie.vue" y la de la derecha el hijo de tipo detalle "DetailSerie.vue" ❹. Si pulsamos uno de los hijos del primer tipo "ItemSerie.vue", este componente hijo emitirá el evento "$emit('selected-item', propSerie)".

Será capturado por el padre gracias a "v-on:selected-item" y asignará a la variable "selectedItem" del "data" del padre el valor de la serie seleccionada. Al cambiar el valor de esta variable, será enviada desde el padre al segundo hijo de tipo "DetailSerie.vue", donde visualizamos la ficha de esta serie ❺.

❷

```
<template>
  <v-card v-if="propSerie">
    <v-layout row wrap style="text-align:center">
      <v-flex md3>{{propSerie.Cadena}}</v-flex>
      <v-flex md6 class="underline"><h3>{{propSerie.Titulo}}</h3></v-flex>
      <v-flex md3>{{propSerie.Anyo}},T[{{propSerie.Temporadas}}]</v-flex></v-layout>
    <v-layout row wrap>
      <v-flex md12>{{propSerie.Descripcion}}</v-flex>
    </v-layout>
  </v-card>
  <span v-else>Seleccione un elemento para mostrar detalle.</span>
</template>

<script>
import ActionsDetail from '@/components/ActionsDetail.vue'
export default { name: 'detail-serie', props:{ propSerie:Object } }
</script>
<style scoped>
  .v-card{color:black !important; font-weight: normal;}
  .underline{text-decoration:underline}
</style>
```

Propiedad "propSerie" donde recibiremos el objeto serie a representar

❹

❸

```
<template>
  <div class="historial">
    <h1>Historial</h1>
    <h2 v-if="selectedItem">Elemento seleccionado: {{ selectedItem.Titulo }}</h2>
    <v-btn color="primary" :class="{active : selected=='All'}"
    <v-btn dark :class="{active : selected=='HBO'}"
    <v-btn color="red" :class="{active : selected=='Netflix'}"
    <v-divider></v-divider>
    <v-layout row wrap>
        <v-flex md6> <item-serie v-for="serie in seriesFiltered" :key="serie.Id"
                :propSerie="serie" :propFilter="filter"
                v-on:selected-item="selectedItem = $event"> </item-serie> </v-flex>
        <v-flex md6> <detail-serie :propSerie="selectedItem"></detail-serie> </v-flex>
    </v-layout>
  </div>
</template>
```

Segundo hijo DetailSerie.vue

Se le pasa la serie seleccionada por el primer hijo ItemSerie.vue

❺

Comunicación – Bus de eventos

IMPORTANTE

Utilice el bus de eventos siempre que un componente no tenga comunicación directa con el componente que quiera comunicarse.

❶

Bus de Eventos

Una alternativa de comunicación es el uso de un bus de eventos, basado en patrón de subscripción. Especialmente útil para intercomunicar componentes no relacionados entre sí a través de un canal.

Intercomuniquemos el hijo de un componente hijo con el hermano de su padre ❶. Para ello crearemos el componente "ActionsDetail.vue", un botón de acción hijo de "DetailsSerie.vue".

En el componente padre "Historial.vue" importamos la librería Vue y creamos un canal de comunicación "BusHistorial", que no es más que una instancia de Vue.

```
import Vue from 'vue'

export const BusHistorial = new Vue();
```

Este canal lo exportamos para ser accesible desde componentes. Capturando el evento "clear" del canal mediante "InstanciaBus .$on(NombreEvento,Función)" en el método "created()", lugar ideal para capturar eventos ❷.

```
,created() { … BusHistorial.$on('clear',()=>  { … }); }
```

Ya podemos trabajar con el canal desde el componente "ActionsDetail.vue" ❸. Donde importamos el bus del padre:

```
import { BusHistorial } from '@/views/Historial.vue'
```

Posteriormente, agregamos al "data" la instancia del bus importada:

```
data(){ return { BusHistorial } }
```

❷

```
,created() {
  var self = this;

  this.seriesFiltered = this.series;

  BusHistorial.$on('clear',()=>  {
    self.selectAll = 'All';
    self.seriesFiltered = this.series;
    self.selectedItem = null;
  });
}
```

Referenciamos la instancia del componente

Capturamos el evento 'clear' que sea recibido por bus

❸

```
<template>
  <div>
    <v-divider></v-divider>
    <v-btn color="primary" class="right"
        @click="BusHistorial.$emit('clear')">Deseleccionar</v-btn>
  </div>
</template>
<script>
  import { BusHistorial } from '@/views/Historial.vue'
  export default {
    name: 'actions-detail',
    data(){ return { BusHistorial } } }
</script>
<style scoped> .right{ float:right; } </style>
```

Al pulsar el botón, emitimos un evento a través del bus del historial

Importamos el bus del componente Historial.vue

Agregamos el bus del historial, al data del componente

Por último, en la plantilla de este componente agregamos un botón donde, al pulsarlo, emitimos un evento a través del canal "BusHistorial":

```
<v-btn color="primary" class="right" @click="BusHistorial.$emit('clear')"> De-
seleccionar </v-btn>
```

Finalmente agregamos este componente acciones dentro de "DetailsSerie.vue" ❹.

Si abrimos la ruta "/historial", al seleccionar una serie se muestra su ficha detalle, junto con el componente de acciones, botón "Deseleccionar" ❺. Al pulsar este botón emitimos un evento al bus de comunicación que será recibido por los subscriptores de este evento, en este caso "Historial.vue", mediante "BusHistorial.$on", ejecutando su función, es decir, limpiar elemento seleccionado y filtro por defecto ❻.

Para dejar de capturar el evento "clear" por el bus de comunicaciones, en el evento "beforeDestroy" del ciclo de vida de vue en "Historial.vue" ejecutamos "$off" del bus:

```
,beforeDestroy(){ BusHistorial.$off('clear',()=>{}); }
```

❹

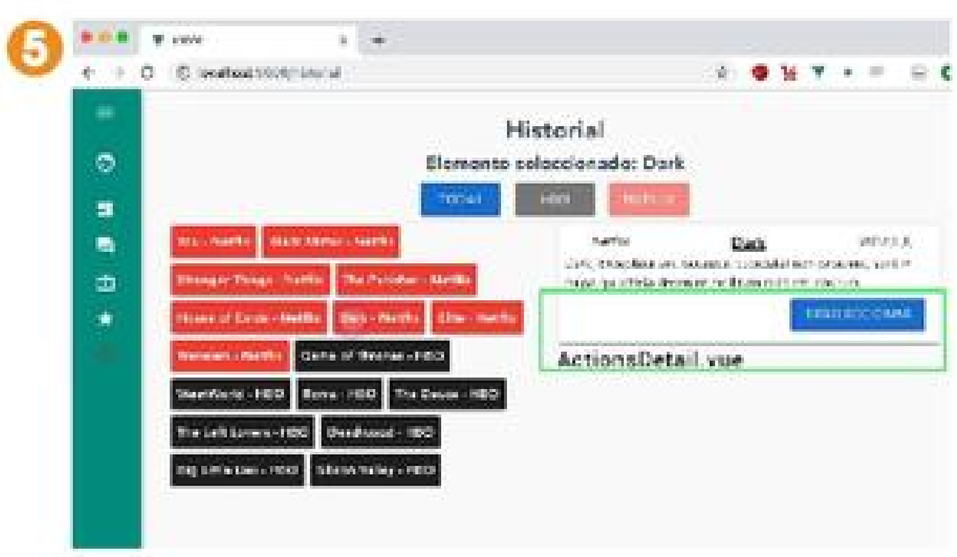
```
<template>
    <v-card v-if="propSerie">
      <v-layout row wrap style="text-align:center"></v-layout>
      <v-layout row wrap> <v-flex md12>{{propSerie.Descripcion}}</v-flex> </v-layout>
      <v-layout row wrap>
        <v-flex md12> <actions-detail></actions-detail> </v-flex>
      </v-layout>
    </v-card>
    <span v-else>Seleccione un elemento para mostrar detalle.</span>
</template>

<script>
import ActionsDetail from '@/components/ActionsDetail.vue'
export default { name: 'detail-serie', props:{ propSerie:Object }
                ,components:{ actionsDetail: ActionsDetail } }
</script>
```

Agregamos el componente de acciones a la plantilla del detalle

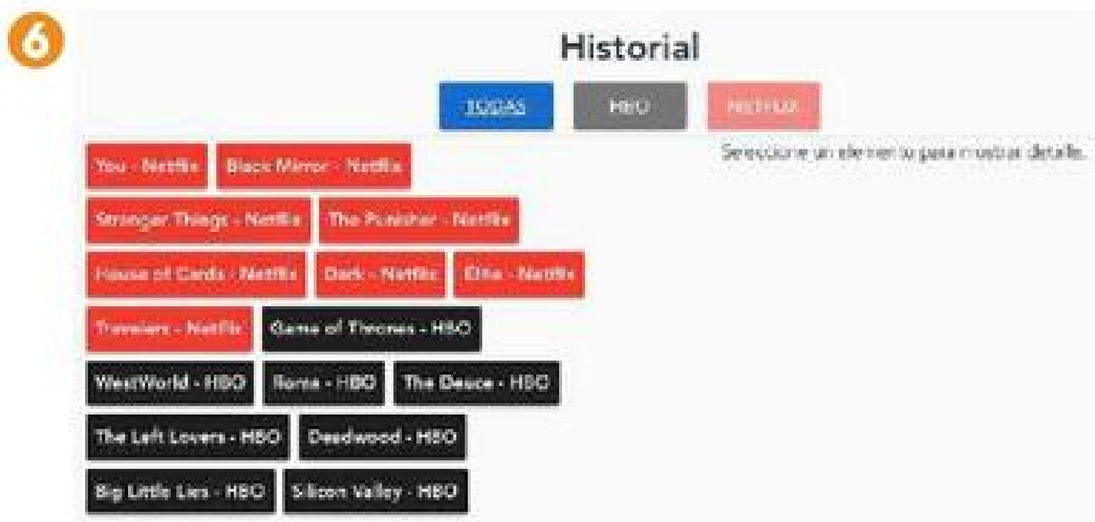
Importamos el componente

Lo agregamos al componente detalle

❺



❻



Comunicación – Mensajería Global

IMPORTANTE

Centralizar en métodos de un bus global es una buena práctica, evitando llenar de "$emit" nuestro código para emitir el mismo nombre de evento.

Mensajería global

Definir el bus de eventos a nivel global permite a todos los componentes enviar eventos a través de este canal de comunicación, centralizando eventos en un mismo punto.

Crearemos un sistema de mensajería global centralizada para tratar mensajes de éxito o error en un mismo punto.

Comenzamos por crear el bus de eventos en un nuevo fichero "src/globalBus.js" ❶. Es una instancia de Vue, como ya hemos visto en el ejercicio anterior, salvo que, en esta ocasión, definimos dos métodos para centralizar la invocación de eventos, uno para mostrar mensajes de error y otro para satisfactorios.

Una vez disponemos de este bus global, abrimos la plantilla de la aplicación "App.vue" y creamos un "snackbar", componente de mensajería de Vuetify:

❶
```
import Vue from 'vue'

export const GlobalBus = new Vue({
  methods:{
    showSuccess(message)
      { this.$emit('show-message', { message, type: 'info' } ) }
    ,showError(message)
      { this.$emit('show-message', { message, type: 'error' } ) }
  }
});
```

Centralizamos eventos del bus

❷
```
import { GlobalBus } from '@/globalBus.js'

export default {
  computed:{
    ...mapGetters({ loading:keyNames.GET_LOADING })
  }
  ,data() {
    return { snackbar:false, snackbarMessage:null, snackbarColor:'primary' }
  }
  ,created() {
    GlobalBus.$on('show-message',(data)=>{
      this.snackbar=true
      this.snackbarMessage=data.message
      this.snackbarColor= (data.type == 'info' )? 'primary' : 'red'
    })
  }
  ,beforeDestroy(){ GlobalBus.$off() }
```

Importamos el bus de eventos global

Variables del data

Iniciamos la captura de eventos del bus

Detenemos captura de todo evento del bus

```
<v-snackbar v-model="snackbar" :color="snackbarColor" right>
   {{snackbarMessage}} <v-btn flat @click.native=" snackbar = false">Cerrar</v-btn>
</v-snackbar>
```

Definimos unas variables para el "snackbar" en el data de "App.vue":

- "snackbar": para mostrar el mensaje.
- "snackbarColor": para el color de fondo, utilizando el azul para mensajes de éxito y el rojo para errores.
- "snackbarMessage": contiene el mensaje a mostrar.

En este mismo componente principal, importamos el "GlobalBus" así como en el evento "created()" iniciamos la escucha del evento "show-message" por parte del bus global ❷. Al capturar este evento, mostramos el mensaje recibido en la ventana flotante.

Detenemos la captura de todo evento por parte del bus global en "beforeDestroy()" mediante "$off", sin incluir el nombre de evento.

```
,beforeDestroy(){ GlobalBus.$off() }
```

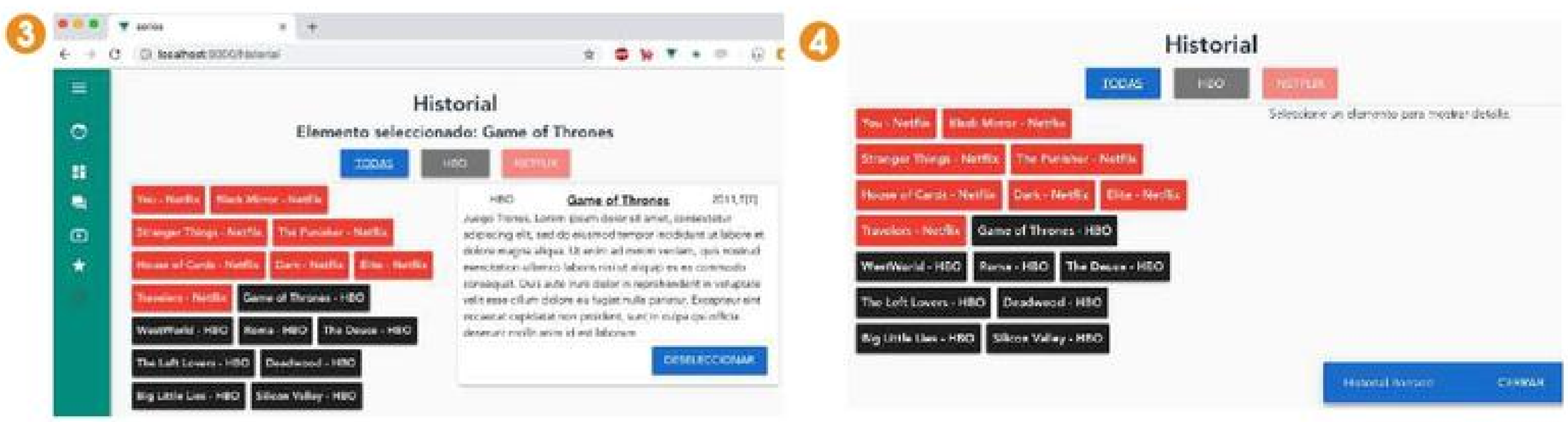
Finalmente, desde el componente "Historial.vue" importamos el bus global:

```
import { GlobalBus } from '@/globalBus.js'
```

De tal forma que en el "created()" de "Historial.vue", al capturar "BusHistorial" el evento "clear" para limpiar la selección, utilizaremos el método "showSuccess" del bus global para enviar el mensaje de éxito:

```
,created() {… BusHistorial.$on('clear',()=> { self.GlobalBus.showSuccess('Historial
Borrado') }); }
```

Abrimos la página historial y seleccionamos un elemento ❸. Cuando pulsemos el botón "Deseleccionar", "Historial.vue" capturará el evento "clean", donde se limpiará la selección y posteriormente utilizando el método "showSuccess" del bus global, le pasará el mensaje a mostrar. El bus global lo recibirá y emitirá un evento mediante "this.$emit('show-message')", capturado por "App.vue" donde mostrará el mensaje de éxito ❹.



Comunicación bidireccional

IMPORTANTE

Las promesas utilizan el método "resolve()" para retornar un valor como respuesta satisfactoria, así como el "reject()" para respuestas erróneas.

Bidireccionalidad

La comunicación entre componentes es unidireccional, sin embargo, podemos conseguir bidireccionalidad utilizando promesas ❶. Como ejercicio, recibamos la opción elegida tras abrir una ventana modal desde un componente.

Añadimos el método "confirm" al "globalBus.js", recibiendo el título y mensaje. Aquí creamos una promesa de retorno al componente. De la promesa obtenemos su parámetro "resolve", que será emitida en el evento ❷.

En la plantilla del componente principal "App.vue" agregamos un "v-dialog" de Vuetify para abrir una ventana modal con dos botones. Al pulsar cualquiera de estos botones, retornamos la respuesta de la promesa ❸.

En el "data" de este componente, agregamos las siguientes variables:

- confirm: para mostrar la ventana modal.
- confirmTitle: contiene el título de la ventana.
- confirmText: contiene el mensaje de la ventana.
- confirmPromise: contiene la promesa de retorno de respuesta.

```
,data() { … ,confirm : false, confirmTitle : null,
confirmText : null, confirmPromise : null }
```

Y capturamos el evento "show-confirmation" en el método "created" de este componente, donde mostramos una ventana con el título y mensaje, al cual le proporcionamos de la promesa para retornar la respuesta.

```
GlobalBus.$on('show-confirmation',(promise, data)
=> { this.confirm=false;

this.$nextTick(()=>{ this.confirm = true; this.con-
firmTitle = data.title; this.confirmText = data.message

this.confirmPromise=promise; } ) } )
```

Ahora en el componente "ActionsDetail.vue" cambiamos el clic del botón de limpieza, ejecutando el método "clear":

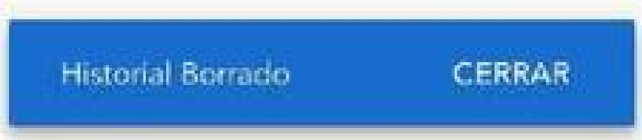
```
<v-btn color="primary" class="right" @
click="clear">Deseleccionar</v-btn>
```

Importamos en este componente el bus global:

❹

❺

❻



```
import { GlobalBus } from '@/globalBus.js'
```

Agregamos este método, que invoca el método "confirm" del bus. Al retornar una promesa, recibimos por el "then" la respuesta de la ventana. Ejecutando la deselección en caso afirmativo y un mensaje en caso negativo.

```
methods:{ clear(){ this.GlobalBus.confirm('Limpiar Selección','¿Desea limpiar selección?')
    .then((response)=>{ if(response) this.BusHistorial.$emit('clear');
        else this.GlobalBus.showError('Selección abortada');}) }
```

Si seleccionamos en el historial una serie, al pulsar el botón "Deseleccionar" aparece una ventana con dos botones ❹. Si cancelamos se muestra un mensaje ❺ mientras que si, por el contrario, aceptamos, se limpiará la selección ❻.

❶

❸

```
<v-dialog v-model="confirm">
    <h2>{{confirmTitle}}</h2>
    <h4>{{confirmText}}</h4>
    <v-divider></v-divider>
    <v-btn color="error" @click="confirmPromise(false); confirm=false"> Cancelar </v-btn>
    <v-btn color="primary" @click="confirmPromise(true); confirm=false"> Aceptar </v-btn>
  </v-card>
</v-dialog>
```

Retornamos respuesta - resolve(true) o resolve(false) de la promesa

❷

```
import Vue from 'vue'
export const GlobalBus = new Vue({
  methods:{
    showSuccess(message) { this.$emit('show-message', { message, type: 'info' } ) }
    ,showError(message) { this.$emit('show-message', { message, type: 'error' } ) }
    ,confirm(title,message) {
      let result;
      let promise=new Promise((resolve,reject) => { result = resolve })
      this.$emit('show-confirmation',result,{title,message})
      return promise;
    }
  }
});
```

Creamos una promesa

Emitimos evento con el "resolve" de la promesa

Retornamos promesa a quien invocó el método

Formulario – Campo Texto

IMPORTANTE

Para prevenir recargar la página al realizar un "submit", tras pulsar el botón del formulario debemos usar "@click.prevent", equivalente a "event.preventdefault()".

Texto

Las aplicaciones web suelen incluir formularios con distintos tipos de campos. En este ejercicio recogeremos información de un campo de tipo texto, simulando su envío.

Empecemos por crear una nueva ruta para la vista "FichaUsuario.vue" que crearemos. Para ello añadimos en las rutas del router "router.js" la ruta hacia la ficha de inserción de nuevos usuarios:

```
routes: [ … ,{ path: '/nuevoUsuario', name: 'nuevoUsuario', component: () => import('@/views/FichaUsuario.vue') } ]
```

En el componente "MenuLateral.vue" agregamos un nuevo elemento al menú para acceder a esta ruta:

```
rutas: [ … { titulo: 'Usuario', icono: 'add', to: { name: 'nuevoUsuario' } } ]
```

Estamos listos para crear la vista del componente "views/FichaUsuario.vue", que incluirá un formulario en su interior.

Utilizando los componentes de Vuetify, empezamos creando la etiqueta del formulario "v-form", que contendrá los campos que conformen el formulario, así como un botón de envío de los datos ❶.

❶
```
<template>
  <div class="fichaUsuario">
    <h2>Nuevo Usuario</h2>
    <v-form ref="form">
      <v-text-field name="Name" label="Nombre" v-model="model.Name"
              :maxlength="10" :counter="20" placeholder="Introduzca Nombre">
      </v-text-field>

      <v-divider></v-divider>

      <v-btn color="primary"  @click.prevent="sendForm">Enviar</v-btn>
    </v-form>
  </div>
</template>
```

Formulario que contiene los campos a enviar

Campo tipo texto, utiliza v-model para enlazar con modelo

Botón que realiza el submit o envio de datos

❷
```
<script>
  import { GlobalBus } from '@/globalBus.js'

  export default {
    name:'fichaUsuario'
    ,data(){ return { GlobalBus, model:{ Name:null } } }
    ,methods:{
      sendForm(){ this.GlobalBus.showSuccess(JSON.stringify(this.model)) }
    }
  }
</script>
```

Modelo del formulario

Método para enviar formulario

El elemento de tipo texto utilizado es "v-text-field" de Vuetify, un "input" de tipo texto con estilos y con los atributos:

- maxlength: límite máximo de caracteres del campo.
- counter: muestra un contador de caracteres introducidos.
- placeHolder: muestra unas instrucciones cuando no hay texto escrito.
- name: nombre del campo.
- label: título del campo.
- v-model: enlaza los datos introducidos en el campo con el modelo del data.

Añadimos al formulario un botón para invocar el método "sendForm" al pulsarlo y enviar el modelo.

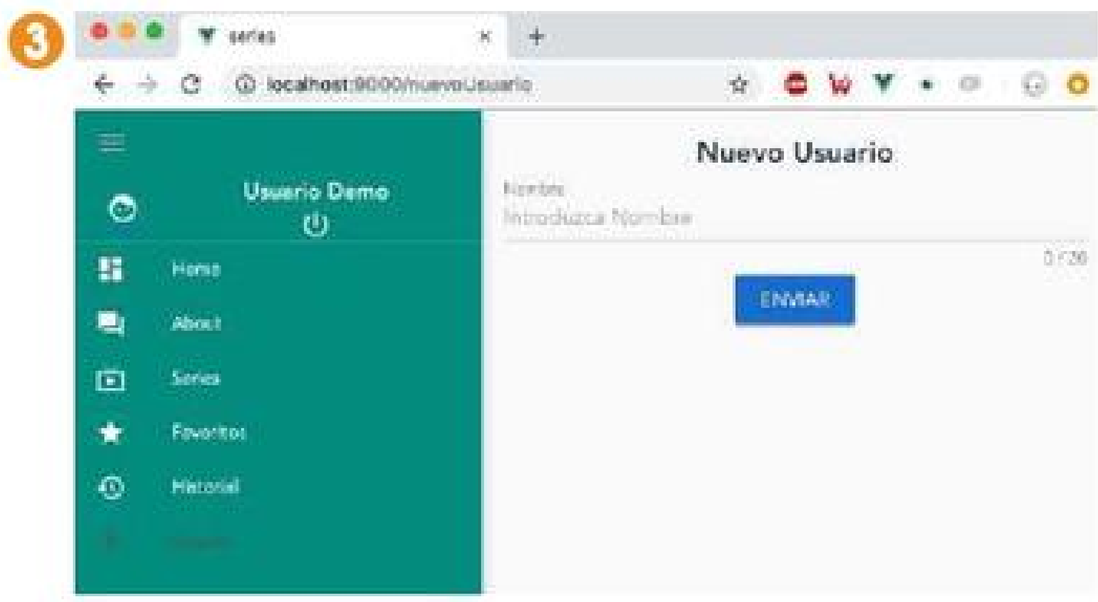
En cuanto al código ❷ del componente "FichaUsuario.vue", importamos el bus global para mostrar mensajes por pantalla, así como agregamos en el data el modelo que utilizará el formulario.

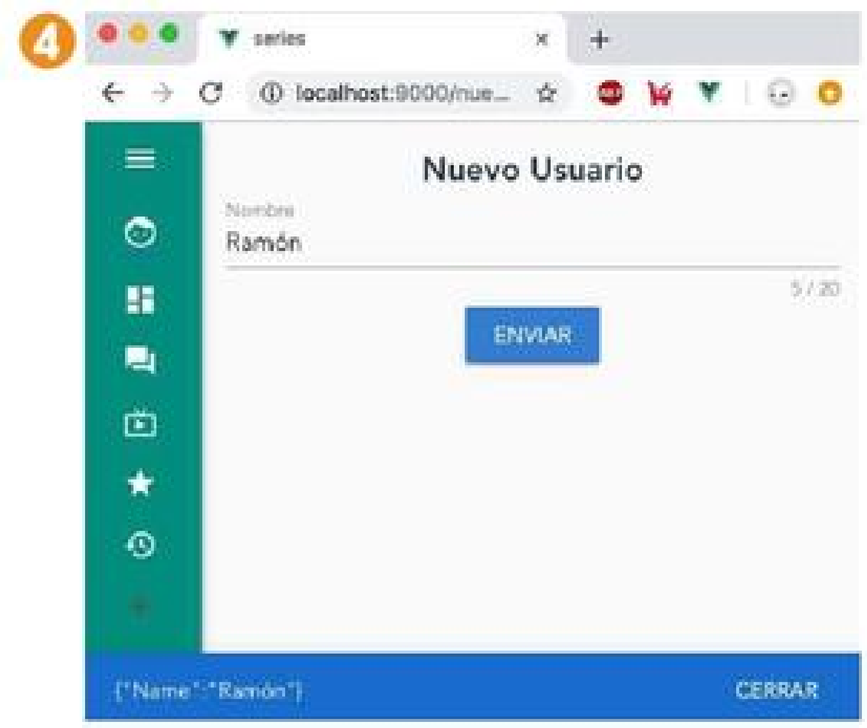
```
,data(){ return { GlobalBus, model:{ Name:null } } }
```

Finalmente, recogemos en el método "sendForm" el modelo del data y lo mostramos por pantalla:

```
sendForm(){ this.GlobalBus.showSuccess(JSON.stringify(this.model)) }
```

Si abrimos la ficha de nuevos usuarios, visualizamos el formulario con un campo de tipo texto y un botón ❸. Al introducir un texto en el campo nombre, automáticamente se enlaza con el modelo gracias a "v-model", con la propiedad "Name" de "model", es decir "model.Name". Implica que al introducir el valor "Ramón", el modelo del data, sería "model: { Name:'Ramón' }". Si pulsamos el botón enviar, visualizamos un mensaje con el modelo al serializarlo mediante "JSON.stringify" ❹.





Formularios – Email con validación

IMPORTANTE

Utilizar validaciones a nivel de campo permite validar en tiempo de edición el campo, el formato, tipo, tamaño o, si es requerido, evitando enviar datos inválidos al servidor.

Email

En esta ocasión vamos a realizar un ejercicio en el que añadiremos un campo de tipo texto, validaremos si es un email correcto y, en caso de no serlo, no podremos enviar el formulario.

Reutilizando el formulario del ejercicio anterior, añadiremos un campo nuevo ❶ de tipo texto, para representar el email; su "v-model" estará enlazado en esta ocasión con la propiedad del modelo del data "Email". Para dotar a este campo de la funcionalidad de validar si el valor es un email válido utilizaremos la propiedad "rules" de Vuetify, un array que puede contener diversas validaciones a nivel de campo, en este caso añadiremos la validación de email.

En cuanto al código, añadimos en el data del componente "FichaUsuario.vue" un nuevo atributo "Email" para el modelo:

```
   ,data(){ return { GlobalBus, model:{ Name:null,
Email:null } …}
```

Así como también añadiremos al data anterior un array de reglas o validaciones, llamado "rules", al que añadiremos la validación de email, que utilizará el método "test" para ejecutar si la expresión regular es válida para el email.

```
, rules:{ email:(email)=>{
   return    /^([a-z0-9_-]+\.)*[a-z0-9_-]+@[a-z0-9_-
]+(\.[a-z0-9_-]+)*\.[a-z]{2,6}$/.test(email)      ||
            'Mail Inválido' } }
```

❶

```
<template>
  <div class="fichaUsuario">
    <h2>Nuevo Usuario</h2>
    <v-form ref="form">   ref-> Se utiliza para localizar un elemento en el DOM
      <v-text-field name="Name" label="Nombre" v-model="model.Name"></v-text-field>
  Campo tipo texto con validación de email
      <v-text-field name="Email" label="E-Mail" v-model="model.Email"
              :maxlength="100" :counter="100" placeholder="Introduzca E-Mail"
              :rules="[rules.email]"></v-text-field>
                        rules-> Array que contiene validaciones a realizar
      <v-divider></v-divider>

      <v-btn color="primary"  @click.prevent="sendForm">Enviar</v-btn>
    </v-form>
  </div>
</template>
```

Finalmente, antes de enviar el formulario tras pulsar el botón, comprobaremos si existe alguna validación fallida en el formulario. Para esto, antes tenemos que localizar el formulario del DOM, que será posible gracias al uso del atributo referencia o "ref" en el formulario :

```
<v-form ref='form'>
```

Con esto indicamos que el formulario lo referenciamos mediante el nombre "form", quedando accesible mediante "this.$refs.form". Por lo que ahora, una vez localizado el formulario, podemos comprobar si no ha pasado alguna validación a través de su método "validate" de "this.$refs.form.validate()" ❷.

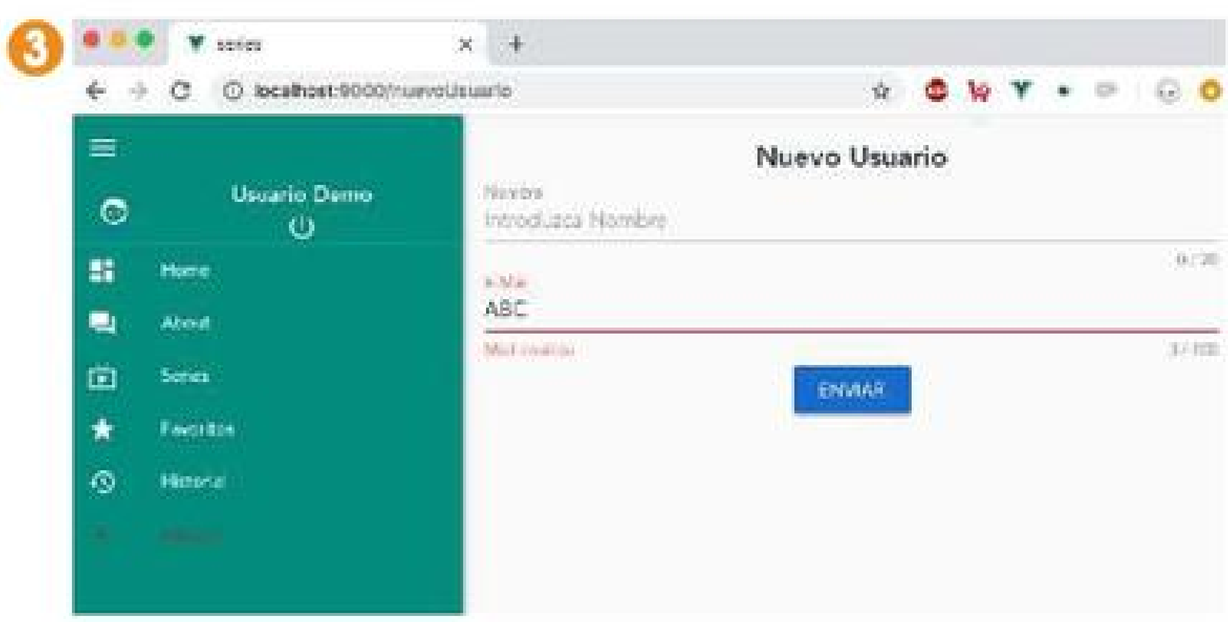
Si abrimos la ruta de introducir nuevo usuario en la aplicación observamos el nuevo campo email debajo del campo nombre, de modo que, al introducir un email erróneo, por ejemplo "ABC" ❸, aparece una validación errónea debajo del campo email indicando "Mail Inválido", imposibilitando enviar datos del formulario si pulsamos el botón enviar, dado que hay validaciones incorrectas en el formulario. De modo que si ahora corregimos el valor del email por uno válido, la validación errónea desaparecerá y podremos enviar los datos, mostrándolos por pantalla ❹.

❷
```
<script>
  import { GlobalBus } from '@/globalBus.js'

  export default {
    name:'fichaUsuario'
    ,data(){
        return { GlobalBus
            , model:{ Name:null, Email:null }
            , rules:{ email:(email)=>{
                  return /^([a-z0-9_-]+\.)*[a-z0-9_-]+@[a-z0-9_-]+(\.[a-z0-9_-]+)*\.[a-z]{2,6}$/
                         .test(email) || 'Mail Inválido' } }
    }
    }
    ,methods:{ sendForm(){ if(this.$refs.form.validate())
                              this.GlobalBus.showSuccess(JSON.stringify(this.model)) }
    }
  }
</script>
```

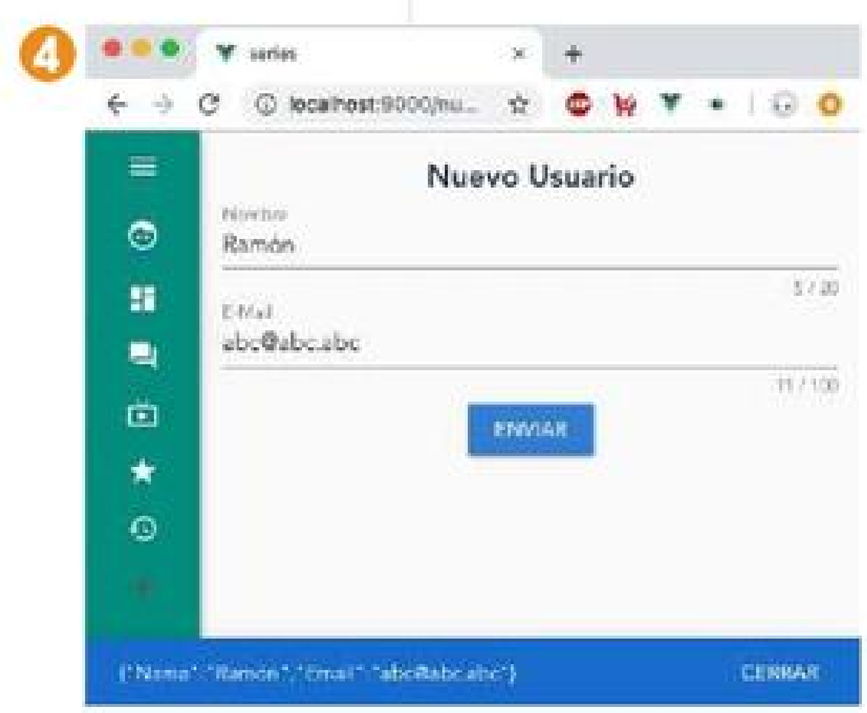
Valicación de email a ejecutar en rules.email del campo

Comprobar si no hay validaciones fallidas en formulario

❸



❹



Formularios – Fechas moment

IMPORTANTE

Es recomendable importar la configuración regional de fecha mediante "locale" de la librería "moment" para tratar las fechas independientemente del país en que se encuentre.

Fechas

Trabajar con fechas es muy común en los formularios, utilizaremos la librería "moments", que nos permite aplicar distintos formatos, así como también utilidades para trabajar con la fecha.

Lo primero que debemos hacer es incorporar esta librería al proyecto, para ello instalaremos el paquete "moment":

```
npm install moment
```

Una vez instalado, en las dependencias del proyecto veremos esta librería añadida en "package.json":

```
"dependencies": { … "moment": "^2.24.0"},
```

De modo que podemos empezar a utilizarla, para ello abrimos el componente "FichaUsuario.vue", donde incluimos un nuevo campo de tipo fecha en la plantilla del componente ❶, indicado mediante "type='date'". Además, añadimos una validación de fecha mediante las "rules", que nos permitirán validar una fecha errónea:

```
<v-text-field name="Date" label="Fecha Nacimiento" v-
model="model.Date"
   placeholder="Introduzca fecha nacimiento"
type="date" rules="[rules.date]"></v-text-field>
```

Una vez definido el campo fecha, visualizaremos a través de una propiedad computada la fecha en formato extenso, "Día de la semana," + "Nombre del mes," + "Día del mes," + "Año" + "Hora", mediante el formato "LLLL" de la librería "moment", como veremos posteriormente. Añadamos en la plantilla el resultado de esta propiedad computada:

```
{{fechaNacimiento}}
```

Una vez preparada la plantilla, en la sección del código importamos la librería "moment" para poder utilizarla, así como su configuración de fecha española, importada mediante "momento/locale/es".

```
import * as moment from 'moment';
import 'moment/locale/es';
```

Añadimos la propiedad computada para mostrar la fecha que escribamos en el formato extenso:

```
   ,computed: { fechaNacimiento(){  return (this.model.Date)? moment(this.model.
Date).format('LLLL') : ''  }  }
```

En las reglas de validación "rules" del "data" añadimos una nueva para validar fechas:

```
, rules:{…, date:(date)=>{ return /^(1|2)\d{3}-\d{2}-\d{2}$/.test(date) || 'Fecha
Inválida'} }
```

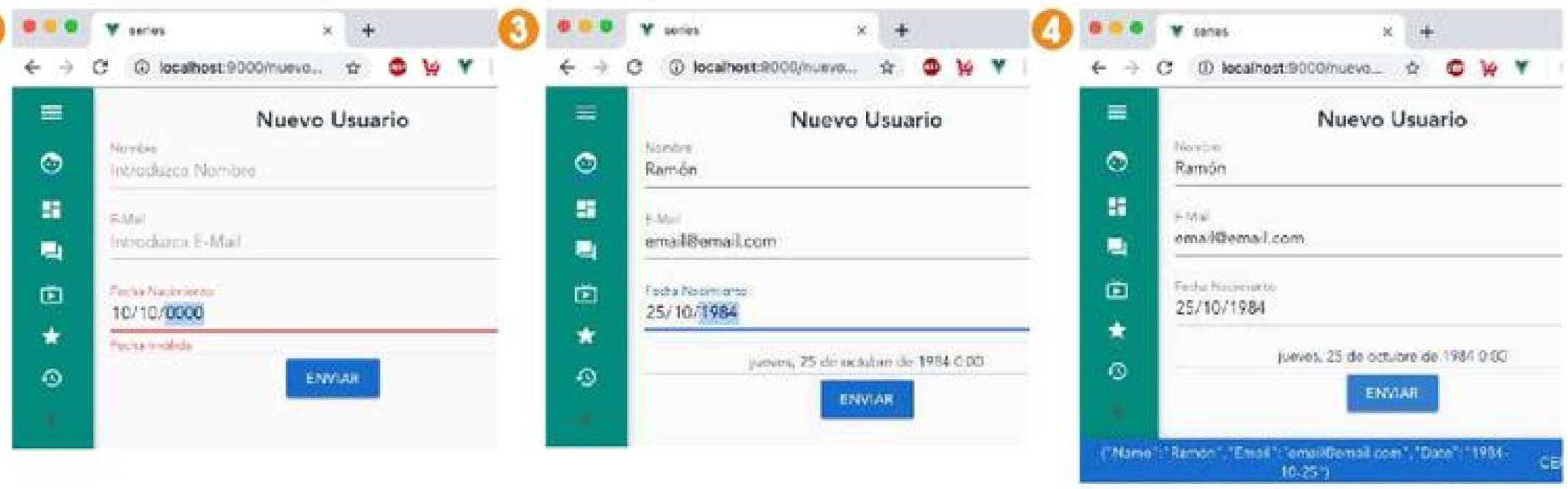
Así como también agregamos al "data" el atributo "Date" del modelo, que enlaza con la fecha:

```
, model:{ Name:null, Email:null, Date:null}
```

Si abrimos la ruta de la ficha para introducir nuevo usuario e introducimos una fecha errónea, se mostrará el mensaje de "Fecha Inválida" ❷, de modo que, si corregimos la fecha, se borrará esta validación y aparecerá bajo la fecha, el formato extenso para esta fecha ❸. Finalmente, si enviamos el formulario, como no tiene fallos de validación, se enviarán los datos ❹.

❶
```
<template>
  <div class="fichaUsuario">
    <h2>Nuevo Usuario</h2>
    <v-form ref="form">
      <v-text-field name="Name" label="Nombre" v-model="model.Name"></v-text-field>
      <v-text-field name="Email" label="E-Mail" v-model="model.Email"></v-text-field>
                                                                  Campo tipo Fecha
      <v-text-field name="Date" label="Fecha Nacimiento" v-model="model.Date"
                placeholder="Introduzca fecha nacimiento"
                type="date"  :rules="[rules.date]"></v-text-field>
      {{fechaNacimiento}}                Validacion fecha
          Propiedad Computada formato fecha
      <v-divider></v-divider>

      <v-btn color="primary"  @click.prevent="sendForm">Enviar</v-btn>
    </v-form>
  </div>
</template>
```

❷ ❸ ❹



Formularios – Opciones Checkbox

IMPORTANTE

Si en lugar de "v-model" queremos controlar su selección manualmente, deberemos de utilizar la propiedad ":checked" del "checkbox" y comprobar el elemento seleccionado.

Checkbox

Cuando queremos enviar una selección de múltiples opciones, debemos utilizar los "checkbox", los conocemos de html, únicamente que desde Vue podemos interaccionar con estos de otra forma.

Como ejercicio vamos a crear tres componentes de tipo "checkbox" que nos permitan seleccionar una serie de opciones, en este caso serán subscripciones a contenido de vídeo.

Agregamos en la plantilla de "FichaUsuario.vue" ❶ las tres opciones que el usuario podrá seleccionar:

```
<v-checkboxv-model="model.subscribe"label="Subscribir
a Netflix" value="netflix"></v-checkbox>

<v-checkbox                    v-model="model.subscribe"
label="Subscribir a HBO"    value="hbo"></v-checkbox>

<v-checkboxv-model="model.subscribe"label="Subscribir
a Amazon"  value="amazon"></v-checkbox>
```

❶

```
<template>
  <div class="fichaUsuario">
    <h2>Nuevo Usuario</h2>
    <v-form ref="form">
      <v-text-field name="Name" label="Nombre" v-model="model.Name"></v-text-field>
      <v-text-field name="Email" label="E-Mail" v-model="model.Email"></v-text-field>
      <v-text-field name="Date" label="Fecha Nacimiento" v-model="model.Date"></v-text-field>
      {{fechaNacimiento}}
                                                                          Opciones múltiples
      <v-checkbox v-model="model.subscribe" label="Subscribir a Netflix" value="netflix"></v-checkbox>
      <v-checkbox v-model="model.subscribe" label="Subscribir a HBO"     value="hbo"></v-checkbox>
      <v-checkbox v-model="model.subscribe" label="Subscribir a Amazon"  value="amazon"></v-checkbox>
                  Array de opciones                                      Valor de la opción
      <v-divider></v-divider>

      <v-btn color="primary"  @click.prevent="sendForm">Enviar</v-btn>
    </v-form>
  </div>
</template>
```

Como podemos observar, el modelo utiliza el atributo "subscribe" del modelo del "data" del componente, se trata de un array. Cuando trabajamos con opciones múltiples, el modelo es un array, donde cada vez que seleccionamos una opción, esta se añade al array. Es por este motivo que utilizan las tres opciones el mismo atributo de tipo array, para que su valor se incluya en la misma lista. El valor que tomará el array al marcar la opción será el indicado en el atributo "value" del "v-checkbox".

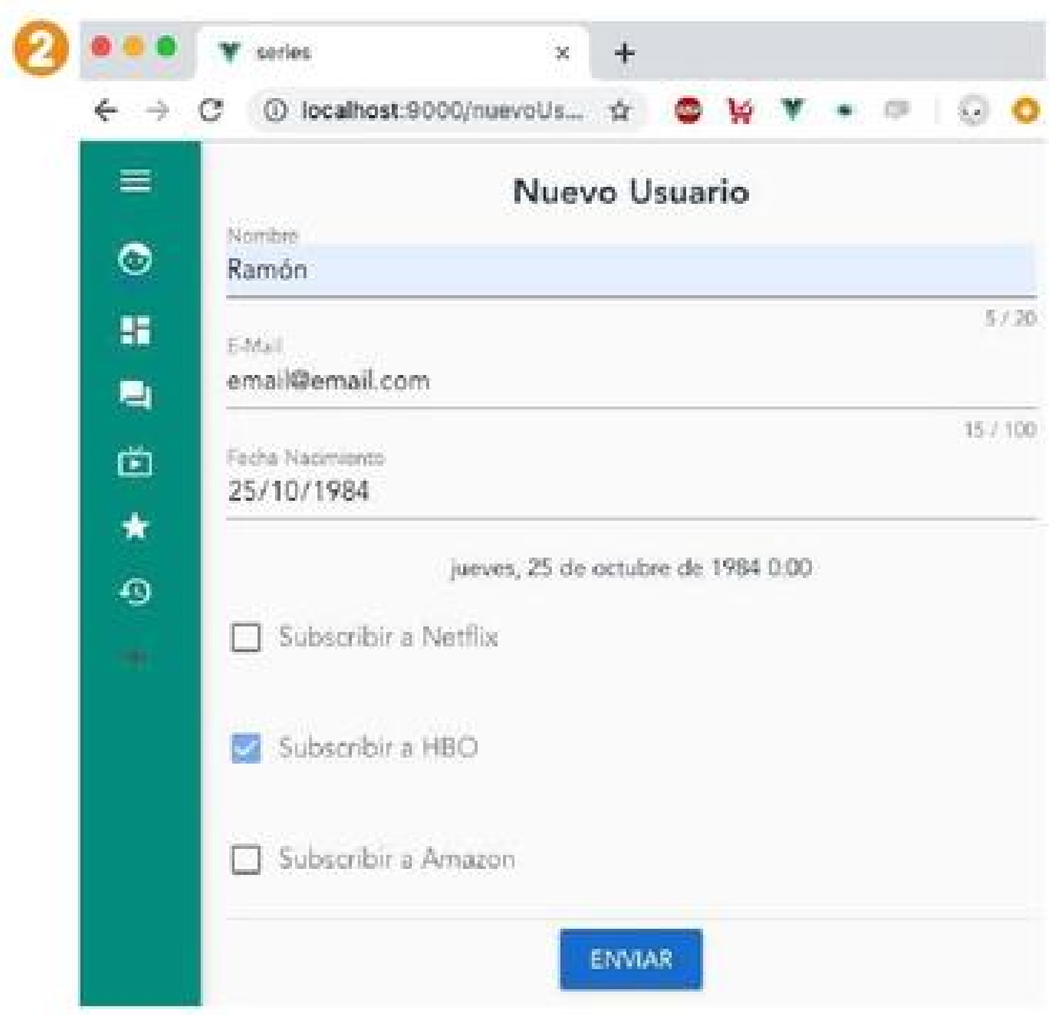
De esta forma, al seleccionar por ejemplo la opción Netflix, el valor del array para este atributo será:

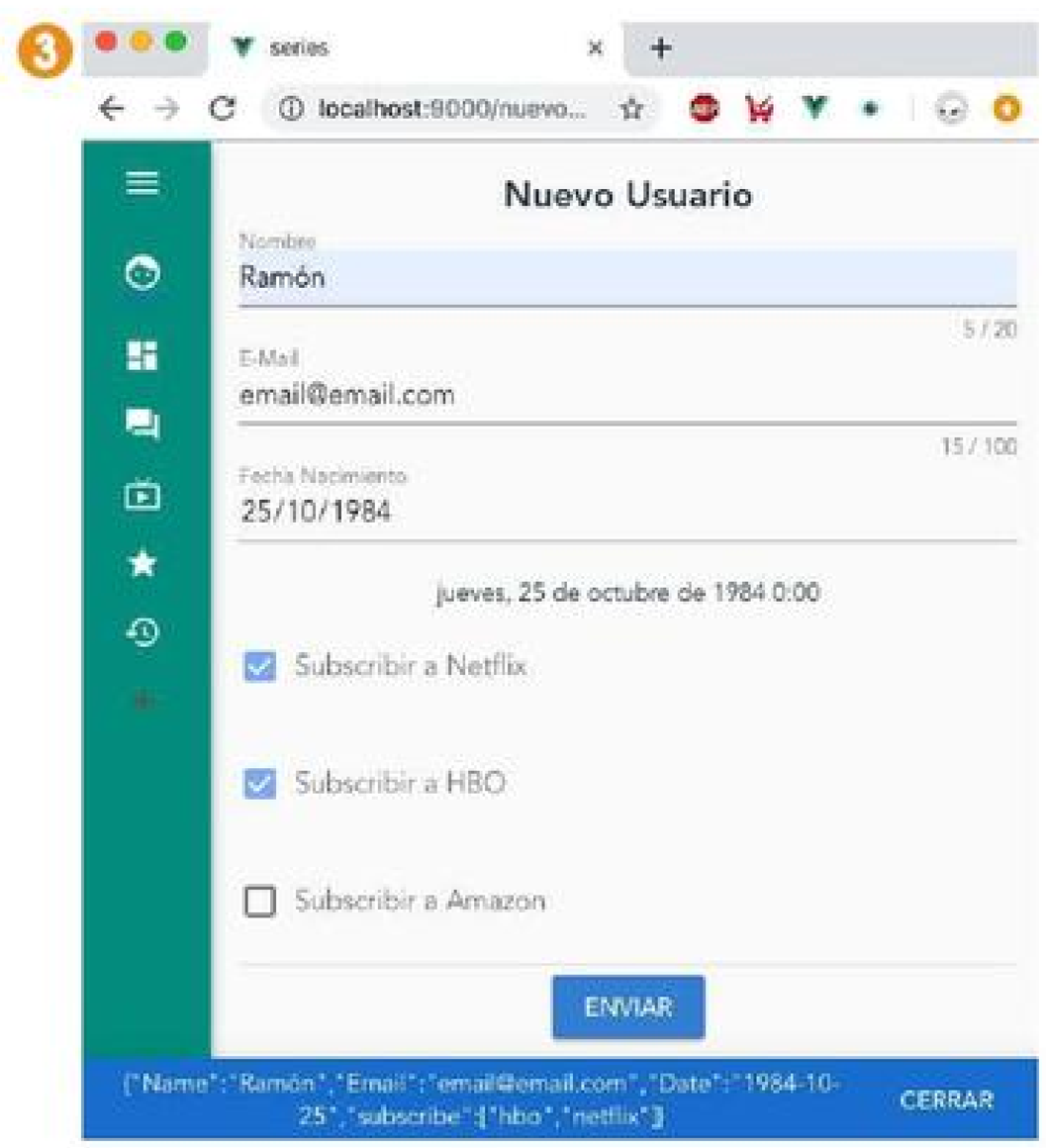
```
subscribe: [ 'netflix' ]
```

A nivel de código, añadimos este array al modelo del formulario en el data del componente, estableciendo la opción HBO, marcada por defecto al cargar el formulario:

```
, model: { Name: null, Email: null, Date: null, subscribe: [ 'hbo' ] }
```

Finalmente, si abrimos la ficha del formulario aparecerá marcada por defecto la opción "Subscribir a HBO" ❷, debido a que así lo establecimos en el modelo mediante "subscribe: ['hbo']". Si marcamos otra opción, este array contendrá dos valores por lo que, si enviamos los datos, veremos estos dos valores enviados ❸.





Formularios – Opción única Radiobutton

IMPORTANTE

Podemos agrupar las opciones forma horizontal o vertical, únicamente deberemos definir en la agrupación "v-radio-group" la propiedad "column" para vertical y "row" para horizontal.

RadioButton

Cuando necesitamos seleccionar una única opción dentro de un conjunto de valores posibles reducido, lo más conveniente es utilizar los "radiobutton", que podemos agrupar en grupos de radio a su vez.

Definamos pues dos opciones de selección única "v-radio", en la plantilla del componente "FichaUsuario.vue" ❶, incluyéndolos dentro de un grupo mediante "v-radio-group":

```
<v-layout row wrap> <v-flex md6>
    <v-radio-group v-model="model.sex" row>
        <v-radio label="Hombre" value="man"></v-radio>
        <v-radio label="Mujer" value="women"></v-radio>
    </v-radio-group>  </v-flex> … </v-layout>
```

En esta ocasión será el grupo quien vincule mediante "v-model" el atributo del modelo con el valor seleccionado del listado de opciones. A su vez, cada radio tendrá declarado el valor que lo identifica y será vinculado con el modelo.

Dentro de este layout que contiene 6 columnas para representar

❶

```
<v-form ref="form">
  <v-text-field name="Name" label="Nombre" v-model="model.Name"></v-text-field>
  <v-text-field name="Email" label="E-Mail" v-model="model.Email"></v-text-field>
  <v-text-field name="Date" label="Fecha Nacimiento" v-model="model.Date"></v-text-field>
{{fechaNacimiento}}

<v-checkbox v-model="model.subscribe" label="Subscribir a Netflix" value="netflix"></v-checkbox>
<v-checkbox v-model="model.subscribe" label="Subscribir a HBO"     value="hbo"></v-checkbox>
<v-checkbox v-model="model.subscribe" label="Subscribir a Amazon"  value="amazon"></v-checkbox>

<v-layout row wrap>                                  Grupo de radio opciones
      <v-flex md6> <v-radio-group v-model="model.sex" row>
                    <v-radio label="Hombre" value="man"></v-radio>          Opción Radio
                    <v-radio label="Mujer" value="women"></v-radio>
                  </v-radio-group> </v-flex>
      <v-flex md6 v-if="model.sex">
                  <v-btn color="info" @click="model.sex=null">Deseleccionar</v-btn> </v-flex>
</v-layout>

<v-divider></v-divider>

<v-btn color="primary"  @click.prevent="sendForm">Enviar</v-btn>
```

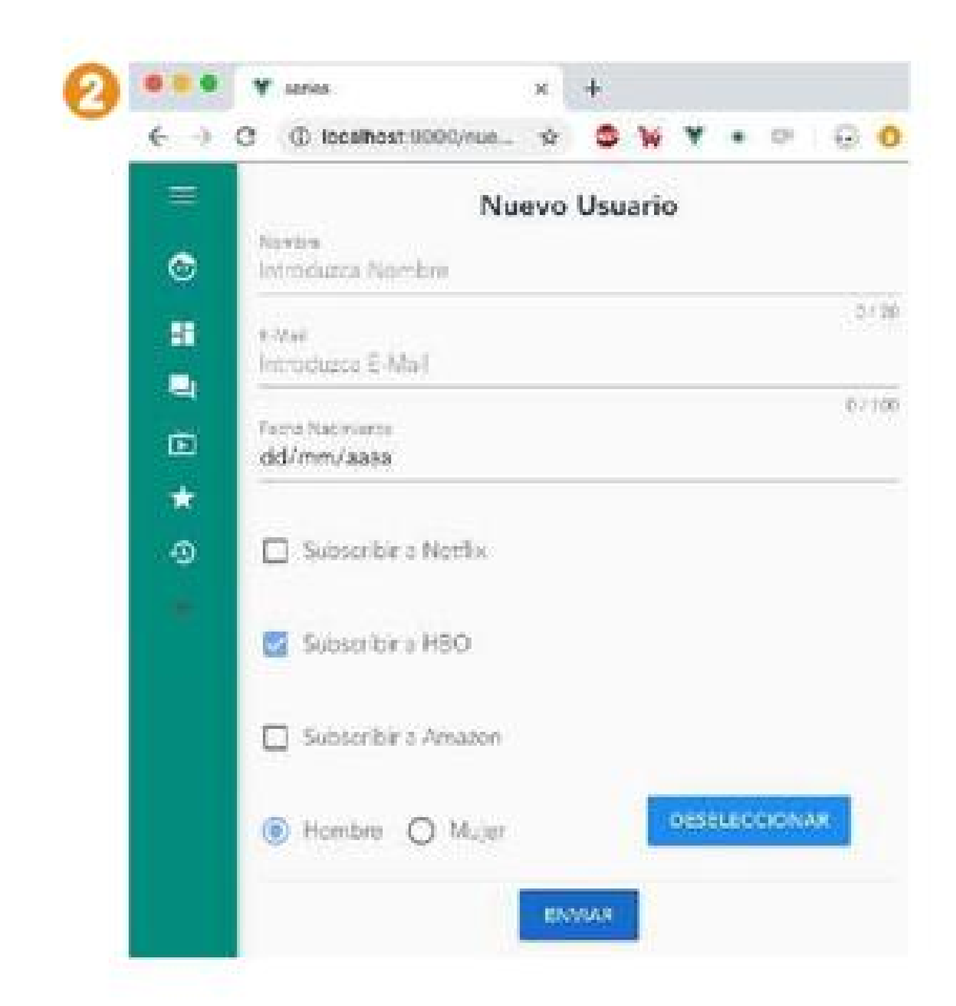
las opciones, añadimos otras 6 columnas a la derecha para que, en caso de tener un valor seleccionado el "radiobutton", se muestre el botón "Deseleccionar" para limpiar la selección de este radio, esto es útil debido a que una vez tengamos una opción seleccionada, no podemos deseleccionar.

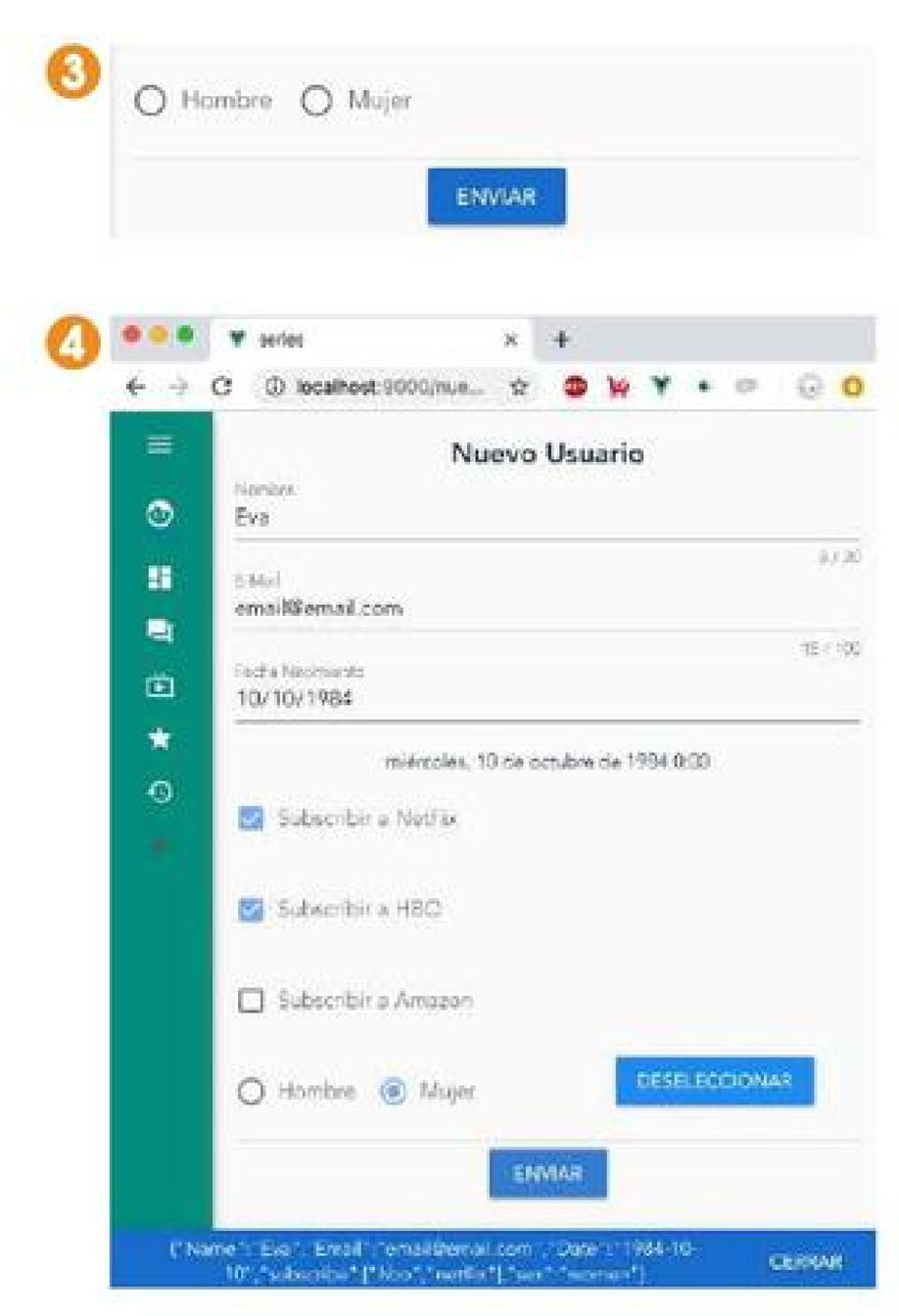
```
<v-flex md6 v-if="model.sex"> <v-btn color="info" @click="model.
sex=null">Deseleccionar</v-btn> </v-flex>
```

De este modo, añadimos en el modelo del data de este componente un atributo adicional llamado "sex" que identifique el valor del sexo seleccionado. En el caso de los "radiobutton", no se almacena un array, debido a que únicamente un valor es el seleccionado. Marquemos la opción hombre como el valor por defecto a marcar en el formulario:

```
, model:{ Name: null, Email: null, Date: null, subscribe: ['hbo'], sex: 'man' }
```

Si abrimos la ficha de introducción de nuevo usuario aparecerá seleccionada la opción hombre, del "radiobutton" ❷, así como el botón de borrar selección. De forma que, si borramos la selección pulsando este botón, la opción seleccionada en el radio se borrará, desapareciendo el botón "Deseleccionar" a su vez ❸. Rellenamos el formulario y, si pulsamos enviar, podremos observar cómo el valor enviado es el seleccionado ❹.





Formularios – Combobox

IMPORTANTE

Para poblar opciones con objetos JSON, en lugar de un array de strings el componente buscaría la propiedad "text" y "value" para asignar las opciones.

Combobox única opción

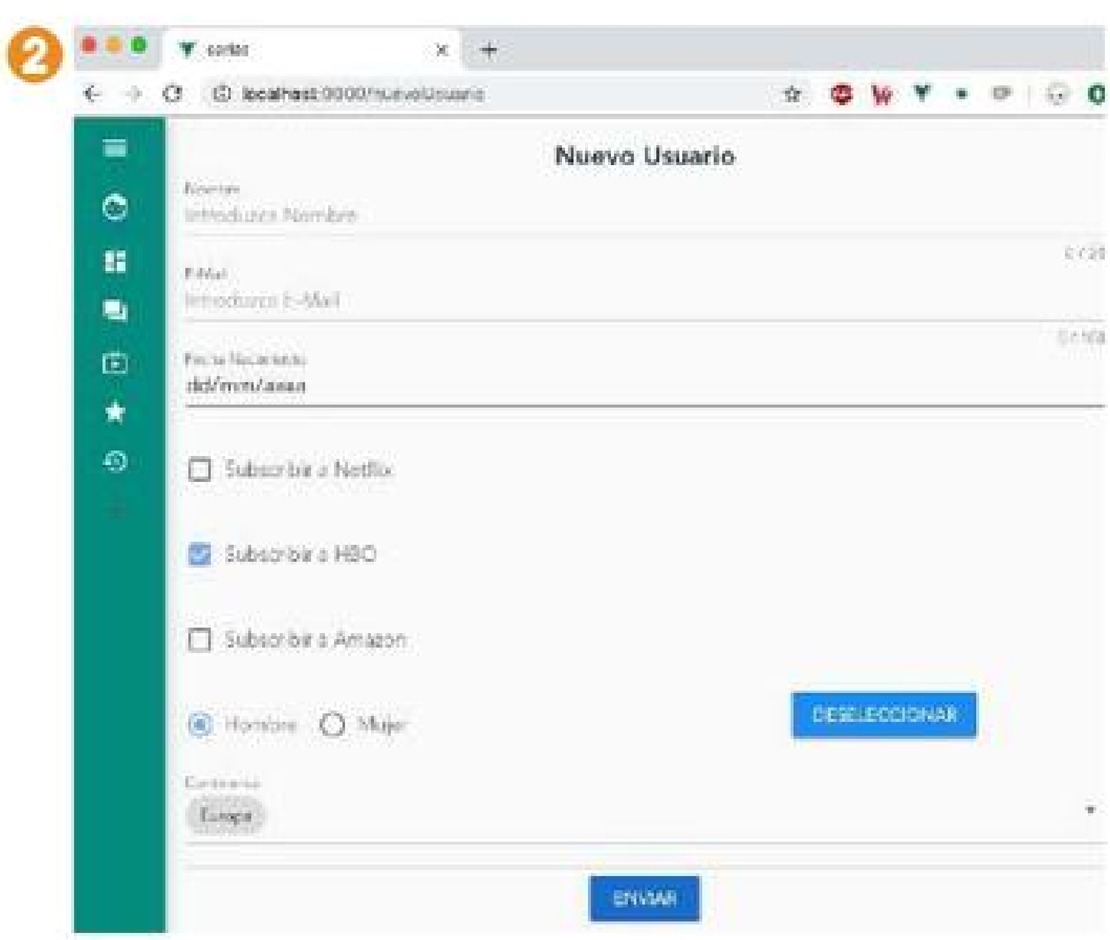
Cuando necesitamos seleccionar una única opción dentro de un conjunto de valores mayor que en los de tipo radio, lo más conveniente es utilizar los "combobox", que nos permite filtrar los resultados posibles mientras escribimos ❶.

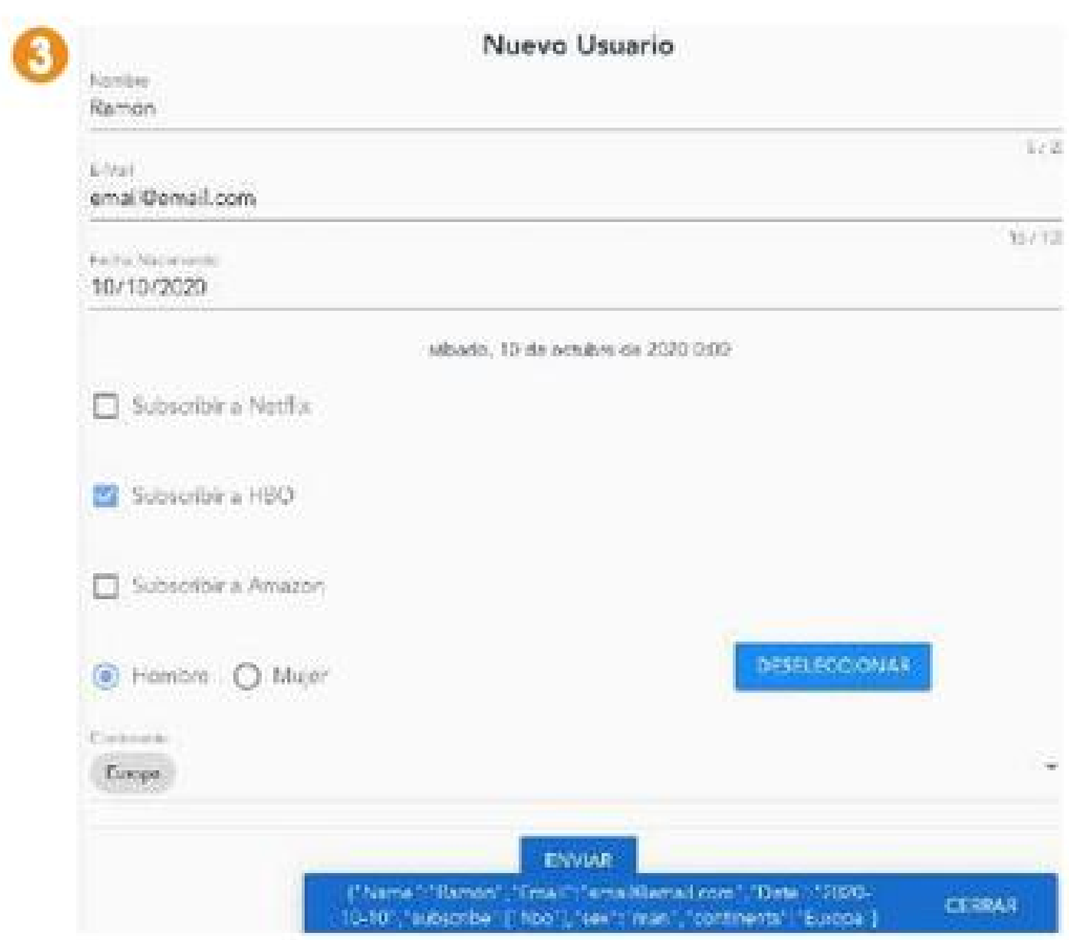
Definamos pues un combo que contenga una lista de continentes, de los que únicamente seleccionaremos uno. Para ello utilizaremos el componente "v-combobox" de Vuetify, en la plantilla del componente "FichaUsuario.vue":

```
<v-combobox                    v-model="model.continents"
label="Continente" chips

        :items="['Asia','América','África','Antártida
','Europa','Oceanía']"></v-combobox>
```

El atributo responsable de poblar las opciones en el combo es "ítems", contiene un array con los nombres de las opciones. Indicamos mediante la propiedad "chips", la forma de representar el valor seleccionado, es decir forma redondeada. Finalmente indicamos en "v-model" el atributo del modelo con el nombre del valor seleccionado.

Añadimos en el modelo del data del este componente el atributo "continents", que identifica el valor del continente seleccionado. En el caso de los combos de opción única, no se almacena un array,





debido a que únicamente un valor es el seleccionado. Marquemos la opción "Europa" como el valor por defecto a marcar en el formulario:

```
, model:{ Name:null, Email:null, Date:null, subscribe:['hbo'], sex:'man',
continents:'Europa' }
```

Si abrimos la ficha de introducción de nuevo usuario, aparecerá seleccionada la opción Europa, del "combobox" ❷. Rellenamos el formulario y, si pulsamos enviar, podremos observar cómo el valor enviado es el seleccionado ❸.

Combobox Múltiple opción

Los combos pueden seleccionar múltiples opciones ❹, para ello únicamente añadimos la propiedad "multiple" en el "v-combobox". Este escenario lo haremos mediante un combo de categorías de películas, para seleccionar varias:

```
<v-combobox v-model="model.categories" label="Categorías" chips multiple
  :items="['Humor','Terror','Amor','Ciencia Ficción','Aventuras','Acción']"></v-
combobox>
```

Añadimos en el modelo del data del componente "FichaUsuario.vue" el atributo "categories", que identifica los valores de las categorías seleccionadas. En el caso de los combos de opción múltiple se almacena un array, marquemos las opciones "Acción" y "Terror" como valores por defecto a marcar en el formulario:

```
, model:{ …, categories: [ 'Acción', 'Terror' ] }
```

Si abrimos la ficha de introducción de nuevo usuario aparecerán seleccionadas las opciones terror y acción, del combo categorías ❺. Si rellenamos el formulario y pulsamos enviar, observamos los valores seleccionados ❻.

Formularios – Campo Personalizado

IMPORTANTE

Debemos tener muy claro el flujo de comunicación entre hijo y padre cuando el componente hijo personalizado controla la lógica, de lo contrario nos perderemos entre eventos.

Campo Dirección

Hasta ahora hemos utilizado campos simples, sin embargo lo más común es crear nuestros campos personalizados, adaptados a nuestras necesidades y obtener los datos desde el componente principal.

Vamos a crear el componente "components/Address.vue", representa una dirección compuesta por un texto, el nombre de la calle, así como dos numéricos, uno para el número y otro para el código postal ❶.

El data de este componente contiene el modelo "model", utilizado por los tres campos, que será inicializado por el valor proporcionado a las props "propModel":

```
,data () { return { model:this.propModel } }
```

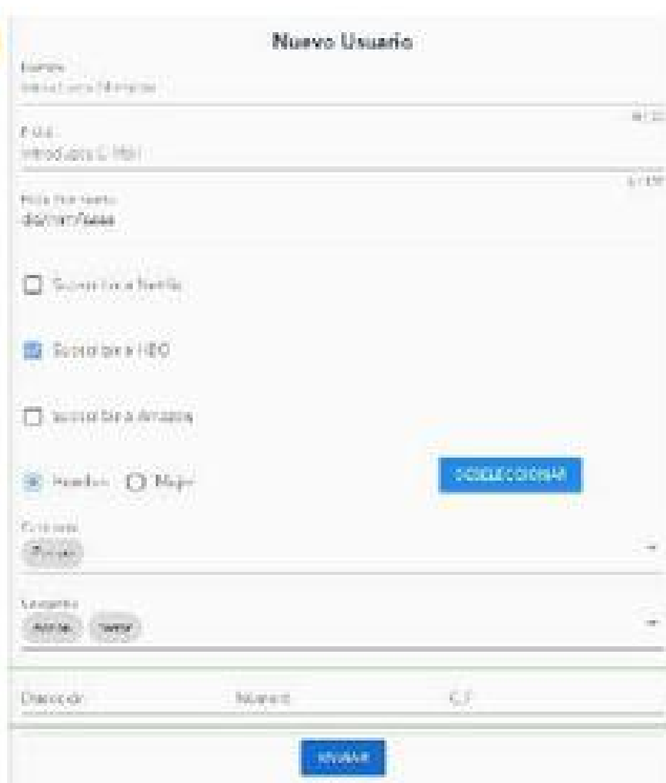
De este modo, cualquier cambio que se produzca en el valor del modelo, será capturado en el "watch", informando a su componente padre "FichaUsuario.vue", emitiendo un evento con el modelo del componente.

```
,watch:{        model(){        this.$emit('address-change',this.model); } }
```

Ahora nos vamos al componente padre "FichaUsuario.vue", y agregamos este nuevo componente en la plantilla:

```
<custom-address        :propModel="model.address"
v-on:address-change="onAddressChange"></custom-address>
```

❷



Mediante "propModel" enviaremos el valor del atributo "address" del modelo del formulario, capturando mediante "v-on" el evento emitido por cambios producidos en el componente dirección, e invocaremos el método "onAddressChange" para recoger el valor que envía el componente dirección y asignarlo al modelo.

Agregamos al data del componente el atributo "address", que contiene el modelo vacío para dicho componente:

```
, model:{ … ,address:{ } }
```

Importamos el componente dirección que acabamos de crear:

```
import Address from '@/components/Address.vue'
```

Así como agregamos a la lista de componentes que utiliza "FichaUsuario.vue" el componente dirección:

```
,components:{ customAddress: Address }
```

Finalmente agregamos el método onAddressChange, para capturar el valor emitido por el componente dirección, que asociaremos al atributo "address" del modelo del formulario.

```
,methods:{ … ,onAddressChange(event) { this.model.address=event; } }
```

Si abrimos la ficha de introducción de nuevo usuario, aparecerá el campo dirección compuesto por sus tres campos ❷. Rellenamos el formulario y, si pulsamos enviar, observamos cómo el valor enviado es el introducido en la dirección ❸.

❶

```
<template>                                                          Componente Dirección
  <v-layout md12 wrap component="customAddress">      Modelo para campo dirección
    <v-flex md6><v-text-field                  v-model="model.address"     label="Dirección">
                </v-text-field></v-flex>                  Modelo para campo número
    <v-flex md3><v-text-field type="number" v-model="model.number"      label="Número">
                </v-text-field></v-flex>                  Modelo para campo codigo postal
    <v-flex md3><v-text-field type="number" v-model="model.postalCode" label="C.P">
                </v-text-field></v-flex>
  </v-layout>
</template>
<script>
export default {
  name: 'CustomAddress'        Modelo del componente, inicialmente cargado por el modelo proporcionado a las props
  ,data () { return { model:this.propModel } }
  ,props:{ propModel:{ type:Object, default() { return {} } } }
  ,watch:{ model(){ this.$emit('address-change',this.model); } }
}                              Cualquier cambio que se produzca en el modelo del componente, se envia al padre por evento
</script>
```

❸

Formularios – Validación Vuelidate

IMPORTANTE

Cuando creamos un proyecto desde cero es conveniente utilizar Vuelidate, ya que nos proporciona unas validaciones base, que podemos ir enriqueciendo.

VUELIDATE

Una alternativa a realizar las validaciones manualmente mediante el atributo "rules" en los campos, podemos utilizar la librería "Vuelidate", puede facilitarnos la vida mucho, ya que posee un gran número de validaciones a utilizar.

Para utilizarla instalamos el paquete de "Vuelidate":

```
npm install vuelidate -save
```

Tras su instalación, veremos en "package.json" que la dependencia está agregada:

```
"dependencies": {…    "vuelidate": "^0.7.4", … },
```

Para agregar globalmente el uso de validaciones mediante esta librería en todo el proyecto, agregamos en "main.js":

```
import Vuelidate from 'vuelidate'

Vue.use(Vuelidate)
```

Estamos listos para utilizarla dentro de nuestro componente "FichaUsuario.vue", empecemos por importar algunas de las validaciones que utilizaremos de esta librería en nuestro componente:

```
      import { required, minLength, email, between
} from 'vuelidate/lib/validators'
```

Asignemos en la instancia vue del componente algunas validaciones, indicándolo en la propiedad "validations" de la instancia, la cual contiene aquellas validaciones a aplicar en determinados campos del modelo:

```
,validations:{ model:{ Name: { required, minLength:
minLength(3) }

      , Email: { email }

      , address: { postalCode: { between: between(1000,
52999) } }  }  }
```

- Validaremos en el campo nombre ❶, ser requerido (required) y con longitud mínima de 3 (minLength), mostrando mediante "$v.AtributoModelo.Validacion" un mensaje en caso de error de validación.
- Validaremos el campo email ❷, para que únicamente se permita introducir emails válidos, para ello eliminaremos la an-

terior validación que implementamos en ejercicios anteriores, sobre "rules".

- Validaremos el atributo código postal del campo dirección ❸, para que muestre error en caso de que el código introducido no esté comprendido entre (between) 1000 y 52999.
- Finalmente añadimos al componente un estilo para que los mensajes de error se muestren en rojo:

```
<style scoped> .invalid { line-height: 1; color: #ff5252 !important; } </style>
```

Si abrimos la ficha de introducción de nuevo usuario, aparecerá el mensaje de que nombre es un campo requerido. Si escribimos 2 caracteres en dicho nombre, el mensaje de error desaparecerá, mostrando uno nuevo, al no cumplir la validación de longitud mínima. Si introducimos un email no válido, se indicará mediante un mensaje. Finalmente, si en la dirección introducimos un código postal cuyo valor no está entre 1000 y 52999, se mostrará un mensaje.

❶
```
<v-text-field name="Name" label="Nombre" v-model="model.Name"
        :maxlength="10" :counter="20" placeholder="Introduzca Nombre"></v-text-field>
<div v-if="!$v.model.Name.required" class="invalid">Nombre es requerido</div>
<div v-if="!$v.model.Name.minLength" class="invalid">Nombre longitud mínima es 3</div>
```

Textos a mostrar cuando se produce un error de validacion

❷
```
<!--:rules="[rules.email]"-->
<v-text-field name="Email" label="E-Mail" v-model="model.Email"
        :maxlength="100" :counter="100" placeholder="Introduzca E-Mail"
        ></v-text-field>
<div v-if="!$v.model.Email.email" class="invalid">No es un email válido</div>
```

Comentamos la anterior validación de rules que implementamos

En caso de fallar la validación de email, se muestra el mensaje

❸
```
<custom-address :propModel="model.address" v-on:address-change="onAddressChange"></custom-address>
<div v-if="!$v.model.address.postalCode.between" class="invalid">Código postal debe estar entre
  {{$v.model.address.postalCode.$params.between.min}} y
  {{$v.model.address.postalCode.$params.between.max}}</div>
```

Mensaje a mostrar cuando el valor no está comprendido entre un rango de valores

Formularios – Validación personalizada

IMPORTANTE

Cuando las validaciones proporcionadas por Vuelidate no sean suficiente, personalícelas libremente definiendo sus condiciones.

Vuelidate Personalizado

Una vez conocidas las validaciones que proporciona Vuelidate, vamos a aprender cómo crear una nueva validación personalizada cuando la librería no puede proporcionárnosla.

Crearemos una validación que verifique si la fecha de nacimiento introducida por el usuario en el campo fecha del formulario está entre dos fechas.

En el componente "FichaUsuario.vue", creamos una constante llamada "mustBeDateBetween", que recibirá dos parámetros, el primero para la mínima fecha válida y el segundo para la máxima fecha aceptada. De modo que una función javascript, que recibe por el parámetro "value" el valor del campo a validar, y por "component", la instancia del componente vue, que retornará el valor booleano de una condición, en la que verificamos mediante la librería "momentjs" si la fecha enviada por valor, está comprendida o no entre estas.

```
const mustBeDateBetween = (min, max) => {
return (value, component) => ( typeof moment===' function' && moment(value) >= moment(min) && moment(value) <= moment(max)) }
```

Añadimos en las validaciones del modelo el atributo "Date" a validar, cuya validación será "mustBeDateBetween", a la cual enviaremos como fecha mínima válida los nacidos en 1900 y como máxima, los nacidos en la fecha actual.

```
,validations:{ model:{ …,Date:{ mustBeDateBetween: mustBeDateBetween('01-01-1900',new Date()) } } }
```

Si ahora abrimos el formulario de ficha de usuario e intentamos introducir una fecha de nacimiento superior o inferior a la fecha válida, indicará que es una fecha errónea ❶.

Validar formulario

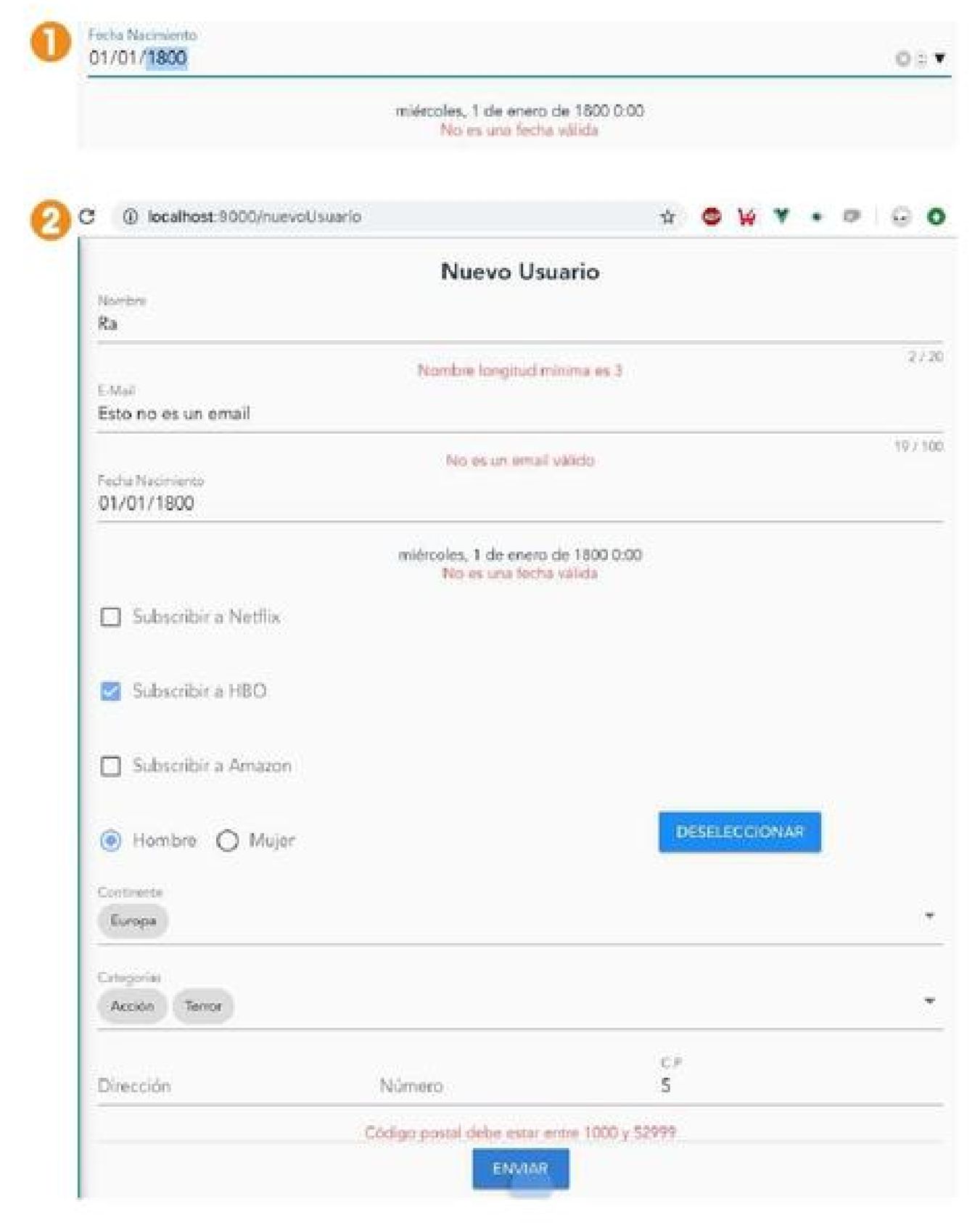
Vuelidate pone a nuestra disposición una serie de métodos a utilizar para comprobar si el formulario es válido o no. Este método es "this.$v.$invalid", permitiéndonos no enviar el formulario en caso de que alguna validación fallara.

Otro método es "$v.$touch", que nos permitirá comprobar si el usuario tocó el campo que contiene validación para seguir mostrando el mensaje de error o no, de modo que para que continúe mostrándose el error, lo utilizaremos.

Por tanto, vamos a modificar el método "sendForm" de este componente para verificar si alguna validación de Vuelidate ha fallado, para evitar enviar el formulario.

```
,methods:{ sendForm(){ this.$v.$touch();
   if(!this.$v.$invalid       &&this.$refs.form.validate())       this.GlobalBus.
showSuccess(JSON.stringify(this.model)) }    }
```

Si ahora abrimos el formulario de ficha de usuario e intentamos errar en cualquiera de los campos de Vuelidate, podemos observar que aparecerán los errores y el formulario no se enviará ❷.



Pruebas Unitarias – Jest

IMPORTANTE

Una buena práctica es declarar las pruebas unitarias en una misma carpeta, alternativamente podría incluirse dentro de la carpeta del componente a validar.

Instalar Jest

Toda fase de desarrollo debe incluir pruebas para validar el código. Uno de estos frameworks basado en Javascript es "Jest", fácilmente integrable en proyectos Vue. Para utilizar "Jest" tenemos varias opciones, creando un nuevo proyecto, incluyendo la opción "unit testing" o partiendo de un proyecto sin "Jest". Para el último caso ejecutamos:

```
vue add @vue/unit-jest
```

Adicionalmente instalaremos la librería de pruebas unitarias para Vue:

```
npm install --save-dev @vue/test-utils
```

Tras esto se habrá agregado la configuración de jest al fichero "jest.config.js", como las extensiones a interpretar (moduleFileExtensions), o la resolución de rutas que contienen "@" en lugar de "src" (moduleNameMapper).

Al añadir jest al proyecto se agrega una prueba de ejemplo "example.spec.js" ❶ a la carpeta "tests/unit", contiene:

- describe: identifica un bloque de pruebas unitarias, que utilizan un mismo contexto.
- shallowMount: se utiliza para cargar el componente sobre el que ejecutar las pruebas.
- it: identifica una prueba unitaria con un título y una función flecha con la prueba a ejecutar.
- wrapper: componente cargado y montado.
- expect: ejecución de una condición que valide la prueba unitaria.

Ejecutar Prueba Unitaria

Para ejecutar toda prueba unitaria existente en el proyecto ❷ de las carpetas donde ubicamos los test, como se ha definido en la propiedad "testMatch" del fichero "jest.config.js", es decir, en las carpetas "tests/unit" o "__tests__":

```
testMatch:                    ['**/tests/unit/**/*.spec.
(js|jsx|ts|tsx)|**/__tests__/*.(js|jsx|ts|tsx)']
```

Empleamos la opción de Jest "watchAll", creando un nuevo co-

mando en el fichero "package.json" para utilizarlo:

```
"scripts":{ …, "test:unit": "vue-cli-service test:unit --watchAll",…}
```

De esta forma, toda prueba unitaria será ejecutada al invocar el comando anterior:

```
npm run test:unit
```

Podemos observar cómo se ha encontrado una prueba, llamada "example.spec.js", y el resultado ❸ es satisfactorio.

Para ejecutar un test en concreto definiremos en "package.json", un nuevo comando al que añadiremos la opción "--testPathPattern" seguida del nombre que debe contener el test para ejecutarlo:

```
"test:single":"vue-cli-service test:unit --testPathPattern=example"
```

Por lo que, si invocamos el nuevo comando, se ejecutará todo test que contenga el nombre "example" ❹.

❶

```
Importar las Funciones a utilizar en pruebas unitarias
import { shallowMount } from '@vue/test-utils'
import HelloWorld from '@/components/HelloWorld.vue'
Importamos componente que queremos probar
describe('HelloWorld.vue', () => {
  it('renders props.msg when passed', () => {
    const msg = 'new message'
    const wrapper = shallowMount(HelloWorld, {
      propsData: { msg }        Cargar componente
    })
                                Prueba unitaria, validación
    expect(wrapper.text()).toMatch(msg)
  })
})
Test
Bloque que agrupa varios tests y utiliza un mismo contexto
```

❸

```
PASS  tests/unit/example.spec.js
  HelloWorld.vue
    ✓ renders props.msg when passed (24ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        8.294s
Ran all test suites.

Watch Usage: Press w to show more.
```

❹

```
mbp-de-ramon:series Eclipselunar$ npm run test:single

> series@0.1.0 test:single /Users/Eclipselunar/DeveloperApplications/100EjerciciosVue/VueTest/series
> vue-cli-service test:unit --testPathPattern=example

 PASS  tests/unit/example.spec.js
  HelloWorld.vue
    ✓ renders props.msg when passed (22ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        3.604s
Ran all test suites matching /example/i.
```

Pruebas Unitarias – Creación

IMPORTANTE

Para cargar un componente es preferible "shallowMount" ya que representa el componente sin montar los componentes hijos, a "mount", que sí carga componente e hijos.

Prueba unitaria

Partiendo del componente "components/DetailSerie.vue"❶ definido en ejercicios anteriores, deseamos verificar que se carga correctamente. Recordemos que este componente se encargaba de pintar por pantalla la ficha detalle de una serie, que recibía por parámetro en las "props" la información necesaria para representar la serie.

Vamos a crear por tanto una nueva prueba unitaria de este componente, para ello añadimos en la carpeta "tests/unit" un nuevo fichero llamado "detailSerie.spec.js" ❷.

Lo primero que importamos es la función de carga de componentes "shallowMount" de la librería de utilidades de test, así como el componente que deseamos probar.

Posteriormente definimos en un bloque "describe" al que le asociamos un título para agrupar todas las pruebas unitarias asociadas al bloque, que utilizan un mismo contexto:

```
describe('Pruebas Componente DetailSerie',
() => {…})
```

Antes de ejecutar cualquiera de las pruebas unitarias en "beforeEach()", que definiremos en este bloque "describe()", vamos a obtener el componente a verificar, montado en la variable

❶

```
<template>
    <v-card v-if="propSerie">
      <v-layout row wrap style="text-align:center">
        <v-flex md3>{{propSerie.Cadena}}</v-flex>
        <v-flex md6 class="underline"><h3>{{propSerie.Titulo}}</h3></v-flex>
        <v-flex md3>{{propSerie.Anyo}},T[{{propSerie.Temporadas}}]</v-flex></v-layout>
      <v-layout row wrap> <v-flex md12>{{propSerie.Descripcion}}</v-flex> </v-layout>
      <v-layout row wrap>
        <v-flex md12> <actions-detail></actions-detail> </v-flex>
      </v-layout>
    </v-card>
    <span v-else>Seleccione un elemento para mostrar detalle.</span>
</template>
<script>
import ActionsDetail from '@/components/ActionsDetail.vue'
export default { name: 'detail-serie', props:{ propSerie:Object }
                 ,components:{ actionsDetail: ActionsDetail } }
</script>
<style scoped>
  .v-card{color:black !important; font-weight: normal;}
  .underline{text-decoration:underline}
</style>
```

"wrapper", mediante "shallowMount":

```
beforeEach(()=>{ wrapper = shallowMount(DetailSerie,{}) console.
log(wrapper.html()) })
```

El primer parámetro de "shallowMount", es el componente para montar, y el segundo son las opciones para enviarle al componente. Una vez asignado el wrapper, mostramos el contenido html tras cargar el componente.

Posteriormente, definimos la prueba unitaria que queremos ejecutar, dentro de la función flecha de "it()":

```
it('Se representa el componente vista sin errores', ()=>{…})
```

Será en esta función flecha donde evaluaremos una aserción, empleando para ello "expect", seguido de un valor, en este caso el valor es si el wrapper que debería contener la instancia del componente montado, es una instancia Vue. En caso afirmativo, lo validaremos mediante la aserción "toBeTruthy", que dará validez a la aserción o no.

```
expect(wrapper.isVueInstance()).toBeTruthy()
```

Si ejecutamos las pruebas mediante el comando "npm run unit:test" veremos que aparecen dos pruebas unitarias a ejecutar, la anterior prueba por defecto y la nueva que acabamos de crear llamada "detailSerie". Ambas han pasado satisfactoriamente su prueba. En el caso de la segunda, "detailSerie" se muestra el contenido html del "v-else" de la plantilla del componente, debido a no haber proporcionado un valor a la propiedad "propSerie" ❸.

❷
```
import { shallowMount } from '@vue/test-utils'
import DetailSerie from '@/components/DetailSerie.vue'

describe('Pruebas Componente DetailSerie', () => {
  let wrapper
  beforeEach(()=>{
    wrapper = shallowMount(DetailSerie,{})
    console.log(wrapper.html())
  })
  it('Se representa el componente vista sin errores', ()=>{
    expect(wrapper.isVueInstance()).toBeTruthy()
  })
})
```

❸
```
PASS  tests/unit/example.spec.js
PASS  tests/unit/detailSerie.spec.js
  ● Console
```

Console.log muestra la condición v-else de la plantilla

```
console.log tests/unit/detailSerie.spec.js:8
  <span>Seleccione un elemento para mostrar detalle.</span>

Test Suites: 2 passed, 2 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        9.716s
Ran all test suites.
```

Pruebas Unitarias – Props

IMPORTANTE

Añadir clases, identificadores o atributos como "test" a los elementos por localizar agiliza la fase de búsqueda para comparar valores de los elementos en el test.

PropsData

Los componentes suelen tener parámetros de entrada definidos por las props que pueden condicionar su comportamiento. Por tanto, quizás queramos analizar el comportamiento del componente en base a unas entradas.

Creemos el test "detailSerieProps.spec.js" para validar la representación de una serie proporcionada al componente "DetailSerie.vue" ❶ a través de su propiedad "propSerie". Añadiendo en la plantilla de este componente el atributo "test='description'" al elemento descripción de la serie para localizar el elemento en la fase de testing.

En el test que acabamos de crear ❷, importamos la función "shallowMount" y el componente a validar. En el bloque "describe", creamos el modelo a proporcionar al componente, variable "model".

Antes de ejecutar cualquier prueba, montamos el componente y le proporcionamos mediante el atributo "propsData" la propiedad "propSerie" al componente, con el valor del modelo a representar.

```
beforeEach(()=>{ wrapper = shallowMount(DetailSerie,{
propsData: { propSerie: model } }) })
```

La primera prueba unitaria será comprobar la carga del componente, como hicimos en el ejercicio anterior.

❶

```
<template>
    <v-card v-if="propSerie">
      <v-layout row wrap style="text-align:center">
        <v-flex md3>{{propSerie.Cadena}}</v-flex>
        <v-flex md6 class="underline"><h3>{{propSerie.Titulo}}</h3></v-flex>
        <v-flex md3>{{propSerie.Anyo}},T[{{propSerie.Temporadas}}]</v-flex></v-layout>
      <v-layout row wrap> <v-flex md12 test="description">{{propSerie.Descripcion}}
                          </v-flex> </v-layout>   Propiedad para encontrar elemento Descripción
      <v-layout row wrap>
        <v-flex md12> <actions-detail></actions-detail> </v-flex>
      </v-layout>
    </v-card>
    <span v-else>Seleccione un elemento para mostrar detalle.</span>
</template>
<script>
import ActionsDetail from '@/components/ActionsDetail.vue'
export default { name: 'detail-serie', props:{ propSerie:Object }
                 ,components:{ actionsDetail: ActionsDetail } }
</script>
```

La segunda contendrá dos validaciones, por un lado buscaremos en la plantilla del componente el elemento título contenido en una etiqueta "<h3>" precedido por la clase "underline". Filtramos los elementos mediante el "find", una vez encontrado obtenemos su valor mediante "text" comparándolo mediante "toBe" con el "Título" del modelo.

```
expect(wrapper.find('.underline h3').text()).toBe(model.Titulo)
```

De igual manera, verificamos si el texto de la descripción de la plantilla es idéntico al valor de esta propiedad del modelo, realizando la búsqueda por el atributo "test", añadido anteriormente para rápida localización.

```
expect(wrapper.find('[test="description"]').text()).toBe(model.Descripcion)
```

Si ejecutamos las pruebas se encuentran 3 ficheros de pruebas, que contienen 4 pruebas superadas en total ❸, son 4 debido a que el último fichero contenía dos pruebas.

❷

```
import { shallowMount } from '@vue/test-utils'
import DetailSerie from '@/components/DetailSerie.vue'

describe('Pruebas Componente DetailSerie con Props', () => {
  let wrapper
  let model={ "Id": 1, "Anyo": 2018, "Cadena": "Netflix", "Temporadas": 10
              ,"Descripcion": "Descripción 1", "Titulo": "Título 1" }
  beforeEach(()=>{
    wrapper = shallowMount(DetailSerie,{ propsData: { propSerie: model } })
    console.log(wrapper.html())
  })
  it('Se representa el componente vista sin errores', ()=>{
    expect(wrapper.isVueInstance()).toBeTruthy()
  })

  it('Buscamos si existen el título y descripción', ()=>{
    expect(wrapper.find('.underline h3').text()).toBe(model.Titulo)
    expect(wrapper.find('[test="description"]').text()).toBe(model.Descripcion)
  })
})
```

❸

```
console.log tests/unit/detailSerieProps.spec.js:10
    <v-card><v-layout row="" wrap="" style="text-align: center;"
><v-flex md3="">Netflix</v-flex> <v-flex md6="" class="underline">
<h3>Titulo 1</h3></v-flex> <v-flex md3="">2018,T[10]</v-flex></v-l
ayout> <v-layout row="" wrap=""><v-flex md12="" test="description"
>Descripción 1
                          </v-flex></v-layout> <v-layout row=""
wrap=""><v-flex md12=""><actions-detail-stub></actions-detail-stub
></v-flex></v-layout></v-card>


Test Suites: 3 passed, 3 total
Tests:       4 passed, 4 total
Snapshots:   0 total
Time:        10.619s
Ran all test suites.
```

Pruebas Unitarias – Aislar componentes

IMPORTANTE

Para aislar un componente automáticamente sin utilizar stubs, montaremos el componente mediante shallowMount, en lugar de mount.

Vuetify

Al utilizar Vuetify necesitamos resolver las etiquetas de los componentes de diseño, de lo contrario nuestras pruebas contendrán un gran número de advertencias que pueden inducir errores, si estamos evaluando el DOM.

Si ejecutamos las pruebas existentes en el proyecto observamos advertencias ❶. Para resolverlas incluimos Vuetify en la configuración global de Jest. Para ello, abrimos el fichero "jest.config.js", y agregamos la configuración:

```
module.exports = { … ,setupTestFrameworkScriptFile:
'<rootDir>/tests/jest-setup.js' }
```

En el directorio "tests" crearemos un fichero llamado "jest-setup.js" donde incorporemos la configuración global de Jest. Importaremos vue, así como vuetify, asignada a vue.

```
import Vue from 'vue';

import Vuetify from 'vuetify';

Vue.use(Vuetify);
```

Si volvemos a ejecutar los tests observaremos cómo ya no aparecen advertencias de este tipo en la consola ❷.

Aislamiento

Gracias a los "stubs" podemos aislar la ejecución de pruebas de un componente de sus componentes hijos. Creemos la prueba "seriesStubs.spec.js" ❸ para aislar el componente "Series.vue" de su componente hijo "Tabla.vue".

Al utilizar vuex el componente "Series.vue" a probar, importamos

esta librería y creamos una instancia vue "localVue" para que la use. Además importamos tanto el componente serie como el componente tabla.

Vuex requiere un store, que nosotros inicializamos vacío, dado que no falsearemos el estado del componente:

```
let store = new Vuex.Store({})
```

Antes de ejecutar pruebas, montamos el componente "Series.vue" con la instancia local vue, el almacén y el atributo "stubs", donde indicamos cada componente a aislar. Será aquí donde ignoremos el componente Tabla.

```
beforeEach(()=>{ wrapper = mount( Series, { localVue, store, stubs:{ Tabla: true } }) })
```

Finalmente se verifica si el componente tabla existe o no dentro de Series.

```
expect(wrapper.find(Tabla).exists()).toBeTruthy()
```

Si ejecutamos la prueba es válida 4 y aparece el componente hijo pero solo a nivel de etiqueta, sin representar. Si queremos que se represente el hijo, bastará con cambiar el valor del stub de la tabla a falso.

3

```
import Vuex from "vuex"
import { mount, createLocalVue } from '@vue/test-utils'
import Series from '@/views/Series.vue'          Componente Padre a probar
import Tabla from '@/components/Tabla.vue';      Componente Hijo

const localVue = createLocalVue()                Series utiliza Vuex, lo incorporamos
localVue.use(Vuex)
let store = new Vuex.Store({})

describe('Pruebas Componente Series', () => {
  let wrapper
  beforeEach(()=>{                               stubs->Aislar Componente hijo (Tabla)
    wrapper = mount(Series,{localVue,store, stubs:{ Tabla:false } })
    console.log(wrapper.html())
  })
  it('Se representa el componente vista sin errores', ()=>{
    expect(wrapper.find(Tabla).exists()).toBeTruthy()
  })                 Verificar si existe el componente hijo o no
})
```

4

```
PASS tests/unit/seriesStubs.spec.js
 ● Console

    console.error node_modules/vuex/dist/vuex.common.js:378
      [vuex] unknown mutation type: series_MUTATE_ALL_SERIES
    console.error node_modules/vuex/dist/vuex.common.js:412
      [vuex] unknown action type: series_ACTION_GET_ALL_SERIES
    console.error node_modules/vuex/dist/vuex.common.js:888
      [vuex] unknown getter: series_ALL_SERIES
    console.log tests/unit/seriesStubs.spec.js:14
      <div class="series"><h1>Series</h1> <tabla-stub proprows="" propcolumns="[object
Object],[object Object],[object Object],[object Object],[object Object]"></tabla-stub>
<!----></div>
```

Pruebas Unitarias – Mock de Vuex

IMPORTANTE

El término mock implica falsear un estado o llamada para proporcionarle una entrada falseada al componente, para obtener siempre un mismo comportamiento.

Vuex

Componentes que utilizan Vuex requieren simular el estado y las respuestas de llamadas externas. Un buen candidato que utiliza Vuex es "Series.vue", simulemos su almacén.

Creamos la prueba unitaria "series.spec.js" importando ❶ Vuex, las utilidades de test, así como los componentes "Series.vue" y su componente hijo "Tabla.vue". Además, creamos la instancia local de vue "localVue" y le asignamos usar Vuex. Finalmente creamos el modelo a simular, que será un array de objetos de tipo serie.

Creamos un nuevo grupo de pruebas ❷, antes de ejecutar cada prueba creamos una instancia de Vuex, mockeando el estado de la variable series con el valor del modelo:

```
store = new Vuex.Store( { state: { series: model },…
```

Simulamos el getter de obtener dicho estado:

```
getters: { [keyNames.ALL_SERIES]: state => { return
state.series; } },
```

Simulamos las mutaciones, para evitar que el componente series, vacíe la lista de series inicializadas, empleando la función de "Jest.fn()" para falsear la llamada:

```
mutations:{[keyNames.MUTATE_ALL_SERIES]: jest.fn() },
```

Finalmente, mockeamos también las acciones:

```
actions= { [keyNames.ACTION_GET_ALL_SERIES]: jest.
fn() }
```

Una vez hemos mockeado el almacén para el componente series,

❶
```
import Vuex from "vuex"
import * as keyNames from '@/keyNames.js'
import {mount,createLocalVue } from '@vue/test-utils'
import Series from '@/views/Series.vue'
import Tabla from '@/components/Tabla.vue';

const localVue = createLocalVue()
localVue.use(Vuex)
```

Instancia Local, donde asignamos Vuex

Modelo que utilizamos para mockear la respuesta de la api

```
let model=[{ "Id": 1, "Anyo": 2018, "Cadena": "Netflix", "Temporadas": 10
            ,"Descripcion": "Descripción 1", "Titulo": "Título 1" }
          ,{ "Id": 2, "Anyo": 2019, "Cadena": "Hbo", "Temporadas": 8
            ,"Descripcion": "Descripción 2", "Titulo": "Título 2" } ]
```

lo montamos con la instancia local y su almacén:

```
wrapper = mount(Series,{localVue,store})
```

Estamos listos para crear varios test, el primero de ellos verifica si el componente Tabla existe:

```
expect(wrapper.find(Tabla).exists()).toBeTruthy()
```

El segundo, verifica si la acción de obtener series se ha invocado, mediante el método "toHaveBeenCalled()":

```
expect(actions[keyNames.ACTION_GET_ALL_SERIES]).toHaveBeenCalled();
```

Que, si recordamos en el componente series, en el método "beforeCreate", se realizaba dicha invocación ❸.

Por último, creamos un tercer test, para obtener las filas de la tabla del componente series:

```
let rows = wrapper.find(Tabla).findAll('tbody>tr').wrappers
```

Donde verificamos si el número de filas encontradas es igual que el número de elementos proporcionados al modelo:

```
expect(rows.length).toBe(model.length)
```

Si ejecutamos los test, veremos cómo son satisfactorios ❹.

❷
```
describe('Pruebas Componente Series', () => {
  let wrapper, store, actions
  beforeEach(()=>{ actions= { [keyNames.ACTION_GET_ALL_SERIES]: jest.fn() }
    store = new Vuex.Store({
      state: {series: model},
      getters: { [keyNames.ALL_SERIES]: state => { return  state.series; } },
      mutations:{[keyNames.MUTATE_ALL_SERIES]: jest.fn() },
      actions }})
    wrapper = mount(Series,{localVue,store})
    console.log(wrapper.html())
  }})
  it('Se representa el componente vista sin errores'
    ,()=>{ expect(wrapper.find(Tabla).exists()).toBeTruthy() })
  it('Se realiza la llamada al método sin errores'
    , ()=>{ expect(actions[keyNames.ACTION_GET_ALL_SERIES]).toHaveBeenCalled(); })
  it('La tabla se rellena con dos filas'
    , ()=>{ let rows = wrapper.find(Tabla).findAll('tbody>tr').wrappers
            expect(rows.length).toBe(model.length) })
})
```

Función de Jest para espiar comportamiento de una función
Almacén de Vuex
Asignamos al estado, el modelo de series
Retornamos el estado mockeado
Ignoramos mutaciones
Mockeamos las acciones de llamar al servidor para obtener las series
Test unitario para comprobar si el componente Tabla existe dentro del padre
Comprobamos si se ha invocado la llamada de obtener series
Buscamos todas las filas de la tabla
Verificamos si el número de filas de la tabla, es igual que el número de elementos del modelo

❸ Componente Series.vue
```
export default {
  name: 'series'
  ,computed:{ ...mapGetters({ rows: keyNames.ALL_SERIES }) }
  ,data(){=
  ,beforeCreate(){
    this.$store.commit(keyNames.MUTATE_ALL_SERIES,[])
    this.$store.dispatch(keyNames.ACTION_GET_ALL_SERIES)
  }
```
Invocar servidor para obtener series

❹
```
Test Suites: 5 passed, 5 total
Tests:       8 passed, 8 total
Snapshots:   0 total
Time:        18.312s
Ran all test suites.
```

Pruebas Unitarias – Router

IMPORTANTE

Utiliza Vue.nextTick() para obtener el resultado en el siguiente pulso de Vue, utilizando el "done()" para indicar el fin de la prueba.

Router

El sistema de rutas de Vue también podemos probarlo, sobre todo nos interesa saber s,i al abrir una ruta, esta cargará el contenido esperado o incluso realizará una redirección.

Desde la vista principal de la aplicación "App.vue", realizaremos la carga de una ruta inexistente, siendo el resultado esperado, la redirección a la página "404.vue" que representa las páginas no encontradas.

Comencemos por crear una nueva prueba "404.spec.js" en el que importaremos ❶ Vuex, el router, la instancia local de Vue, así como los componentes "App.vue", "About.vue" y "NotFound".

Asignamos a la variable "router" las rutas que queramos simular, en este caso son "about" y la redirección de rutas cuando no se encuentre una definida en esta lista de rutas.

Por último incluimos en la instancia de Vue local el uso de Vuex y del router. El motivo de incluir Vuex es debido a que montaremos el componente "App.vue" que contiene "getters" que debemos mockear.

Ahora procedemos a crear las pruebas unitarias dentro del "describe" ❷, para ello antes de ejecutar las pruebas, debemos montar la aplicación, con las rutas especificadas y el almacén de vuex. En este almacén de Vuex, simulamos los getters de "loading" y de "user_loged".

❶

```
import Vuex from "vuex"
import VueRouter from 'vue-router'
import * as keyNames from '@/keyNames.js'
import {mount, createLocalVue } from '@vue/test-utils'
import NotFound from '@/views/404.vue'
import About from '@/views/About.vue'
import App from '@/App.vue'
```

Componentes

Rutas

```
const router = new VueRouter({
  routes: [ { path: '404', name:'noEncontrada', component:NotFound}
            ,{ path: '*', redirect:{ name:'noEncontrada' } }
            ,{ path: '/about', name: 'about', component:About} ] })
```

```
const localVue = createLocalVue()
localVue.use(VueRouter)
localVue.use(Vuex)
```

Instancia Local, cargamos Router y Vuex

```
beforeEach(()=>{ … wrapper = mount(App, {localVue, router, store }) })
```

La primera prueba para realizar es abrir la ruta inexistente "ningunaRutaConocida". Esta incluye el parámetro "done", utilizado para indicar el fin de la prueba. Al tratarse de una acción resuelta en la siguiente iteración de Vue, incluiremos la validación dentro de "nextTick". Allí verificamos si la ruta abierta, utiliza el componente "NotFound".

```
it('Redirigir a 404', done => { router.push('/ningunaRutaConocida')
    localVue.nextTick(()=>{ expect(wrapper.find(NotFound).exists()).toBe(true)     done() }) })
```

Ahora si probamos a abrir otra ruta existente como "about", debería mostrarla:

```
it('Abrir About', done => { router.push({name: 'about'})
    localVue.nextTick(()=>{          expect(wrapper.find(About).exists()).toBe(true)
done() })   })
```

Si ejecutamos las pruebas, deberíamos obtener un resultado satisfactorio ❸, por lo que el resultado de todas las pruebas en conjunto debería serlo también ❹.

❷

```
describe('Pruebas Componente Series', () => {
  let wrapper, store, actions
  beforeEach(()=>{ actions= { [keyNames.ACTION_GET_ALL_SERIES]: jest.fn() }
    store = new Vuex.Store({
      state: {series: model},
      getters: { [keyNames.ALL_SERIES]: state => { return  state.series; } },
      mutations:{[keyNames.MUTATE_ALL_SERIES]: jest.fn() },
      actions })
    wrapper = mount(Series,{localVue,store})
    console.log(wrapper.html())
  })
  it('Se representa el componente vista sin errores'
    ,()=>{ expect(wrapper.find(Tabla).exists()).toBeTruthy() })
  it('Se realiza la llamada al método sin errores'
    , ()=>{ expect(actions[keyNames.ACTION_GET_ALL_SERIES]).toHaveBeenCalled(); })
  it('La tabla se rellena con dos filas'
    , ()=>{ let rows = wrapper.find(Tabla).findAll('tbody>tr').wrappers
            expect(rows.length).toBe(model.length) })
})
```

Función de Jest para espiar comportamiento de una función
Almacén de Vuex
Asignamos al estado, el modelo de series
Retornamos el estado mockeado
Ignoramos mutaciones
Mockeamos las acciones de llamar al servidor para obtener las series
Test unitario para comprobar si el componente Tabla existe dentro del padre
Comprobamos si se ha invocado la llamada de obtener series
Buscamos todas las filas de la tabla
Verificamos si el número de filas de la tabla, es igual que el número de elementos del modelo

❸

```
PASS  tests/unit/404.spec.js (6.076s)
  ● Console
```

❹

```
Test Suites: 6 passed, 6 total
Tests:       10 passed, 10 total
Snapshots:   0 total
Time:        12.955s
Ran all test suites.
```

Pruebas desde UI

Interfaz de usuario

El CLI de Vue nos proporciona una interfaz desde la cual podemos gestionar dependencias del proyecto, distintos modos de compilación del proyecto, así como ejecutar pruebas unitarias existentes.

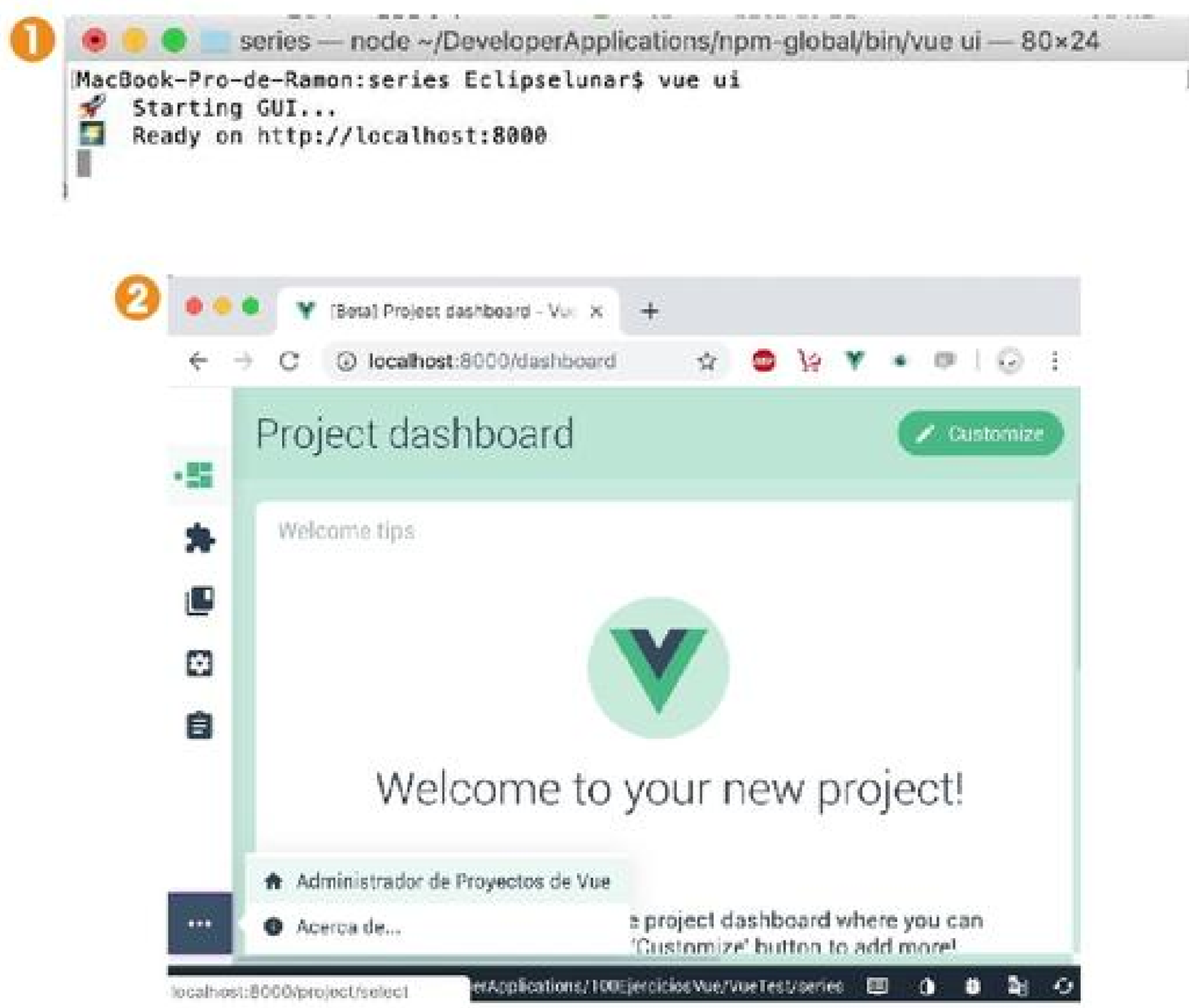
Para abrir esta interfaz de usuario debemos abrir la consola y ejecutar el comando "vue ui" ❶, esto abrirá en el navegador la interfaz del CLI de Vue, donde seleccionamos el proyecto en el menú "Administrador de Proyectos de Vue" ❷. Será aquí donde seleccionemos el proyecto "VueTest" ❸.

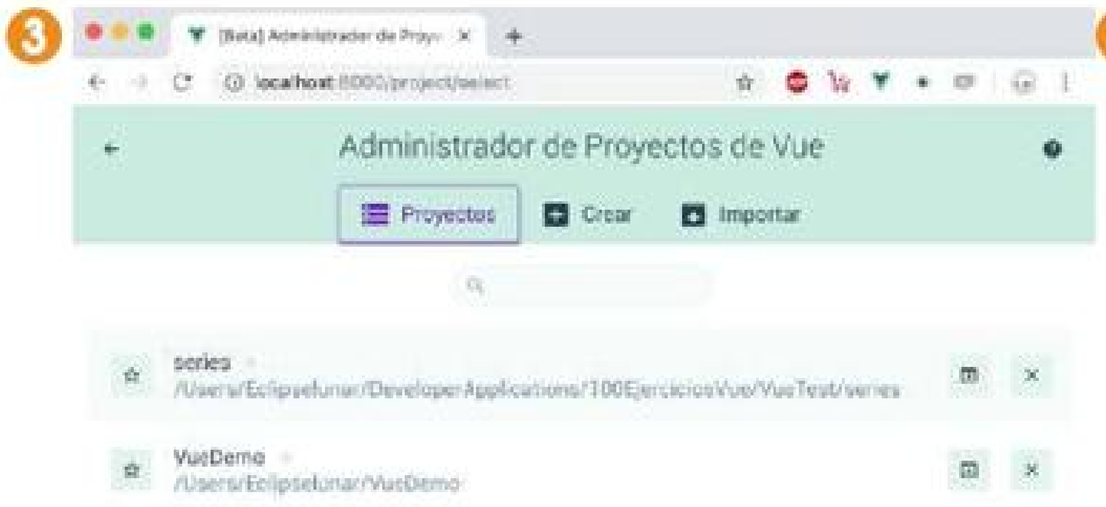
Tras seleccionar el proyecto podemos visualizar complementos instalados como "Jest", así como agregar otros ❹.

También podemos instalar paquetes desde la sección de dependencias, donde podremos visualizar los actuales e instalar otros ❺.

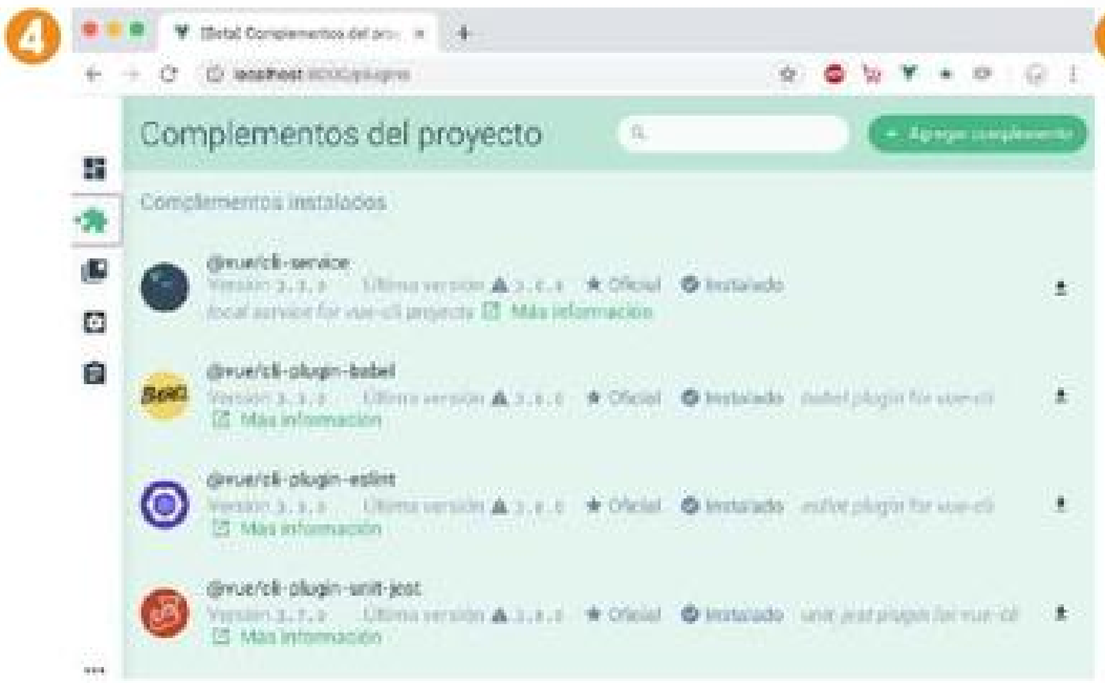
La configuración del proyecto también puede realizarse desde la sección de configuración ❻.

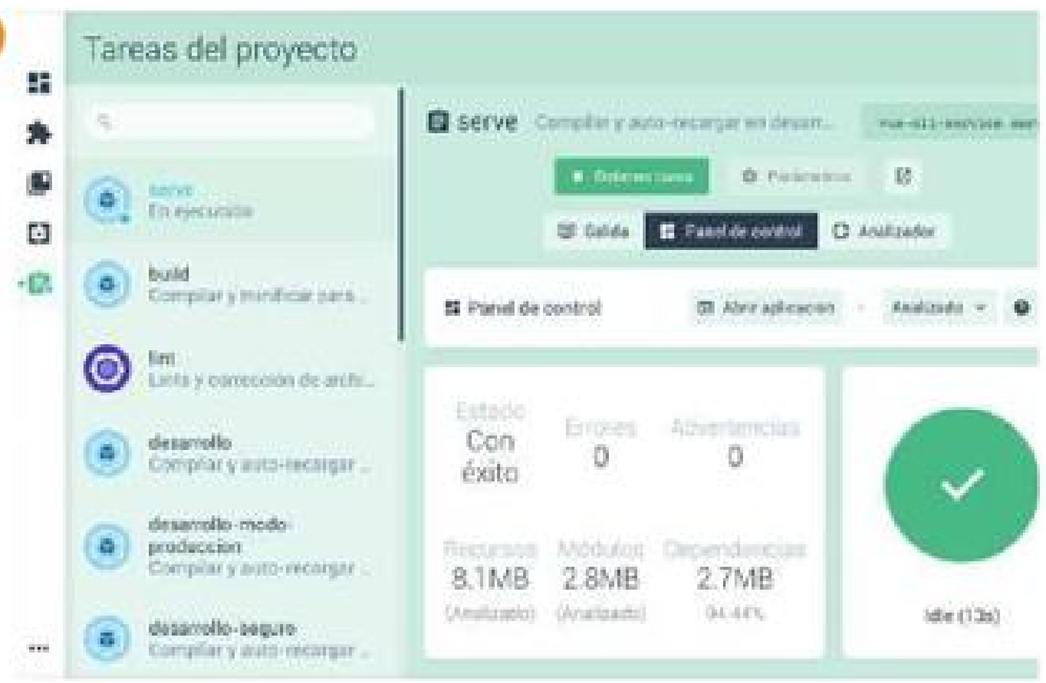
La ultima sección que encontramos es la de tareas del proyecto, donde podemos compilar el proyecto ❼, abrir su entorno web para visualizarlo, además de ejecutar las pruebas unitarias del proyecto ❽ y visualizar sus resultados.

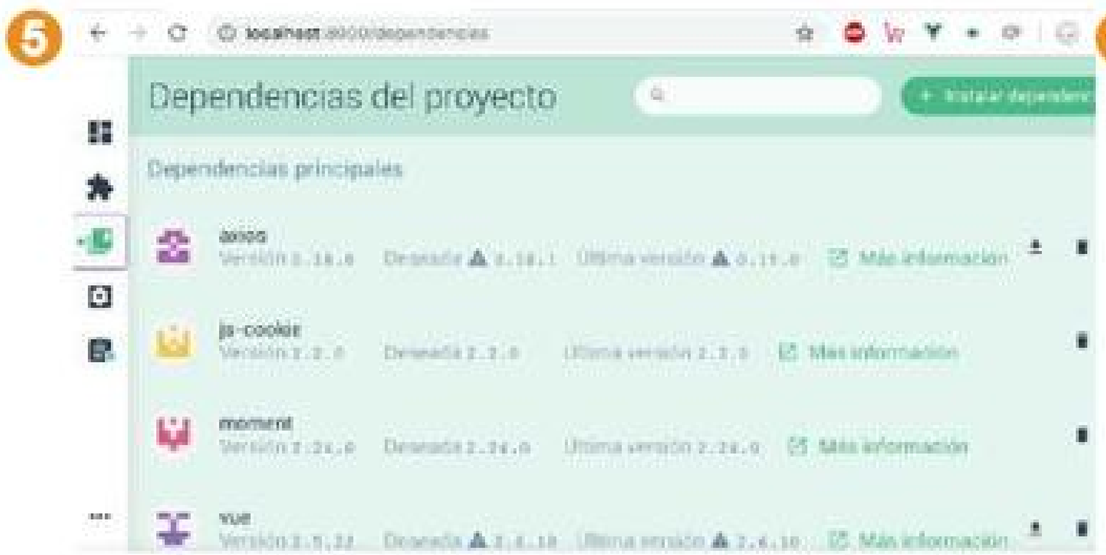



3
Administrador de Proyectos de Vue
Proyectos
Crear
Importar
series
VueDemo

6
Configuración del proyecto
Configuraciones generales
Public Path
Directorio de salida


4
Complementos del proyecto
Complementos instalados
@vue/cli-service
@vue/cli-plugin-babel
@vue/cli-plugin-eslint
@vue/cli-plugin-unit-jest


7
Tareas del proyecto
serve
build
lint
desarrollo
desarrollo-modo-producción
desarrollo-seguro
Panel de control
Estado
Con éxito
Errores
0
Advertencias
0
Recursos
8.1MB
Módulos
2.8MB
Dependencias
2.7MB


5
Dependencias del proyecto
Dependencias principales
axios
js-cookie
moment
vue

8
Tareas del proyecto
produccion-modo-desarrollo
produccion-report
production-modern
production-watch
test:unit
test:single
inspect
Salida

Para continuar aprendiendo...

SI ESTE LIBRO HA COLMADO SUS EXPECTATIVAS

Este libro forma parte de una colección en la que se cubren los programas informáticos de más uso y difusión en todos los sectores profesionales.

Todos los libros de la colección tienen el mismo planteamiento que este que acaba de terminar. Así que, si con este hemos conseguido que aprenda a utilizar VueJS o ha aprendido algunas nuevas técnicas que le han ayudado a profundizar su conocimiento de esta plataforma, no se detenga aquí: en la página siguiente encontrará otros libros de la colección que pueden ser de su interés.

PÍDALOS EN SU LIBRERÍA HABITUAL...Y, SI NO LOS ENCUENTRA, SOLICÍTELOS A

MARCOMBO, Gran Via de les Corts Catalanes, 594, 1º 2ª 08007 Barcelona - Tel. 933 180 079

LENGUAJE DE PROGRAMACIÓN

"Aprender JavaScript con 100 ejercicios prácticos" le permitirá adquirir los conocimientos básicos para asignar interactividad a los elementos de una página web o una aplicación.

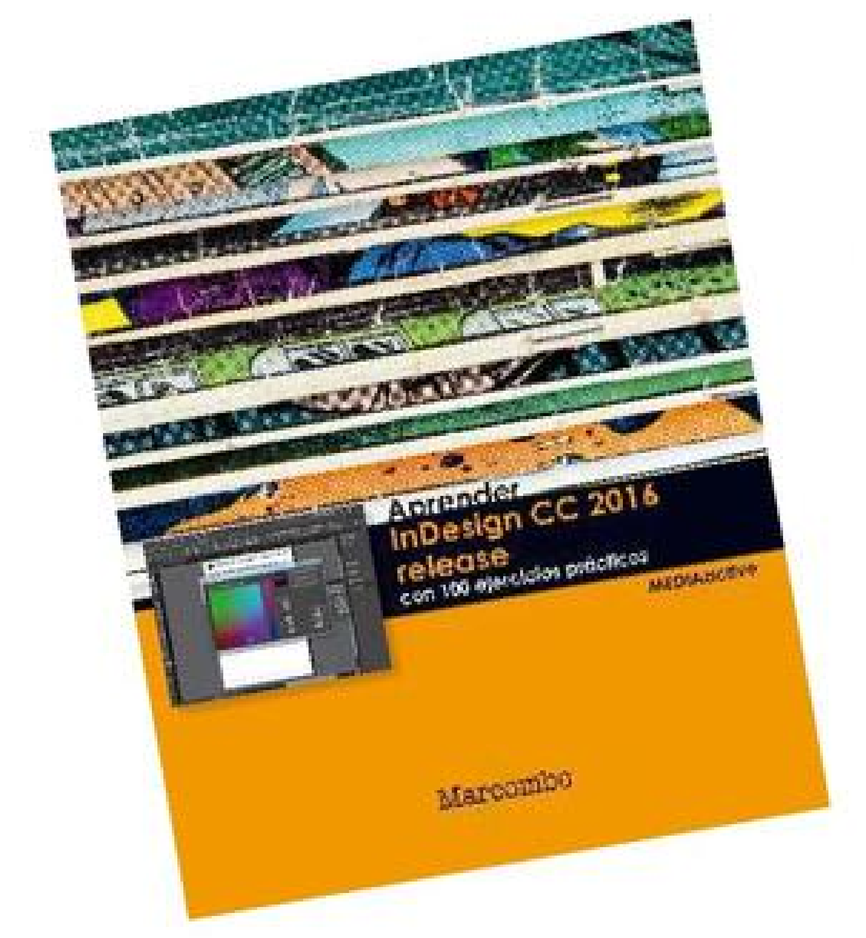
JavaScript es el lenguaje de programación que se ocupa de asignar la interactividad a cada uno de los elementos que conforman una página web o una aplicación. Este lenguaje, complicado pero a la vez muy mecánico, funciona mediante la asignación de elementos de diferentes tipos, cuyo uso y aplicación resulta imprescindible conocer.

Con este libro:

- Conocerá el lenguaje y la gramática de JavaScript.
- Aprenderá la interacción de HTML y CSS con JavaScript.
- Realizará pases de imágenes interactivos excepcionales y otros efectos.
- Trabajará con las librerías de jQuery para aportar todavía más potencia a sus sitios web.
- Conocerá algunos de los plugins más conocidos con distintos fines.

MAQUETACIÓN

Por otro lado si su interés está más cerca de la maquetación de libros, revistas, folletos, etc., entonces su libro ideal es "Aprender InDesign CC 2016 release con 100 ejercicios prácticos".



InDesign es el programa líder en el sector del diseño gráfico para diseñar y maquetar composiciones de todo tipo, desde libros hasta revistas, folletos publicitarios, etc. Con este manual aprenderá a utilizar este impresionante software, mejorado y ampliado, con sus propias creaciones. En esta versión de InDesign, Adobe ha incluido distintas novedades, tanto en su interfaz como en sus herramientas.

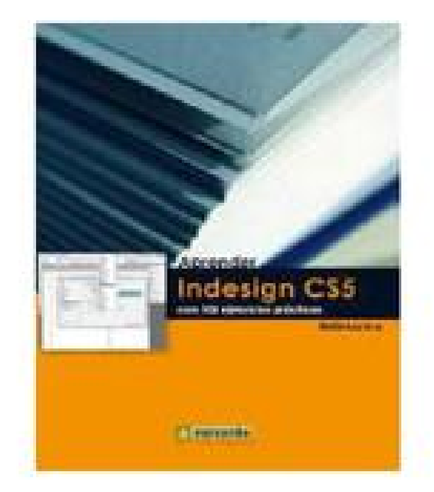
Con este libro:

- Diseñe fácilmente atractivas composiciones para la edición digital e impresa
- Genere rápidamente documentos PDF interactivos
- Obtenga elegantes composiciones de página
- Cree documentos complejos con tablas, numeraciones...
- Corrija errores durante la maquetación gracias a la Comprobación preliminar interactiva

COLECCIÓN APRENDER...CON 100 EJERCICIOS

DISEÑO Y CREATIVIDAD ASISTIDOS

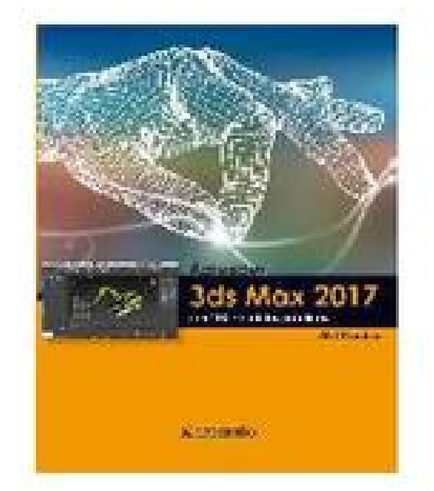
Hoy en día, gran parte del trabajo de los diseñadores gráficos se lleva a cabo con la inestimable ayuda de las herramientas digitales, en constante evolución. A ellas están dedicados los títulos de esta categoría.



- 3ds Max 2017
- AutoCAD 2017
- Flash CC
- Illustrator CC
- InDesign CC
- Photoshop CC
- Retoque fotográfico con Photoshop CS6

INTERNET

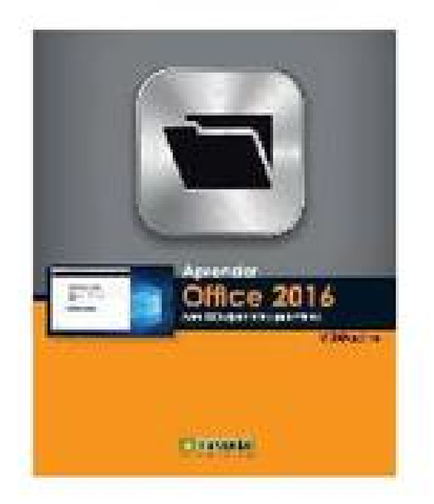
Gracias a Internet, millones de personas de todo el mundo tienen acceso fácil e inmediato a una cantidad enorme y diversa de información en línea. Consulte estos manuales para conocer sus múltiples utilidades.



- Dreamweaver CC
- Internet Explorer 10
- Windows Live

OFIMÁTICA

El término Ofimática se refiere al equipamiento utilizado para crear, guardar, manipular y compartir digitalmente información, tanto a nivel profesional como a nivel particular. En esta categoría agrupamos los títulos:



- Access 2016
- Office 2016
- PowerPoint 2016
- Word 2016
- Word y PowerPoint 2016

SISTEMAS OPERATIVOS

Los sistemas operativos se encargan de gestionar y coordinar las actividades realizadas por un ordenador. Estos manuales describen las principales funciones de Windows.

- Las novedades de Windows 7
- Windows 7 Multimedia y Nuevas tecnologías
- Windows 10