

UN RECOPIIATORIO  
DE POSTS PUBLICADOS EN  
EL ABISMO DE NULL



# DESARROLLA APLICACIONES CON VUEJS

JOSÉ ANTONIO DONGIL SÁNCHEZ

---

---

# Tabla de contenido

<a href="#">Acerca de este manual</a>	1.1
<b>Bloque 1. Los conceptos básicos</b>	<b>1.2</b>
<a href="#">Capítulo 1. The Progressive JavaScript Framework</a>	1.2.1
<a href="#">Capítulo 2. Trabajando con templates</a>	1.2.2
<a href="#">Capítulo 3. Enlazando clases y estilos</a>	1.2.3
<b>Bloque 2. La creación de componentes</b>	<b>1.3</b>
<a href="#">Capítulo 4. Creando componentes</a>	1.3.1
<a href="#">Capítulo 5. El ciclo de vida de un componente</a>	1.3.2
<a href="#">Capítulo 6. Definiendo componentes en un único fichero</a>	1.3.3
<b>Bloque 3. La gestión de rutas con vue-router</b>	<b>1.4</b>
<a href="#">Capítulo 7. Introduciendo rutas en nuestra aplicación</a>	1.4.1
<a href="#">Capítulo 8. Interceptores de navegación</a>	1.4.2
<a href="#">Capítulo 9. Conceptos avanzados</a>	1.4.3
<b>Bloque 4. La gestión de estados con vuex</b>	<b>1.5</b>
<a href="#">Capítulo 10. Introducción</a>	1.5.1
<a href="#">Capítulo 11. Los estados y getters</a>	1.5.2
<a href="#">Capítulo 12. Las mutaciones y acciones</a>	1.5.3
<a href="#">Capítulo 13. Los módulos</a>	1.5.4
<b>Bloque 5. El empaquetado de la aplicación con webpack</b>	<b>1.6</b>
<a href="#">Capítulo 14. Conceptos básicos</a>	1.6.1
<a href="#">Capítulo 15. Configurando nuestra primera build</a>	1.6.2
<a href="#">Capítulo 16. Caching, Shimming &amp; Splitting</a>	1.6.3
<b>Bloque 6. Renderizado en servidor con vue-server-renderer</b>	<b>1.7</b>
<a href="#">Capítulo 17. Introducción a Server-Side Rendering</a>	1.7.1
<a href="#">Capítulo 18. Configurando Webpack para SSR</a>	1.7.2
<a href="#">Capítulo 19. Adaptando tu proyecto a SSR</a>	1.7.3
<b>Bloque 7. Otras herramientas</b>	<b>1.8</b>
<a href="#">Capítulo 20. Aplicaciones universales con Nuxt</a>	1.8.1
<b>Bloque 8. Testeando nuestra aplicación con vue-test-utils</b>	<b>1.9</b>
<a href="#">Capítulo 21. Introducción</a>	1.9.1

---

Capítulo 22. Testeando nuestros componentes	1.9.2
Capítulo 23. Testeando nuestros stores	1.9.3

---



## Acerca de este manual

Este manual es una recopilación de posts extraídos del blog [El Abismo de Null](#) sobre el desarrollo de aplicaciones hechas en JavaScript con el nuevo framework VueJS.

Como podrás ver a lo largo de los capítulos, esta herramienta supone una forma más progresiva de adentrarse a aplicaciones que necesiten de un mayor escalado y mantenimiento de módulos.

## Antes de empezar

El manual intenta ser lo más explícito posible en las explicaciones de todos esos conceptos que puedan resultarte nuevos. Sin embargo, sí espero una predisposición de conocimientos previos por tu parte.

Por ejemplo, el manual se encargará de explicar todo los conceptos importantes de:

- Vue
- Vue-Router
- Vue-Cli
- Vuex
- Webpack
- Nuxt

Pero no se encargará de ser una guía sobre:

- JavaScript
- EcmaScript 6
- CSS
- HTML

Estos conceptos, aunque pueden ser referenciados o explicados en algún momento en concreto, no son el objetivo principal y tendrás que tenerlos en cuenta antes de empezar la lectura del manual.

## Estructura

El manual se encuentra compuesto por 8 bloques. Cada bloque se encargará de explicar, de forma progresiva, uno de los grandes elementos del framework.

Si eres nuevo en VueJS, te recomiendo que empieces desde el Capítulo 1 a leer y que prosigas para un aprendizaje adecuado. Si ya tienes conocimientos previos sobre VueJS, siéntete libre de viajar por el manual como te plazca.

## Aviso importante

El manual no supone un sustitutivo de la documentación oficial y solo quiere ser un apoyo o un aglutinador de conceptos base que todo desarrollador debería tener para ser competente en dicha herramienta.

El manual nace como una ayuda suplementaria para todas aquellas personas que no tengan un nivel fluido de inglés y quiera leer documentación sobre VueJS en su lengua materna.

Hay que recordar que el manual es una recopilación de post de un blog. Un blog que usa, en muchas ocasiones, lenguaje coloquial y que es un medio para expresar impresiones e inquietudes técnicas más.

No tomes esta guía como si se tratase de un ensayo académico formal.

## Comparte

La guía no se ha creado con ningún fin lucrativo. No se esperan aportaciones económicas, ni se cuenta con ningún derecho de autor explícito.

Simplemente ha sido creada por un afán por compartir y evangelizar sobre esta gran herramienta de trabajo.

Si te gusta lo que has leído y crees que puede aportar valor a otras personas, comparte. Cambia, elimina o añade todo aquello que tú creas que aporta y hazla tuya.

Puedes compartir por cualquier red social, blog o podcasts que creas conveniente.

- [Tuitea](#)

## Descarga

El manual se encuentra disponible en la plataforma GitBook de manera pública. **Están habilitados los formatos en MARKDOWN, HTML, PDF, EPUB y MOBI** para que puedas disfrutar de ella desde cualquier tipo de dispositivo.

Aunque las 5 formas han sido probadas, puede ser que se me hayan pasado aquellos detalles de maquetación o de visualización más específicos, por lo que de ser así, te agradecería que me lo comentaras o que me enviaras una Pull Request con los cambios.

## Feedback

Estoy disponible en todo momento para que puedas decirme qué te ha parecido el manual y estoy abierto a cualquier comentario sobre cómo mejorarlo.

[Puedes hablar conmigo por twitter](#) o [escribirme algún comentario en mi blog](#), o [escribirme directamente un correo electrónico](#).

Te intentaré contestar en la mayor brevedad posible. Lo prometo.

## Agradecimientos

Gracias por leer esta guía y confiar en ella para dar tus primeros pasos en VueJS. Creo que el tiempo dedicado te merecerá la pena por la forma y el mimo con el que este framework ha sido desarrollado.

Tu productividad y tu equipo agradecerá que hayas apostado por una tecnología como esta.

Agradezco mucho la ayuda de las siguientes personas que han colaborado a hacer este manual un poco mejor:

- Correctores ortográficos y estilísticos:

- [Rafa García](#)
- [Raúl Arrieta](#)

En fin...empecemos :)

# Capítulo 1. The Progressive JavaScript Framework

Hoy comienza una nueva serie en El Abismo. Hoy empezamos un nuevo viaje hacia nuevas tierras tecnológicas. Hemos hablado largo y tendido en el blog sobre buenas prácticas, patrones, paradigmas y pequeños trucos que nos han ayudado a hacer mejor JavaScript. Creo que es hora de que aprovechemos todo ese conocimiento que hemos ido adquiriendo, y que lo empleemos en conocer mejor nuestras herramientas de trabajo.

Como desarrolladores, tenemos muchas herramientas de trabajo, pero nada suele ser tan importante como el conjunto de librerías o frameworks a los que nos atamos en los proyectos. Incluir estas dependencias en nuestro proyectos va a determinar la forma en la que trabajamos y el estilo que tendremos que usar. De ahí la importancia de elegir aquellas herramientas que realmente necesitamos y que nos harán sentirnos cómodos.

Como antes de poder incluir cualquier herramienta en producción, hay que probarla y aprenderla, creo que es un buen momento para que nos detengamos unas semanas y prestemos atención a una de las librerías JavaScript que más está llamando la atención en los últimos meses: VueJS.

No sé cuánto durará esta serie, así que lo mejor es que nos pongamos al lío y dediquemos nuestro tiempo a entenderla y a saber cómo se usa. ¿Te apetece unirte? ¿Estás con ganas de aprender la enésima librería de JavaScript? ¿Sí? Pues sigamos:

## ¿Qué es VueJS?

Vue (pronunciado como viú) es la nueva herramienta JavaScript creada por [Evan You](#), miembro bastante conocido en la comunidad por participar en el desarrollo de [Meteor](#) y por ser desarrollador de Google durante varios años.

Evan You define su herramienta [como un framework progresivo](#). Progresivo porque el framework se encuentra dividido en diferentes librerías bien acotadas que tienen una responsabilidad específica. De esta manera, el desarrollador va incluyendo los diferentes módulos según las necesidades del contexto en el que se encuentre. No es necesario incluir toda la funcionalidad desde el principio como en el caso de frameworks como [AngularJS 1.x](#) o [EmberJS 1.x](#).

Es un sistema de modularización bastante parecido al de ReactJS. Facebook desarrolló un core para poder trabajar con vistas, pero a partir de ahí se han ido creando toda una serie de librerías (tanto por parte de Facebook como de la comunidad) que permite trabajar de una manera eficiente en un SPA. Aquí todas las piezas importantes se enmarcan dentro del proyecto de [VueJS](#) creado por Evan You.

A lo largo de estos primeros capítulos nos centraremos en el estudio del core del framework, por ahora dejaremos de lado a [vue-router](#) y a [vuex](#), aunque en algún momento llegaremos hasta ellos.

El core principal permite el desarrollo de componentes de UI por medio de JavaScript. La librería se enmarca dentro las arquitecturas de componentes (que tan de moda están) con una gestión interna de modelos basada en el patrón [MVVM](#). Esto quiere decir que los componentes, internamente, tienen mecanismos de doble 'data-binding' para manipular el estado de la aplicación.

Presume de ser una librería bastante rápida que consigue renderizar en mejores tiempos que ReactJS. Estos son los tiempos que podemos encontrar en su documentación:

Vue	React	
23ms	63ms	Fastest
42ms	81ms	Median
51ms	94ms	Average
73ms	164ms	95th Perc.
343ms	453ms	Slowest

Como curiosidad: VueJS tiene un tamaño de 74.9 KB (la versión al hacerse el post es la v2.2.6).

## ¿Qué caracteriza a VueJS?

Pero ¿qué define a VueJS? ¿Qué lo diferencia o lo asemeja al resto de alternativas? ¿Por qué se está poniendo tan de moda? Intentemos explicar algunas de sus características para que vosotros mismos veáis si el framework tiene la potencia que nos dicen:

- **Proporciona componentes visuales de forma reactiva.** Piezas de UI bien encapsulados que exponen una API con propiedades de entrada y emisión de eventos. Los componentes reaccionan ante eventos masivos sin que el rendimiento se vea perjudicado.

- **Cuenta con conceptos de directivas, filtros y componentes bien diferenciados.** Iremos definiendo y explicando estos elementos a lo largo de la serie.
- **La API es pequeña y fácil de usar.** Nos tendremos que fiar por ahora si ellos lo dicen :)
- **Utiliza Virtual DOM.** Como las operaciones más costosas en JavaScript suelen ser las que operan con la API del DOM, y VueJS por su naturaleza reactiva se encontrará todo el rato haciendo cambios en el DOM, cuenta con una copia cacheada que se encarga de ir cambiando aquellas partes que son necesarias cambiar.
- **Externaliza el ruteo y la gestión de estado** en otras librerías.
- **Renderiza `templates` aunque soporta JSX.** JSX es el lenguaje que usa React para renderizar la estructura de un componente. Es una especie de HTML + JS + vitaminas que nos permite, en teoría, escribir plantillas HTML con mayor potencia. VueJS da soporte a JSX, pero entiende que es mejor usar plantillas puras en HTML por su legibilidad, por su menor fricción para que maquetadores puedan trabajar con estos templates y por la posibilidad de usar herramientas de terceros que trabajen con estos templates más estándar.
- **Permite focalizar CSS para un componente específico.** Lo que nos permitirá crear contextos específicos para nuestros componentes. Todo esto sin perder potencia en cuanto a las reglas de CSS a utilizar. Podremos usar todas las reglas CSS3 con las que se cuentan.
- **Cuenta con un sistema de efectos de transición** y animación.
- **Permite renderizar componentes para entornos nativos** (Android e iOS). Es un soporte por ahora algo inmaduro y en entornos de desarrollo, pero existe una herramienta creada por Alibaba llamada [Weex](#) que nos permitiría escribir componentes para Android o iOS con VueJS si lo necesitáramos.
- **Sigue un flujo one-way data-binding** para la comunicación entre componentes.
- **Sigue un flujo doble-way data-binding** para la comunicación de modelos dentro de un componente aislado.
- **Tiene soporte para TypeScript.** Cuenta con decoradores y tipos definidos de manera oficial y son descargados junto con la librería.
- **Tiene soporte para ES6.** Las herramientas y generadores vienen con Webpack o Browserify de serie por lo que tenemos en todo momento un Babel transpilando a ES5 si queremos escribir código ES6.

- **Tiene soporte a partir de Internet Explorer 9.** Según el proyecto en el que estemos esto puede ser una ventaja o no. Personalmente cuanto más alto el número de la versión de IE mejor porque menos pesará la librería, pero seguro que tendréis clientes que os pongan ciertas restricciones. Es mejor tenerlo en cuenta.
- **Permite [renderizar las vistas en servidor](#).** Los SPA y los sistemas de renderizado de componentes en JavaScript tienen el problema de que muchas veces son difíciles de utilizar por robots como los de Google, por lo tanto el SEO de nuestra Web o aplicación puede verse perjudicado. VueJS permite mecanismos para que los componentes puedan ser renderizados en tiempo de servidor.
- **Es extensible.** Vue [se puede extender mediante plugins](#).

## ¿Cómo empezamos?

Para empezar, lo único que tendremos que hacer es incluir la dependencia de VueJS a nuestro proyecto. Dependiendo de nuestras necesidades, [podremos hacer esto de varias maneras](#). Yo me voy a quedar con la forma habitual de añadir dependencias a un proyecto de NodeJS.

Lo primero que hacemos es ejecutar los siguientes comandos en el terminal:

```
$ mkdir example-vue
$ cd example-vue
$ npm init
```

Esto nos genera una nueva carpeta para el proyecto y nos inicia un paquete de NodeJS. Una vez que tengamos esto, añadiremos VueJS como dependencia de la siguiente manera:

```
$ npm install vue --save
```

De esta forma, ya tendremos todo lo necesario para trabajar en nuestro primer ejemplo. Lo que esta dependencia nos descarga son diferentes VueJS dependiendo del entorno que necesitemos.

Lo que hacemos ahora es añadir un fichero `index.html` en la raíz e incluimos tanto la librería de Vue, como nuestro fichero JS, donde desarrollaremos este primer ejemplo:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>VueJS Example</title>
</head>
<body>
  <script src="node_modules/vue/dist/vue.js"></script>
  <script src="app.js"></script>
</body>
</html>
```

Si os dais cuenta, hemos añadido la librería VueJS de desarrollo y no la minificada. Esto es así porque la librería de desarrollo nos lanzará un montón de advertencias y errores que nos ayudarán a aprender y trabajar con VueJS.

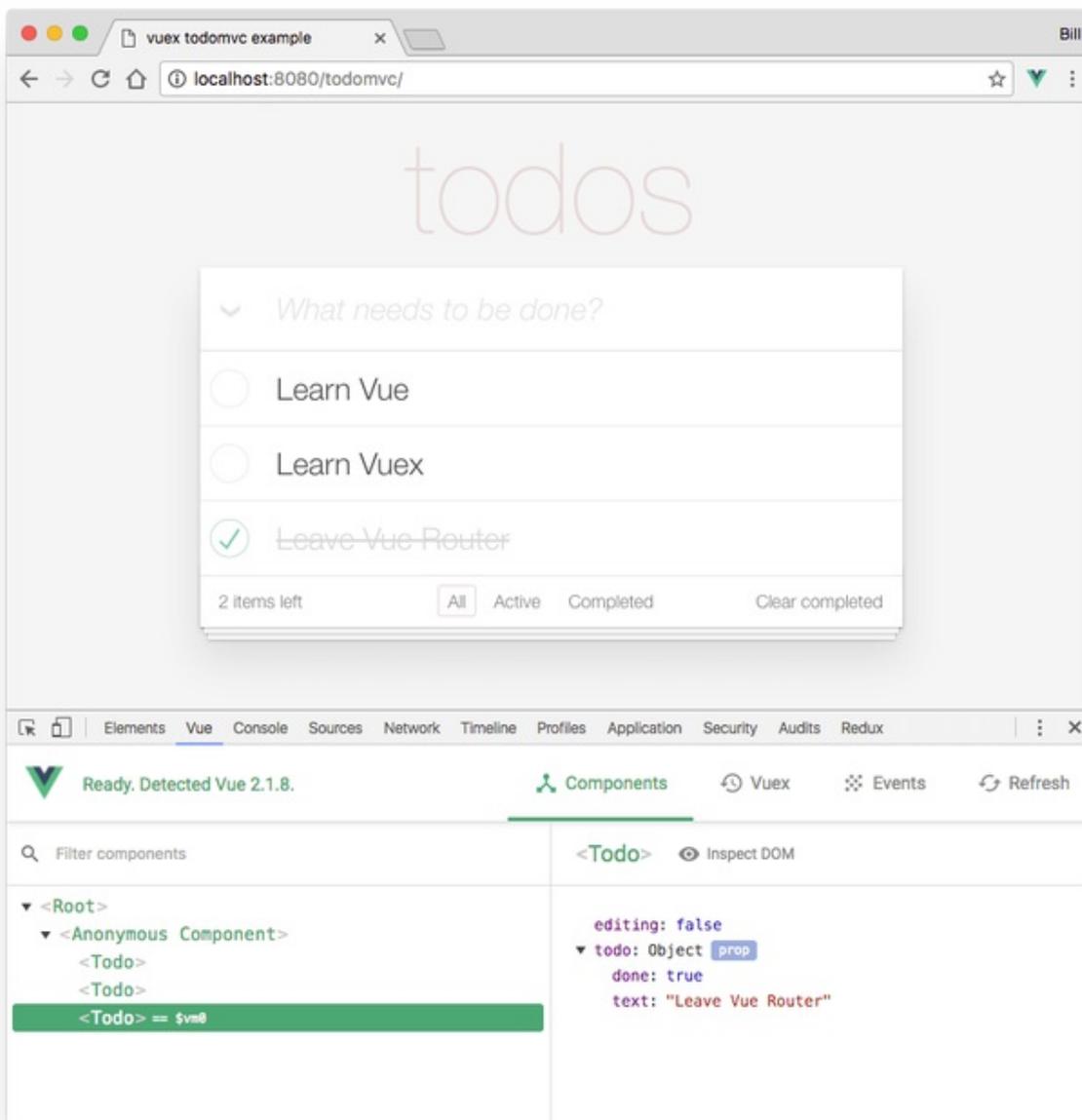
La librería tiene una gestión de errores súper descriptiva por lo que os recomiendo encarecidamente usarla en tiempo de desarrollo. Lógicamente tendremos que cambiarla por la minificada cuando nos encontremos en producción, pero bueno... para eso todavía queda. Una vez que tenemos esto, estamos preparados para trabajar en nuestro primer ejemplo.

Antes de que pasemos al código, me gustaría recomendaros un par de herramientas que nos pueden ser muy útiles para trabajar con VueJS.

## Depurando el código

Casi todos los frameworks importantes cuentan con una herramienta específica para poder depurar y analizar ciertos aspectos de nuestras aplicaciones. VueJS no iba a ser menos y [cuenta con un plugin para Firefox y Chrome](#) que se integra con el resto de herramientas de desarrollo de ambos navegadores.

He podido probar la herramienta de Chrome y me ha sorprendido lo fácil que es de usar. En la versión con la que he podido trastear, es posible inspeccionar los componentes que se encuentran en nuestro HTML. Esta inspección nos permite ver los modelos que se encuentran enlazados y la posibilidad de depurar esta información.



Otro de los apartados está dedicado a la posibilidad de gestionar el estado de nuestra aplicación por medio de `vuex`. La funcionalidad nos permite ver las transiciones en las que se mueve nuestra aplicación como si de un vídeo que se pueda rebobinar se tratase.

La otra funcionalidad que incluye es la posibilidad de detectar todos los eventos que se están generando en nuestra aplicación VueJS. Como iremos viendo a lo largo de los posts, VueJS genera un importante número de eventos para comunicar los diferentes componentes de nuestra interfaz. Poder ver en tiempo real y por medio de notificaciones como se van generando, es una maravilla a la hora de saber lo que pasa.

El plugin es mantenido por la propia comunidad de VueJS, por lo que, por ahora, su actualización se da por descontado.

## Gestionando un proyecto real

La forma en la que hemos instalado la primera dependencia de VueJS es algo rudimentaria. Cuando empezamos un proyecto, suele ser buena idea empezar desde un código base, una plantilla que contenga aquella parte de flujo de trabajo, necesario para trabajar en una plataforma en concreto.

Como en el caso de Angular, la gente de VueJS nos proporciona una herramienta de terminal muy útil para nuestro día a día: [vue-cli](#). Esta herramienta nos permitirá generar plantillas de nuestro proyecto, generar componentes e incluso personalizar plantillas a nuestro gusto.

`vue-cli` no solo nos ayudará a empezar rápidamente en un proyecto, sino que además, nos ayudará a entender cómo estructurar nuestro código y a seguir una serie de buenas prácticas debemos tener en cuenta si queremos que nuestro proyecto escale.

Para instalarlo como dependencia global, solo tendremos que lanzar el siguiente comando:

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

## Desarrollando más rápido

Los nuevos editores de texto permiten incluir pequeños plugins para añadir funcionalidad extra. En [Visual Studio Code](#) contamos con unos cuantos plugins que nos pueden ayudar a desarrollar más rápido. Dos de los más usados son:

- **Syntax Highlight for VueJS:** plugin para remarcar todas aquella sintaxis y palabras reservadas a VueJS. Este plugin nos permite localizar elemento de una forma más rápida y cómoda.
- **Vue 2 Snippets:** plugin que contiene pequeños '[snippets](#)' para que añadir nuestro código VueJS sea más rápido. De esta forma nos ayuda también como '[intellisense](#)'.

## Primer ejemplo

Para mostrar un poco de código, vamos a crear un pequeño ejemplo. La idea es crear una pequeña aplicación que nos permita añadir nuevos juegos a mi listado de juegos favoritos - sí, lo siento, no deja de ser el `TODO LIST` de toda la vida :).

Para conseguir esto, lo primero que vamos a hacer es añadir un elemento HTML que haga de contenedor de nuestra aplicación VueJS:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>VueJS Example</title>
</head>
<body>
  <div id="app"></div>

  <script src="node_modules/vue/dist/vue.js"></script>
  <script src="app.js"></script>
</body>
</html>
```

De esta manera, conseguimos delimitar el contexto en el que puede actuar nuestra aplicación. Es una buena forma de poder crear tantas aplicaciones VueJS necesitemos en un proyecto o incluso de poder alternar tecnologías.

Lo siguiente que hacemos es crear una instancia de nuestra aplicación VueJS en nuestro fichero `app.js` :

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

Lo que le decimos a VueJS es que genere una nueva instancia que tenga como referencia al elemento HTML que tenga como identificador único la palabra reservada `app` .

Lo siguiente que vamos a hacer es añadirle una pequeña plantilla con el HTML de nuestra aplicación:

```
const app = new Vue({
  el: '#app',
  template: [
    '<div class="view">',
    '  <game-header></game-header>',
    '  <game-add @new="addNewGame"></game-add>',
    '  <game-list v-bind:games="games"></game-list>',
    '</div>'
  ].join('')
});
```

Lo que hacemos es configurar la plantilla que queremos que renderice VueJS. Bueno, parece que la plantilla tiene bastante magia y que poco tiene que ver con HTML. Tenéis razón hay muchas cosas que VueJS está haciendo aquí. Veamos qué ocurre:

- [línea 4]: Defino toda la plantilla dentro de un elemento `div`. Todo componente o instancia tiene que encontrarse dentro de un elemento raíz. VueJS nos devuelve un error de no ser así y renderizará mal el HTML.
- [línea 5]: Aquí he definido un componente. Con VueJS puede extender nuestro HTML con nueva semántica. En este caso un componente que va a hacer de cabecera.
- [línea 6]: Vuelvo a definir otro componente. En este caso, es un componente que cuenta con un `input` y un `button`. Lo veremos más tarde. De este elemento, lo más destacable es el atributo `@new="addNewGame"`. Es un nuevo atributo que no se encuentra en el estándar de HTML ¿Qué es entonces? Estos elementos personalizados son lo que en VueJS se entiende como directivas. Son un marcado personalizado que aporta nueva funcionalidad al elemento HTML marcado. En este caso, lo que estamos haciendo es añadir un listener que escucha en el evento `new`, cada vez que el componente `game-add` emite un evento `new`, el elemento padre se encuentra escuchando y ejecuta la función `addNewGame`. No os preocupéis si no lo entendéis ahora porque lo explicaremos en un post dedicado a la comunicación entre componentes.
- [línea 7]: En esta línea hemos añadido otro componente. Este componente `game-list` se encarga de pintar por pantalla el listado de videojuegos favoritos. Como vemos, tiene una nueva directiva que no conocíamos de VueJS: `v-bind`. Esta directiva lo que hace es enlazar una propiedad interna de un componente con un modelo del elemento padre, en este caso el modelo `games`.

Vale, parece que la plantilla se puede llegar a entender.

Si nos damos cuenta hay dos elementos que hemos usado en la plantilla que no hemos definido en ninguna parte en nuestra primera instancia de VueJS: `addNewGame` y `games`. Para definirlos los hacemos de la siguiente manera:

```

const app = new Vue({
  el: '#app',
  template: [
    '<div class="view">',
    '  <game-header></game-header>',
    '  <game-add @new="addNewGame"></game-add>',
    '  <game-list v-bind:games="games"></game-list>',
    '</div>'
  ].join(''),
  data: {
    games: [
      { title: 'ME: Andromeda' },
      { title: 'Fifa 2017' },
      { title: 'League of Legend' }
    ]
  },
  methods: {
    addNewGame: function (game) {
      this.games.push(game);
    }
  }
});

```

Lo que hemos hecho es meter el método y el modelo en las zonas reservadas para ello. Todos los modelos que una instancia o un componente defina internamente, se tienen que incluir dentro de `data` y todos los métodos dentro de `methods`.

Teniendo esto, tenemos el 50% hecho de nuestra "aplicación". Lo siguiente que vamos a ver es la definición de los tres componentes visuales en los que he dividido la interfaz.

Empecemos por el componente `game-header` :

```

Vue.component('game-header', {
  template: '<h1>Video Games</h1>'
});

```

Nada del otro mundo. Lo único que estamos haciendo es registrar un componente de manera global con la etiqueta `game-header` . De esta forma ya podrá usar en las instancias de Vue. Internamente definimos un `template` sencillo con el título.

El siguiente componente tiene un poco más de chicha. Se trata del componente `game-add` , el combobox encargado de incluir nuevos juegos.

```
Vue.component('game-add', {
  template: [
    '<div>',
    '  <input type="text" v-model="titleGame" />',
    '  <button @click="emitNewGame">Añadir</button>',
    '</div>'
  ].join(''),
  data: function () {
    return {
      titleGame: null
    }
  },
  methods: {
    emitNewGame: function () {
      if (this.titleGame) {
        this.$emit('new', { title: this.titleGame });
        this.titleGame = null;
      }
    }
  },
});
```

Miremos un poco en detalle:

- **[línea 3]:** Volvemos a definir una plantilla HTML con un único elemento raíz.
- **[línea 4]:** El elemento tiene una directiva `v-model` que nos va a permitir ir obteniendo el valor del `input` e ir incluyéndolo en la variable `titleGame`.
- **[línea 5]:** El elemento tiene una directiva `@click` que lo que nos permite es registrar una función cuando se genere el evento clic sobre el botón.
- **[línea 8]:** El elemento data se inicializa, en un componente, con una función y no con un objeto. En su post veremos la razón de esto. Si ponemos un objeto, recibiremos un error de VueJS.
- **[línea 14]:** La función se encarga de ver si el input se encuentra vacío y emitir un evento hacia componentes padres con el nuevo título del juego.

Los siguientes componentes se encargan de pintar el listado de juegos. Son el componente `game-list` y el componente `game-item`:

```

Vue.component('game-list', {
  props: ['games'],
  template: [
    '<ol>',
    '<game-item v-for="item in games" :game="item" :key="item.id"></game-item>'
  ],
  '</ol>'
  ].join('')
});

Vue.component('game-item', {
  props: ['game'],
  template: '<li>{{ game.title }}</li>'
});

```

El componente `game-list` recibe un modelo como propiedad. Se trata del listado de juegos a mostrar. En el `template` vemos la directiva `v-for` encargado de iterar los juegos e ir pintando diferentes componentes `game-item`.

El componente `game-item` recibe un modelo y lo pinta.

El sistema es reactivo, es decir que si yo inserto un nuevo elemento en el array de juegos, VueJS es lo suficientemente inteligente para saber que tiene que renderizar los elementos precisos.

En el siguiente ejemplo podemos ver todo junto:

```

Vue.component('game-add', {
  template: [
    '<div>',
    '<input type="text" v-model="titleGame" />',
    '<button @click="emitNewGame">Añadir</button>',
    '</div>'
  ].join(''),
  data: function () {
    return {
      titleGame: null
    }
  },
  methods: {
    emitNewGame: function () {
      if (this.titleGame) {
        this.$emit('new', { title: this.titleGame });
        this.titleGame = null;
      }
    }
  },
});

```

```
Vue.component('game-list', {
  props: ['games'],
  template: [
    '<ol>',
    '<game-item v-for="item in games" :game="item" :key="item.id"></game-item>'
  ],
  '</ol>'
  ].join('')
});

Vue.component('game-item', {
  props: ['game'],
  template: '<li>{{ game.title }}</li>'
});

Vue.component('game-header', {
  template: '<h1>Video Games</h1>'
});

const app = new Vue({
  el: '#app',
  template: [
    '<div class="view">',
    '<game-header></game-header>',
    '<game-add @new="addNewGame"></game-add>',
    '<game-list v-bind:games="games"></game-list>',
    '</div>'
  ].join(''),
  data: {
    message: 'Video Games',
    games: [
      { title: 'ME: Andromeda' },
      { title: 'Fifa 2017' },
      { title: 'League of Legend' }
    ]
  },
  methods: {
    addNewGame: function (game) {
      this.games.push(game);
    }
  }
});
```

## Conclusión

Nos queda mucho camino por recorrer, pero parece que la filosofía de VueJS tiene sentido. Hay un equipo de personas muy competentes del mundo JavaScript que han sabido extraer de las herramientas que han usado en el pasado, todas las características buenas y las han

desarrollado aquí.

El ejemplo es simple pero si nos puede dar una idea de lo intuitivo y fácil que puede llegar a ser. Si vienes de utilizar frameworks y librerías orientadas a componentes no te costará el cambio.

Si vienes de un mundo más artesano que hace uso de jQuery y/o Handlebars, el aprendizaje progresivo que propone y el sistema de plugins te pueden ayudar e incluso llegar a sonar muy parecido. Y... si eres nuevo en el mundo JavaScript... bienvenido, cualquier sitio es bueno para empezar.

Nos leemos :)

## Capítulo 2. Trabajando con templates

Cuando trabajamos en Web, una de las primeras decisiones que solemos tomar es la forma en que se van a pintar los datos de los usuarios por pantalla:

Podemos optar por guardar una serie de HTMLs estáticos que ya cuentan con la información del usuario necesaria incrustada y almacenados dentro de base de datos. Este es el modelo que suelen desarrollar los CMS de muchos e-commerces o blogs personales.

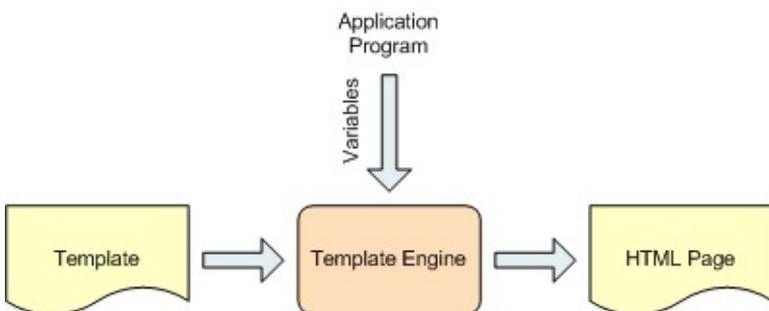
Otra opción sería la de renderizar o pintar estos datos dentro del HTML de manera dinámica. La idea subyace en crear una plantilla del HTML que queremos mostrar al usuario, dejando marcados aquellos huecos donde queremos que se pinten variables, modelos o entidades determinadas. Este es el sistema que suelen usar la gran mayoría de WebApps actuales que se basan en SPA.

Cada una de las dos opciones es válida y dependerá del contexto en el que nos encontremos la forma en que actuemos. VueJS por ejemplo, opta por la segunda opción. VueJS nos permite desarrollar plantillas HTML que se pueden renderizar tanto en tiempo de back como de front de manera dinámica.

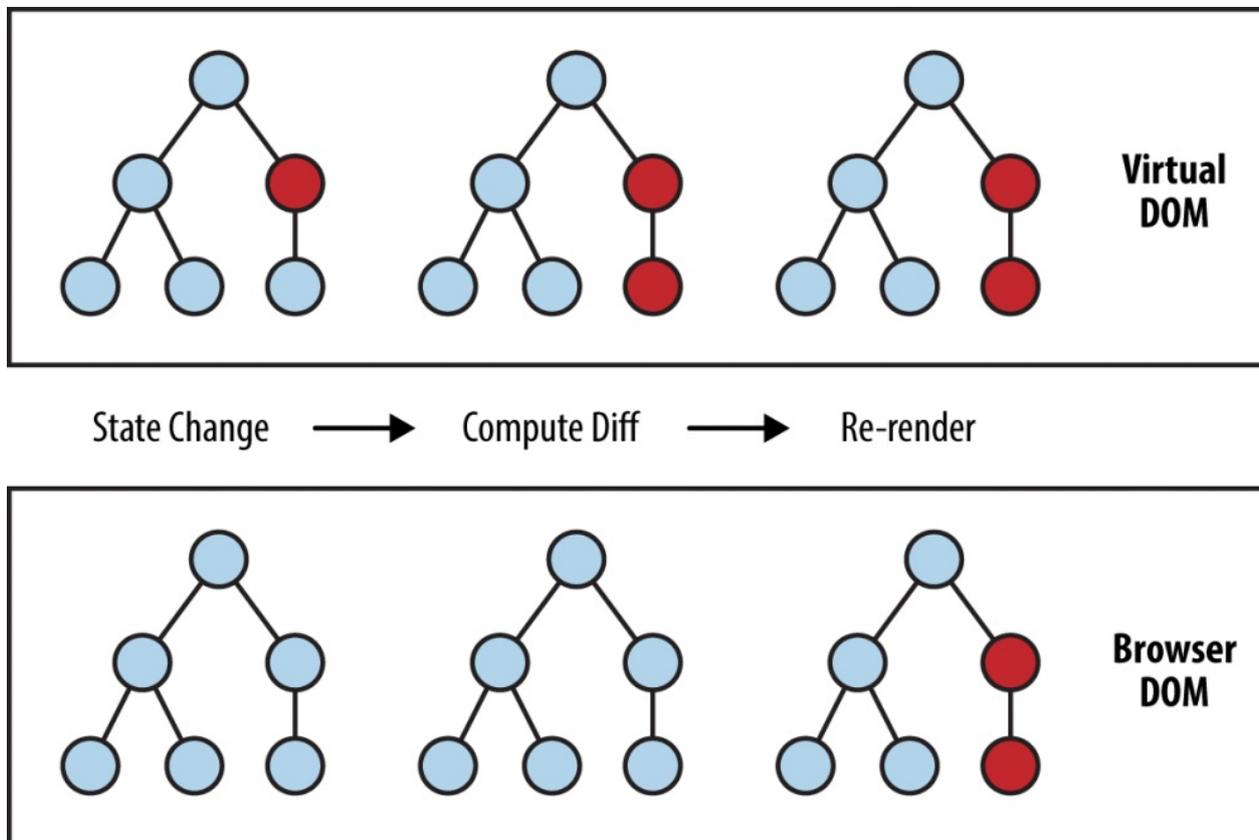
El post de hoy trata de explicar toda esta sintaxis mediante la explicación los conceptos de interpolación, directiva y filtro. Espero que os guste:

### ¿Cómo funciona?

Como decíamos en la introducción, VueJS cuenta con un motor de plantillas muy parecido al de librerías como HandlebarsJS. Lo que nos permite es crear HTML + una sintaxis especial que nos permite incluir datos del usuario. Aunque tiene soporte para JSX, se ha optado por un marcado mucho más ligero y entendible por un abanico de desarrolladores mayor. Tanto maquettadores, como fronts, como ciertos perfiles back están muy acostumbrados a este tipo de marcado dinámico.



Lo bueno de VueJS es que el cambio de estas plantillas en DOM no se produce de manera directa. Lo que VueJS hace es mantener una copia del DOM cacheada en memoria. Es lo que se suele denominar Virtual DOM y es lo que permite que el rendimiento de este tipo de frameworks no se vea penalizado por la cantidad de cambios que se pueden producir de manera reactiva.



## La interpolación

Una interpolación es la posibilidad de cambiar partes de una cadena de texto por variables. Para indicar dónde queremos un comportamiento dinámico de una cadena de texto dentro de VueJS, lo podemos indicar, marcando la variable que queremos interpolar con las dobles llaves (también conocidos como 'bigotes'):

```
<h1>Bienvenido {{ user.name }}</h1>
```

## Interpolando HTML

Este sistema nos sirve para interpolar variables que no contienen HTML. Si necesitáramos que la inserción sea interpretada como HTML tendríamos que indicarlo con la siguiente directiva `v-html` :

```
<div v-html="rawHtml"></div>
```

Este caso puede ser una mala práctica si lo que intentamos es estructurar nuestras vistas por medio de este método. El concepto de componente es el idóneo para reutilizar elementos. Intentemos usar este método solo en casos en los que no es posible otro método y teniendo en cuenta que el HTML incrustado sea controlado 100% por nosotros y no por el usuario, de esta manera podremos evitar ataques por XSS.

## Interpolando atributos

VueJS no solo nos deja interpolar textos de nuestros elementos HTML, también nos va a permitir tener control sobre los valores de nuestros atributos. Podríamos querer indicar si un botón tiene que estar habilitado o deshabilitado dependiendo de la lógica de la aplicación:

```
<button type="submit" v-bind:disabled="isEmpty">Entrar</button>
```

Podríamos tener este comportamiento con cualquier atributo de un elemento HTML.

## Interpolando por medio de expresiones

En VueJS se puede interpolar texto por medio de pequeñas expresiones. Es una posibilidad de incluir un poco de lógica en nuestra plantilla. Podríamos por ejemplo evaluar una expresión para renderizar o no un elemento de esta manera:

```
<div class="errors-container" v-if="errors.length !== 0">
```

Esto renderizaría el elemento dependiendo de si evalúe a true o a false. Aunque contemos con la posibilidad, es buena práctica que todas estas evaluaciones nos las llevemos a nuestra parte JavaScript. De esta manera separamos correctamente lo que tiene que ver con la vista de lo que tiene que ver con la lógica. Seguimos respetando los niveles de responsabilidad.

Si no tenéis mas remedio que usar una expresión de este estilo, hay que tener en cuenta que ni los flujos ni las iteraciones de JavaScript funcionan en ellos. La siguiente interpolación de una expresión daría un error:

```
{{ if (ok) { return message } }}
```

## Las directivas

Las directivas son atributos personalizados por VueJS que permiten extender el comportamiento por defecto de un elemento HTML en concreto. Para diferenciar un atributo personalizado de los estándar, VueJS añade un prefijo `v-` a todas sus directivas.

Por ejemplo, y como ya hemos ido viendo, contamos con directivas como esta:

```
<input id="username" type="text" v-model="user.name" />
```

`v-model` nos permite hacer un doble data-binding sobre una variable específica. En este caso `user.name`.

Una directiva puede tener o no argumentos dependiendo de su funcionalidad. Por ejemplo, una directiva con argumento sería la siguiente:

```
<a v-bind:href="urlPasswordChange" target="_blank">
  ¿Has olvidado tu contraseña?
</a>
```

Lo que hace `v-bind` con el argumento `href` es enlazar el contenido de `urlPasswordChange` como url del elemento `a`.

Las directivas son un concepto bastante avanzado de este tipo de frameworks. Hay que tener en cuenta en todo momento que la ventaja de trabajar con estos sistemas es que el comportamiento es reactivo. Esto quiere decir, que si el modelo al que se encuentra enlazado un elemento HTML se ve modificado, el propio motor de plantillas se encargará de renderizar el nuevo elemento sin que nosotros tengamos que hacer nada.

## Directivas modificadoras

Una directiva, en particular, puede tener más de un uso específico. Podemos indicar el uso específico accediendo a la propiedad que necesitamos incluir en el elemento. Un caso muy común es cuando registramos un evento determinado en un elemento y queremos evitar el evento por defecto que tiene el elemento. Por ejemplo, en un evento `submit` queremos evitar que nos haga un post contra el servidor para así realizar la lógica que necesitamos en JavaScript. Para hacer esto, lo haríamos así:

```
<form class="login" v-on:submit.prevent="onLogin">
```

Dentro de la API de VueJS contamos con todos estos modificadores.

## Atajo para la directiva `v-bind` o `v-on`

Una de las directivas más utilizadas es `v-bind`, lo que nos permite esta directiva es enlazar una variable a un elemento ya sea un texto o un atributo. Cuando lo usamos para enlazar con un atributo como argumento, podríamos usar el método reducido y el comportamiento sería el mismo. Para usar el atajo ahora debemos usar `:`. Para ver un ejemplo veremos cuando enlazamos una url a un elemento a. Podemos hacerlo de esta manera:

```
<button type="submit" v-bind:disabled="isFormEmpty">Entrar</button>
```

O de esta otra que queda más reducido:

```
<button type="submit" :disabled="isFormEmpty">Entrar</button>
```

Los dos generarían el mismo HTML:

```
<button type="submit" disabled="disabled">Entrar</button>
```

En el caso de registrar un evento, tenemos algo parecido. No hace falta que indiquemos `v-on` sino que podemos indicarlo por medio de una arroba `@` de esta manera:

```
<form class="login" @submit.prevent="onLogin">
```

El comportamiento sería el mismo que con `v-on`.

En este post hemos explicado algunas de [las directivas que hay, para entender el concepto, pero la API cuenta con todas estas](#).

## Las filtros

Cuando interpolamos una variable dentro de nuestro HTML, puede que no se pinte todo lo bonito y claro que a nosotros nos gustaría. No es lo mismo almacenar un valor monetario que mostrarlo por pantalla. Cuando yo juego con 2000 euros. Dentro de mi variable, el valor será 2000 de tipo number, pero cuando lo muestro en pantalla, el valor debería ser 2.000,00 € de tipo string.

Hacer esta conversión en JavaScript, rompería con su responsabilidad específica dentro de una web, no solo estamos haciendo que trabaje con lógica, sino que la conversión implica hacer que trabaje en cómo presenta los datos por pantalla.

Para evitar esto, se han inventado los filtros. Un filtro modifica una variable en tiempo de renderizado a la hora de interpolar la cadena. Para cambiar el formato de una variable tenemos que incluir el carácter `|` y el filtro que queremos usar como transformación.

Este sería un caso donde convertimos el texto de la variable en mayúsculas:

```
<h1>Bienvenido {{ user.name | uppercase }}</h1>
```

Los filtros se comportan como una tubería de transformación por lo que yo puedo concatenar todos los filtros que necesite de esta manera:

```
<h1>Bienvenido {{ user.name | filter1 | filter2 | filterN }}</h1>
```

En VueJS v1 se contaba dentro del core con una serie de filtros por defecto que en VueJS v2 han quitado para agilizar su tamaño. [Para incluir estos filtros podemos insertar la siguiente librería que extiende el framework.](#)

## Todo junto

Los ejemplos que hemos ido viendo forman parte del ejemplo que hemos desarrollado para este post. Lo que hemos hecho es desarrollar una plantilla en VueJS sobre una vista típica de login. El marcado es el siguiente:

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <title>Example templates</title>
</head>

<body>
  <div id="app" v-cloak>
    <h1>Bienvenido {{ user.name | uppercase }}</h1>

    <div class="login-errors-container" v-if="errors.length !== 0">
      <ol class="login-errors">
        <li v-for="error in errors"> {{ error }}</li>
      </ol>
    </div>

    <form class="login" v-on:submit.prevent="onLogin">
      <div class="login-field">
        <label for="username">Nombre de usuario</label>
        <input id="username" type="text" v-model="user.name" />
      </div>

      <div class="login-field">
        <label for="password">Contraseña</label>
        <input id="password" type="password" v-model="user.password" />
      </div>

      <div class="login-field">
        <button type="submit" v-bind:disabled="isFormEmpty">Entrar</button>
      </div>
    </form>

    <a v-bind:href="urlPasswordChange" target="_blank">
      ¿Has olvidado tu contraseña?
    </a>
  </div>

  <script src="node_modules/vue/dist/vue.js"></script>
  <script src="app.js"></script>
</body>

</html>
```

Creo que con lo visto en el post, todos los elementos se explican y no es necesario que profundicemos.

Os dejo también la instancia de VueJS donde se ve la lógica creada para la vista:

```
const app = new Vue({
  el: '#app',
  data: {
    user: { name: null, password: null },
    urlPasswordChange: 'http://localhost:8080',
    errors: []
  },
  computed: {
    isEmpty: function () {
      return !(this.user.name && this.user.password);
    }
  },
  methods: {
    onLogin: function () {
      this.errors = [];

      if (this.user.name.length < 6) {
        this.errors.push('El nombre de usuario tiene que tener al menos 6 caracteres');
      }

      if (this.user.password.length < 6) {
        this.errors.push('La contraseña tiene que tener al menos 6 caracteres');
      }
    }
  },
  filters: {
    uppercase: function (data) {
      return data && data.toUpperCase();
    }
  }
});
```

## Conclusión

Parece una tontería, pero saber desarrollar plantillas en VueJS nos ayuda mucho en nuestro camino a comprender el framework. No olvidemos que estamos aprendiendo sobre un framework progresivo y que, quizá, existan desarrolladores que con esto tengan más que suficiente para desarrollar aplicaciones con VueJS y que no necesiten muchas más funcionalidades.

Los que hayan usado en su vida profesional motores de plantillas, habrán visto que el sistema es el mismo de siempre. Esto es otra de las ventajas de VueJS: el framework intenta ser todo lo práctico que puede y no intentar reinventar la rueda si es posible. Si algo como las plantillas HTML ha funcionado ya en otros sistemas, por qué no aprovecharnos de ellos.

Si a alguno de vosotros, al leer este post sobre plantillas, el sistema de directivas y filtros presentado se le queda un poco pequeño, es bueno recordar que VueJS permite su extensión por medio de plugins y que crear nuevas directivas y nuevos filtros estará disponible para nosotros. Hablaremos en El Abismo más adelante de cómo hacer esto.

Por el momento, nos quedamos aquí.

Nos leemos :)

## Capítulo 3. Enlazando clases y estilos

Una de las cosas que más me gustaban cuando usaba jQuery era la posibilidad de incluir o quitar clases desde mi JavaScript en cualquier momento. Con dos simples métodos -

```
addClass yremoveClass
```

 - podía hacer de todo.

Me daba una versatilidad y tal toma de decisión sobre el cómo, cuándo y por qué incluir o eliminar una clase a un elemento HTML, que el mecanismo que suelen proponer los frameworks modernos, no me acababa de convencer o de darme esa comodidad que yo ya tenía.

En su día HandlebarsJS no consiguió convencerme del todo, AngularJS contiene tantas funcionalidades que el proceso se vuelve demasiado complejo y verboso. En VueJS nos pasa parecido, pero por lo menos la sintaxis es clara y sencilla.

El post de hoy vamos a dedicarlo a explicar cómo podemos incluir o quitar clases o estilos dependiendo del estado de nuestra aplicación, un breve post con el que cerraremos la introducción a la serie:

### Enlazando clases

Para enlazar una variable a una clase, hacemos uso de la directiva `v-bind:class` o su alternativa corta `:class`. Con esta declaración, podemos guardar el nombre de clases en variables y jugar con ello a nuestro antojo de manera reactiva.

Por ejemplo, yo podría hacer esto:

```
<div :class="player1.winner">
```

De esta manera, he enlazado la variable `winner` de mi modelo `player1` a la directiva `:class`. Lo que VueJS va a intentar es coger el contenido de `winner` y lo va a insertar como una clase.

Esto nos da poca versatilidad porque nos hace acoplarnos demasiado a una sola variable y no nos permite incluir más en un elemento. Sin embargo, VueJS acepta otras formas de enlazar modelos para conseguir funcionamientos más flexibles.

Dentro de la directiva `v-bind:class` podemos enlazar tanto un objeto como un array. Veamos qué nos aporta cada caso:

## Enlazando un objeto

Podemos enlazar un objeto en un elemento para indicar si una clase se tiene que renderizar en el HTML o no. Lo hacemos de la siguiente manera:

```
<div :class="{ winner: player1.winner }">
```

Cuando la variable `player1.winner` contenga un `true` la clase `winner` se renderizará. Cuando contenga `false` no se incluirá. De esta manera puedo poner el toggle de las funciones que quiera. Este objeto, me lo puedo llevar a mi parte JavaScript y jugar con él como necesite.

## Enlazando un array

Puede darse el caso también, que no solo necesite hacer un toggle de clases. Puede que quiera indicar un listado de clases enlazadas. Yo podría hacer lo siguiente:

```
<div :class="['box', winner]">
```

En este caso lo que quiero es que el elemento `div` tenga la clase `box` y lo que contenga internamente la variable `winner`. Con esto se puede jugar bastante y crear híbridos como el siguiente:

```
<div :class="['box', { winner: player1.winner }]">
```

En este caso, hago que `winner` se incluya o no dependiendo del valor de `player1.winner`.

Al igual que pasaba con los objetos, esta estructura de datos puede ser almacenada en JS y ser enlazada directamente.

Un pequeño apunte a tener en cuenta al enlazar en un componente

Podemos enlazar variables a la definición de un componente que se encuentre en nuestra plantilla.

Teniendo la definición del siguiente componente:

```
Vue.component('pokemon', {
  template: [
    '<div class="pokemon">',
    '  <div class="pokemon-head"></div>',
    '  <div class="pokemon-body"></div>',
    '  <div class="pokemon-feet"></div>',
    '</div>'
  ].join('')
});
```

Yo podría hacer esto en el template al usarlo:

```
<pokemon :class="player1.pokemon.name"></pokemon>
```

El resultado del HTML generado, en este caso, sería el siguiente:

```
<div class="pokemon pikachu">
  <div class="pokemon-head"></div>
  <div class="pokemon-body"></div>
  <div class="pokemon-feet"></div>
</div>
```

VueJS primero coloca las clases que tenía definido en su plantillas interna y luego incluye las nuestras. De esta manera podremos pisar los estilos que deseemos. Es una buena forma de extender el componente a nivel de estilos.

## Enlazando estilos

También podemos enlazar estilos directamente en un elemento. En vez de usar la directiva `v-bind:class`, tenemos que usar la directiva `v-bind:style`. Haríamos esto:

```
<div v-bind:style="{ color: activeColor, fontSize: fontSize + 'px' }"></div>
```

Hemos incluido un objeto que lleva todas las propiedades de CSS que queramos. Este objeto también podría estar en nuestro JS para jugar con él.

### Como un array

Puede que necesitemos extender estilos básicos de manera inline. VueJS ya ha pensado en esto:

```
<div v-bind:style="[baseStyles, overridingStyles]">
```

Me detengo menos en esta directiva porque creo que es mala práctica incluir estilos en línea, simplemente es bueno que sepamos que existe la posibilidad por si en algún caso en concreto no quedase más remedio de hacer uso de esta técnica.

## A tener en cuenta

Cuando incluimos un elemento CSS que suele llevar prefijos debido a que no está estandarizado todavía (imaginemos en transform por ejemplo), no debemos preocuparnos, pues VueJS lo tiene en cuenta y el mismo añadirá todos los prefijos necesarios

## Todo junto

Los ejemplos que hemos ido viendo en el post son sacado del siguiente ejemplo:

Hemos creado un pequeño juego que te permite enfrentar en un combate a tu pokemon favorito contra otros. La idea está en elegir dos pokemons y ver quién de los dos ganaría en un combate. Una tontuna que nos sirve para ver mejor cómo enlazar clases en VueJS.

El ejemplo consta de estos tres ficheros:

```
.box { display: inline-block; padding: 1rem; margin: 1rem; }
.box.winner { background: green; }

.pokemon { width: 3rem; display: inline-block; margin: 1rem; }
.pokemon-head, .pokemon-body { height: 3rem; }
.pokemon-feet { height: 1rem; }

.pokemon.bulvasaur .pokemon-head { background: #ff6a62; }
.pokemon.bulvasaur .pokemon-body { background: #62d5b4; }
.pokemon.bulvasaur .pokemon-feet { background: #317373; }

.pokemon.squirtle .pokemon-head,
.pokemon.squirtle .pokemon-feet { background: #8bc5cd; }
.pokemon.squirtle .pokemon-body { background: #ffe69c; }

.pokemon.charmander { background: #de5239; }

.pokemon.pikachu { background: #f6e652; }
```

Estas son las clases que dan forma a nuestros cuatro pokemons y las que vamos a ir enlazando de manera dinámica según lo que el usuario haya elegido.

```
Vue.component('pokemon', {
  template: [
    '<div class="pokemon">',
    '  <div class="pokemon-head"></div>',
    '  <div class="pokemon-body"></div>',
    '  <div class="pokemon-feet"></div>',
    '</div>'
  ].join('')
});

const app = new Vue({
  el: '#app',
  data: {
    player1: { pokemon: {}, winner: false },
    player2: { pokemon: {}, winner: false },
    pokemons: [
      { id: 0, name: 'pikachu', type: 'electro' },
      { id: 1, name: 'bulvasaur', type: 'planta' },
      { id: 2, name: 'squirtle', type: 'agua' },
      { id: 3, name: 'charmander', type: 'fuego' }
    ],
    results: [
      [0, 2, 1, 0],
      [1, 0, 2, 2],
      [2, 1, 0, 1],
      [0, 1, 2, 0],
    ]
  },
  methods: {
    fight: function () {
      const result = this.results[this.player1.pokemon.id][this.player2.pokemon.
id];

      const selectWinner = [
        () => { this.player1.winner = true; this.player2.winner = true; },
// empate
        () => { this.player1.winner = true; this.player2.winner = false; },
// gana jugador 1
        () => { this.player1.winner = false; this.player2.winner = true; }
// gana jugador 2
      ];

      selectWinner[result]();
    },
    resetWinner: function () {
      this.player1.winner = false;
      this.player2.winner = false;
    }
  }
});
```

El código anterior define un componente pokemon con diferentes `divs` para simular la anatomía 'estilo lego' de un pokemon.

La instancia de VueJS define una aplicación que simula la batalla. Lo que hacemos es definir dos jugadores (líneas 14 y 15), un listado de pokemons (líneas de la 16 a la 20) y una tabla de resultados posibles donde x e y indican quién ganaría entre los pokemons seleccionados por ambos jugadores (líneas 22 a la 26).

Hemos definido dos métodos para simular el combate. El método `fight` obtiene el `id` de ambos jugadores y busca la posición en la tabla de resultados. Dependiendo del resultado dado, se indica el jugador que ha ganado. El método `resetWinner` nos permite reiniciar la partida para empezar una nueva.

La plantilla que permite mostrar todo esta lógica por pantalla es el siguiente:

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <title>Example classes</title>

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

<body>
  <div id="app">
    <div class="actions-container">
      <button @click="fight">Luchar</button>
    </div>

    <!-- Casilla del jugador 1 -->
    <div :class="['box', { winner: player1.winner }]">
      <select v-model="player1.pokemon" @change="resetWinner">
        <option v-for="pokemon in pokemons" v-bind:value="pokemon">{{ pokemon.
name }}</option>
      </select>
      <pokemon :class="player1.pokemon.name"></pokemon>
    </div>

    <label>VS</label>

    <!-- Casilla del jugador 2 -->
    <div :class="['box', { winner: player2.winner }]">
      <pokemon :class="player2.pokemon.name"></pokemon>
      <select v-model="player2.pokemon" @change="resetWinner">
        <option v-for="pokemon in pokemons" v-bind:value="pokemon">{{ pokemon.
name }}</option>
      </select>
    </div>
  </div>

  <script src="node_modules/vue/dist/vue.js"></script>
  <script src="app.js"></script>
</body>
</html>
```

Hemos definido dos contenedores para todo lo relativo a cada uno de los jugadores (esto en el futuro serán componentes, pero para que el ejemplo quede claro, hemos preferido dejarlo así).

En la línea 18 podemos ver cómo usamos un enlace de clases por medio de array y de objeto. Combinar ambos métodos nos da mucha versatilidad. Como yo necesito indicar dos clases, uso el array. La primera clase es una fija. No necesito dinamismo en tiempo de JS

con lo que indico directamente como un string la clase `box`. La segunda clase está enlazada al modelo del jugador. La clase `winner` se activará cuando tengamos un ganador de la partida.

El otro elemento donde tenemos enlace de clases es en la línea 22. En este caso estoy enlazando una clase dinámica a un componente. Como vimos anteriormente, esto es posible y lo que nos va a permitir es pintar los colores del pokémon elegido. Ese modelo variará dependiendo de lo seleccionado en el elemento select.

Si queréis ver el ejemplo funcionando podéis hacerlo en [este jsfiddle](#).

## Conclusión

Con el post de hoy lo que hemos hecho es aumentar nuestros conocimientos en la sintaxis que podemos usar para las plantillas de VueJS. Nada nuevo ni revolucionario. Nada que otras alternativas no nos permitan.

Conocerlo es nuestra obligación para ser buenos desarrolladores de VueJS, así que aunque este post parezca un trámite, es necesario conocerlo y asimilarlo.

En el próximo post de la serie, elevaremos el nivel de dificultad y nos centraremos en la creación de componentes, la piedra angular de este framework progresivo.

Por el momento es todo.

Nos leemos :)

# Capítulo 4. Creando componentes

Estamos en pleno boom de los frameworks y librerías front orientados a componentes. Parecen ser la mejor forma de modularizar nuestro código y de conseguir piezas reutilizables y mantenibles con un alto nivel de cohesión interno y un bajo acoplamiento entre piezas.

VueJS no iba a ser menos y basa su funcionamiento en el aislamiento de estados y comportamientos en pequeños componentes que se encarguen de llevar a cabo el ciclo de vida entero de una mínima parte de la UI que estamos creando.

Como en casi todos los casos - quizá a excepción de Polymer - sufre de todo aquello bueno y malo de estas soluciones. Para explicar cómo trabajar con componentes en VueJS lo mejor será desarrollar un ejemplo. En este ejemplo vamos a explicar qué son las propiedades, qué son los eventos personalizados y qué son 'slots'. ¿Empezamos?

## El ejemplo

El desarrollo que vamos a hacer es un pequeño 'marketplace' para la venta de cursos online. El usuario va a poder indicar el tiempo en meses que desea tener disponible la plataforma en cada uno de los cursos.

## Creando la instancia

Para ello lo que vamos a crear es un primer componente llamado `course`. Para hacer esto, tenemos que registrar un nuevo componente dentro del framework de la siguiente manera:

```
Vue.component('course', {
  data: function () {
    return {
      months: 0,
      styleClass: null,
      header: {
        title: 'Course default',
        image: 'http://lorempixel.com/64/64/'
      }
    }
  },
  // ... more code
});
```

Con esto ya podremos hacer uso de él en cualquier plantilla en el que necesitemos un ítem curso dentro de nuestra app. Hemos incluido unos datos de inicialización del componente. En este caso datos de la cabecera. Como puedes apreciar, en un componente, `data` se define con una función y no con un objeto.

## Incluyendo propiedades

Un componente no deja de ser una caja negra del cual no sabemos qué HTML se va a renderizar, ni tampoco sabemos como se comporta su estado interno. Este sistema de cajas negras es bastante parecido al [comportamiento de una función pura en JavaScript](#).

Una función, para poder invocarse, debe incluir una serie de parámetros. Estos parámetros son propiedades que nosotros definimos al crear una función. Por ejemplo, puedo tener una función con esta cabecera:

```
function createCourse(title, subtitle, description) {  
  ...  
}
```

Si yo ahora quiero hacer uso de esta función, simplemente lo haría así:

```
createCourse(  
  'Curso JavaScript',  
  'Curso Avanzado',  
  'Esto es un nuevo curso para aprender'  
);
```

Dados estos parámetros, yo espero que la función me devuelva lo que me promete: un curso.

Pues en VueJS y su sistema de componentes pasa lo mismo. Dentro de un componente podemos definir propiedades de entrada. Lo hacemos de la siguiente manera:

```
Vue.component('course', {  
  // ... more code  
  props: {  
    title: { type: String, required: true },  
    subtitle: { type: String, required: true },  
    description: { type: String, required: true },  
  },  
  // ... more code  
});
```

Estamos indicando, dentro del atributo `props` del objeto `options`, las propiedades de entrada que queremos que tenga nuestro componente, en nuestro caso son 3: `title`, `subtitle` y `description`, al igual que en la función.

Estas propiedades, ahora, pueden ser usadas en su template. Es buena práctica dentro de cualquier componente que indiquemos estas propiedades y que les pongamos validadores.

En nuestro caso, lo único que estamos diciendo es que las tres propiedades sean de tipo String y que sean obligatorias para renderizar correctamente el componente. Si alguien no usase nuestro componente de esta forma, la consola nos mostrará un warning en tiempo de debug.

Ahora ya podemos usar, en nuestro HTML, nuestro componente e indicar sus propiedades de entrada de esta forma:

```
<course
  title="Curso JavaScript"
  subtitle="Curso Avanzado"
  description="Esto es un nuevo curso para aprender">
</course>
```

Como vemos, es igual que en el caso de la función.

Hay que tener en cuenta que las propiedades son datos que se propagan en una sola dirección, es decir, de padres a hijos. Si yo modifico una propiedad dentro de un componente hijo, ese cambio no se propagará hacia el padre y por tanto no provocará ningún tipo de reacción por parte del sistema.

[Aunque hablaremos más de props en el futuro, aquí tienes más documentación.](#)

## Personalizando eventos

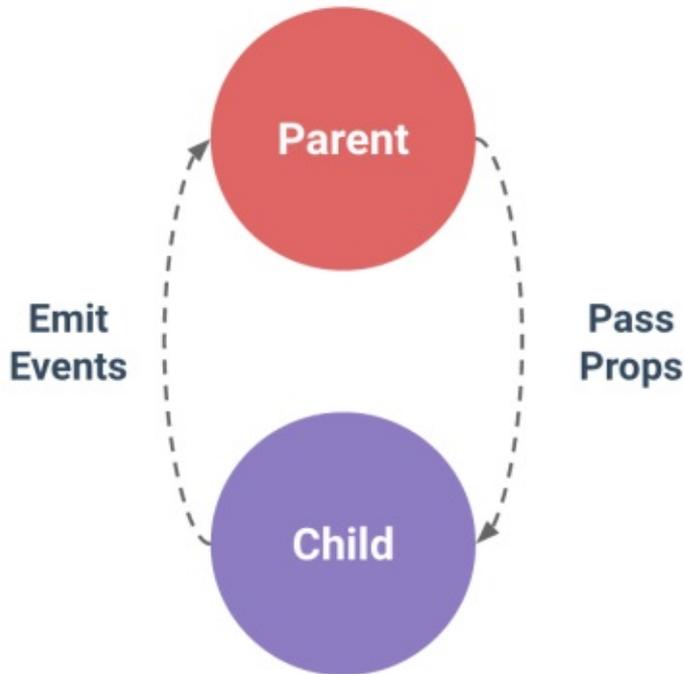
Hemos visto cómo el componente padre consigue comunicar datos a un componente hijo por medio de las propiedades, pero ¿Qué ocurre cuando un hijo necesita informar de ciertos datos o acciones que están ocurriendo en su interior? ¿Cómo sería el return de un componente en VueJS si fuese una función JavaScript?

Bueno, la opción por la que ha optado VueJS, para comunicar datos entre componentes hijos con componentes padres, ha sido por medio de eventos personalizados. Un componente hijo es capaz de emitir eventos cuando ocurre algo en su interior.

Tenemos que pensar en esta emisión de eventos como en una emisora de radio. Las emisoras emiten información en una frecuencia sin saber qué receptores serán los que reciban la señal. Es una buena forma de escalar y emitir sin preocuparte del número de

oyentes.

En los componentes de VueJS pasa lo mismo. Un componente emite eventos y otros componentes padre tienen la posibilidad de escucharlos o no. Es una buena forma de desacoplar componentes. El sistema de comunicación es este:



En nuestro caso, el componente va a contar con un pequeño `input` y un botón para añadir cursos. Lo que ocurrirá es que el componente `course` emitirá un evento de tipo `add` con un objeto que contiene los datos del curso y los meses que se quiere cursar:

```
Vue.component('course', {
  // ... more code
  methods: {
    add: function () {
      this.$emit('add', { title: this.title, months: this.months });
    }
  },
  // ... more code
});
```

Lo que hacemos es usar el método del componente llamado `$emit` e indicar un 'tag' para el evento personalizado, en este caso `add`.

Ahora, si queremos registrar una función cuando hacemos uso del componente, lo haríamos de la siguiente manera:

```
<course
  title="Curso JavaScript"
  subtitle="Curso Avanzado"
  description="Esto es un nuevo curso para aprender"
  @add="addToCart">
</course>
```

Hemos registrado un evento de tipo `@add` que ejecutará la función `addToCart` cada vez que el componente emita un evento `add`.

Aunque hablaremos más de eventos en el futuro, aquí tienes más documentación.

## Extendiendo el componente

Una vez que tenemos esto, hemos conseguido definir tanto las propiedades de entrada como los eventos que emite mi componente. Podríamos decir que tenemos un componente curso base.

Ahora bien, me gustaría poder definir ciertos estados y comportamientos dependiendo del tipo de curso que quiero mostrar. Me gustaría que los cursos de JavaScript tuviesen un estilo y los de CSS otro.

Para hacer esto, podemos extender el componente base `course` y crear dos nuevos componentes a partir de este que se llamen `course-js` y `course-css`. Para hacer esto en VueJS, tenemos que hacer lo siguiente:

```
const course = {
  props: {
    title: { type: String, required: true },
    subtitle: { type: String, required: true },
    description: { type: String, required: true },
  },
  data: function () {
    return {
      months: 0,
      styleClass: null,
      header: {
        title: 'Course default',
        image: 'http://lorempixel.com/64/64/'
      }
    }
  },
  methods: {
    add: function () {
      this.$emit('add', { title: this.title, months: this.months });
    }
  },
}
```

```
template: [
  '<div :class=["course", styleClass]>',
    '<header class="course-header" v-once>',
      '',
      '<h2>{{ header.title }}</h2>',
    '</header>',
    '<main class="course-content">',
      '',
      '<section>',
        '<h3>{{ title }}</h3>',
        '<h4>{{ subtitle }}</h4>',
        '<p> {{ description }}</p>',
      '</section>',
    '</main>',
    '<footer class="course-footer">',
      '<label for="meses">MESES</label>',
      '<input id="meses" type="number" min="0" max="12" v-model="months" />'
  ,
    '<button @click="add">AÑADIR</button>',
  '</footer>',
  '</div>'
].join('')
};

Vue.component('course-js', {
  mixins: [course],
  data: function () {
    return {
      styleClass: 'course-js',
      header: {
        title: 'Curso JS',
        image: 'http://lorempixel.com/64/64/'
      }
    }
  },
});

Vue.component('course-css', {
  mixins: [course],
  data: function () {
    return {
      styleClass: 'course-css',
      header: {
        title: 'Curso CSS',
        image: 'http://lorempixel.com/64/64/'
      }
    }
  },
});
```

Lo que hemos hecho es sacar todo el constructor a un objeto llamado `course`. Este objeto contiene todo lo que nosotros queremos que el componente tenga como base. Lo siguiente es definir dos componentes nuevos llamados `course-js` y `course-css` donde indicamos en el parámetro `mixins` que queremos que hereden.

Por último, indicamos aquellos datos que queremos sobrescribir. Nada más. VueJS se encargará de componer el constructor a nuestro gusto y de generar los componentes que necesitamos. De esta forma podemos reutilizar código y componentes. Ahora podemos declarar nuestros componentes dentro del HTML de la siguiente forma:

```
<course-js
  title="Curso JavaScript"
  subtitle="Curso Introdutorio"
  description="Esto es un nuevo curso para aprender"
  @add="addToCart">
</course-js>
<course-css
  title="Curso CSS Avanzado"
  subtitle="Curso Avanzado"
  description="Esto es un nuevo curso para aprender"
  @add="addToCart">
</course-css>
```

Ambos componentes tienen la misma firma pero internamente se comportan de diferente manera.

En el futuro hablaremos más de mixins. Si necesitas saber más sobre ello, aquí puedes.

## Refactorizando el componente

Después de crear dos componentes más específicos, se me viene a la cabeza que ese template que estamos usando en `course`, presenta una estructura bastante compleja. Sería buena idea que refactorizásemos esa plantilla en trozos más pequeños y especializados que nos permiten centrarnos mejor en el propósito y la responsabilidad de cada uno de ellos.

Sin embargo, si vemos los componentes en los que podríamos dividir ese template, nos damos cuenta que por ahora, no nos gustaría crear componentes globales sobre estos elementos. Nos gustaría poder dividir el código pero sin que se encontrase en un contexto global. Esto en VueJS es posible.

En VueJS contamos con la posibilidad de tener componentes locales. Es decir, componentes que simplemente son accesibles desde el contexto de un componente padre y no de otros elementos.

Esto puede ser una buena forma de modularizar componentes grandes en partes más pequeñas, pero que no tienen sentido que se encuentren en un contexto global ya sea porque su nombre pueda chocar con el de otros, ya sea porque no queremos que otros desarrolladores hagan un uso inadecuado de ellos.

Lo que vamos a hacer es coger el siguiente template:

```

template: `
  <div :class="['course', styleClass]">
    <header class="course-header" v-once>
      
      <h2>{{ header.title }}</h2>
    </header>
    <main class="course-content">
      
      <section>
        <h3>{{ title }}</h3>
        <h4>{{ subtitle }}</h4>
        <p> {{ description }}</p>
      </section>
    </main>
    <footer class="course-footer">
      <label for="meses">MESES</label>
      <input id="meses" type="number" min="0" max="12" v-model="months" />
      <button @click="add">AÑADIR</button>
    </footer>
  </div>
`

```

Y lo vamos a convertir en los siguiente:

```

template: `
  <div :class="['course', styleClass]">
    <course-header :title="header.title" :image="header.image"></course-header>
    <course-content :title="title" :subtitle="subtitle" :description="description"></course-content>
    <course-footer :months="months" @add="add"></course-footer>
  </div>
`

```

Hemos sacado el header, el content y el footer en diferentes componentes a los que vamos pasando sus diferentes parámetros.

Los constructores de estos componentes los definimos de esta manera:

```
const courseHeader = {
  props: {
    image: { type: String, required: true },
    title: { type: String, required: true }
  },
  template: [
    '<header class="course-header" v-once>',
    '  ',
    '  <h2>{{ title }}</h2>',
    '</header>'
  ].join('')
};

const courseContent = {
  props: {
    title: { type: String, required: true },
    subtitle: { type: String, required: true },
    description: { type: String, required: true }
  },
  template: [
    '<main class="course-content">',
    '  ',
    '  <section>',
    '    <h3>{{ title }}</h3>',
    '    <h4>{{ subtitle }}</h4>',
    '    <p> {{ description }}</p>',
    '  </section>',
    '</main>'
  ].join('')
};

const courseFooter = {
  props: {
    months: { type: Number, required: true }
  },
  template: [
    '<footer class="course-footer">',
    '  <label for="meses">MESES</label>',
    '  <input id="meses" type="number" min="0" max="12" v-model="months" />',
    '  <button @click="add">AÑADIR</button>',
    '</footer>'
  ].join(''),
  methods: {
    add: function () {
      this.$emit('add', this.months );
    }
  },
};
```

Estos constructores podrían ser usados de forma global, y no estaría mal usado. Sin embargo, para el ejemplo, vamos a registrarlos de forma local en el componente `course` de esta manera:

```
const course = {
  // ... more code
  components: {
    'course-header': courseHeader,
    'course-content': courseContent,
    'course-footer': courseFooter
  },
  // ... more code
};
```

Todos los componentes cuentan con este atributo `components` para que registremos constructores y puedan ser usados.

Personalmente, creo que pocas veces vamos a hacer uso de un registro local, pero que contemos con ello, creo que es una buena decisión de diseño y nos permite encapsular mucho mejor a la par que modularizar componentes.

## Creando un componente contenedor

Una vez que hemos refactorizado nuestro componente `course`, vamos a crear un nuevo componente que nos permita pintar internamente estos cursos. Dentro de VueJS podemos crear componentes que tengan internamente contenido del cual no tenemos control.

Estos componentes pueden ser los típicos componentes layout, donde creamos contenedores, views, grids o tablas donde no se sabe el contenido interno. En VueJS esto se puede hacer gracias al elemento slot. Nuestro componente lo único que va a hacer es incluir un div con una clase que soporte el estilo flex para que los elementos se pinten alineados.

Es este:

```
Vue.component('marketplace', {
  template: [
    '<div class="marketplace">'
    '  <slot></slot>'
    '</div>'
  ].join('')
});
```

Lo que hacemos es definir un 'template' bastante simple donde se va a encapsular HTML dentro de slot. Dentro de un componente podemos indicar todos los slot que necesitemos. Simplemente les tendremos que indicar un nombre para que VueJS sepa diferenciarlos.

Ahora podemos declararlo de esta manera:

```
<marketplace>
  <component
    v-for="course in courses"
    :is="course.type"
    :key="course.id"
    :title="course.title"
    :subtitle="course.subtitle"
    :description="course.description"
    @add="addToCart">
  </component>
</marketplace>
```

Dentro de marketplace definimos nuestro listado de cursos.

Fijaros también en el detalle de que no estamos indicando ni `course` ni `course-js` ni `course-css`. Hemos indicado la etiqueta `component` que no se encuentra definida en ninguno de nuestros ficheros.

Esto es porque `component` es una etiqueta de VueJS en la que, en combinación con la directiva `:is`, podemos cargar componentes de manera dinámica. Como yo no se que tipo de curso va a haber en mi listado, necesito pintar el componente dependiendo de lo que me dice la variable del modelo `course.type`.

Para saber más sobre slots, tenemos esta parte de la documentación.

## Todo junto

Para ver todo el ejemplo junto, contamos con este código:

```
body {
  background: #FAFAFA;
}

.marketplace {
  display: flex;
}

.course {
  background: #FFFFFF;
  border-radius: 2px;
```

```
    box-shadow: 0 2px 2px rgba(0,0,0,.26);
    margin: 0 .5rem 1rem;
    width: 18.75rem;
}

.course .course-header {
    display: flex;
    padding: 1rem;
}

.course .course-header img {
    width: 2.5rem;
    height: 2.5rem;
    border-radius: 100%;
    margin-right: 1rem;
}

.course .course-header h2 {
    font-size: 1rem;
    padding: 0;
    margin: 0;
}

.course .course-content img {
    height: 9.375rem;
    width: 100%;
}

.course .course-content section {
    padding: 1rem;
}

.course .course-content h3 {
    padding-bottom: .5rem;
    font-size: 1.5rem;
    color: #333;
}

.course .course-content h3,
.course .course-content h4 {
    padding: 0;
    margin: 0;
}

.course .course-footer {
    padding: 1rem;
    display: flex;
    justify-content: flex-end;
    align-items: center;
}

.course .course-footer button {
    padding: 0.5rem 1rem;
}
```

```

    border-radius: 2px;
    border: 0;
  }

.course .course-footer input {
  width: 4rem;
  padding: 0.5rem 1rem;
  margin: 0 0.5rem;
}

.course.course-js .course-header,
.course.course-js .course-footer button {
  background: #43A047;
  color: #FFFFFF;
}

.course.course-css .course-header,
.course.course-css .course-footer button {
  background: #FDD835;
}

```

```

const courseHeader = {
  props: {
    image: { type: String, required: true },
    title: { type: String, required: true }
  },
  template: [
    '<header class="course-header" v-once>',
    '  ',
    '  <h2>{{ title }}</h2>',
    '</header>'
  ].join('')
};

const courseContent = {
  props: {
    title: { type: String, required: true },
    subtitle: { type: String, required: true },
    description: { type: String, required: true }
  },
  template: [
    '<main class="course-content">',
    '  ',
    '  <section>',
    '    <h3>{{ title }}</h3>',
    '    <h4>{{ subtitle }}</h4>',
    '    <p> {{ description }}</p>',
    '  </section>',
    '</main>'
  ].join('')
};

```

```

const courseFooter = {
  props: {
    months: { type: Number, required: true }
  },
  template: [
    '<footer class="course-footer">',
      '<label for="meses">MESES</label>',
      '<input id="meses" type="number" min="0" max="12" v-model="months" />',
      '<button @click="add">AÑADIR</button>',
    '</footer>'
  ].join(''),
  methods: {
    add: function () {
      this.$emit('add', this.months );
    }
  },
};

const course = {
  props: {
    title: { type: String, required: true },
    subtitle: { type: String, required: true },
    description: { type: String, required: true }
  },
  components: {
    'course-header': courseHeader,
    'course-content': courseContent,
    'course-footer': courseFooter
  },
  data: function () {
    return {
      months: 0,
      styleClass: null,
      header: {
        title: 'Course default',
        image: 'http://lorempixel.com/64/64/'
      }
    }
  },
  template: [
    '<div :class=["course", styleClass]>',
      '<course-header :title="header.title" :image="header.image"></course-header>',
      '<course-content :title="title" :subtitle="subtitle" :description="description"></course-content>',
      '<course-footer :months="months" @add="add"></course-footer>',
    '</div>'
  ].join(''),
  methods: {
    add: function (months) {
      this.$emit('add', { title: this.title, months: months });
    }
  }
};

```

```
    }
  };

  Vue.component('course-js', {
    mixins: [course],
    data: function () {
      return {
        styleClass: 'course-js',
        header: {
          title: 'Curso JS',
          image: 'http://lorempixel.com/64/64/'
        }
      }
    },
  });

  Vue.component('course-css', {
    mixins: [course],
    data: function () {
      return {
        styleClass: 'course-css',
        header: {
          title: 'Curso CSS',
          image: 'http://lorempixel.com/64/64/'
        }
      }
    },
  });

  Vue.component('marketplace', {
    template: [
      '<div class="marketplace">',
      '  <slot></slot>',
      '</div>'
    ].join('')
  });

  const app = new Vue({
    el: '#app',
    data: {
      courses: [
        {
          id: 1,
          title: 'Curso introductorio JavaScript',
          subtitle: 'Aprende lo básico en JS',
          description: 'En este curso explicaremos de la mano de los mejores profesores JS los principios básicos',
          type: 'course-js'
        },
        {
          id: 2,
          title: 'Curso avanzado JavaScript',
          subtitle: 'Aprende lo avanzado en JS',
```

```
        description: 'En este curso explicaremos de la mano de los mejores pro  
fesores JS los principios avanzados',  
        type: 'course-js'  
    },  
    {  
        id: 3,  
        title: 'Curso introductorio Cascading Style Sheets',  
        subtitle: 'Aprende lo básico en CSS',  
        description: 'En este curso explicaremos de la mano de los mejores pro  
fesores CSS los principios básicos',  
        type: 'course-css'  
    }  
],  
    cart: []  
},  
methods: {  
    addToCart: function (course) {  
        this.cart.push(course);  
    }  
}  
});
```

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <title>Example components</title>

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

<body>
  <div id="app">
    <marketplace>
      <component
        v-for="course in courses"
        :is="course.type"
        :key="course.id"
        :title="course.title"
        :subtitle="course.subtitle"
        :description="course.description"
        @add="addToCart">
      </component>
    </marketplace>

    <ul class="cart">
      <li v-for="course in cart">{{ course.title }} - {{ course.months }} meses</li>
    </ul>
  </div>

  <script src="node_modules/vue/dist/vue.js"></script>
  <script src="app.js"></script>
</body>

</html>
```

## Conclusión

Hemos explicado todo lo que tiene que ver con el corazón de la librería. Controlando y sabiendo cómo funcionan los componentes en VueJS, tendremos mucho recorrido ganado en poder desarrollar aplicaciones del mundo real.

Las propiedad, los eventos y los slots son una buena forma para diseñar componentes de una forma versátil y dinámica. Diseñar bien nuestros componentes será un primer paso a tener en cuenta si queremos que nuestra arquitectura triunfe, pero sí es importante tener en cuenta qué posibilidades nos da VueJS para hacer este diseño más robusto y constante.

No te preocupes si el ejemplo te parece bastante enrevesado o sin sentido. Por culpa de tener que explicar todos los casos posibles que se pueden dar en un componente, hemos tenido que complicarlo todo. En el futuro veremos que muchas de estas decisiones que hemos tomado, como la herencia o el registro local, se podría haber solucionado con el paso de un nuevo parámetro y el registro global.

En próximos posts, seguiremos hablando sobre componentes y seguiremos entendiendolos mejor.

Nos leemos :)

## Capítulo 5. El ciclo de vida de un componente

Todo componente tiene un ciclo de vida con diferentes estados por los que acaba pasando. Muchos de los frameworks y librerías orientados a componentes nos dan la posibilidad de incluir funciones en los diferentes estados por los que pasa un componente.

En el caso de VueJS existen 4 estados posibles. El framework nos va a permitir incluir acciones antes y después de que un componente se encuentre en un estado determinado. Estas acciones, conocidas como hooks, tienen varios propósitos para el desarrollador:

- Lo primero que nos van a permitir es conocer el mecanismo interno de cómo se crea, actualiza y destruye un componente dentro de nuestro DOM. Esto nos ayuda a entender mejor la herramienta.
- Lo segundo que nos aporta es la posibilidad de incluir trazas en dichas fases, lo que puede ser muy cómodo para aprender al principio cuando somos novatos en una herramienta y no sabemos como se va a comportar el sistema, por tanto, es un buen sistema para trazar componentes.
- Lo tercero, y último, es la posibilidad de incluir acciones que se tienen que dar antes o después de haber llegado a un estado interno del componente.

A lo largo del post vamos a hacer un resumen de todos los posibles hooks. Explicaremos en qué momento se ejecutan y cómo se encuentra un componente en cada uno de esos estados. Por último, pondremos algunos ejemplos para explicar la utilidad de cada uno de ellos:

### Creando el componente

Un componente cuenta con un estado de creación. Este estado se produce entre la instanciación y el montaje del elemento en el DOM. Cuenta con dos hooks. Estos dos hooks son los únicos que pueden interceptarse en renderizado en servidor (dedicaremos una entrada a esto en próximos capítulos), el resto, debido a su naturaleza, sólo pueden ser usados en el navegador.

**beforeCreate**

Este hook se realiza nada más instanciar un componente. Durante este hook no tiene sentido acceder al estado del componente pues todavía no se han registrado los observadores de los datos, ni se han registrado los eventos.

Aunque pueda parecer poco útil, utilizar este hook puede ser es un buen momento para dos acciones en particular:

- Para configurar ciertos parámetros internos u opciones, inherentes a las propias funcionalidad de VueJS. Un caso de uso común es cuando queremos evitar referencias circulares entre componentes. Cuando usamos una herramienta de empaquetado de componentes, podemos entrar en bucle infinito por culpa de dichas referencias. Para evitar esto, podemos cargar el componente de manera 'diferida' para que el propio empaquetador no se vuelva loco.

```
const component1 = {
  beforeCreate: function () {
    this.$options.components.Component2 = require('./component2.vue');
  }
};
```

- Para iniciar librerías o estados externos. Por ejemplo, imaginemos que queremos iniciar una colección en `localStorage` para realizar un componente con posibilidad de guardado offline. Podríamos hacer lo siguiente:

```
const component = {
  beforeCreate: function () {
    localStorage.setItem('tasks', []);
  }
};
```

## created

Cuando se ejecuta este hook, el componente acaba de registrar tanto los observadores como los eventos, pero todavía no ha sido ni renderizado ni incluido en el DOM. Por tanto, tenemos que tener en cuenta que dentro de `created` no podemos acceder a `$el` porque todavía no ha sido montado.

Es uno de los más usados y nos viene muy bien para iniciar variables del estado de manera asíncrona. Por ejemplo, necesitamos que un componente pinte los datos de un servicio determinado. Podríamos hacer algo como esto:

```
const component = {
  created: function () {
    axios.get('/tasks')
      .then(response => this.tasks = response.data)
      .catch(error => this.errors.push(error));
  }
};
```

## Montando el componente

Una vez que el componente se ha creado, podemos entrar en una fase de montaje, es decir, que se renderizará e insertará en el DOM. Puede darse el caso que al instanciar un componente no hayamos indicado la opción `el`. De ser así, el componente se encontraría en estado creado de manera latente hasta que se indique o hasta que ejecutemos el método `$mount`, lo que provocará que el componente se renderice pero no se monte (el montaje sería manual).

### beforeMount

Se ejecuta justo antes de insertar el componente en el DOM, justamente, en tiempo de la primera renderización de un componente. Es uno de los hooks que menos usarás y, como muchos otros, se podrá utilizar para trazar el ciclo de vida del componente.

A veces se usa para iniciar variables, pero yo te recomiendo que delegues esto al hook `created`.

### mounted

Es el hook que se ejecuta nada más renderizar e incluir el componente en el DOM. Nos puede ser muy útil para inicializar librerías externas. Imagínate que estás haciendo uso, dentro de un componente de VueJS, de un plugin de jQuery. Puede ser buen momento para ejecutar e iniciarlo en este punto, justamente cuando acabamos de incluirlo al DOM.

Lo usaremos mucho porque es un hook que nos permite manipular el DOM nada más iniciarlo. Un ejemplo sería el siguiente. Dentro de un componente estoy usando el plugin `button` de jQuery UI (Imaginemos que es un proyecto legado y no me queda otra). Podríamos hacer esto:

```
const component = {
  mounted: function () {
    $(".selector").button({});
  }
};
```

## Actualizando el componente

Cuando un componente ha sido creado y montado se encuentra a disposición del usuario. Cuando un componente entra en interacción con el usuario pueden darse eventos y cambios de estados. Estos cambios desembocan en la necesidad de tener que volver a renderizar e incluir las diferencias provocadas dentro del DOM de nuevo. Es por eso que el componente entra en un estado de actualización que también cuenta con dos hooks.

### beforeUpdate

Es el hook que se desencadena nada más que se provoca un actualización de estado, antes de que se comience con el re-renderizado del Virtual DOM y su posterior 'mapeo' en el DOM real.

Este hook es un buen sitio para trazar cuándo se provocan cambios de estado y producen renderizados que nosotros no preveíamos o que son muy poco intuitivos a simple vista. Podríamos hacer lo siguiente:

```
const component = {
  beforeUpdate: function () {
    console.log('Empieza un nuevo renderizado de component');
  }
};
```

Puedes pensar que es un buen sitio para computar o auto calcular estados a partir de otros, pero esto es desaconsejado. Hay que pensar que estos hooks son todos asíncronos, lo que significa que si su algoritmo interno no acaba, el componente no puede terminar de renderizar de nuevo los resultados. Con lo cual, cuidado con lo que hacemos internamente de ellos. Si necesitamos calcular cálculos, contamos con funcionalidad específica en VueJS por medio de Watchers o Computed properties.

### updated

Se ejecuta una vez que el componente ha re-renderizado los cambios en el DOM real. Al igual que ocurría con el hook `mounted` es buen momento para hacer ciertas manipulaciones del DOM externas a VueJS o hacer comprobaciones del estado de las variables en ese momento.

Puede que tengamos que volver a rehacer un componente que tenemos de jQuery, Aquí puede ser buen momento para volver a lanzarlo y hacer un refresh o reinit:

```
const component = {
  updated: function () {
    $(".selector").button("refresh");
  }
};
```

## Destruyendo el componente

Un componente puede ser destruido una vez que ya no es necesario para el usuario. Esta fase se desencadena cuando queremos eliminarlo del DOM y destruir la instancia de memoria.

### `beforeDestroy`

Se produce justamente antes de eliminar la instancia. El componente es totalmente operativo todavía y podemos acceder tanto al estado interno, como a sus propiedades y eventos.

Suele usarse para quitar eventos o escuchadores. Por ejemplo:

```
const component = {
  beforeDestroy() {
    document.removeEventListener('keydown', this.onKeyDown);
  }
};
```

### `destroyed`

Tanto los hijos internos, como las directivas, como sus eventos y escuchadores han sido eliminados. Este hook se ejecuta cuando la instancia ha sido eliminada. Nos puede ser muy útil para limpiar estados globales de nuestra aplicación.

Si antes habíamos iniciado el `localStorage` con una colección para dar al componente soporte offline, ahora podríamos limpiar dicha colección:

```
const component = {
  destroyed: function () {
    localStorage.removeItem('tasks');
  }
};
```

## Otros hooks

Existen otros dos hooks que necesitan una explicación aparte. Dentro de VueJS yo puedo incluir componentes dinámicos en mi DOM. De esta manera, yo puedo determinar, en tiempo de JavaScript, que componente renderizar. Esto lo veíamos en el post anterior y nos puede ser muy útil a la hora de pintar diferentes vistas de una WebApp.

Pues bien, VueJS no cuenta solo con eso, sino que cuenta con una etiqueta especial llamada `keep-alive`. Esta etiqueta, en combinación con la etiqueta `component`, permite cachear componentes que han sido quitados del DOM, pero que sabemos que pueden ser usados en breve. Este uso hace que tanto las fases de creación, como de destrucción, no se ejecuten por obvias y que de tal modo, se haya tenido que dar una opción.

VueJS nos permite engancharse a dos nuevos métodos cuando este comportamiento ocurre. Son los llamados `activated` y `deactivated`, que son usados del mismo modo que el hook `created` y el hook `beforeDestroy` por los desarrolladores.

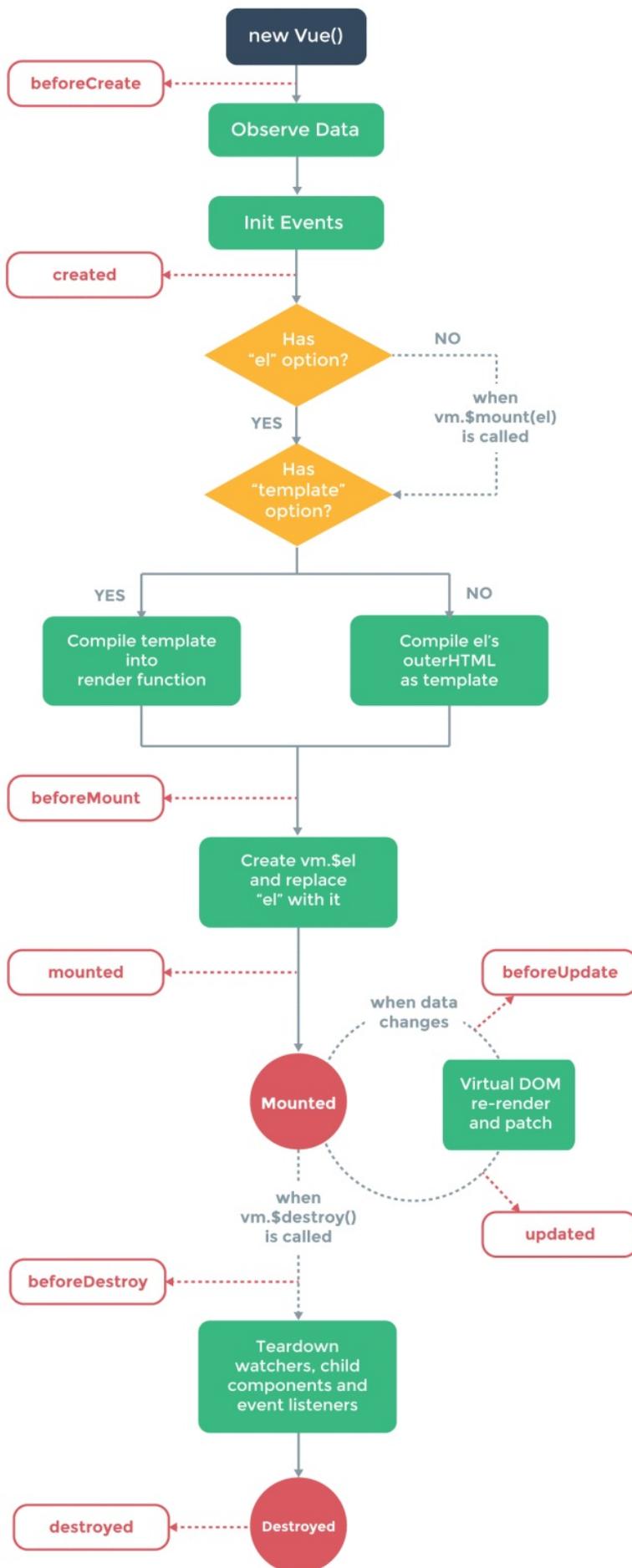
## Conclusión

Conocer el ciclo de vida de un componente nos hace conocer mejor VueJS. Nos permite saber cómo funciona todo y en qué orden.

Puede que en muchas ocasiones no tengamos que recurrir a ellos, o puede que en tiempo de depuración tengamos un mixin que trace cada fase para obtener información. Quién sabe. Lo bueno es la posibilidad de contar con ello. Lo bueno es el poder registrarnos en estos métodos y no depender tanto de la magia interna de un framework.

A mi por lo menos, que un framework cuente con estos mecanismos, me suele dar seguridad para llevar productos a producción con él.

Os dejo el diagrama que resume el ciclo de vida de un componente:



Nos leemos :)

## Capítulo 6. Definiendo componentes en un único fichero

En el post de hoy, nos vamos a centrar en cómo organizar nuestros componentes. Hemos trabajado con ejemplos de código muy sencillos que no contaban con más de dos o tres componentes. El sistema usado puede funcionar en aplicaciones pequeñas, aplicaciones con poca lógica interna o en la creación de un pequeño widget que queramos insertar en otra aplicación.

Cuando queremos hacer aplicaciones más grandes, el sistema utilizado (todos los componentes en un único fichero y registrado directamente en el framework) no escala. Necesitamos una forma de poder separar los componentes en diferentes ficheros y en usar herramientas que nos permitan empaquetar toda nuestra aplicación en un flujo dinámico y cómodo.

Lo que haremos, será explicar cómo empezar un proyecto VueJS a partir de las plantillas establecidas por la comunidad como estándar, y a partir de ahí, empezar a explicar las formas en las que podremos organizar las diferentes partes de nuestro código.

### Creando un proyecto con `vue-cli`

Cuando hemos decidido dar el paso de realizar nuestro próximo proyecto con VueJS, tendremos que tener claro si nos queremos meter en el ecosistema de esta plataforma. Hacer un SPA va mucho más allá de crear componentes, y casarnos con VueJS sin conocerlo bien, puede traer consecuencias.

Si hemos decidido que es el camino a seguir, VueJS no nos deja solos, sino que nos sigue ayudando en nuestra comprensión progresiva del framework. Lo mejor que podemos hacer para empezar un proyecto es hacer uso de su herramienta `vue-cli`. Esta herramienta es una interfaz de línea de comandos que nos va a permitir generar un proyecto con todo aquello necesario para empezar con VueJS.

Para instalar la herramienta, necesitamos tener instalado NodeJS y NPM. Lo siguiente es ejecutar el siguiente comando en el terminal:

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

Esto lo que hace es instalar la herramienta de `vue-cli` de manera global en el sistema para que hagamos uso de ella. Para saber si la herramienta se ha instalado correctamente, ejecutaremos el siguiente comando:

```
$ vue -V
```

Esto nos dirá la versión de `vue-cli` que tenemos instalada. En mi caso la 2.8.1.

Lo siguiente es hacer uso de ella. Vayamos desde el terminal a aquella carpeta donde queremos que se encuentre nuestro nuevo proyecto de VueJS. Lo siguiente es comprobar las plantillas que nos ofrece la herramienta. Para ello ejecutamos el siguiente comando:

```
$ vue list
```

Esto nos listará todas las plantillas. En el momento de crear este post contábamos con 5 maneras:

- **browserify**: nos genera una plantilla con todo lo necesario para que el empaquetado de nuestra SPA se haga con browserify.
- **browserify-simple**: es parecida a la anterior. Empaqueta con browserify, pero la estructura en carpetas será más simple. Nos será útil para crear prototipos.
- **simple**: Es una plantilla sencilla, muy parecida a la de los ejemplos de posts anteriores.
- **webpack**: igual que la de browserify, pero con el empaquetador webpack.
- **webpack-simple**: igual que browserify-simple, pero con webpack.

Nosotros nos vamos a basar en la plantilla de `webpack` para empezar nuestro proyecto. Para empezar el proyecto ejecutamos el siguiente comando:

```
$ vue init webpack my-new-app
```

Lo que este comando hace es generar una aplicación llamada `my-new-app` con la plantilla de `webpack`.

Lo que hará `vue-cli` es una serie de preguntas para que configuremos ciertos aspectos a nuestro gusto. En el momento de creación del post, eran las siguientes:

- **? Project name**: nos deja elegir un nombre para el proyecto, podemos coger el que hemos indicado por defecto.
- **? Project description**: una descripción que será incluida en el `package.json` del proyecto.
- **? Author**: El auto a incluir en el `package.json`
- **? Runtime + Compiler or Runtime-only**: nos deja elegir si queremos incluir el

compilador dentro de la solución.

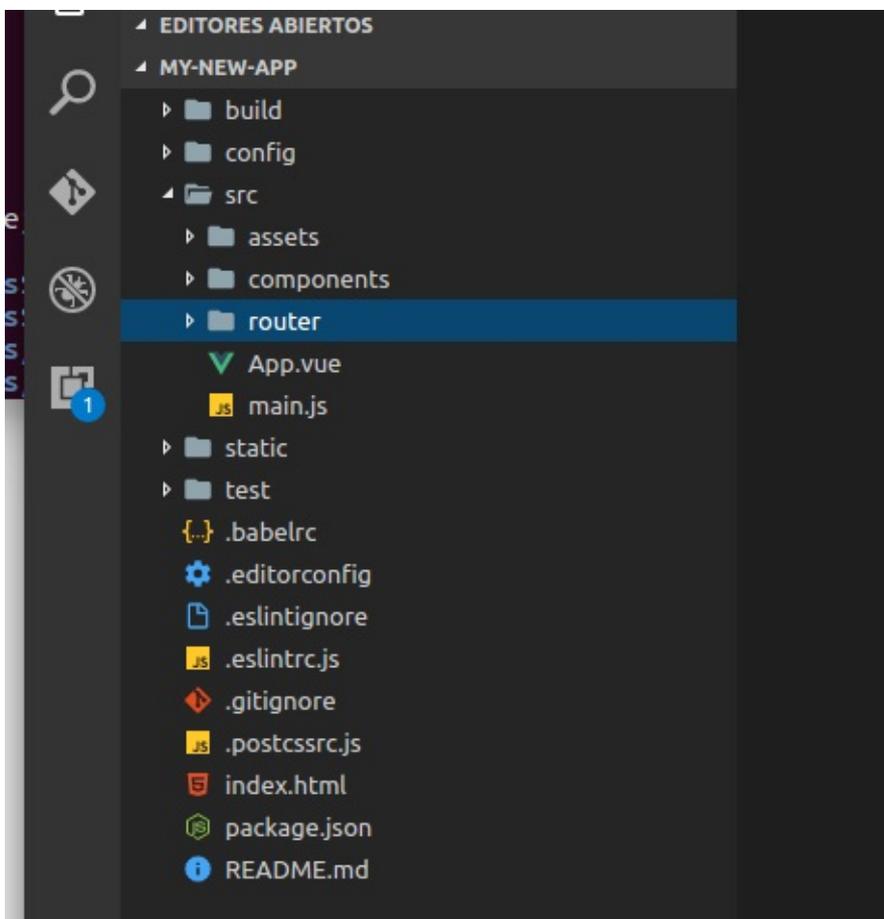
- **? Install vue-router:** Nos incluye un router por defecto en la solución y las dependencias necesarias.
- **? Use ESLint to lint your code:** Nos permite incluir un linter con la plantilla que deseemos para las reglas genéricas.
- **? Setup unit tests with Karma + Mocha:** Nos incluye las dependencias de test unitarios si lo deseamos.

Cuando hayamos contestado a ellas tendremos nuestra plantilla lista. Para cerciorarnos de que todo fue correctamente, lanzamos los siguientes comandos:

```
$ cd my-new-app
$ npm install
$ npm run dev
```

Lo que hace esto es navegar hasta la carpeta del proyecto generado, instalar todas las dependencias del proyecto y ejecutar la tarea de npm llamada dev que nos compila y empaqueta todo, lanza la app en el navegador y se queda escuchando a posibles cambios. Si todo fue bien se abrirá una web simplona.

Y ¿Qué ha hecho por debajo ese init? Pues nos ha generado una estructura en carpetas parecida a esta:



Explicamos cada carpeta y fichero a continuación:

- **build**: en esta carpeta se encuentran todos los scripts encargados de las tareas de construcción de nuestro proyecto en ficheros útiles para el navegador. Se encarga de trabajar con webpack y el resto de loaders (no entro más en webpack, porque además de no tener ni idea ahora mismo, le dedicaremos una serie en este blog en el futuro cuando por fin lo hayamos aprendido, por ahora tendremos que fiarnos de su magia :)).
- **config**: contiene la configuración de entornos de nuestra aplicación.
- **src**: El código que los desarrolladores tocarán. Es todo aquello que se compilará y formará nuestra app. Contiene lo siguiente:
  - **assets**: Aquellos recursos como css, fonts o imágenes.
  - **components**: Todos los componentes que desarrollaremos.
  - **router**: La configuración de rutas y estados por los que puede pasar nuestra aplicación.
  - **App.vue**: Componente padre de nuestra aplicación.
  - **main.js**: Script inicial de nuestra aplicación.
- **static**: Aquellos recursos estáticos que no tendrán que renderizarse, ni optimizarse. Pueden ser htmls o imágenes o claves.
- **test**: Toda la suite de test de nuestra aplicación
- **.babelrc**: Esta plantilla está configurada para que podamos escribir código ES6 con babel. Dentro de este fichero podremos incluir configuraciones de la herramienta.
- **.editorconfig**: Nos permite configurar nuestro editor de texto.
- **.eslintignore**: Nos permite indicar aquellas carpetas o ficheros que no queremos que tenga en cuenta eslint.
- **.eslintrc.js**: Reglas que queremos que tengan en cuenta eslint en los ficheros que está observando.
- **.gitignore**: un fichero que indica las carpetas que no se tienen que versionar dentro de nuestro repositorio de git.
- **.postcssrc.js**: Configuración de PostCSS.
- **index.html**: El html inicial de nuestra aplicación.
- **package.json**: El fichero con la meta información del proyecto (dependencias, nombre, descripción, path del repositorio...).
- **README.md**: Fichero markdown con la documentación del proyecto.

Esta estructura es orientativa y podremos cambiar aquello que no se adecue a nuestro gusto. Esta estructura es una entre muchas posibles. Lo bueno de usar esta plantilla o alguna similar es que, si mañana empezamos en otro proyecto que ya usaba VueJS, la fricción será menor.

## Formas de escribir el componente

Una vez que tenemos esto, podemos empezar a desarrollar componentes. Si vamos a la carpeta `@/src/components/` tendremos los componentes. Los componentes terminan con la extensión `.vue`. En este caso de la plantilla, encontraréis un componente llamado `Hello.vue` donde se encuentra todo lo necesario sobre el. Tanto el html, como su css, como su js, se encuentran aquí.

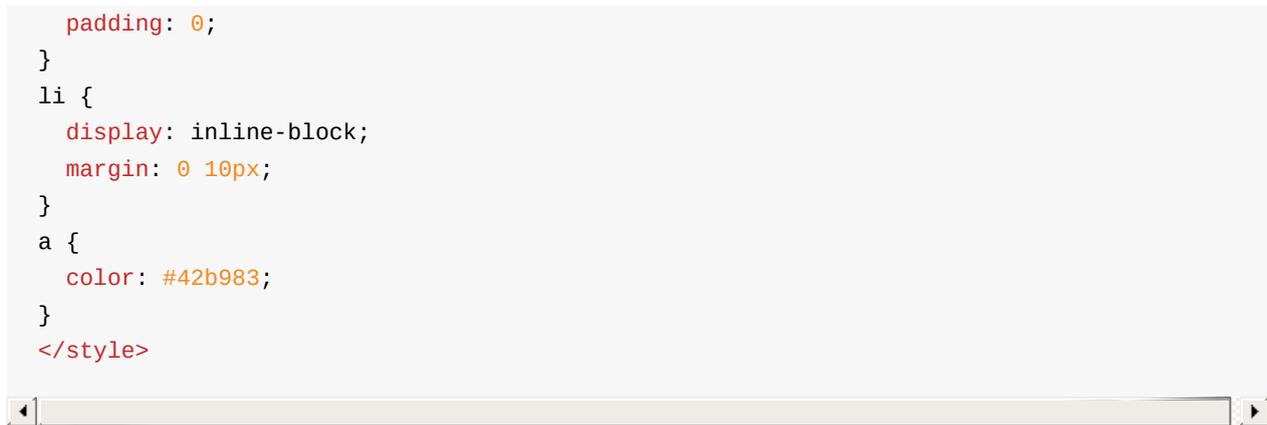
Es lo que VueJS llama como componente en un único fichero. El fichero internamente tiene una forma como la siguiente:

```
<template>
  <div class="hello">
    <h1>{{ msg }}</h1>
    <h2>Essential Links</h2>
    <ul>
      <li><a href="https://vuejs.org" target="_blank">Core Docs</a></li>
      <li><a href="https://forum.vuejs.org" target="_blank">Forum</a></li>
      <li><a href="https://gitter.im/vuejs/vue" target="_blank">Gitter Chat</a></li>
      <li><a href="https://twitter.com/vuejs" target="_blank">Twitter</a></li>
      <br>
      <li><a href="http://vuejs-templates.github.io/webpack/" target="_blank">Docs for
This Template</a></li>
    </ul>
    <h2>Ecosystem</h2>
    <ul>
      <li><a href="http://router.vuejs.org/" target="_blank">vue-router</a></li>
      <li><a href="http://vuex.vuejs.org/" target="_blank">vuex</a></li>
      <li><a href="http://vue-loader.vuejs.org/" target="_blank">vue-loader</a></li>
      <li><a href="https://github.com/vuejs/awesome-vue" target="_blank">awesome-vue</a
></li>
    </ul>
  </div>
</template>

<script>
export default {
  name: 'hello',
  data () {
    return {
      msg: 'Welcome to Your Vue.js App'
    }
  }
}
</script>

<!-- Add "scoped" attribute to limit CSS to this component only -->
<style scoped>
h1, h2 {
  font-weight: normal;
}
ul {
  list-style-type: none;
```

```
padding: 0;
}
li {
  display: inline-block;
  margin: 0 10px;
}
a {
  color: #42b983;
}
</style>
```



Encontramos tres etiquetas especiales: `template`, `script` y `style`, delimitan el html, el js y el css de nuestro componente respectivamente. El loader de VueJS es capaz de entender estas etiquetas y de incluir cada porción en sus paquetes correspondientes. Nosotros no tenemos que preocuparnos de ellos.

Esto nos da muchas posibilidades porque nos aísla muy bien. Si el día de mañana yo necesito un componente en otro proyecto, simplemente me tendré que llevar este fichero `.vue` y el componente seguirá funcionando en teoría igual.

Una buena característica es que el loader de VueJS (la pieza encargada de compilar el código en webpack) tiene compatibilidad con otros motores de plantillas como pug, con preprocesadores css como SASS o LESS y con transpiladores como Babel o TypeScript, con lo que no estamos limitados por el framework.

Además, cada parte está bien delimitada y no se sufren problemas de responsabilidad, ya que la presentación, la lógica y el estilo están bien delimitados. Es bastante bueno, porque el loader de VueJS nos va a permitir aislar estilos para un componente en particular. No hace uso de Shadow DOM como el estándar, pero si tiene un sistema para CSS Modules que nos va a permitir encapsular estilos si no nos llevamos demasiado bien con la cascada nativa (Esto se haría marcando la etiqueta `style` con el atributo `scope`).

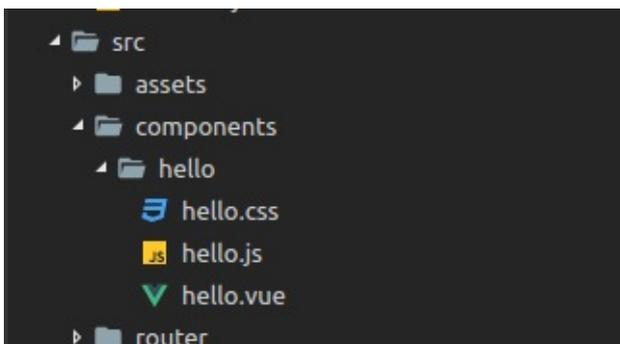
Si la forma del fichero no nos gusta, podemos separar las responsabilidades a diferentes ficheros y enlazarlos en el `.vue`. Podríamos hacer lo siguiente:

```

<template>
  <div class="hello">
    <h1>{{ msg }}</h1>
    <h2>Essential Links</h2>
    <ul>
      <li><a href="https://vuejs.org" target="_blank">Core Docs</a></li>
      <li><a href="https://forum.vuejs.org" target="_blank">Forum</a></li>
      <li><a href="https://gitter.im/vuejs/vue" target="_blank">Gitter Chat</a></li>
      <li><a href="https://twitter.com/vuejs" target="_blank">Twitter</a></li>
      <br>
      <li><a href="http://vuejs-templates.github.io/webpack/" target="_blank">Docs for
This Template</a></li>
    </ul>
    <h2>Ecosystem</h2>
    <ul>
      <li><a href="http://router.vuejs.org/" target="_blank">vue-router</a></li>
      <li><a href="http://vuex.vuejs.org/" target="_blank">vuex</a></li>
      <li><a href="http://vue-loader.vuejs.org/" target="_blank">vue-loader</a></li>
      <li><a href="https://github.com/vuejs/awesome-vue" target="_blank">awesome-vue</a
></li>
    </ul>
  </div>
</template>
<script src="./hello.js"></script>
<style src="./hello.css" scoped></style>

```

Ahora nuestro componente no se encuentra en un solo fichero, sino en 3. La carpeta es la que indica el componente entero. Esto quedaría así:



Puede ser un buen método si quieres que varios perfiles trabajen en tus componentes. De esta forma un maquetaador, o alguien que trabaje en estilos no tiene ni porqué saber que existe VueJS en su proyecto ya que el css está libre de nada que tenga que ver con ello.

## Conclusión

El post de hoy es corto, pero conciso. La forma en cómo organizas tu código es clave a la hora de desarrollar mejor. Saber dónde se encuentra cada cosa hará que ganemos en agilidad y mantenimiento. Aislar los componentes de esta forma puede ayudarnos mucho en el futuro.

La forma en que decidamos organizar internamente un componente, no deja de ser una cuestión de gusto, por lo que en esa cuestión no me meteré. Lo importante es que si dentro de un proyecto decides hacerlo de una manera, seas obcecado y siempre lo hagas de esa manera, hará que tu código sea más previsible y aburrido - y ser aburrido en ciertos aspectos es bueno :).

A partir de ahora, los ejemplos de la serie estarán escritos en este sistema de fichero único y sabiendo que usaremos la plantilla que `vue-cli` nos genera por defecto, por lo que puede ser buena idea practicar ahora para poder seguir la serie en el futuro.

Dejaremos por ahora los componentes y nos adentraremos durante unas semanas en el uso de vue-router, otra de las piezas importantes si hemos decidido hacer nuestra próxima SPA con VueJS. Por ahora, esto es todo amigos.

Nos leemos :)

## Capítulo 7. Introduciendo rutas en nuestra aplicación

Dejamos por un momento los componentes y nos centramos en una nueva funcionalidad del framework: el enrutamiento. Una vez que hemos creado nuestros componentes base o comunes (botones, inputs, listados, items, cartas...) y nuestros componentes de negocio (formularios, widgets, buscadores...) puede ser que necesitemos crear componentes vista o página para crear una aplicación completa (un SPA).

Dependiendo del número de vistas de las que se componga mi aplicación, la navegación entre ellas será más o menos fácil. Por ello, cuando una aplicación empieza a crecer en el número de vistas, es buena idea incluir algún mecanismo o herramienta que nos permita escalar este problema de la manera adecuada.

En el post de hoy - y en los sucesivos - veremos las formas en las que podemos incluir un sistema de navegación en nuestra aplicación VueJS de una manera escalable y poco intrusiva. Sígueme, por favor:

### ¿Qué es un sistema de rutas en mi aplicación?

Es un sistema que permite configurar la navegación de nuestra aplicación. Suele componerse de una librería que se encuentra interceptando la ruta que indicamos en nuestro navegador para saber en todo momento a qué estado de la aplicación debe moverse.

Un enrutador nos permite decir, para una url determinada, qué componente renderizar. Está muy basado en los sistemas de Modelo-Vista-Controlador y el diseño de API Rest. Suelen ser muy útiles para gestionar de una manera centralizada el comportamiento y el viaje que debe llevar un usuario por nuestra aplicación. Al final no deja de ser una forma de solucionar las diferentes direcciones web con las que cuenta mi aplicación en un sistema SPA.

### ¿Y si no necesitamos un sistema de rutas?

En muchas ocasiones, incluimos una librería compleja de gestión de la navegación sin plantearnos siquiera si lo necesitamos. Imaginad que hemos desarrollado una pequeña web donde presentamos nuestro producto, la típica web estática que no presenta más de 7 u 8 páginas diferentes.

Si por un casual, hemos decidido desarrollarla con VueJS, puede ser tentador usar su librería hermana vue-router. Sin embargo, quizá añadamos complejidad sin necesidad.

Y si no añadimos un enrutador ¿Cómo lo hacemos? Podemos preparar nosotros una pequeña solución que permita este dinamismo. Por ejemplo, con algo parecido a esto:

```
const NotFound = { template: '<p>Page not found</p>' };
const Home = { template: '<p>home page</p>' };
const About = { template: '<p>about page</p>' };

const routes = {
  '/': Home,
  '/about': About
};

new Vue({
  el: '#app',
  data: {
    currentRoute: window.location.pathname
  },
  computed: {
    ViewComponent () {
      return routes[this.currentRoute] || NotFound
    }
  },
  render (h) { return h(this.ViewComponent) }
});
```

Lo que hacemos, en este caso, [es apoyarnos en la funcionalidad de propiedades computadas que nos ofrece el framework](#) para conseguir dinamismo. Lo que conseguimos es que cada vez que la variable `currentRoute` cambie, se ejecute la función `ViewComponent` que devuelve el componente que hayamos configurado en nuestro array `routes`. Si la ruta puesta en el navegador no es correcta, renderizamos el componente `NotFound`.

La implementación es bastante sencilla y nos va a permitir la navegación por nuestra aplicación sin hacer mucho más. Si nuestra aplicación empieza a crecer, tenemos que tener en cuenta que una solución como esta es limitada y que deberemos ir pensando en incluir algo más elaborado.

## Y si lo necesitamos ¿Cómo empezamos?

Si necesitamos algo más elaborado, [puede ser una buena idea que diseñemos nuestra propia librería como hace Juanma en este post](#), o que usemos una de las librerías con las que ya contamos en la comunidad como es el caso de Director.

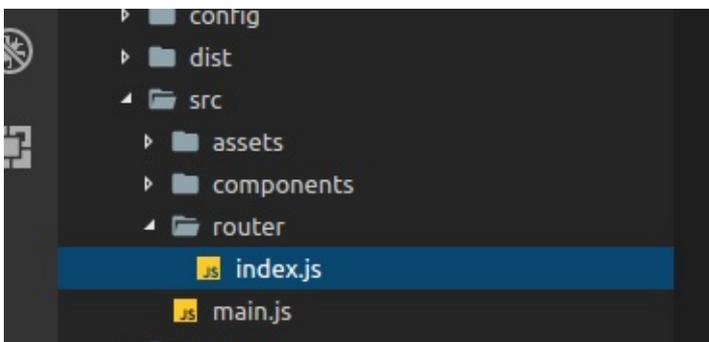
Tenemos también la opción de extender VueJS con una nueva librería llamada `vue-router`, esta es la opción que vamos a explicar. Es la opción que mejor se adapta al propio framework y que nos va a proporcionar todo lo necesario.

Para usarla tenemos varias formas: si es un proyecto que ya tenemos empezado, por medio de la línea de comandos:

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

O creando un proyecto desde 0 con `vue-cli`, indicando en la configuración del proyecto, que queremos una plantilla con `vue-router`, como ya explicamos en el post anterior.

Una vez que tenemos esto, lo siguiente será configurar nuestras rutas e indicar a VueJS que incluya esta configuración en su contexto. Para hacer esto, creamos una carpeta en la raíz del proyecto de esta manera:



Dentro del fichero `index.js` vamos a ir incluyendo toda la configuración de rutas de nuestra aplicación. Dentro de este fichero incluimos las siguientes líneas:

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

Vue.use(Router);

export default new Router({});
```

Lo que estamos haciendo es importar tanto la librería de `vue` como la de `vue-router`. Lo siguiente es extender VueJS por medio de `Vue.use(Router)`. De esta manera, extendemos la funcionalidad con un plugin. Por último, devolvemos la instancia del router que vamos a configurar.

Para terminar de integrar `vue-router` totalmente, solo nos falta inyectar esta instancia en todos los componentes. Esto lo conseguimos yendo al fichero `main.js`, donde se encuentra el 'setup' inicial de mi aplicación `vue`. Dentro ponemos lo siguiente:

```
import Vue from 'vue';
import router from './router';
import App from './components/app/app.vue';

new Vue({
  el: '#app',
  router,
  render: h => h(App)
});
```

Lo único que hacemos es inyectar en la instancia principal de nuestra aplicación vue nuestro router para que sea accesible a todo el árbol de componentes.

Si ahora queremos que estos componentes se pinten, `vue-router` cuenta con un componente específico donde se irá incluyendo el componente que la ruta nos indique. En nuestro componente app, hay que añadir el componente `<router-view>`. Lo único que hace este componente es sustituirse por nuestra vista.

Ya está. No necesitamos más fontanería. Ya podemos empezar a configurar rutas.

## ¿Cómo configuramos rutas?

Dentro de una instancia del router contamos con un parámetro llamado `routes` que permite configurar nuestras rutas. Por ejemplo, podríamos hacer lo siguiente:

```
// router/index.js
import HomeView from '@components/views/home-view.vue';

export default new Router({
  routes: [
    { path: '/home', component: HomeView }
  ]
});
```

De esta forma, cuando un usuario ponga en el navegador la ruta `www.mi-spa.com/#/home`, vue renderizará mi componente `HomeView`.

Dentro de este objeto podemos incluir otro nuevo parámetro llamado `name`. Este parámetro está muy bien para dar un nombre a nuestra ruta. De esta forma los desarrolladores desacoplan la url física del estado al que nos tenemos que dirigir y hace que podamos renombrar rutas muy largas. Por ejemplo:

```
// router/index.js
import HomeView from '@components/views/home-view.vue';

export default new Router({
  routes: [
    {
      path: 'my/shop/detail/go/www/43544352/app/home',
      name: 'home',
      component: HomeView
    }
  ]
});
```

Ahora podemos usar el nombre para referenciarla sin tener que usar toda la ruta. Es recomendable siempre ponerlo y usar ese nombre dentro de nuestra app.

Cómo decíamos, al final nuestro sistema de rutas está muy pensado para cargar estados o recursos en nuestros componentes de página y por ello, además de rutas estáticas, podemos usar rutas dinámicas. Esto significa que yo puedo indicar que se renderice siempre un componente para una serie de rutas que tienen patrones en común. Por ejemplo:

```
// router/index.js
import ProductDetailView from '@components/views/product-view.vue';

export default new Router({
  routes: [
    {
      path: 'products/:productId',
      name: 'product-detail',
      component: ProductDetailView
    }
  ]
});
```

Lo que hemos conseguido con esto es que tanto `/products/1234` como `products/3452` nos renderice el mismo componente. Siempre que queramos incluir una parte dinámica a nuestra ruta, tenemos que indicarlo con dos puntos `:`.

Este dinamismo nos puede ser muy útil para obtener productos por un id determinado de servidor dado que la parte dinámica es inyectada dentro de nuestros componentes en el campo `$route.params`.

Con este comportamiento se puede hacer cualquier cosa que se nos ocurra ya que `vue-router` usa la librería `path-to-regexp` para relacionar rutas por medio de expresiones regulares. Si necesitas algo mucho más específico sería bueno que le echases una ojeada.

Puedes preguntarte qué ocurriría si más de una ruta de las que has configurado coincide con la ruta especificada por el usuario. Las rutas son registradas en vue por orden en el array. Por tanto la prioridad será el orden en la que se especificó. En cuanto vue-router encuentra una coincidencia recorriendo el arreglo, ejecuta su renderizado. Tenlo en cuenta.

## ¿Cómo navegamos a nuevas rutas?

Una vez que hemos configurado nuestras rutas, podemos navegar entre ellas para que el usuario pueda realizar las acciones necesarias. Esta navegación la podemos hacer de dos maneras: de manera semántica por medio del componente `<router-link>` o de manera programática.

Si lo hacemos de manera semántica tendríamos que hacerlo de esta manera dentro de nuestros templates:

```
<router-link to="/home">Voy a home</router-link>
```

Esto se renderiza por lo siguiente:

```
<a href="/home">Voy a home</a>
```

El parámetro `to` nos acepta un objeto como valor para conseguir ciertas cosas, por ejemplo indicar parámetros. Por ejemplo, con el caso anterior de los productos, yo puedo hacer lo siguiente:

```
<router-link :to="{ name: 'product-detail', params: { productId: 1234 }}">
  Voy a ver el producto 1234
</router-link>
```

Ese objeto se puede definir dentro del JS perfectamente.

Si queremos crear navegaciones en tiempo de lógica del componente, contamos con 3 métodos para poder hacerlo. Yo puedo hacer lo siguiente:

```
$router.push('home');
```

Donde `home` es la ruta a la que me quiero dirigir. Este método tiene la siguiente firma:

```
$router.push(location, onComplete?, onAbort?);
```

Puede también que queramos navegar una ruta, pero que no queramos que se guarde en el histórico de navegación. En este caso usaríamos el método `replace` de esta manera:

```
$router.replace('home');
```

En este caso, iremos a la vista `home` y reemplazará a la ruta en la que estamos actualmente en el histórico.

Por último, contamos con un método llamado `go` que nos permite decir el número de saltos hacia delante o hacia detrás que queremos dar en el histórico de navegación:

```
$router.go(-1);
```

Esto iría a la ruta anteriormente visitada.

Una de las cosas que me gusta de esta API de navegación es que los nombres no han sido puesto de manera caprichosa. Si nos fijamos en ellos, son un mapeo 1 a 1 de la API `History`. De manera nativa los navegadores cuentan con `window.history.pushState`, `window.history.replaceState` y `window.history.go`. Esto hace que si hemos usado la API nativa, usarla en `vue` nos resulte muy intuitivo.

## Conclusión

Nunca se sabe de qué manera puede crecer una aplicación, por lo que suele ser difícil de antemano saber si se va a necesitar un sistema como `vue-router` o no. Es por ello que la forma en la que el ecosistema de `VueJS` nos permite ir incluyendo estas pequeñas funcionalidades, de manera progresiva, me parece todo un acierto para el aprendizaje y para la complejidad de nuestro proyecto.

Lo bueno de un sistema de enrutado como este es que, por lo general, se cuenta con una API muy sencilla, y aprender su mecanismo suele costar poco. La parte más difícil a la hora de desarrollar nuestra aplicación se encuentra en la parte de diseño. La parte donde tenemos que decidir como va a ser el flujo y la experiencia del usuario entre pantallas. Si tenemos claro en qué estado se tiene que encontrar en cada momento nuestro usuario, el resto es pan comido.

En los próximos posts, profundizaremos en la librería y aprenderemos sobre el ciclo de vida que tiene una ruta en nuestro sistema y en cómo sacar partido a sus hooks. Hasta el momento, esto es todo.

Nos leemos :)



## Capítulo 8. Interceptores de navegación entre rutas

Una vez que hemos visto los conceptos básicos de la librería de rutas, es el turno de profundizar en otros conceptos. Dentro de nuestra SPA, nos vendría bien tener algún tipo de mecanismo para saber, en ciertos momentos de una navegación, qué hacer en ciertas situaciones.

La librería de vue-router cuenta en su haber con una funcionalidad para esto llamada Navigation Guards o interceptores de navegación. Estos interceptores son una serie de funciones que nos van a permitir realizar diferentes acciones entre la navegación de una ruta a otra.

Por ejemplo, estos interceptores nos pueden venir bien para hacer ciertas comprobaciones o modificaciones del estado de la aplicación. Nos pueden venir bien para realizar ciertas redirecciones o para abortar una navegación si algo no se encuentra cómo el sistema espera.

vue-router cuenta con varias opciones para incluir estos interceptores que van desde el registro del interceptor de manera global hasta el registro del interceptor de manera local. A lo largo del post vamos a explicar cada uno de ellos y el uso que le podemos dar:

### Interceptores globales

#### beforeEach

Dentro de vue-router contamos con la posibilidad de registrar un interceptor que se ejecutará cada vez que se realice un cambio de ruta. Este interceptor se ejecuta de manera global, es decir, para toda las rutas a las que naveguemos, justamente antes de producirse la navegación.

La forma de registrar un interceptor global es con la siguiente sintaxis:

```
const router = new VueRouter({ ... });

router.beforeEach((to, from, next) => {
  // ...
});
```

beforeEach nos inyecta tres parámetros en la función de callback:

- **to**: Es el objeto router con la información de la ruta a la que voy.
- **from**: Es el objeto router con la información de la ruta de la que vengo.
- **next**: Es la función que me permite reanudar la navegación. Es una función que tiene un comportamiento bastante complejo ya que nos permite diferentes datos de entrada.

Por ejemplo:

- Si la ejecutamos sin parámetro ( `next()` ), la navegación se reanudará hacia la ruta indicada en `to` .
- Si la ejecutamos con una cadena que nos indique otra ruta ( `next('/')` ), nos redirigirá a la ruta que le hemos indicado.
- Si indicamos un valor booleano `false` ( `next(false)` ), abortará la redirección.
- Incluso si yo paso la instancia de un error ( `next(new Error())` ), la navegación se abortará y se inyectará con esta instancia el callback de la función registrada en `onError` . Como veis, muy completito. Siempre debemos ejecutar esta función para que el flujo continúe.

Puedo incluir dentro de mi aplicación todos los interceptores globales que yo necesite. Por ejemplo, puedo tener esto:

```
const router = new VueRouter({ ... });

router.beforeEach((to, from, next) => {
  // ...
});

router.beforeEach((to, from, next) => {
  // ...
});
```

El orden de ejecución es FIFO (El primero en registrarse es el primero en ejecutarse).

Los interceptores globales nos pueden venir muy bien para comprobar algún estado global de la aplicación. Por ejemplo, puede venirnos muy bien para comprobar si un usuario en particular va a poder tener acceso a una determinada parte de la aplicación o no.

```
function existToken() {
  return !!localStorage.token;
}

router.beforeEach((to, from, next) => {
  if (to.path !== '/login' && existToken()) {
    next();
  } else {
    next('login');
  }
});
```

El ejemplo es un caso muy simplificado de control de acceso en la parte privada de una aplicación. Lo que hace es comprobar si la aplicación va a navegar a una ruta diferente de login. De ser así, comprueba que tenga un token de sesión con la API, lo que significa que se tiene acceso. Si lo tiene, continúa con la navegación normal. De no ser así, redirige a la vista de login.

Será uno de los interceptores que más usemos.

## afterEach

Este interceptor se ejecuta después de que todos los interceptores de los componentes se hayan ejecutado. Su sintaxis es esta:

```
router.afterEach((to, from) => { // ... });
```

Como podemos apreciar, en este caso no se inyecta la función `next` por lo que no será posible influir en la navegación.

Es un interceptor que vamos a usar poco y que nos puede venir bien para depurar las rutas de navegación.

## Interceptores locales a una ruta

En ocasiones, puede que necesitamos influir en la navegación de una sola ruta y no en cada una de ellas. Cuando necesitamos un uso más específico en una ruta, podemos registrar una función en tiempo de configuración.

Por ejemplo, podemos hacer esto:

```
const router = new VueRouter({
  routes: [
    {
      path: '/login',
      component: LoginView,
      beforeEnter: (to, from, next) => {
        delete localStorage.token;
        next();
      }
    }
  ]
});
```

Lo que hacemos es limpiar la sesión del usuario antes de navegar a login. De esta manera nunca se nos olvidará quitar privilegios al usuario si se ha hecho un logout.

Este interceptor puede sernos útil para evitar ir a rutas intermedias en un proceso de compra. Por ejemplo, podemos evitar mostrar la vista detalle si el usuario no ha seleccionado antes en una vista previa ningún producto.

## Interceptores locales a un componente

Podemos localizar todavía más cuándo interceptar la navegación. Podemos incluir interceptores a nivel de un componente. Dependiendo del contexto de navegación en el que se encuentre un componente, podremos hacer unas acciones u otras.

Este mecanismo interno en un componente es muy importante porque la librería `vue-router` cachea los componentes y evita que ciertos hooks - como los de creación o destrucción - no sean ejecutados siempre. Por tanto, estos interceptores nos podrán ayudar porque tienen acceso a la instancia interna del propio componente.

Si yo quisiera interceptar comportamientos de navegación para un componente, lo haría de la siguiente forma:

```
const CartSummary = {
  template: `...`,
  beforeRouteEnter (to, from, next) { },
  beforeRouteUpdate (to, from, next) { },
  beforeRouteLeave (to, from, next) { }
};
```

Estudiemos cada uno de ellos:

### `beforeRouteEnter`

En este interceptor entramos cuando la navegación ha sido confirmada, pero todavía no se ha creado, ni renderizado el componente. Nos puede venir bien usarlo para saber si el componente tiene algún comportamiento de navegación especial antes de pintarse.

Si nosotros queremos influenciar en el estado de un componente, tenemos que esperar a que sea creado. Para conseguir esto, podemos hacerlo pasándole una función a `next()` de la siguiente manera:

```
const CartSummary = {
  template: `...`,
  beforeRouteEnter (to, from, next) {
    next(vm => vm.products = []);
  }
};
```

Este interceptor, es un buen sitio para iniciar el estado de un componente con datos de un servicio externo. Por ejemplo, voy a traer los productos de mi backend y a cargarlos:

```
const CartSummary = {
  template: `...`,
  beforeRouteEnter (to, from, next) {
    axios.get('/products', (res) => {
      next(vm => vm.products = res.data);
    });
  }
};
```

Como los interceptores detienen la navegación hasta que se ejecuta la función `next()`, nos vienen bien para gestionar este asincronismo.

## beforeRouteUpdate

Se ejecuta cuando la ruta de navegación cambia y el componente está siendo reutilizado para la siguiente ruta o por la misma. Por ejemplo, imaginemos que la ruta ha cambiado en la parte dinámica, en sus parámetros.

Como dijimos, los componentes se encuentran cacheados y no ejecutan sus hooks de creación y destrucción. Como este interceptor tiene acceso a la instancia del componente, puede ser un buen momento para manipular el estado según las necesidades nuevas.

```
const CartSummary = {
  data() {
    return {
      products: null,
    };
  },
  template: `...`,
  beforeRouteUpdate (to, from, next) {
    this.products = [];
  }
};
```

Sin callbacks ni artificios. Directamente accediendo a la instancia porque el componente ya se encuentra instanciado. Nada de fontanería.

## beforeRouteLeave

Por último, contamos con este interceptor que se ejecuta antes de cambiar de ruta y sabiendo que el componente no va a ser utilizado.

Como se ejecuta antes de ir a la nueva navegación, tiene acceso al estado del componente. Es muy utilizado para evitar navegaciones involuntarias y sin querer. Imagínate que el usuario ha dado sin querer a salir de la compra y tiene todo relleno.

Podemos usar ese interceptor para poner un popup e indicar si el usuario está de acuerdo con no guardar los cambios realizados.

```
const CartSummary = {
  template: `...`,
  beforeRouteLeave (to, from, next) {
    this.popup()
      .then(next)
      .catch(() => next(false));
  }
};
```

## ¿Cuál es el flujo de ejecución de estos Guards?

Como parece un poco confuso cuando se ejecuta cada uno de los interceptores. Os pongo una guía del flujo que se sigue:

1. La navegación es activada.
2. Se llama a todos los `beforeRouterLeave` que se hayan registrado y que no van a ser reutilizados en la siguiente ruta a la que voy.
3. Se llama a todos los interceptores globales `beforeEach`.
4. Se llama a todos los `beforeRouteUpdate` de los componentes que van a ser reutilizados en la siguiente ruta a la que voy.
5. Se llama a `beforeEnter` que hemos configurado en la ruta a la que voy.
6. Se resuelven toda la asincronía de componentes de esa ruta.
7. Se llama a `beforeRouteEnter` de los componentes que van a estar activos.
8. Se da la navegación como confirmada.
9. Se llama al interceptor `afterEach`.
10. Se llama a los callbacks pasados a `next` in `beforeRouteEnter`.

11. Y vuelta a empezar cuando se lanza una nueva navegación.

## Conclusión

Una de las cosas malas de usar frameworks y librerías de terceros es que te tienes que adaptar a lo que ellos entienden por un buen momento para que tu enganches tu funcionalidad. Quizá para muchos estos interceptores cumplan con sus exigencias, quizá para otros esto sea un impedimento.

Lo bueno es que VueJS ha pensado en ello y nos da bastantes opciones para poder redirigir o abortar navegaciones o incluso hacer acciones de borrado, actualización o creación de estados influidos por la navegación que se nos pide.

El problema de los interceptores suele ser el de siempre: el de tener funcionalidad ejecutándose de una manera asíncrona, lo que hace más difícil depurar si no somos metódicos y no tenemos en cuenta cuando incluirlos y cuando no.

Me he encontrado muchas veces interceptores dios que hacen más de lo que deben o que incluso que no respetan la regla de única responsabilidad. Tengamos en cuenta que este uso debería ser un recursos limitado y muy justificado en nuestro contexto.

Nos leemos :)

## Capítulo 9. Conceptos avanzados

Con lo aprendido hasta ahora sobre `vue-router`, podríamos cubrir gran parte de la funcionalidad necesaria para un buen número de aplicaciones.

Sin embargo, cuando nos enfrentamos a aplicaciones más grandes, tener en cuenta otras posibilidades nos puede ayudar en términos de reutilización y buenas prácticas.

El posts de hoy está dedicado a estudiar todos aquellos conceptos avanzados de `vue-router` que pueden ayudarnos a mejorar y a darnos mayor versatilidad cuando desarrollamos en un proyecto con `vue`.

Para conseguir esto, nos centraremos en el anidamiento de rutas, el paso de propiedades a un `ComponentView`, la inclusión de meta información y el cambio de comportamiento del scroll de nuestra web. Vayamos con ello:

### Anidar rutas

Durante esta serie hemos hecho ejemplos con rutas bastantes simples. Los sistemas de navegación de los ejemplos siempre han sido de una ruta a otra pero, ¿qué ocurre cuando dentro de nuestro sistema existe una navegación entre subrutas?

Imaginemos por un momento, que tenemos que desarrollar una aplicación bancaria y que estamos implementando la funcionalidad de transferencias entre cuentas. Nuestra aplicación podría contar, por ejemplo, con un proceso dividido en 3 pantallas:

- Una pantalla para configurar los datos de la transferencia: En esta pantalla, el usuario indica el importe y el destinatario al que realizar la transferencia.
- Una pantalla para mostrar el detalle de la transferencia: Nos puede ser muy útil para mostrar el estado actual de la cuenta, el día en que se hará la transferencia y el cálculo de las comisiones que conlleva la operación.
- Y una pantalla de confirmación de la transferencia realizada: Se le muestra al usuario cuando toda la operación de transferencia fue correctamente.

Para realizar esto, podríamos diseñar 3 vistas que se corresponderían con estas rutas:

```
/transfer/config  
/transfer/detail  
/transfer/confirm
```

Como vemos, la propia funcionalidad, me lleva a tener un subenrutado o anidamiento de rutas.

Estos procesos suelen conllevar una interfaz parecida en cada vista para favorecer la navegación al usuario. Por ejemplo, las 3 vistas van a compartir unas cabeceras y un componente de navegación que nos indique en qué paso de la realización de la transferencia nos encontramos.

Para desarrollar esto, podemos hacerlo de 3 maneras diferentes:

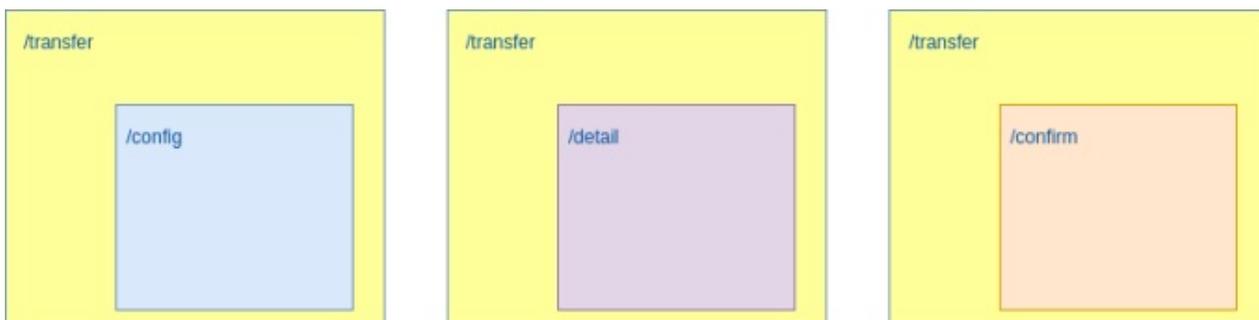
1. Creando un HTML diferente para cada una de las vistas.
2. Extrayendo todo el HTML común y creando componentes que se utilizasen en las 3 vistas.
3. Creando una 'layout' que contenga toda la parte común a las 3 vistas e ir rellenando de manera dinámica la diferencia entre vistas.

La opción 1 la descartamos porque no respeta los principios de reutilización. Si necesito hacer un cambio en cualquier momento, tengo que cambiar 3 vistas.

La opción 2 es bastante buena, pero no llega a ser lo suficientemente reutilizable; Sí, estamos reutilizando componentes comunes, pero aún hay partes que pueden cambiar en el tiempo y que me hagan trabajar más de la cuenta, manteniendo código idéntico.

La opción 3 es la mejor porque respeta los principios de reutilización ya que toda la parte común se encuentra encapsulada en un componente padre que orquesta a 3 componentes hijos.

La idea es hacer algo como esto:



Desarrollar un componente común que se llame `TransferView` que haga de componente contenedor, y 3 componentes hijos, más específicos, llamados `ConfigView`, `DetailView` y `ConfirmView`.

Aunque con vue tendríamos soporte para desarrollar esto, vue-router nos da una funcionalidad que se apoya en un sistema de rutas anidadas. Yo podría configurar mi ruta de la siguiente manera:

```
const TransferView = {
  template: `
    <div>
      <h2>Cabecera func. transferencia</h2>
      <router-view></router-view>
    </div>
  `
};

const ConfigView = {
  template: '<h3>Paso configuración</h3>'
};

const DetailView = {
  template: '<h3>Paso detalle</h3>'
};

const ConfirmView = {
  template: '<h3>Paso confirmación</h3>'
};

const router = new VueRouter({
  routes: [
    {
      path: '/transfer',
      component: TransferView,
      children: [
        { path: 'config', component: ConfigView },
        { path: 'detail', component: DetailView },
        { path: 'confirm', component: ConfirmView }
      ]
    }
  ]
});
```

Lo que hacemos es incluir un nuevo parámetro en la ruta llamado `children`. En este campo, podemos configurar todas las rutas anidadas que necesitemos. En nuestro caso 3.

Lo último que hemos hecho es añadir el componente `router-view` en nuestro componente padre `TransferView`. De esta manera vue-router sabe en qué parte tiene que renderizar el componente hijo.

Este anidamiento puede ser todo lo profundo que queramos, simplemente tenemos que incluir el parámetro `children`, configurar las rutas con su componente hijo que queramos e indicar en el componente padre dónde pintarlo por medio de `router-view`.

El anidamiento suele utilizarse también para crear el layout principal de nuestra aplicación. Como todas las vistas contarán con el header, el menú y el footer, lo que se hace es crear un componente `AppView` que contiene estos elementos y un componente `router-view`

donde se renderizará la parte de la vista más específica.

## Pasar propiedades a un ComponentView

Aunque un `ComponentView` es un componente muy específico dentro de nuestra aplicación, y es bastante improbable que pueda ser reutilizado por estar tan acoplado con el negocio, es buena idea usar las mismas buenas prácticas que usamos en componentes más específicos.

Cuando un componente de vista hace uso de parámetros para su correcto funcionamiento, es necesario que no lo acoplemos, dentro de lo posible, a `vue-router` y que usemos la funcionalidad de propiedades de entrada especificada por vue.

Por tanto, intentemos evitar este tipo de accesos:

```
const ProductDetailView = {
  template: '<label>ProductId {{ $route.params.productId }}</label>'
};

const router = new VueRouter({
  routes: [{
    path: '/products/:id',
    component: ProductDetailView
  }]
});
```

Por el siguiente:

```
const ProductDetailView = {
  props: ['productId'],
  template: '<label>ProductId {{ productId }}</label>'
};

const router = new VueRouter({
  routes: [{
    path: '/products/:id',
    component: ProductDetailView,
    props: true
  }]
});
```

Lo que hemos hecho es decir a la ruta que active el paso de parámetros por medio de propiedades al componente. De esta forma el componente está más desacoplado y nos ayuda a su reutilización y a ser probado de manera aislada.

El atributo `props` nos permite varios valores según nuestras necesidades. Por ejemplo, puedo pasar un objeto para 'mapear' manualmente las propiedades con los parámetros. O incluso, puede que la ruta no cuente con parámetros, pero sí con propiedades que hay que iniciar. Este caso nos puede ser útil.

Un caso puede ser este.

```
const router = new VueRouter({
  routes: [
    {
      path: '/promotion/from-newsletter',
      component: PromotionView,
      props: { newsletterPopup: false }
    }
  ]
});
```

Hacemos que el popup del componente no se muestre al principio al navegar a esta ruta. No recibimos parámetros de la ruta, pero si pasamos propiedades al componente.

Puedo pasar también una función. Nos puede ayudar a tomar decisiones sobre el valor que quiero incluir en la propiedad del componente dependiendo de la ruta. Por ejemplo:

```
const router = new VueRouter({
  routes: [
    {
      path: '/search',
      component: SearchView,
      props: (route) => ({ query: route.query.q })
    }
  ]
});
```

En este caso usamos la función para incluir la query realizada en la url como propiedad del componente `SearchView`.

## Incluir meta-información de una ruta

Puede darse el caso que necesitemos incluir información específica para una ruta. Puede sernos útil marcar ciertas rutas para influir en su comportamiento. Esto se puede hacer incluyendo nuevos campos en el atributo `meta` del objeto `route` de la siguiente manera:

```
const router = new VueRouter({
  routes: [
    {
      path: '/login',
      component: LoginView,
      meta: {
        isPublic: true
      }
    }
  ]
});
```

De esta manera estamos marcando que la ruta login es pública para cualquier usuario. Hay que tener cuidado con esta meta información porque es heredada de padres a hijos por lo que tengámoslo en cuenta si pensamos que algo está yendo mal.

Esta información es inyectada a los componentes dentro de `$route.matched` y es accesible tanto dentro de los componentes como de los interceptores de navegación. Por ejemplo, podemos combinar este campo con el interceptor `beforeEach` :

```
router.beforeEach((to, from, next) => {
  if (!to.matched.some(record => record.meta.isPublic) && !auth.loggedIn()) {
    next({
      path: '/login', query: { redirect: to.fullPath }
    });
  } else {
    next();
  }
});
```

Si la ruta no es pública, ni el usuario tiene autorización en el sistema, se devuelve al usuario a la pantalla de login. Esto se ejecutará para todas las rutas a las que naveguemos.

## Poner nuestro sistema de rutas en modo History de HTML5

Un SPA suele contar con un sistema de rutas precedido por una almohadilla. De esta forma, el navegador sabe que no tiene que resolver la ruta contra un servidor, ni hace una recarga de la página, sino que intenta solucionar la ruta a nivel interno de navegador. De esta manera, la librería de rutas intercepta el nuevo valor y actúa según su configuración.

Los que ya hemos trabajado con otros SPAs sabemos que las rutas suelen tener esta apariencia:

```
https://my-web.com/#/app/login
```

Evitar una URL como esta puede deberse a 3 razones:

- Puede que desde negocio se quiera trabajar en un sistema de rutas usable para que el usuario pueda recordarlas o guardarlas en favoritos.
- Puede que no nos interese como desarrolladores dar pistas al usuario del tipo de herramientas que usamos para nuestro desarrollo (Cuando en una web se ve un hash de este tipo es indicativo de SPA).
- Puede que tengamos problemas de SEO al no poder acceder a ciertas rutas.

Sería bastante bueno que nuestra librería pudiese entender correctamente la siguiente ruta:

```
https://my-web.com/app/login
```

Esto es posible en vue gracias a nuestra librería de rutas `vue-router`. Yo puedo configurarlo de esta manera para que se comporte de forma nativa a como lo hace el navegador:

```
const router = new VueRouter({
  mode: 'history',
  routes: [...]
});
```

De esta forma, tenemos el comportamiento esperado. Lo malo de esto es que ahora todas las rutas harán una petición a nuestro servidor de aplicaciones y que dependiendo de la herramienta que usemos, el error será gestionado de una u otra manera, pudiendo romper el funcionamiento correcto de nuestra aplicación.

Este mecanismo se puede configurar de diferentes formas. Por ejemplo, si nuestra aplicación es servida por un Apache, debemos registrar la siguiente regla en su configuración:

```
<IfModule mod_rewrite.c>
  RewriteEngine On
  RewriteBase /
  RewriteRule ^index\.html$ - [L]
  RewriteCond %{REQUEST_FILENAME} !-f
  RewriteCond %{REQUEST_FILENAME} !-d
  RewriteRule . /index.html [L]
</IfModule>
```

Esta regla nos redirige a `index.html` cada vez que no encuentra la ruta especificada. De esta forma, la aplicación sabrá reiniciarse correctamente.

Si usamos un NGINX, lo deberíamos hacer así:

```
location / {
    try_files $uri $uri/ /index.html;
}
```

Si queremos que el propio NodeJS nos gestione esto, contamos con un middleware de Express que nos permite configurar este redireccionamiento a nivel de servidor: `connect-history-api-fallback`.

El problema que seguimos teniendo con esto es que si la ruta no existe se nos seguirá redirigiendo a `index.html` no dando información al usuario de que esa ruta no existe. Para solucionar esto podemos registrar una ruta genérica en `vue-router` que siempre se ejecutará cuando ninguna otra regla haya conseguido relacionarse:

```
const router = new VueRouter({
  mode: 'history',
  routes: [{
    path: '*',
    component: NotFoundView
  }]
});
```

Para cualquier ruta (wildcard `*`), mostramos el componente `NotFoundView`. De esta forma siempre damos una información adecuada al usuario.

## Cambiar el comportamiento del scroll

Otro 'daño colateral' de usar el modo histórico de HTML5 y no el comportamiento de URLs por defecto de un SPA, es que podemos gestionar en qué parte del scroll queremos colocar nuestra nueva vista; Cuando naveguemos podemos preferir que la nueva vista siempre se sitúe en su parte superior o que mantenga el scroll de la ruta de la que procedemos.

En el objeto `router` contamos con otro parámetro para gestionar esto:

```
const router = new VueRouter({
  routes: [...],
  scrollBehavior (to, from, savedPosition) {
    return { x: 0, y: 0 }
  }
});
```

Esta función `scrollBehavior` nos permite acceder a los datos de la ruta de la que vengo y de la que voy. Además, se cuenta con un parámetro opcional que solo es inyectado si el cambio de ruta es provocado por los botones de navegación del propio navegador.

En el ejemplo de arriba, hemos indicado que scroll siempre se sitúe en la parte superior e izquierda de la página. Puede darse el caso que queramos simular una navegación hasta un 'anchor' determinado. Esto lo podríamos conseguir así:

```
scrollBehavior (to, from, savedPosition) {
  if (to.hash) {
    return {
      selector: to.hash
    }
  }
}
```

Esta función tiene acceso a los datos de la meta información de una ruta, por lo que el comportamiento del scroll es tan configurable como nosotros necesitamos.

Nota: Esta funcionalidad, como decíamos, solo funciona con el modo 'history' activo.

## Todo junto

Ahora vamos a juntar todos estos conocimientos y a hacer un pequeño ejemplo. Vamos a implementar del todo la funcionalidad de transferencias de apartados anteriores.

Lo primero que hacemos es crear el servidor de NodeJS que servirá nuestra aplicación:

```

const express = require('express');
const history = require('connect-history-api-fallback');
const app = express();

app.use(history());
app.use(express.static('public'));

const server = app.listen(3001, "localhost", () => {
  const host = server.address().address;
  const port = server.address().port;
  console.log('Running at http://' + host + ':' + port);
});

```

Hemos incluido el fallback del histórico ya que nuestro ejemplo estará configurado con modo histórico de HTML5, el modo que ponía las URLs 'bonitas', recuerda.

Lo siguiente es desarrollar los componentes y la configuración de rutas:

```

const TransferView = {
  template: `
    <div>
      <h2>Transferencia nacional</h2>
      <router-view></router-view>
    </div>
  `,
};

const ConfigView = {
  data() {
    return {
      amount: 0,
      iban: 'ES671234567898761232'
    }
  },
  template: `
    <div>
      <h3>Configurar transferencia</h3>
      <form @submit.prevent="showDetail">
        <label for="amount">Cantidad</label>
        <input id="amount" type="text" v-model="amount"/>
        <label for="iban">IBAN</label>
        <input id="iban" type="text" v-model="iban"/>
        <button>Realizar transferencia</button>
      </form>
    </div>
  `,
  methods: {
    showDetail() {
      this.$router.push({
        name: 'detail',

```

```
        params: { amount: this.amount, iban: this.iban }
      });
    }
  }
};

const DetailView = {
  props: ['amount', 'iban'],
  template: `
    <div>
      <h3>Detalle de la transferencia que va a realizar</h3>
      <p>Cantidad: {{ amount }} €</p>
      <p>IBAN: {{ iban }}</p>
      <router-link :to="{ name: 'confirm' }">Confirmar</router-link>
    </div>
  `
};

const ConfirmView = {
  template: `
    <div>
      <h3>Transferencia realizada correctamente</h3>
      <router-link to="/">Volver</router-link>
    </div>
  `
};

const NotFoundView = {
  template: `
    <div>
      <router-link :to="{ name: 'config' }">Empezar transferencia</router-link>
    </div>
  `
};

const routes = [
  {
    path: '/transfer',
    component: TransferView,
    children: [
      { path: 'config', name: 'config', component: ConfigView },
      { path: 'detail', name: 'detail', component: DetailView, props: true },
      { path: 'confirm', name: 'confirm', component: ConfirmView },
    ]
  },
  { path: '*', name: 'not-found', component: NotFoundView }
];

const router = new VueRouter({
  mode: 'history',
  routes,
  scrollBehavior(to, from, savedPosition) {
```

```
    return { x: 0, y: 0 }  
  }  
});  
  
const app = new Vue({ router }).$mount('#app');
```

Si vemos el ejemplo con detalle, veremos que se incluye la funcionalidad de history mode HTML5, el anidamiento de rutas, el paso de parámetros como propiedad y el control del scroll.

## Conclusión

A lo largo de estos 3 últimos posts hemos hecho un repaso a todo lo que puede aportarnos una librería como `vue-router`. No es una librería innovadora, ni lo necesita ser, simplemente es una opción que se integra perfectamente con el ecosistema de vue para la gestión de rutas.

Terminado este capítulo, entraremos en una nueva fase de la serie donde estudiaremos la gestión del estado en una aplicación. Estamos muy cerca de contar con todas las piezas necesarias para poder hacer una aplicación completa, robusta y escalable con el ecosistema de vue.

Por ahora la experiencia está mereciendo la pena y lo aprendido es coherente con lo que muchos fronts han demandado a lo largo de los años a un framework JavaScript.

Nos leemos :)

## Capítulo 10. Introducción

Cuando decidimos apostar por una arquitectura de componentes, uno de los primeros problemas con los que nos enfrentamos cuando nuestra aplicación empieza a crecer es la dificultad que solemos tener para comunicar componentes que se encuentran a distintos niveles de nuestro árbol.

Una vez que hemos tratado el problema de diseñar buenos componentes, que hemos hablado de cómo dividir nuestra aplicación en diferentes vistas de las que podemos navegar sin problema, llega el turno de cómo gestionar el estado interno de una aplicación de página única.

A lo largo de los próximos posts, hablaremos de cómo atajar este problema dentro del ecosistema de vue. Seguimos dando un paso más en nuestro camino progresivo hacia la construcción de aplicaciones del mundo real con VueJS. Continuemos ;):

### Antes de empezar...

#### ¿Qué es el estado de una aplicación?

Cuando hablamos de estados en una aplicación, hablamos del conjunto completo de variables y constantes que configuran nuestra aplicación. Los estados son todas las formas posibles en las que se puede encontrar mi sistema en un momento determinado.

La forma en que gestionemos y estructuremos el estado de nuestra aplicación puede ser la clave para evitar bugs innecesarios. Saber cómo, cuándo, dónde y por qué muta un estado en particular y de la manera más rápida posible, nos ayudará en nuestro día a día.

#### ¿Qué son las acciones de una aplicación?

Son todos aquellos métodos, funciones, o procedimientos que se encargan de mutar los estados de nuestra aplicación. Se encargan de que dado  $x$  este pueda crear un nuevo estado  $y$ . La definición matemática de una función es este:

$$f(x) = y$$

Dado un estado  $x$  que es pasado a una función  $f$  es obtenido el nuevo estado  $y$  los paradigmas de programación se articulan como propuestas para gestionar y mutar los estados por medios de diferentes tipos de cómputos.

## ¿Cómo puedo comunicarme entre componentes?

Al trabajar con arquitecturas de componentes jerarquizadas en forma de árbol, uno de nuestros trabajos consiste en conseguir comunicar estados de unos componentes a otros. Cuando empezamos a desarrollar nuestro árbol, nuestras arquitecturas son sencillas y el comunicar componentes padres con componentes hijos es relativamente fácil.

El problema viene cuando nuestro árbol empieza a crecer y tenemos que comunicar componentes hermanos - componentes que comparten el mismo componente que los instanció - o componentes que no tienen ningún parentesco dentro de nuestro árbol.

Cuando ocurre esto, tenemos que empezar a pensar en alguna estrategia que nos sea útil y fácilmente de mantener. Una de estas estrategias nos la proporciona la propia librería de vue. En vue podemos comunicar diferentes componentes creando un bus de datos interno.

Este bus de datos se consigue creando una nueva instancia de la clase Vue. Yo por ejemplo, podría hacer esto para comunicar dos componentes sin parentesco:

```
const bus = new Vue();

const componentA = {
  methods: {
    doAction() {
      bus.$emit('increment', 1);
    }
  }
};

const componentB = {
  data() {
    return {
      count: 0
    }
  },
  created() {
    bus.$on('increment', (num) => {
      this.count += num;
    });
  }
};
```

Lo que hago es crear una instancia de Vue que cuenta con un método `$emit` para emitir eventos y un método `$on` para registrarme a eventos. Con esto, lo que hago es, en el componente B, registrarme al evento `increment` cuando el componente ya ha sido creado y esperar a que el componente A emita nuevos cambios al ejecutar su método `doAction`. Es muy parecido a la comunicación que tiene un hijo con un padre, pero esta vez sin parentesco.

Este sistema nos puede funcionar puntualmente para aplicaciones pequeñas o en casos aislados. Cuando el sistema empieza a crecer, empieza a hacerse inmantenible. Nos dificulta las labores de reutilizar acciones y de compartir estados comunes.

Por tanto, tenemos que buscar otras alternativas.

Podría ser buena idea, para aplicaciones medias, hacer uso de un lugar centralizado donde compartir estos estados y métodos. Una pequeña librería que usen los componentes.

Podríamos pensar en algo como esto:

```
var store = {
  debug: true,
  state: {
    message: 'Hello!'
  },
  setMessageAction (newValue) {
    this.debug && console.log('setMessageAction triggered with', newValue)
    this.state.message = newValue
  },
  clearMessageAction () {
    this.debug && console.log('clearMessageAction triggered')
    this.state.message = ''
  }
};

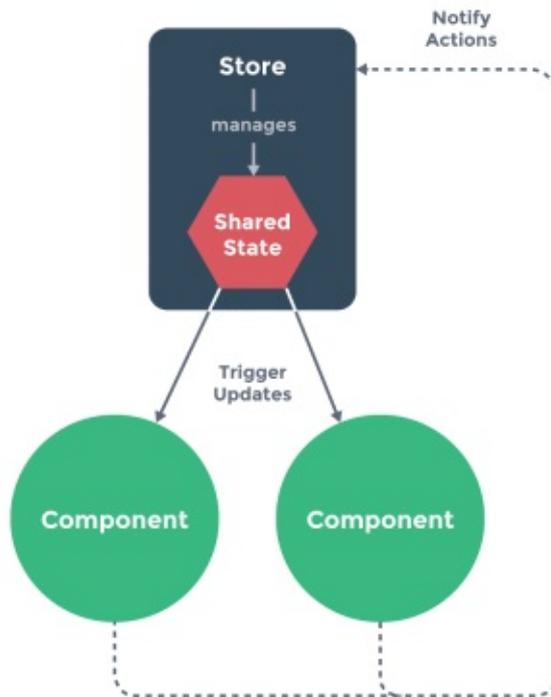
var vmA = new Vue({
  data: {
    privateState: {},
    sharedState: store.state
  }
});

var vmB = new Vue({
  data: {
    privateState: {},
    sharedState: store.state
  }
});
```

Lo que hacemos es crear un objeto que contiene el estado a compartir y unos métodos que se encargan de mutar este estado. De esta manera centralizamos los estados y las acciones.

Ahora, podemos crear instancias de componentes que compartan parte del estado. Si uno de los componentes quieren mutar un estado compartido, hacen uso de uno de los métodos del store. Cómo el objeto se encuentra referenciado en todos los componentes que deseamos, el cambio se realiza en todos.

Con esto, conseguiríamos una arquitectura muy parecida a la del siguiente dibujo:



La solución no me convence del todo, porque no dejamos de tener un estado global con vía libre para realizar cambios a cualquier componente. No existe ningún control y puede ser difícil para trazar qué componente es el responsable en el cambio de un estado. No hay encapsulamiento, y si no somos cuidadosos, podemos liarla parda. Entre tener esto y nada, lo mismo.

Como decimos, este sistema nos puede funcionar, pero cuando contamos con aplicaciones más grandes aún, donde modularizar también este store será necesario, necesitaremos una librería más elaborada, más robusta y que no de tantas posibilidades para manipular externamente el estado. Es aquí donde entra en juego alternativas como vuex.

## ¿Qué es vuex?

Cuando el ejemplo anterior se nos queda demasiado corto, es hora de incluir a nuestra aplicación de vue una alternativa llamada vuex. Vuex es la implementación que ha hecho la comunidad del patrón de diseño creado por Facebook llamado flux.

Cuando llega ese momento en el que tienes que compartir demasiados estados comunes entre componentes, que tienes que hacer virguerías para comunicarte en tu propia comunidad, que tienen métodos o acciones muy parecidas en muchos componentes que te gustaría refactorizar o que empiezas a tener problemas para seguir la trazabilidad por la que pasa un estado en particular es momento de plantearse usar este tipo de arquitecturas.

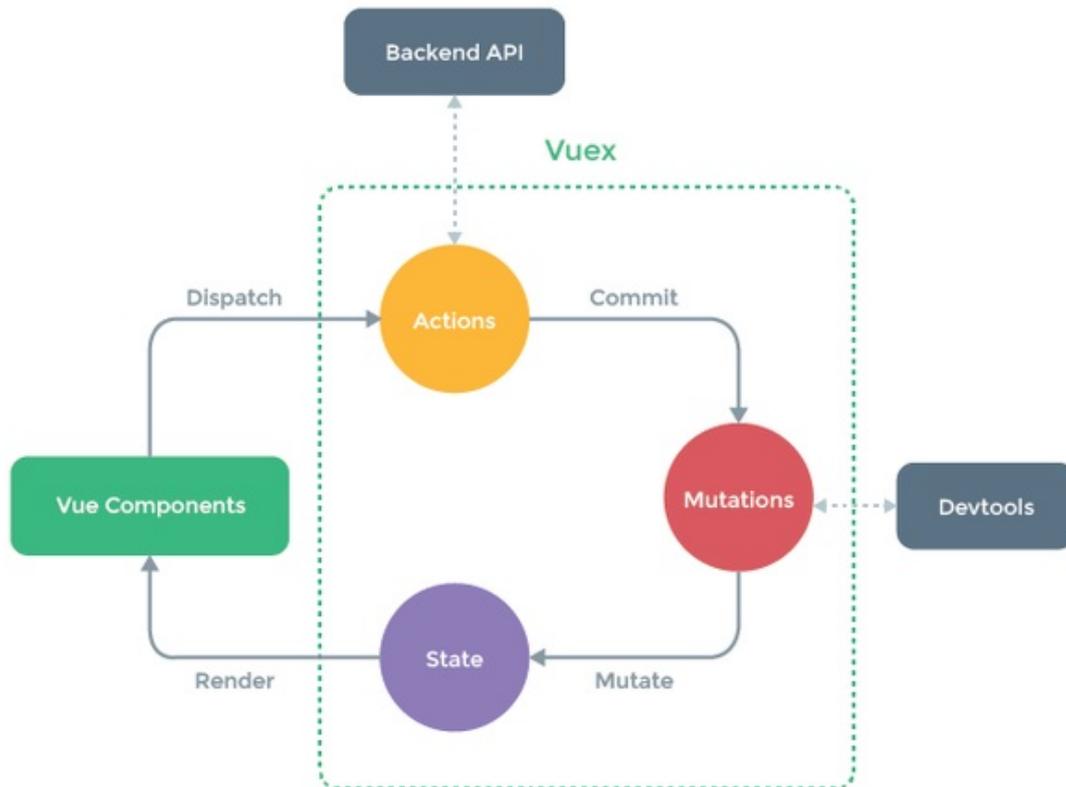
Flux es una arquitectura que gestiona el estado en un objeto singleton global donde su labor es crear mecanismos para evitar que otros componentes puedan cambiar el estado de una aplicación sin su control.

Dentro de la comunidad se han desarrollado muchas implementaciones de flux, pero una de las más conocidas es redux. Redux es una librería, de estilo funcional, muy utilizada por ser agnóstica al framework.

Esto quiere decir que la librería puede ser utilizada tanto con Angular como con React sin sufrir fricciones con los diferentes planteamientos ya que se encarga de la gestión de forma que no se acopla con ninguna plataforma. Siempre necesitaremos conectores específicos para usarlo con cada una de ellas. [En vue también se cuenta con un conector para redux, por si los desarrolladores desean hacer uso de él.](#)

Sin embargo, como decíamos, en vue se ha optado por desarrollar una implementación específica del patrón llamada vuex y que se acopla mucho mejor a la filosofía de vue como iremos viendo a lo largo de estos posts.

La arquitectura de vuex está muy bien esquematizada en esta imagen:



Aunque entraremos en detalle más adelante (no hoy, tranquilos :smile:) expliquemos cada elemento:

- El cuadro verde representa nuestra arquitectura de componentes, los cuales se presentan ahora como la estructura de un edificio esperando a estar habitado por los estados de la aplicación.
- El círculo morado son estos estados que los componentes usan. No se encuentran internamente dentro de los componentes, sino que ahora están gestionados por Vuex y vinculados a los componentes por medio de observadores. Vuex se adecua muy bien al sistema reactivo de la plataforma y lo que hace es, que cuando un estado dentro de su sistema de almacenamiento muta, y se encuentra vinculado con un componente, se provoca la reacción de renderizar de nuevo el componente con el nuevo estado mutado.
- Por otra parte, los componentes son capaces de lanzar acciones. Las acciones son el círculo amarillo y son un buen sitio donde gestionar parte de la lógica más próxima a los datos de nuestra aplicación. Permiten gestionar asincronía, por lo que son el lugar

idóneo para realizar llamadas a servidores externos (caja gris). Cuando una acción ha terminado de realizar sus labores asíncronas (o síncronas), permite realizar confirmaciones (commits) contra el estado.

- Estas confirmaciones lo que provocan es la ejecución de métodos especializados en la mutación de cambios. Esto se puede ver en el círculo rojo. Cuando se ejecutan estos métodos de mutación se desencadenan cambios en el estado que provocan renderizados en el HTML. De esta manera, cerramos el círculo.

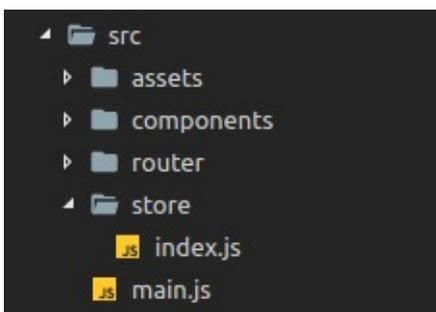
Si nos fijamos, nos encontramos en un flujo unidireccional, lo que nos ayuda a entender en todo momento qué es lo que está ocurriendo en nuestro sistema. Por ahora dejemos esto aquí, en el marco teórico, porque lo explicaremos con ejemplos más adelante. Ahora veamos como integrar la librería de vuex en nuestro SPA.

## ¿Cómo empezamos con vuex?

Bastante simple. Como todo en el mundo NodeJS, lo primero que hacemos es instalar e incluir la dependencia en nuestra aplicación de la siguiente manera:

```
$ npm install vuex --save
```

Como `vue-cli` no nos da soporte para vuex en su generador, lo siguiente será incluir la carpeta donde almacenaremos todo lo necesario para vuex. Lo hacemos de esta manera:



Dentro de este `index.js` crearemos todo nuestro almacenamiento de estados. Lo primero que escribimos es lo siguiente:

```
import Vue from 'vue';
import Vuex from 'vuex';

Vue.use(Vuex);

export default new Vuex.Store({});
```

Donde, como siempre en vue, estamos añadiendo vuex como un plugin, como ya hacíamos con vue-router.

Lo último que hacemos para acabar con la conexión total entre vue y vuex, es incluir la instancia creada del store en la instancia de nuestra aplicación, de la siguiente manera:

```
import Vue from 'vue';
import App from './components/app/app.vue';
import store from './store';

const app = new Vue({
  el: '#app',
  store,
  template: '<App/>',
  components: { App }
});
```

Ya está, ya podemos empezar con el resto de conceptos.

## Recuerda

Como ya explicamos en la introducción, vue cuenta con una buena herramienta de depuración en Chrome y que dentro de ella cuentas con una pestaña de vuex donde podrás inspeccionar en todo momento la instantánea de tu proyecto. Úsala todo lo que puedas para verificar que estás haciendo las cosas como quieres.

## Conclusión

Cuidado con vuex. Muchas veces nos dejamos llevar por la tecnología e introducimos dependencias en nuestras aplicaciones sin saber muy bien si la necesitamos. Vuex puede ser un buen caso de esto. Nuestras aplicaciones evolucionan mucho a lo largo del tiempo e incluir una librería como esta desde el principio, puede entorpecernos más que ayudarnos.

Si una cosa tiene buena vue es que permite ir incluyendo sus diferentes piezas según necesidad, así que si no tenemos muy claro si vamos a necesitar un lugar centralizado para manejar el estado, quizá sea mejor que no lo usemos, más adelante podremos hacerlo.

Hay que tener en cuenta que vuex nos ayuda a mantener y testear mejor nuestra aplicación, pero también tenemos que ser conscientes que vuex nos implica nuevos niveles de abstracción que harán que necesitemos trabajar con nuevos conceptos que harán la

curva de aprendizaje menos accesible para desarrolladores juniors a nuestros proyectos. Por ello, debemos tener cuidado.

De todas las implementaciones de flux, vuex me parece la más intuitiva y coherente con las necesidades del framework, pero eso no significa que no nos vaya a suponer añadir más fontanería de la que quizá nos gustaría.

Nos leemos :)

# Capítulo 11. Los estados y getters

Una vez que hemos visto cómo incluir vuex en nuestra aplicación, es momento de explicar los conceptos básicos de la librería.

En el post de hoy, dedicaremos tiempo a contar cómo podemos almacenar el estado dentro del store. Veremos cómo con muy poco de código podremos crear datos accesibles para varios componentes.

También repasaremos un concepto bastante interesante denominado getters que nos permitirá crear consultas más específicas sobre los datos que nos interesan para cada componente.

## Los estados

Los estados nos van a permitir crear diferentes instantáneas del estado completo de nuestra aplicación. La forma en la que definimos estados dentro de vuex es muy sencilla, solo tenemos que incluir atributos en el objeto state de vuex.Store de la siguiente forma:

```
// store/index.js
...

export default new Vuex.Store({
  state: {
    count: 0
  }
});
```

Con esta simple declaración, ya podemos hacer uso de ello dentro de nuestros componentes:

```
import store from '../store/index';

const componentA = {
  data() {
    return {
      count: null
    }
  },
  create() {
    this.count = store.state.count;
  }
};
```

Lo que hemos hecho es iniciar el estado interno count del componente con el estado global del store. Este código lo podemos mejorar ya que se encuentra algo acoplado. Si recordáis, en el post anterior, inyectamos nuestro store para propagarse por todo el árbol de componentes de esta manera:

```
import Vue from 'vue';
import App from './components/app/app.vue';
import store from './store';

const app = new Vue({
  el: '#app',
  store,
  template: '',
  components: { App }
});
```

Por lo tanto, podríamos hacer esto perfectamente:

```
const componentA = {
  data() {
    return {
      count: null
    }
  },
  create() {
    this.count = this.$store.state.count;
  }
};
```

Con esto quitamos la dependencia 'hardcodeada' y usamos la instancia de store que se encuentra inyectada ( `this.$store` ). De esta manera, quitamos dependencias tediosas en todos nuestros componentes. Ya se cuenta con él.

El ejemplo sigue teniendo algo que huele mal ya que lo que hemos hecho es iniciar el estado interno, pero no hemos creado un componente que reaccione si cambia el estado del store. Para conseguir esto, nos basamos en la funcionalidad de estados computados con la que cuentan todos los componentes.

El ejemplo se convertiría en algo cómo esto:

```
const componentA = {
  computed: {
    count() {
      return this.$store.state.count;
    }
  }
};
```

De esta forma, conseguimos que si el valor de `store.state.count` cambia, se ejecute esta función y el componente reaccione a cambios. Vale, parece que lo tenemos listo. Sigamos.

¿Qué ocurre si quiero incluir muchos estados en un componente? Insertar propiedades computadas de esta manera es algo tedioso. Estamos creando una función para llamar a un estado que se llama igual. Es código innecesario.

Hay una funcionalidad muy chula en vuex que nos permite mapear estados a cálculos de una forma más agradable. Dentro de vuex se cuenta con el método `mapState` que permite esto. Podemos hacer esto, por ejemplo:

```
// ./components/componentA/componentA.js
import { mapState } from 'vuex';

export default {
  computed: mapState()
};
```

Este cambio no nos aporta más que azúcar sintáctico, pues el código de antes y el de ahora se comportan exactamente igual. Lo único que hacemos con esta forma es que el componente se ponga a escuchar todos los estados que existan dentro de nuestro store.

Esto está muy bien, pero es poco real que un componente se ponga a computar todo. Por tanto, ¿cómo puedo con `mapState` seleccionar qué parte del estado me interesa? Tenemos varias formas que, según el contexto, nos podrán interesar más o menos.

Por ejemplo, si tenemos este store:

```
// store/index.js
...

export default new Vuex.Store({
  state: {
    count: 0,
    gists: [],
    user: {}
  }
});
```

Puede que en mi componente solo me interese `count` y `gists`. Esto lo puedo hacer de esta manera:

```
// ../components/componentA/componentA.js
import { mapState } from 'vuex';

export default {
  computed: mapState(['count', 'gists'])
};
```

De esta manera, solo me mapearía esas dos propiedades y no `user`. Puede que, por cierto contexto, el nombre del estado del store y del estado interno que se computa, no tengan que llamarse igual. Podemos mapearlo también, indicando un objeto de esta manera:

```
// ../components/componentA/componentA.js
import { mapState } from 'vuex';

export default {
  computed: mapState({
    countAlias: 'count'
  })
};
```

De esta forma, dentro del template, haríamos uso de `{{ countAlias }}` desacoplando ambas.

Puede que necesitemos calcular un dato a partir de un estado del store. En ese caso yo puedo indicar una función:

```
// ./components/componentA/componentA.js
import { mapState } from 'vuex';

export default {
  computed: mapState({
    countPlus2(state) {
      return state.count + 2;
    }
  })
};
```

Otra opción es que, no solo necesitemos mapear estados del store, sino que el propio componente tenga estados computados internos que no tengan nada que ver con el global. Podemos tener esto:

```
// ./components/componentA/componentA.js
import { mapState } from 'vuex';

export default {
  computed: {
    myLocalComputed() {
      // code
    },
    ...mapState(['count', 'gists'])
  }
};
```

De esta forma, y gracias al nuevo Spread Operator de ES6, podemos combinar objetos diferentes de una manera muy sencilla. Ahora contaría con tres propiedades computadas: las dos del store y la interna ( `myLocalComputed` ).

Estas utilidades con prefijo `map-` serán creadas para todos los elementos de vuex por lo que aprendérselo bien, será necesario ya que lo usaremos en lo que queda de serie.

## Los getters

Una cosa es cómo almaceno los datos en mi store global y otra diferente qué datos me son útiles dentro de un componente. Por ejemplo, puede que dentro de mi store se almacene un número considerable de gists, pero que un componente solo necesite aquellos que son públicos.

Bueno, no parece muy grave, a fin de cuentas, si tengo acceso al store en un componente, podría hacer algo como esto:

```
// ./components/componentA/componentA.js

export default {
  computed: {
    publicGists() {
      return this.$store.state.gists.filter(gist => gist.public);
    }
  }
};
```

Nada mal ¿no? Ahora, el 'pequeño' problema viene cuando otro componente necesita también conocer solo los gists públicos. Para conseguir esto, podría coger la función anterior, llevármela a una librería y reutilizar la utilidad en ambos componentes. Sin embargo, estamos creando código que se encuentra muy acoplado a vuex y que nos va a hacer incluir dependencias en los componentes.

vuex ha pensado en ello y nos ha dado una funcionalidad dentro del store llamada getters. Los getters son funciones que permiten obtener datos parciales de nuestro estado y que son reutilizables por todos los componentes de una forma más cómoda que creando librerías e incluyendo dependencias.

Refactoricemos el componente anterior, llevémoslo a nuestro store de esta manera:

```
// store/index.js
...

export default new Vuex.Store({
  state: {
    count: 0,
    gists: [],
    user: {}
  },
  getters: {
    publicGists(state) {
      return state.gists.filter(gist => gist.public);
    }
  }
});
```

Lo que hago es definir consultas dentro del objeto `getters` de mi store. Ahora en mi componente puedo hacer esto:

```
// ./components/componentA/componentA.js

export default {
  computed: {
    publicGists() {
      return this.$store.getters.publicGists;
    }
  }
};
```

Estamos reutilizando consultas con solo llamar a las propiedades que tengamos almacenadas en getters. Con vuex, puedo componer getters de esta manera:

```
// store/index.js
...

export default new Vuex.Store({
  state: {
    count: 0,
    gists: [],
    user: {}
  },
  getters: {
    publicGists(state) {
      return state.gists.filter(gist => gist.public);
    },
    totalPublicGists: (state, getters) => {
      return getters.publicGists.length;
    }
  }
});
```

Los getters son inyectados internamente. De esta manera, puedes usar los getters dentro de otros getters.

Incluso hay que pensar que esto no se limita a búsquedas estáticas. Puedo indicar desde el componente qué elemento me puede interesar de una colección;

```
// store/index.js
...

export default new Vuex.Store({
  state: {
    count: 0,
    gists: [],
    user: {}
  },
  getters: {
    publicGists(state) {
      return state.gists.filter(gist => gist.public);
    },
    totalPublicGists: (state, getters) => {
      return getters.publicGists.length;
    },
    publicGistById: (state, getters) => (id) => {
      return getters.publicGists.filter(gist => gist.id === id);
    }
  }
});
```

Para usar este getter en un componente, lo haría así:

```
// ../components/componentA/componentA.js

export default {
  computed: {
    publicGists() {
      return this.$store.getters.publicGistById(2);
    }
  }
};
```

También contamos con una utilidad de maps que se comportan igual que el de `mapState`.

En este caso solo tenemos que usar `mapGetters`:

```
// ../components/componentA/componentA.js
import { mapGetters } from 'vuex';

export default {
  computed: mapGetters()
};
```

De esta forma, incluyo todos los getters que existan configurados en el store.

Me gustan los getters porque nos desacoplan mucho (es bueno para testear todo este código de consultas de forma aislada) y nos permiten reutilizar código entre componentes.

## Conclusión

Usemos la conclusión para hacer una pequeña reflexión:

Contar con vuex no significa tener que almacenar exactamente todos los estados de mi aplicación en el store. Se trata de tener claros qué datos pueden ser utilizados por más partes del sistema, qué estados son claves para el módulo o el negocio de la aplicación y encapsularlos en este mecanismo.

Pero esto no quiere decir que, en muchos casos, los componentes no cuenten con su propio estado interno. Habrá estados que solo incumban al correcto funcionamiento interno del componente ¿Qué sentido tendría sacar este estado a un store? ¿Cuándo tiene sentido que se encuentre encapsulado en la pieza de código que más cohesionado se encuentra de él?

Tener esto claro puede suponer, como desarrolladores, un tiempo de análisis y de diseño clave para que nuestro estado global no se convierta en un cajón desastre de variables. Tengámoslo en cuenta.

Nos leemos :)

## Capítulo 12. Las mutaciones y acciones

Con el post anterior, tenemos la mitad del flujo de vuex explicado. Ya somos capaces de obtener estados de nuestro store global y de crear reacciones en los componentes a partir de esto.

Lo siguiente que tenemos que tener en cuenta es cómo poder manipular estos estados para que se conviertan y muten en aquello que necesitamos, ya sea tanto por la interacción del usuario como por los eventos que se encuentran registrados en mi aplicación.

Para conseguir esto, vamos a explicar las dos funcionalidades que nos permiten esto y que, como veremos, tiene muchas similitudes a como lo hacen otras librerías como redux.

### Las mutaciones

En vuex no puedo llegar a una variable del estado y manipularla para que cambie directamente. Si hiciese esto, los componentes no reaccionarían al cambio. Debido a que la librería quiere seguir un sistema de flujo unidireccional, donde todas las fases se encuentren en un ciclo cerrado, deberemos usar un nuevo concepto conocido como mutaciones. Las mutaciones son aquellas funciones que se encargan de cambiar el valor de nuestro estado.

Las mutaciones se comportan de igual manera que un evento. Una mutación cuenta con un tipo y un manejador que debe registrarse dentro de nuestro store. Cuando yo quiero hacer uso de esa mutación, solo tengo que invocarla.

Igual que un evento.

Para conseguir estas mutaciones, tenemos que registrarlas de la siguiente manera:

```
// store/index.js
...

export default new Vuex.Store({
  state: {
    count: 0,
    gists: [],
    user: {}
  },
  mutations: {
    increment: function (state) {
      state.count++
    }
  }
});
```

Dentro de mutations insertamos todos aquellos manejadores que queremos que manipulen nuestros datos. `increment` es el tipo de mutación que queremos y la función el manejador que se disparará. Si nos damos cuenta, el estado se inyecta como parámetro para ser manipulado. Usa esta instancia para que todo funcione correctamente.

Una vez que tenemos esta mutación definida, ya podemos hacer uso de ella en nuestros componentes. Para usarlo, simplemente tendremos que ejecutar

```
store.commit('increment') .
```

En nuestro componente podríamos tener esto:

```
// ../components/componentA/componentA.js

export default {
  methods: {
    increment: {
      store.commit('increment');
    }
  }
};
```

Para evitar redundancias en el código, podemos hacer uso de `mapMutations` :

```
// ../components/componentA/componentA.js
import { mapMutations } from 'vuex';

export default {
  methods: mapMutations();
};
```

Al igual que pasaba con los getters, las mutaciones permiten payloads para manipular el estado. Por ejemplo:

```
// store/index.js
...

export default new Vuex.Store({
  state: {
    count: 0,
    gists: [],
    user: {}
  },
  mutations: {
    increment: function (state, payload) {
      state.count += payload.amount;
    }
  }
});
```

Y para usarlo en un componente:

```
// ../components/componentA/componentA.js

export default {
  methods: {
    increment: {
      store.commit('increment', { amount: 200 });
    }
  }
};
```

Si no nos convence la firma del método `commit` por ser poco explicativa, podemos pasar un objeto indicando cada elemento de la siguiente forma:

```
// ../components/componentA/componentA.js

export default {
  methods: {
    increment: {
      store.commit({
        type: 'increment',
        amount: 200
      });
    }
  }
};
```

Esto ya será al gusto del desarrollador o el equipo.

Para terminar con las mutaciones, vamos a aclarar un par de cosas para que el día de mañana no cometamos fallos innecesarios:

## Cuidado con la reactividad de los objetos

El estado de un store se comporta igual que el data de un componente a nivel de 'reactividad'. Esto quiere decir, que inicies todas las variables de tu estado y que cuando vayas a mutar objetos del estado, ten en cuenta que las propiedades no provocarán reacciones al no contar con observadores ni getters internos en ellas.

Por tanto, si necesitas mutar un objeto, usa la funcionalidad `Vue.set` que te permitirá indicar que una propiedad ha cambiado y que el sistema debe reaccionar. En vez de hacer esto:

```
export default new Vuex.Store({
  state: {
    count: 0,
    gists: [],
    user: {}
  },
  mutations: {
    changeName: function (state, payload) {
      state.user.name += payload.name;
    }
  }
});
```

Haz esto:

```
import Vue from 'vue';

export default new Vuex.Store({
  state: {
    count: 0,
    gists: [],
    user: {}
  },
  mutations: {
    changeName: function (state, payload) {
      Vue.set(state.user, 'name', payload.name);
    }
  }
});
```

Si necesitas hacer un cambio de más propiedades, usa el `Spread Operator` de ES6:

```
export default new Vuex.Store({
  state: {
    count: 0,
    gists: [],
    user: {}
  },
  mutations: {
    changeFullName: function (state, payload) {
      state.user = { ...state.user, ...payload.user };
    }
  }
});
```

Esto generará una nueva instancia del objeto con lo que la reacción sí está asegurada.

## Poner los tipos de las mutaciones como constantes

Otro buen uso dentro de la creación de mutaciones es el llevar las cadenas a constantes. De esta forma cuando crece un proyecto, tengo localizadas todas las mutaciones que se pueden realizar y evito hardcodeados de cadenas, lo que hará que tenga intellisense en mi editor de código favorito. Por tanto, es una buena práctica hacer esto:

```
export const CHANGE_FULLNAME = 'CHANGE_FULLNAME';

export default new Vuex.Store({
  state: {
    count: 0,
    gists: [],
    user: {}
  },
  mutations: {
    [CHANGE_FULLNAME](state, payload) {
      state.user = { ...state.user, ...payload.user };
    }
  }
});
```

De esta forma todo sigue igual. Si os dais cuenta, exporto la constante. Esto es porque ahora en los componentes también puedo hacer uso de estas constantes para realizar los commits. Si yo uso cadenas en mis componentes para referirme a mutaciones, y cambio el nombre a una de ellas, ningún editor podrá avisarme de que hay algo mal.

Sin embargo, si lo hago de esta manera, rápidamente me avisará de que algo no va bien en tiempo de escritura del código. Como digo, esto es solo una buena práctica y no es obligatorio hacer uso de ello.

## No uses procesos asíncronos en tus mutaciones

Otro tema interesante es tener en cuenta es que las mutaciones deben contar con manejadores que sólo gestionen lógica síncrona. Internamente, vuex incluye un proxy a cada mutación para 'logar' el estado antes y después de la mutación.

Estos hooks son usados por las herramientas de depuración para que podamos tener instantáneas en todo momento. Si incluimos operaciones asíncronas, perdemos esta visión porque las herramientas nos mentirán al no poder esperar que la mutación acabe.

Y entonces ¿Cómo gestionamos la asincronía en mis aplicaciones? ¿No voy a poder hacer llamadas a servidor o bases de datos en una aplicación de vue. Pues sí, para eso nacieron las acciones :)

## Las acciones

Las acciones funcionan igual que las mutaciones. Eso sí, no mutan estado - eso lo delegan a las mutaciones - y se permiten todas las operaciones asíncronas que necesitemos. Por ejemplo, imaginemos que tenemos la mutación `ADD_GISTS` de esta forma:

```
export const ADD_GISTS = 'ADD_GISTS';

export default new Vuex.Store({
  state: {
    gists: []
  },
  mutations: {
    [ADD_GISTS](state, gists) {
      state.gists = gists;
    }
  }
});
```

Podríamos indicar una acción que obtenga los gists desde una API externa de la siguiente manera:

```
export const ADD_GISTS = 'ADD_GISTS';

export default new Vuex.Store({
  state: {
    gists: []
  },
  mutations: {
    [ADD_GISTS](state, gists) {
      state.gists = gists;
    }
  },
  actions: {
    fetchGists(context) {
      return axios.get('https://api.github.com/gists')
        .then(response => response.data)
        .then(gists => context.commit(ADD_GISTS, gists));
    }
  }
});
```

Las acciones esperan una promesa para ser resueltas, de ahí que hagamos un return de la promesa que devuelve axios. Cuando axios nos devuelve los gists, podemos ejecutar commits con el tipo de mutación que queramos llevar a cabo. (El uso de constantes tiene mucho sentido por todas las veces que vamos a usarlas durante nuestro proyecto).

Me gusta que separen ambos conceptos en la librería. Las acciones son donde me puedo dedicar a meter mi lógica, mis validaciones, mi comunicación con el exterior, y las mutaciones sólo se preocupan de controlar los estados, de manipularlos. Me parece un proceso bastante natural y bien separado en diferentes conceptos y responsabilidades:

Las acciones se encargan de preparar todo lo necesario para que una mutación confirme un cambio en el estado como si de una transacción se tratase. Es como tener un sistema tricapa en un espacio muy reducido.

Para hacer uso de acciones en un componente, puedo hacerlo por medio del método dispatch. Dentro del componente haré esto:

```
export default {
  name: 'dashboard-view',
  created() {
    this.fetchGists();
  },
  methods: {
    fetchGists() {
      store.dispatch('fetchGists');
    }
  }
};
```

Lo mismo que con el resto, contamos con un `mapActions` que funciona de la misma manera:

```
import { mapActions } from 'vuex';

export default {
  name: 'dashboard-view',
  created() {
    this.fetchGists();
  },
  methods: mapActions()
};
```

Como pasaba con las mutaciones, las acciones también permiten un payload:

```
export default {
  name: 'dashboard-view',
  created() {
    this.fetchGists();
  },
  methods: {
    fetchGists() {
      store.dispatch('fetchGists', 1);
    }
  }
};
```

Detengámonos un poco en el primer parámetro que le pasamos a una acción, el parámetro `context`. Al final `context` es una instancia del propio store con lo que podremos hacer todo aquello que hacemos en un componente. Por ejemplo, yo podría acceder al estado dentro de una acción:

```

export const ADD_GISTS = 'ADD_GISTS';

export default new Vuex.Store({
  state: {
    gists: []
  },
  mutations: {
    [ADD_GISTS](state, gists) {
      state.gists = gists;
    }
  },
  actions: {
    fetchGists({ commit, state }, gistId) {
      if (state.gists.length === 0) {
        return axios.get('https://api.github.com/gists')
          .then(response => response.data)
          .then(gists => context.commit(ADD_GISTS, gists));
      }
    }
  }
});

```

Si dentro del estado ya hay gists, no realizo la llamada. Deshago el objeto de esa forma gracias a ES6; [es la nueva funcionalidad llamada asignación por Destructuring](#).

Si esto es así, si tenemos una instancia del propio store en context, yo podría componer una acción determinada, a partir de otras acciones más específicas. [Podría realizar varias llamadas asíncronas evitando el temido Callback Hell](#). Si lo unimos a los Async function de ES6, podemos tener algo parecido a esto:

```

actions: {
  async actionA ({ commit }) {
    commit('gotData', await getData())
  },
  async actionB ({ dispatch, commit }) {
    await dispatch('actionA')
    commit('gotOtherData', await getOtherData())
  }
}

```

Las posibilidades nos las impondrá negocio, pero con esto estamos más que cubiertos. Tenemos todos los métodos en un sitio centralizado, cohesionado a sus datos y con muchas posibilidades de aislarlos para ser probados.

## Conclusión

Quizá, al igual que me pasó a mí, os sintáis ahora mismo así:



desmotivaciones.es

## ¿Pero cómo he llegado aquí?

Es curioso que vux tenga unos conceptos muy concretos y predefinidos que estudiados por separado son entendibles y coherentes, pero que cuando juntamos todo lo aprendido es un... vale, estoy perdido...

Creo que es normal. Hay mucho concepto de ES6 que se usa para sacarle todo el jugo al lenguaje y que hay que tener muy claros. Además, que usar un patrón más abstracto de lo que hemos visto en otros frameworks, nos va a suponer un esfuerzo. Incluso, aunque todo se conecta, es lógico que veamos mucha magia en sus piezas ya que delegamos mucho trabajo en la herramienta.

Creo que la única forma de enfrentarse a una librería como vux es practicando mucho y haciendo casos de uso que se vayan complicando. Cometeremos errores, pero poco a poco, iremos viendo que todo tiene sentido y que nuestro código tiene cierta 'armonía'.

Muchos pensaréis: ¿Y tanto lío para qué? Yo con mi jQuery era feliz ¿Por qué todo esta complicación? Y tendréis razón. Si no sufres con jQuery, si no tardas años en encontrar un bug, si todo te escala, si tu equipo trabaja bien ¿Para qué cambiar?

El problema viene cuando esto no es así, cuando escalar tus proyectos y equipos está siendo difícil, cuando el ego de cada desarrollador ha hecho que aquello no tenga un estilo predefinido.

Vuex te ayuda a eso, a evitar luchas entre tu equipo por cómo se estructuran o nombras las cosas, a no tener que pensar en eso. ¿Te gusta su estilo? ¿Lo entiendes? ¿Merece la pena el tiempo invertido? Pues adelante, ni lo dudes, pero ten en cuenta que se necesita un proceso de adaptación que tendrá un coste.

Nos leemos :)

## Capítulo 13. Los módulos

Al igual que pasa con nuestra aplicación, cuando un store empieza a crecer demasiado, empieza a ser bastante inmanejable gestionarlo todo en un único fichero. Como Vuex presenta una solución donde se gestiona todo el estado en único objeto, tenemos que pensar una forma para poder modularizar, pero a la vez seguir teniendo esta estructura en árbol único.

Vuex cuenta con una funcionalidad que nos va a permitir dividir nuestro árbol de datos en módulos más específicos que contarán cada uno de ellos con todo lo necesario para gestionar estas porciones.

La forma de crear un módulo es tan fácil como crear un objeto JSON de la siguiente manera:

```
const moduleA = {
  state: {...},
  getters: {...},
  mutations: {...},
  actions: {...}
};
```

Para incluirlo dentro de nuestro store, usamos la propiedad `module` y lo asignamos con el nombre que deseemos que tenga:

```
const store = new Vuex.Store({
  state: {...},
  getter: {...},
  mutations: {...},
  actions: {...},
  modules: {
    a: moduleA
  }
});
```

Con esto, ahora puedo acceder al estado de un módulo, en particular, de esta manera:

```
store.state.a;
```

## Estado local de los módulos

Al contar con una jerarquía de módulos y submódulos, es importante saber cómo acceder a las diferentes partes del estado según en qué zona del store nos encontremos.

Por ejemplo, si me encuentro en un módulo en particular, lo que se inyecta tanto en los getters como en los mutations es el estado local al módulo, de tal manera que yo lo haría así:

```
const moduleA = {
  state: { count: 0 },
  mutations: {
    increment(state) {
      state.count++
    }
  },
  getters: {
    doubleCount (state) {
      return state.count * 2
    }
  }
};
```

Si necesitase acceder al estado del ámbito global, es decir, del estado raíz, podría hacerlo tanto en los getters como en los actions, pues tengo acceso a esta parte del store. En los getters se inyecta un tercer parámetro con este estado y en los actions el objeto context cuenta también con ello. Veamos el ejemplo:

```
const moduleA = {
  getters: {
    sumWithRootCount (state, getters, rootState) {
      return state.count + rootState.count;
    }
  },
  actions: {
    incrementIfOddOnRootSum ({ state, commit, rootState }) {
      if ((state.count + rootState.count) % 2 === 1) {
        commit('increment');
      }
    }
  }
};
```

## Módulos y su namespace

Aunque definas módulos para tener un buen sistema modularizado, puede que no necesites que tus componentes tengan conocimiento de esta modularización, por lo tanto, todos los getters, mutations y actions son incluidos en el ámbito global del store. Si dos acciones o mutaciones son llamadas igual, ambas reaccionan.

El estado sí es dividido a nivel de espacio de nombres para que no haya conflictos a la hora de que sus variables se llamen igual en diferentes módulos.

Si por un casual, deseamos que nuestros getters, mutations y actions se encuentren separados por espacio de nombres, tendremos que poner a true el atributo namespaced en la raíz del módulo. De esta manera, si hacemos esto:

```
const moduleA = {
  namespaced: true,
  state: { count: 0 },
  mutations: {
    increment(state) {
      state.count++
    }
  },
  getters: {
    doubleCount (state) {
      return state.count * 2
    }
  }
};
```

La forma en la que yo accedería ahora ese getter y esa mutation sería de la siguiente manera:

```
this.$store.getters['a/doubleCount'];
this.$store.commit('a/increment');
```

Si vas a usar este sistema de espacio de nombres, ten en cuenta usar esta nomenclatura en formato cadena como si accedieras a un fichero de una carpeta también para los `mapGetters`, `mapActions` y `mapMutations`, ya que se usa el mismo.

## Acceso a elementos globales en los módulos

Puede ocurrir que cuando nuestros módulos se encuentren dentro de un espacio de nombres, queramos acceder a un getter de su módulo superior. Para hacer eso, los getters cuentan con cuarto parámetro donde son inyectados estos rootGetters

```

modules: {
  foo: {
    namespaced: true,
    getters: {
      someGetter (state, getters, rootState, rootGetters) {
        getters.someOtherGetter;
        rootGetters.someOtherGetter;
      },
      someOtherGetter: state => { ... }
    },
  },
}

```

Como vemos, no hay un conflicto de nombres aunque el módulo raíz y el submódulo compartan un método con el mismo nombre porque ambos son accedidos de diferente manera.

Nos ocurre lo mismo con los actions ya que el contexto también cuenta con esta propiedad

`rootGetters` .

```

actions: {
  someAction ({ dispatch, commit, getters, rootGetters }) {
    getters.someGetter;
    rootGetters.someGetter;
  },
  someOtherAction (ctx, payload) { ... }
}

```

Si dentro de esta acción yo quiero llamar a otras acciones o mutaciones, lo que tengo que hacer es pasar un objeto `{ root: true }` como tercer parámetro de la siguiente manera:

```

actions: {
  someAction ({ dispatch, commit, getters, rootGetters }) {
    // -> 'foo/someOtherAction'
    dispatch('someOtherAction');

    // -> 'someOtherAction'
    dispatch('someOtherAction', null, { root: true });

    // -> 'foo/someMutation'
    commit('someMutation');

    // -> 'someMutation'
    commit('someMutation', null, { root: true });
  },
  someOtherAction (ctx, payload) { ... }
}

```

Se pasa como tercer parámetro porque el segundo puede ser el payload.

## Registro de módulos dinámicamente

Puede sernos útil registrar diferentes módulos en otros momentos de la aplicación, sin tener que definirlos todos en el store en un instante determinado. Para lograr esto hacemos uso del método `registerModule` de la siguiente manera:

```
store.registerModule('myModule', { // ... });
```

El primer parámetro indica el nombre que va a tener el módulo y el segundo el módulo en sí. Si necesitamos crear un submódulo dentro de un módulo concreto, podemos registrar estos anidamientos de la siguiente manera:

```
store.registerModule(['nested', 'myModule'], { // ... });
```

Donde el acceso interno sería `this.$store.nested.myModule.state`.

Esta forma de registrar módulos dinámicamente, nos viene muy bien para que plugins de vue puedan hacer uso de la potencia de vuex para que puedan enganchar su módulo de datos a nuestra estructura en árbol del estado.

## Reutilizando módulos

Puede que necesites reutilizar módulos. Sí, tener más de una instancia del mismo módulo en tu store. Si se da este caso, ten cuenta que el estado es un objeto y que si lo reutilizas de esta manera, compartes su referencia, provocando, que si cambias el valor en una de ellas, cambie el valor en todas sus instancias.

Para solucionar esto, hacemos como lo mismo que hacíamos con el objeto data de los componentes: asignar una función que devuelva nuestro objeto. De esta forma cada instancia contará con una referencia diferente del estado (Una pequeña factoría dentro de vuex, fijate tú :). Lo hacemos así:

```
const MyReusableModule = {
  state () {
    return { foo: 'bar' }
  },
  // mutations, actions, getters...
};
```

## Conclusión

Como vemos, tener un árbol de datos complica entender el todo. Tener un cajón desastre de datos no nos va aportar más que quebraderos de cabeza, y saber cuándo crear buenos estancos de datos nos ayudará mucho en el momento que crezca el proyecto. Piensa en tu store como si de un gran buque se tratase, piensa que tienes en tus manos un petrolero.

Estos barcos son tan largos que con el movimiento interno que produce el fuel se podrían llegar a desestabilizar y hundir. Es por eso que existen estancos que dividen estos compartimientos para evitar esto y que el fuel sea transportado. Piensa en tu gran store como si fuera este caso.

Como en todo, el arte del desarrollo está en diseñar y nombrar adecuadamente las cosas. De nada nos sirve contar con esta funcionalidad de modularización sino hemos dedicado un tiempo a comprender el negocio y las formas en que tenemos que moldearlo.

Obsesionarnos con modular en las primeras fases de desarrollo nos va a servir de poco y la refactorización y el tiempo en un proyecto, nos irá diciendo que necesidades pueden surgir. Encontrar ese equilibrio entre no dejarnos llevar por la deuda técnica y empezar a hacer sobre-ingeniería desde el principio es la gran responsabilidad que tendrá todo desarrollador...

...Y con esto acabamos, por ahora, con vuex.

Lo que vamos a hacer en los próximos posts del blog es dejar, por unas semanas, el ecosistema de vue a un lado y centrarnos en algo que creo primordial para poder trabajar cómodamente en la creación de aplicaciones SPA: el aprendizaje de webpack.

Hemos postergado esto y en algunas etapas nos hemos fiado de la magia que desentrañaba esta herramienta y creo que es importante que nos sintamos cómodos con ella, por eso, haremos una pequeña serie que nos ayudará a comprender mejor nuestro stack de trabajo.

Nos leemos :)

## Capítulo 14. Conceptos básicos

Nos detenemos por un momento en el camino de vue y nos ponemos manos a la obra en entender Webpack. Aunque lo parezca, no nos hemos desviado de nuestra trayectoria.

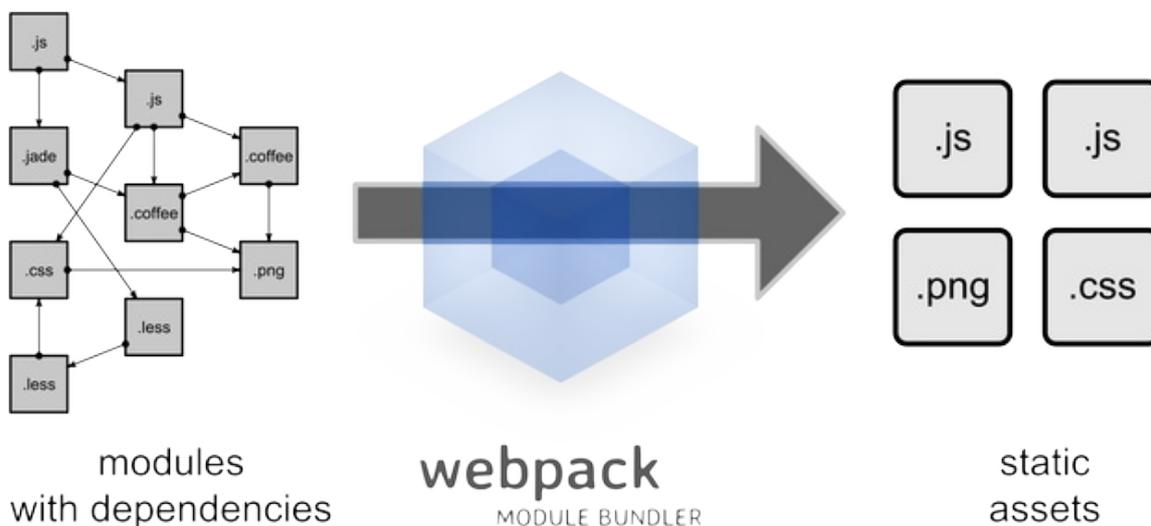
Como pudimos ver en el capítulo donde estudiamos vue-cli, vimos que la plantilla nos generó una serie de ficheros donde ya se encontraba la construcción de nuestra aplicación de una forma mágica, sin que nosotros tuviésemos que desarrollar o crear nada, simplemente ejecutando un comando.

Estos ficheros se encontraban creados con Webpack y hacían todo lo necesario para no tener que mancharnos las manos nosotros. Cómo no vamos a basar el destino de un proyecto en la magia y los fuegos artificiales, es hora de que nos atemos los machos y entendamos qué estaba haciendo vue-cli por nosotros.

Puede ser que algún día necesitemos incluir algo en la configuración que no viene por defecto y que estos conocimientos nos sean muy útiles. Así que vamos al lío:

### ¿Qué es?

Webpack es un empaquetador de módulos creado con NodeJS. Entiéndase por módulos todo aquello que se encuentra separado en diferentes ficheros dentro de nuestra aplicación. Y no nos estamos refiriendo solo a los ficheros JS. Todo fichero, con la extensión que tenga, nos da igual, es un buen candidato para poder ser empaquetado en un producto final.



Los empaquetadores de módulos tienen su hueco en el mundo Web por dar solución a dos problemas que hemos sufrido casi todos los desarrolladores: el manejo de un elevado número de ficheros y la gestión, a veces muy complicada, de las dependencias de nuestro proyecto y sus acoplamientos.

Expliquemos cada uno de ellos:

## **Manejo elevado de fichero**

Cuando un proyecto crece, ya estemos trabajando en una SPA o en una aplicación con varios HTML, empezamos a tener un problema a la hora de organizar nuestro código y nuestros assets. Cuando nuestros ficheros empiezan a crecer, es una buena práctica seguir ciertas recomendaciones para dividir el problema de nuestra solución en porciones más pequeñas. Los principios que seguimos basan el desarrollo en crear funcionalidades que tienen una única responsabilidad en ficheros por separado.

Por tanto, el número de nuestros ficheros empiezan a crecer y el número de dependencias que tenemos que incluir en nuestros HTML también crece. Llega un momento que la gran cantidad de scripts, hojas de estilo e imágenes a enlazar es tan grande que empieza a ser inmanejable.

Los empaquetadores de módulos como Webpack son buenas herramientas para solucionar esto porque nos permitirán seguir escribiendo nuestras funcionalidades en ficheros por separado, pero nos proporcionarán un proceso automático, donde todo se concentra en un único fichero para que sea más manejable.

Además, por la propia arquitectura de HTTP 1.1, no podemos olvidar que nos encontramos comunicándonos con servidores sin una sesión específica. Esto quiere decir que cada vez que un navegador hace una petición sobre un recurso (ya sea un JS, un CSS o un PNG) se crea un proceso de petición-respuesta con inicio y fin.

Ninguna de las peticiones que hagamos comparten un canal de difusión de recursos. Esto hace que el consumo de datos de red y los tiempos de carga aumenten y que los procesos de optimización sean importantes para todo desarrollador. Si conseguimos que todos nuestro código JS se encuentre en un único fichero, si conseguimos que todas nuestras clases estén en un único CSS, reduciremos el número de llamadas a servidor y por tanto nuestra aplicación será mejor.

Por tanto, ya sea Webpack u otro empaquetador de módulos, necesitamos una herramienta así.

## **La gestión de dependencias y su acoplamiento**

El tener muchos ficheros enlazados en mi HTML no es solo un problema de rendimiento, es también un problema a la hora de no cometer errores. Nos pasa con la Web que tenemos que ser muy cuidadosos para no mantener sucio el contexto global ya sea para no pisar variables de manera indeseable ya sea por no cometer otros fallos.

El orden en que yo ordeno la carga de ficheros en mi HTML puede ser clave para que todo funcione correctamente. Puede que muchos de mis ficheros hagan uso de jQuery, pero puede pasar que si yo no coloco la librería exactamente en el lugar preciso donde el navegador tenga que cargarla, cuando vaya a usar la librería, mi código no funcione.

Los empaquetadores de módulos trabajan muy bien resolviendo dependencias. Webpack por ejemplo, una de las primeras cosas que hace es acceder al fichero de entrada de mi aplicación y construir un grafo con las dependencias que se van teniendo. De esta forma va incluyendo todo aquello que se necesita. Estos empaquetadores usan los sistemas de módulos como pueda ser el descrito por CommonJS, AMD, la implementación nativa de ES6 o el `@import` de CSS3.

Lo bueno de usar un empaquetador como Webpack es que no tendremos el problema de dependencias muertas. ¿No os ha pasado alguna vez que habéis incluido una librería en vuestro index.html y que por lo que sea no la habéis usado porque no hacía falta? Sin darnos cuenta estamos penalizando el rendimiento de nuestra webapp con algo que no se utiliza.

Con Webpack esto no puede pasar. Como el empaquetador se encuentra construyendo el grafo de dependencias desde el fichero inicial, no hay manera de incluir una librería que no estaba indicada como dependencia, haciendo nuestras aplicaciones más livianas y específicas.

## ¿En qué más nos ayuda Webpack?

Pero lo bueno de Webpack es que no es un simple empaquetador de ficheros al uso. Webpack nos ayuda en varias tareas más.

### Permite transformaciones del código

Con Webpack podemos dejar de lado gestores de tareas como gulp o grunt ya que vamos a poder crear un sinfín de tareas en tiempo de empaquetado. Una de estas tareas es la transformación del código. Dentro de Webpack podemos escribir código en diferentes lenguajes como TypeScript, CoffeScript o ES6 y Webpack se encargará de transpilarlos en JavaScript.

No solo ocurre esto con los ficheros en lenguaje JavaScript. Como decíamos, Webpack trabaja con muchos tipos de ficheros y nos permite escribir ficheros en Jade y ser transformados en HTML al ser utilizados. Podemos escribir nuestros estilos con SASS, Stylus o LESS y Webpack nos devolverá un CSS minificado y optimizado. Lo mismo con las imágenes: Webpack nos permitirá transformar imágenes en cadenas de base64.

Webpack es un sistema muy modularizado por lo que todas estas funcionalidades podrán ser añadidas en cualquier momento.

## **Permite empaquetados parciales**

Puede darse el caso que nuestra aplicación sea tan grande, que no deseemos crear un solo paquete de la aplicación. O puede darse el caso de que no estemos en una SPA y que queramos hacer paquetes diferentes para los HTML de nuestra aplicación.

Podremos crear módulos intermedios que interaccionan con otros módulos y que se podrán ir componiendo en otros paquetes según necesidad. Las alternativas de configuración aquí son las que nos de la imaginación y los límites que desde negocio nos impongan.

## **Permite configurar empaquetados según el entorno y el target**

También será posible el crear diferentes empaquetados dependiendo del entorno en el que deseemos desplegar. Puede darse el caso que ciertas partes de la construcción no tengan que ser iguales para desarrollo que para producción.

Puede darse el caso también de que el paquete que queremos utilizar, se vaya a usar en más de un tipo de contexto diferente. Como sabemos, JavaScript es cada vez más agnóstico a la plataforma y esto hace que seamos capaces de ejecutar código JavaScript tanto en navegadores, como en servidores, como en aplicaciones de escritorio y móvil. Puede que según las necesidades de la plataforma no queramos que nuestros módulos tengan la misma funcionalidad o que se encuentran empaquetados de formas distintas.

Webpack cuenta con una serie de plugins y librerías que nos harán este trabajo más fácil a los desarrolladores.

## **Permite hacer compilaciones en caliente**

Webpack cuenta con una funcionalidad que puede ayudar mucho en tiempo de desarrollo y es que existe algo llamado HMR (Hot Module Replacement). Lo que hace esta tecnología es mantener en tiempo de desarrollo una especie de compilador que nos permite actualizar aquellas partes del código que hemos ido cambiando sin tener que refrescar el navegador.

De esta manera, un desarrollador que ha encontrado un bug, puede cambiar su código, hacer CTRL + S y Webpack se encargará de actualizar el código y compilarlo de tal manera que mi webapp no se vea afectada. De esta forma evitamos perdida de datos o estados de manera innecesaria cuando estamos haciendo alguna prueba en concreto. Es algo que ayuda mucho a la hora de perfeccionar una funcionalidad.

## ¿Cómo empezar?

Cómo utilizar todas funcionalidad tiene que ser uno de nuestros objetivos a lo largo de esta serie, vamos a empezar a explicar cómo utilizar y configurar todo lo necesario para conseguirlo:

Lo primero que tenemos que hacer es crear un nuevo proyecto de Node como es de costumbre:

```
$ mkdir testing-webpack
$ cd testing-webpack
$ npm init
```

Instalamos Webpack como dependencia del proyecto. Aunque Webpack presenta una CLI, es mejor que la usemos de manera local como dependencia para evitar los problemas de versiones entre proyectos:

```
$ npm install webpack --save
```

Después de configurar nuestro proyecto, crearemos la siguiente estructura de ficheros:

```
- testing-webpack
|-- dist
|-- src
    |-- utils.ts
    |-- main.ts
    |-- index.html
|-- webpack.config.js
```

Donde `src` es la carpeta en la que desarrollaremos nuestro código, `dist` la carpeta donde se incluirá todo aquello que queremos desplegar a producción y `webpack.config.js` es el fichero donde configuraremos todo lo relativo a Webpack y a cómo se tiene que empaquetar la aplicación.

Los ficheros `utils.ts` y `app.ts` son los dos módulos de la aplicación que queremos empaquetar. Están escritos en TypeScript y son estos:

```
// utils.ts
export default {
  sayMyName(name: string) {
    console.log('Hello ' + name);
  }
};

// main.ts
import utils from './utils';

utils.sayMyName('Heisenberg');
```

Mi fichero `index.html` es este:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>testing webpack</title>
  </head>
  <body>
  </body>
</html>
```

Por último, vamos añadir en el fichero `package.json` el siguiente comando para que luego en el futuro nos sea más intuitivo ejecutar la construcción de nuestra aplicación:

```
// package.json
{
  "name": "testing.webpack",
  "version": "0.0.1",
  "description": "Testing Webpack",
  "author": "jdonsan",
  "scripts": {
    "build": "webpack",
  },
  ...
}
```

Con esto, lo que conseguimos es que si ejecutamos:

```
$ npm run build
```

Se nos lance Webpack yendo al fichero por defecto `webpack.config.js`. Como ahora mismo no tiene configurado el proceso, el comando no hará nada en especial.

Para conseguir que haga cosas, primero tenemos que estudiar un poquito.

## Conceptos básicos

Al final todo parece muy complejo, pero entender Webpack se basa en estudiar sobre estos 4 conceptos:

### Los puntos de entrada

Dentro de una aplicación tenemos que indicar cuales son los puntos de entrada por los que queremos que un empaquetador empiece a trabajar. Tenemos que indicarlo para que se empiece a generar el grafo de dependencias.

En nuestro caso el punto de entrada de la aplicación es `/src/main.ts`. Para indicar a Webpack que empiece la inspección y el empaquetado de módulos por un punto de entrada concreto, lo hacemos así:

```
// webpack.config.js
const config = {
  entry: './src/main.ts'
};

module.exports = config;
```

En nuestro caso, al realizar aplicaciones SPA suele ser normal contar con un solo punto de entrada, pero puede darse el caso de que queramos hacer paquetes separados para las diferentes páginas de nuestra aplicación.

Para lograr eso, podríamos hacer algo tal que así:

```
// webpack.config.js
const config = {
  entry: {
    pageHome: './src/home/main.ts',
    pageProfile: './src/profile/main.ts',
    pageLogin: './src/login/main.ts'
  }
};

module.exports = config;
```

En este caso tenemos tres puntos de entrada para tres paquetes diferentes.

Otro caso puede ser el de que queramos separar nuestro código del resto de librerías externas. Podríamos hacer algo como esto:

```
// webpack.config.js
const config = {
  entry: {
    app: './src/app.ts',
    vendors: './src/vendors.ts'
  }
};
```

## Las salidas

Una vez que hemos decidido que punto o puntos de entrada queremos en nuestra configuración, es el turno de indicarle a Webpack donde tiene que dejar los paquetes que ha ido generando.

Al igual que tenemos un punto de entrada, podemos indicar un punto de salida de la siguiente forma:

```
// webpack.config.js

const config = {
  entry: './src/main.ts',
  output: {
    filename: 'app.min.js',
    path: __dirname + '/dist'
  }
};

module.exports = config;
```

De esta forma estamos indicando que queremos que lo empaquetado se guarde en la ruta `dist` y con el nombre de fichero `app.min.js`.

Si lo que deseamos es dar una salida a una configuración con varios puntos de entrada. Podemos hacerlo de la siguiente manera:

```
// webpack.config.js
const config = {
  entry: {
    home: './src/home/main.js',
    profile: './src/profile/main.js',
    login: './src/login/main.js'
  },
  output: {
    filename: '[name].min.js',
    path: __dirname + '/dist'
  }
}
};

module.exports = config;
```

Con este sistema, lo que le estamos diciendo a Webpack es que nos guarde los tres ficheros dentro de `dist` con el nombre que hayamos indicado en `entry`. De esta manera, nos guardará tres ficheros con el siguiente nombre: `home.min.js`, `profile.min.js`, `login.min.js`.

El atributo `output` cuenta con bastantes parámetros más que iremos viendo a lo largo de los posts de manera salteada. Si deseas estudiar un poco más cuales son, los tienes en este listado.

## Los loaders

Esta parte de la configuración nos sirve para indicar las transformaciones que queremos hacer a los ficheros. Los loaders se comportan muy parecido a los plugins que existen en gestores de tareas como gulp o grunt.

Los loaders son en sí partes modularizadas de Webpack por lo que cualquier desarrollador que necesite una transformación de sus ficheros, puede crear uno y engancharlo en su configuración.

Por ejemplo, yo quiero indicar en mi configuración de Webpack, que todos los ficheros que tengan la extensión `.ts` tengan que ser transpilados con `tsc` para que mi código escrito en TypeScript, se convierta en ES5.

Para conseguir esto, lo primero que tengo que hacer, es descargarme el loader preciso, de la siguiente forma:

```
$ npm install ts-loader --save-dev
```

Con este loader, ahora ya podemos hacer lo siguiente:

```
// webpack.config.js
const config = {
  entry: './src/main.ts',
  output: {
    filename: 'app.min.js',
    path: __dirname + '/dist'
  },
  module: {
    rules: [
      {
        test: /\.ts?$/,
        exclude: /node_modules/,
        loader: "ts-loader"
      }
    ]
  }
};

module.exports = config;
```

Dentro del objeto `module` existe un parámetro llamado `rules`. Este parámetro es un array que nos permite indicar todas las reglas de transformación que queremos que se ejecuten sobre nuestro código.

Lo que tendremos que hacer siempre es indicar la expresión regular del tipo de ficheros al que queremos que el loader haga efecto, en este caso todos aquellos ficheros que terminen en `.ts`.

Si a un fichero le afectan más de una regla porque coincide con la expresión regular, el orden en cómo se encuentren los loaders colocados es importante ya que se ejecutarán por el orden en que se encuentren en el array.

Los loaders pueden tener opciones. Es recomendable que cuando tengas una necesidad, compruebes si ya existe el loader. De ser así, observa la documentación y estudia las opciones que te puede dar.

## Los plugins

Pero no solo de transformaciones vive Webpack. Los loaders tienen bastantes limitaciones. Están muy bien para realizar ciertas transformaciones, pero a veces se quedan cojos en otras tareas que no tienen que ver con el empaquetamiento en sí.

Para esto han venido a ayudarnos los plugins. Son otra forma de extender y añadir funcionalidad extra a Webpack. Por ejemplo, tendremos plugins para incluir trazas entre loader y loader, para saber cómo ha ido la ejecución. Hay plugins para copiar estáticos de

unas carpetas a otras, o par inyectar las dependencias en nuestro index.html. Este será el ejemplo que hagamos.

Lo primero que haremos será incluir la dependencia en nuestro proyecto del plugin:

```
$ npm install html-webpack-plugin --save-dev
```

La forma de hacer uso de este plugin es la siguiente:

```
// webpack.config.js
const HtmlWebpackPlugin = require('html-webpack-plugin');

const config = {
  entry: './src/main.ts',
  output: {
    filename: 'app.min.js',
    path: __dirname + '/dist'
  },
  module: {
    rules: [
      {
        test: /\.ts?$/,
        exclude: /node_modules/,
        loader: "ts-loader"
      }
    ]
  },
  plugins: [
    new HtmlWebpackPlugin({ template: './src/index.html' })
  ]
};

module.exports = config;
```

Existe una sección en nuestro fichero para que los incluyamos llamada `plugins`, obviamente. Esta sección es un array de objetos que hacen cosas. Webpack sabrá cuando ejecutarlos y cuándo hacer uso de ellos.

En nuestro caso hemos instanciado la clase `HtmlWebpackPlugin`. Lo que hace este plugin es generarnos un fichero `index.html` con los paquetes generados referenciados directamente de serie y nos guarda la copia en el `output` que hayamos indicado.

Nosotros le hemos indicado como opción que coja como referencia una plantilla del fichero que se encuentra en `./src/index.html`. Un plugin muy sencillo, muy útil y con un montón más de configuraciones extra.

Y por ahora con lo explicado, podemos trabajar para proyectos que están empezando o que son sencillos de empaquetar, no necesitaremos mucho más.

## Conclusiones

Webpack me parece una herramienta muy potente. Te ofrece todo lo necesario para que con muy poco puedas hacer mucho. Es muy modularizable y fácil de extender. Su forma declarativa hace que incluir una nueva transformación, una nueva funcionalidad sea tan sencillo como copiar y pegar la documentación del desarrollador que lo hizo.

Sin embargo, su mecanismo por medio de configuración me parece personalmente poco intuitivo en ciertas ocasiones. No es difícil aprender y comprender los conceptos básicos, pero cuando se necesita unas configuraciones más complejas, el fichero de configuración llega a ser bastante difícil de manejar. Lo veremos cuando estudiemos los ficheros de construcción que nos genera vue-cli y cómo el potencial que tiene es lastrado por contener una configuración un tanto ilegible.

Esto que digo es una simple objeción personal y no se atiene a ningún hecho técnico. Habrá gente que entienda mejor estos sistemas de configuración declarativos y habrá otros que prefieran sistemas más imperativos como gulp. Como todo, es hacerse a la herramienta y trabajar mucho con ella.

Nos leemos :)

# Capítulo 15. Configurando nuestra primera build

Los conceptos básicos están bien. Ayudan a asentar el conocimiento que necesitamos para desarrollar cosas más complicadas. Pero sin un caso real donde aplicar esos conocimientos, es difícil aprender de forma consistente.

Como no es lo mismo contarlo que vivirlo, hoy vamos a crear una build que podría pasar por la de un proyecto real de tamaño pequeño-medio. A lo largo de los diferentes pasos a seguir para configurar nuestra build, iremos descubriendo nuevos conocimientos sobre Webpack que nos salvarán de más de un apuro.

El proceso que vamos a seguir es uno de los miles que podríamos hacer. Que yo lo haga así, no significa que sea prioritario que se haga así. De hecho, no significa que lo esté haciendo del todo bien. El post es una forma práctica de seguir aprendiendo conceptos, pero el flujo puede variar según vuestras necesidades.

Empecemos:

## Pensando en entornos

Lo primero que haremos será pensar en cómo van a ser los entornos en los que se podrá desplegar nuestra aplicación. Por lo general, se cuenta con 3 entornos en casi todos los proyectos: entorno de desarrollo, entorno de test y entorno de producción.

Creo que los nombres de los entornos son lo suficientemente explícitos como para no tener que explicarlos.

Para simplificar las cosas, dejemos de lado el entorno de test y centrémonos en montar una build que sirva tanto para desarrollo como para producción. Para que la build escale bien, vamos a mantener dos configuraciones separadas y vamos a extraer todo lo que tengan en común en una tercera configuración. De esta manera, deberemos tener en nuestro proyecto algo parecido a esto:

```
- proyecto-real-build
-- /build
---- webpack.base.conf.js
---- webpack.dev.config.js
---- webpack.pro.config.js
```

Lo primero que hemos hecho es meter todas las configuraciones en una carpeta llamada `/build` . organizando los ficheros así, no ensuciamos la carpeta raíz. Como sabemos, la carpeta raíz ya que de por sí está demasiado llena de ficheros de configuración genéricos. Separar todo lo relativo a la build mejora la legibilidad.

Lógicamente, si hacemos esto, cuando lancemos el comando `webpack` en línea de comandos no funcionará. Esto se debe a que Webpack no encuentra la configuración por defecto. Para solucionar esto, insertaremos un par de comandos nuevos en nuestro proyecto Node. En el `package.json` pondremos estas dos líneas dentro de `scripts`:

```
// package.json
{
  ...
  "scripts": {
    "build:dev": "webpack --config ./build/webpack.dev.conf.js",
    "build:pro": "webpack -- config .build/webpack.pro.conf.js"
  }
  ...
}
```

Cuando en mi línea de comandos ejecute `npm run build:dev` , se ejecutará la configuración de desarrollo y cuando se ejecute `npm run build:pro` , la de producción.

Volviendo a nuestra carpeta build, vemos que los nombres de los ficheros son claros e inequívocos. Todo desarrollador tiene claro sobre dónde va a influir cada uno. El fichero base ( `webpack.base.conf.js` ) contendrá la parte de la configuración más genérica. Este fichero es como el padre de las dos configuraciones específicas `webpack.dev.config.js` y `webpack.pro.config.js` .

Para que estos ficheros específicos hereden la configuración del base, tenemos que hacer uso un módulo de Webpack llamado `webpack-merge` . Así que lo primero que hacemos es instalar las dependencias que necesitamos en nuestro proyecto.

```
$ npm install webpack webpack-merge --save-dev
```

Teniendo esto, pongamos la configuración básica de nuestra aplicación. Por ejemplo, ya sea desarrollo o producción, la entrada y la salida las tenemos bastantes claras, y la generación del `index.html` también. Por tanto, incluimos esta configuración:

```
// ./build/webpack.base.conf.js

const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');

const config = {
  entry: ['./src/main.js'],
  output: {
    filename: 'app.min.js',
    path: path.resolve(__dirname, '..', 'dist'),
    publicPath: '/'
  },
  plugins: [
    new HtmlWebpackPlugin({ template: './src/index.html' })
  ]
};

module.exports = config;
```

Como hicimos en el post anterior, tenemos un punto de entrada en `./src/main.js` y una salida en `./dist/app.min.js`. Incluimos el plugin `HtmlWebpackPlugin` para que nos inyecte los scripts necesarios para que funcione nuestro `index.html`.

Ahora lo que hacemos es reusar esa configuración para los dos entornos, de la siguiente manera:

```
// Ponemos lo mismo tanto en ./build/webpack.dev.conf.js
// como ./build/webpack.pro.conf.js

const merge = require('webpack-merge');
const webpackBaseConf = require('./webpack.base.conf');

const config = merge(webpackBaseConf, {

});

module.exports = config;
```

De esta manera, hemos separado las 2 builds.

## Gestión de Assets

En el post anterior, hablamos mucho sobre cómo empaquetar módulos JS, pero hablamos poco de cómo añadir todos aquellos ficheros que no tienen nada que ver con JS. Ya dijimos que todo lo que tenga enlaces o dependencias es un candidato para ser empaquetado,

pero, lógicamente, no todos los ficheros queremos que se empaqueten de la misma manera.

Por ejemplo, habrá equipos que prefieran que el CSS de su proyecto, se añada a la build inyectándolo inline en el HTML, otros, por ejemplo, preferirán que se genere un fichero CSS por separado y que sea enlazado en el HTML. Con las imágenes pasa lo mismo. Habrá equipos que, por rendimiento, las necesitarán optimizadas y transformadas en String Base64 inyectadas en el HTML, y habrá equipos que las necesitará en una carpeta aparte.

Yo voy a usar la forma habitual de ir guardando todo en ficheros por separado. Veamos los diferentes tipos de assets que existen en un proyecto y cómo poder ponerlos donde queramos.

Para explicar el proceso, voy a añadir en src un componente que tenga todo lo necesario para funcionar:

```
- proyecto-real-build
-- /build
-- /src
---- /components
----- /users
----- data.csv
----- icon.png
----- index.js
----- style.css
---- index.html
---- main.js
```

He decidido desarrollar un componente porque va a demostrar mejor que no importa donde se encuentren nuestros assets. Ya no hace falta que se encuentren en carpetas genéricas. Podemos crear cosas más específicas y Webpack se encargará por nosotros de colocar todo donde debe.

Como vemos en este caso, mi componente hace uso de un ficheros de estilos, de una imagen y de un ficheros de datos. Veamos cómo configuro yo esto para que se meta cada cosa en su sitio:

## Cargando CSS

La idea es que, con loaders, podamos hacer todo. Por ejemplo, cada vez que mi JS o mi HTML haga uso de un fichero CSS específico, Webpack tendrá que meterlo en la carpeta dist en un fichero separado. Veamos el código del componente:

```
import './style.css';
import Data from './data.csv';

export default {
  render() {
    const items = Data.reduce((html, user) =>`${html}<li>${user.name} ${user.lastname1} ${user.lastname2}</li>`, '');
    return`<ul class="users">${items}</ul>`;
  }
};
```

Nuestro componente importa su CSS específico (`import './style.css'`). No es la mejor forma de hacerlo y en vue nunca importaremos el CSS así, pero para el ejemplo nos vale.

Lo que hacemos ahora es configurar nuestro `webpack.base.conf.js` para decirle qué tiene que hacer con este código. Lo hacemos así:

```
// webpack.base.conf.js

const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');
const ExtractTextPlugin = require('extract-text-webpack-plugin');

const config = {
  ...
  module: {
    rules: [
      {
        test: /\.css$/,
        use: ExtractTextPlugin.extract({
          use: 'css-loader'
        })
      }
    ]
  },
  plugins: [
    new HtmlWebpackPlugin({ template: './src/index.html' }),
    new ExtractTextPlugin('app.min.css')
  ]
};
```

Con esta nueva regla estamos diciéndole que todos los ficheros que terminen en `.css`, queremos que se encuentren extraídos en el fichero que hayamos indicado en el plugin, en este caso `app.min.css`. Lo que le decimos también es que, mientras tanto, vaya ejecutando el loader `css-loader` que es muy útil para resolver los `@import` de CSS3.

De esta forma, podemos ir escribiendo nuestro CSS en ficheros separados, y Webpack se encargará de ir juntándolos respetando el orden de las dependencias.

## Cargando imágenes

Si observamos el fichero CSS:

```
// ./src/components/users/style.css

.users {
  list-style: none;
}

.users li:before {
  content: '';
  background: url(./icon.png);
  background-size: cover;
  width: 64px;
  height: 64px;
  display: inline-block;
  vertical-align: middle;
  margin: 1rem;
}
```

Vemos que tiene como dependencia una imagen ( `icon.png` ). Necesitamos una manera de indicarle a Webpack la forma en la que tiene que tratar a esta imagen. Para ello, incluiremos un nuevo loader que se encarga de mover ficheros a la ruta que decidamos:

```
// webpack.base.conf.js

const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');
const ExtractTextPlugin = require('extract-text-webpack-plugin');

const config = {
  ...
  module: {
    rules: [
      {
        test: /\.css$/,
        use: ExtractTextPlugin.extract({
          use: 'css-loader'
        })
      },
      {
        test: /\.(png|svg|jpg|gif)$/,
        loader: 'file-loader',
        options: {
          name: 'img/[name].[hash:7].[ext]'
        }
      }
    ]
  },
  ...
};
```

Lo que indico es una nueva regla que se ejecuta cuando, un fichero cargado termine en `.png`, `.svg`, `.jpg` o `.gif` y se ejecutará el loader `file-loader`. Este loader lo que hace es guardar una copia en `dist`. A su vez, cambia la url del resto de ficheros que lo enlazasen para que apunten a la nueva ubicación.

El loader cuenta con una opción de configuración donde podemos indicar el nombre de la carpeta donde queremos guardarlo o nombre específico que debe tener: En este caso, le estamos diciendo, que guarde las imágenes en `./dist/img`.

Hemos metido, en el nombre, un hash que Webpack genera automáticamente. De esta forma, el navegador puede saber si la imagen ha cambiado o sigue siendo la misma y por tanto puede mantenerla cacheada. Esto lo explicaremos en el siguiente post dedicado al Caching.

## Cargar ficheros de datos

Otro tipo de ficheros que pueden existir en nuestra aplicación son JSON, CSV o XML. Existen loaders para estos ficheros. Puedo hacer uso de un CSV dentro de mi aplicación. Lo bueno es que estos loaders se encargan de convertir mis datos en objetos JSON,

incluyéndoles incluso, una pequeña API para hacer iteraciones sobre los datos.

Parece una tontería, pero la forma en la que se suelen cargar estos ficheros de datos, suele ser por medio de llamadas AJAX y parseos manuales. Si el fichero no es muy grande, Webpack nos lo incrustará en nuestra build con este loader:

```
{
  test: /\. (csv|tsv)$/,
  use: 'dsv-loader'
}
```

Todos los ficheros que hayan sido referenciados con formato `.csv` o `.tsv` serán procesados por el loader `dsv-loader`.

En nuestro programa hacemos caso de un CSV de datos con usuarios de ejemplo. Como comprobarás en el repositorio, no hemos tenido que hacer parseos. Para nosotros se ha convertido en un fichero JS más con el que trabajar.

## Otros assets

Las fuentes o los HTML estáticos, por poner otros ejemplos, también tienen sus loaders. Podremos hacer con estos ficheros procesos parecidos a los antes mencionados. Es nuestra responsabilidad comprobar que loaders y plugins existen de forma oficial para realizar todas estas transformaciones y tareas.

Si deseas ver todos los loader con los que cuenta Webpack, puedes hacerlo en el [catálogo oficial de la documentación](#).

## Resultado final

Con la configuración puesta, hemos conseguido crear esta build que funciona como nosotros deseamos. Dentro de `dist` obtendremos estos ficheros:

```
- proyecto-real-build
-- /dist
---- /img
----- icon.b8c6544.png
---- app.min.css
---- app.min.js
---- index.html
```

## Configurando Webpack en desarrollo

¿Qué necesidades sueles tener en desarrollo y quizá no sean necesarias en producción? A mi principalmente se me ocurren 3:

- Poder depurar mi código en el navegador
- Poder contar con un servidor ligero para lanzar el resultado
- Poder ver los resultados que he cambiado sin tener que recargar nada

Empecemos por el primero

## Poder depurar mi código en el navegador

La mejor forma de depurar nuestro código es desde el navegador, desde las devTools de Chrome, por ejemplo. El problema que tienen los empaquetadores tiene que ver con que el código a depurar queda poco legible. Como el código empaquetado, por lo general, cuenta con demasiada fontanería, es difícil saber en que lugar se ha producido un error.

Por esto, es importante contar con una herramienta que permita mapear los ficheros que nosotros hemos escrito con código cargado en el navegador. Esto se puede conseguir por medio de los sourcemaps.

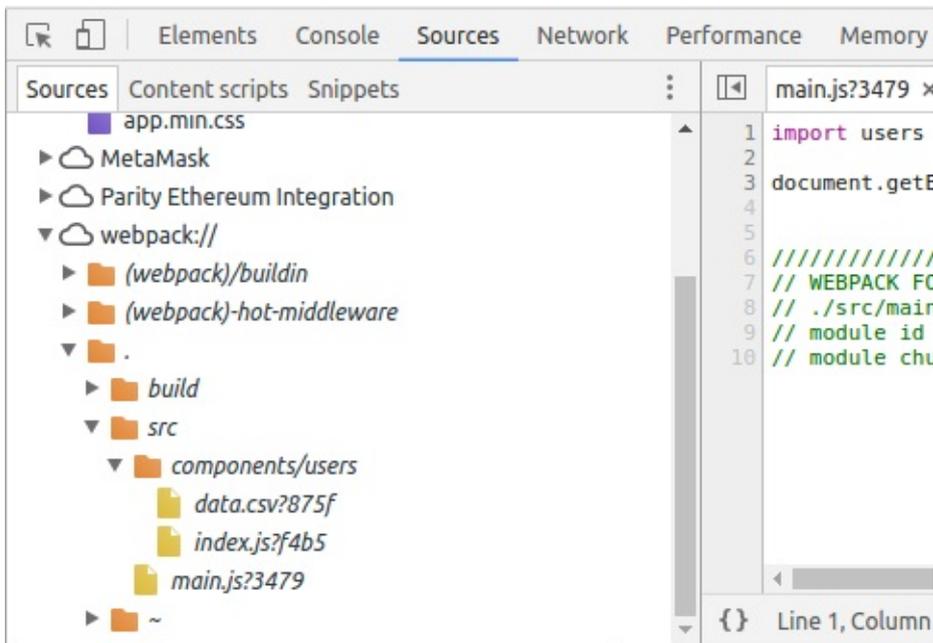
En Webpack es muy sencillo añadir un sourcemap a nuestro paquete. Con poner esta configuración, nos valdría:

```
const merge = require('webpack-merge');
const webpackBaseConf = require('./webpack.base.conf');

const config = merge(webpackBaseConf, {
  devtool: 'cheap-eval-source-map'
});

module.exports = config;
```

Con esto, si vamos al Developer Tools, comprobaremos que cuando volvamos a generar el paquete, tendremos esto en sources:



Ya podemos poner puntos de parada y depurar nuestro código.

## Poder contar con un servidor ligero para lanzar el resultado

Existen muchas formas de levantar un servidor web en Node. Podemos hacer uso de módulos como `http-server` que nos permitirá hacer peticiones sobre nuestra webapp.

Sin embargo, podemos sacarle mucho más potencial a Webpack gracias a un módulo llamado `webpack-dev-middleware`. Este módulo permite cargar nuestra build empaquetada dentro de un servidor Web. De esta forma tenemos mucho más margen para configurar cosas por medio de código.

El módulo es muy potente porque nos cargará la build en memoria. Esto significa que cuando estemos en desarrollo no se creará la carpeta `dist` en disco. lo que provocará que la construcción sea más rápida.

Es buen módulo también porque nos puede servir como proxy. Si nuestra aplicación hace llamadas a una API de nuestro servidor, nos permitirá redirigir el tráfico hacia donde digamos. De esta forma, podemos tener desacopladas aplicaciones front con back que en producción se encontrarán dentro del mismo dominio.

Para poner este modulo tendremos que hacer varias cosas. Lo primero es añadir un fichero `dev-server.js` a la carpeta build. Dentro de este fichero incluiremos la configuración de nuestro servidor:

```
const webpack = require("webpack");
const webpackDevMiddleware = require("webpack-dev-middleware")
const webpackDevConf = require('./webpack.dev.conf');
const app = require("express")();

const compiler = webpack(webpackDevConf);

app.use(webpackDevMiddleware(compiler, {
  publicPath: '/',
  quiet: true
}));

app.listen(3000, function () {
  console.log("Listening on port 3000!");
});
```

Lo que estamos haciendo es levantar un servidor con Express en el puerto 3000. Lo que hacemos es compilar nuestra configuración en tiempo ejecución del script. Luego le pasamos la configuración compilada al módulo `webpack-dev-middleware`. Gracias a esto, conseguimos que nuestro empaquetado final se sirva desde memoria.

Lo bueno de este módulo es que ya pone nuestro proyecto en `mode watch = on`, lo que significa que si hemos cambiado algo de nuestro código, el módulo sabe lanzar de nuevo la compilación.

El único cambio que tenemos que hacer ahora tiene que ver con cómo ejecutamos este build. Volvemos al `package.json`:

```
// package.json
{
  ...
  "scripts": {
    "build:dev": "node ./build/dev-server.js",
    "build:pro": "webpack --config .build/webpack.pro.conf.js"
  }
  ...
}
```

Cuando lanzo `npm run build:dev`. Se levanta un servidor, se compila el proyecto en memoria, me genera un sourcemap para depurar y encima es sensible a cambios.

## Poder ver los resultados que he cambiado sin tener que recargar nada

Lo último que nos queda es que, ese observador que recompila cada vez que haya un cambio, sepa indicar al navegador que se recargue. Como decimos, hay varias formas de hacer esto pues Webpack cuenta con la funcionalidad de carga de módulos en caliente, sin embargo, ya que estamos haciendo uso de un servidor 'ad hoc' con Express, usaremos otro middleware para conseguir esto.

El módulo que necesitamos es el siguiente: `webpack-hot-middleware`. Este módulo es el encargado de indicarle al navegador que existen cambios a actualizar. Para hacer uso de él, tenemos que seguir los siguientes pasos:

Tenemos que indicar a Webpack que nos habilite toda la funcionalidad de HRM. Para eso, incluimos el plugin `webpack.HotModuleReplacementPlugin` de la siguiente manera:

```
// ./build/webpack.dev.conf.js

const webpack = require('webpack');
const merge = require('webpack-merge');
const webpackBaseConf = require('./webpack.base.conf');

const config = merge(webpackBaseConf, {
  devtool: 'cheap-eval-source-map',
  plugins: [
    new webpack.HotModuleReplacementPlugin(),
    new webpack.NoEmitOnErrorsPlugin()
  ]
});

module.exports = config;
```

Lo siguiente es incluir el módulo como middleware de express para que esté monitorizando y comprobando si hay nuevos cambios o nuevas recompilaciones:

```
// ./build/webpack.dev.conf.js

const webpack = require("webpack");
const webpackDevMiddleware = require("webpack-dev-middleware")
const webpackHotMiddleware = require('webpack-hot-middleware');
const webpackDevConf = require('./webpack.dev.conf');
const app = require("express")();
const compiler = webpack(webpackDevConf);

app.use(webpackDevMiddleware(compiler, {
  publicPath: '/',
  quiet: true
}));

app.use(webpackHotMiddleware(compiler));

app.listen(3000, function () {
  console.log("Listening on port 3000!");
});
```

Este módulo es capaz de actualizar el código en caliente en el navegador, pero no es capaz de hacer un reload del navegador. Para ello, el módulo cuenta con una librería que permite suscribirnos a un evento que nos indica si debemos hacer un reload. Lo hacemos de la siguiente manera:

```
const hotClient =
  require('webpack-hot-middleware/client?noInfo=true&reload=true');

hotClient.subscribe(function (event) {
  if (event.action === 'reload') {
    window.location.reload()
  }
});
```

Este fichero lo tenemos que incluir en nuestro compilado. Para ello, cambiamos el `entry` de `dev` para que lo tenga en cuenta. Lo incluimos el primero en el array o no funcionará:

```
const webpack = require('webpack');
const merge = require('webpack-merge');
const webpackBaseConf = require('./webpack.base.conf');

webpackBaseConf.entry = ['./build/dev-client'].concat(webpackBaseConf.entry);

const config = merge(webpackBaseConf, {
  devtool: 'cheap-eval-source-map',
  plugins: [
    new webpack.HotModuleReplacementPlugin(),
    new webpack.NoEmitOnErrorsPlugin()
  ]
});

module.exports = config;
```

En esta línea:

```
webpackBaseConf.entry = ['./build/dev-client'].concat(webpackBaseConf.entry);
```

Incluimos este pequeño script que hemos generado.

De esta forma, cuando cambiemos un módulo de nuestro proyecto, el navegador se actualizará sin que nosotros hagamos nada. Un poco de fontanería, pero bueno. Una vez hecho, no nos molestará más.

## Configurando Webpack en producción

Toda la configuración anterior no es innecesaria en nuestro paquete de producción. La fase de empaquetado en producción es una fase de resultados finales óptimos. Es decir, todo lo que se genera, tienen que ser ficheros estáticos que se encuentren en disco y que tengan un tamaño lo más reducido posible, sin que perdamos funcionalidad.

Tareas como la depuración y la carga dinámica dejan de ser importantes y minificar y comprimir pasan a ser unas de las cuestiones prioritarias.

## Optimizando el index.html

En nuestro caso, estamos generando 3 ficheros que estaría bien que optimizáramos. Para hacerlo incluiremos una serie de plugins que harán por nosotros el trabajo. Lo primero que haremos será indicar que el `index.html` se encuentre minificado. Para esto, lo hacemos así:

```
// ./build/webpack.pro.conf.js

const webpack = require('webpack');
const merge = require('webpack-merge');
const webpackBaseConf = require('./webpack.base.conf');
const HtmlWebpackPlugin = require('html-webpack-plugin');

const config = merge(webpackBaseConf, {
  plugins: [
    new HtmlWebpackPlugin({
      template: './src/index.html',
      minify: {
        removeComments: true,
        collapseWhitespace: true,
        removeAttributeQuotes: true
      }
    })
  ]
});

module.exports = config;
```

Añadimos un nuevo parámetro al plugin `HtmlWebpackPlugin` llamado `minify`. Nos permite configurar varias cosas. Por poner un ejemplo de todo lo que se puede hacer, he decidido eliminar los comentarios del HTML final, los espacios blancos que no aporten marcado y las dobles comillas de atributos que sean posibles quitarla. Con esto reducimos el tamaño y el funcionamiento sigue siendo el mismo.

## Optimizando el app.min.css

Hacemos algo muy parecido con el CSS. Añadimos un nuevo plugin que permite optimizar estilos. Tan fácil como añadir el plugin `OptimizeCSSPlugin` :

```
// ./build/webpack.pro.conf.js

const webpack = require('webpack');
const merge = require('webpack-merge');
const webpackBaseConf = require('./webpack.base.conf');
const HtmlWebpackPlugin = require('html-webpack-plugin');
const OptimiceCSSPlugin = require('optimice-css-assets-webpack-plugin');

const config = merge(webpackBaseConf, {
  plugins: [
    new HtmlWebpackPlugin({
      template: './src/index.html',
      minify: {
        removeComments: true,
        collapseWhitespace: true,
        removeAttributeQuotes: true
      }
    })
    new OptimiceCSSPlugin()
  ]
});

module.exports = config;
```

## Optimizando `app.min.js`

Por último, añadimos plugins para optimizar el JS. Es igual que el anterior, pero con los nuevos plugins:

```
const webpack = require('webpack');
const merge = require('webpack-merge');
const webpackBaseConf = require('./webpack.base.conf');
const OptimizeCSSPlugin = require('optimize-css-assets-webpack-plugin');
const HtmlWebpackPlugin = require('html-webpack-plugin');

const config = merge(webpackBaseConf, {
  plugins: [
    new HtmlWebpackPlugin({
      template: './src/index.html',
      minify: {
        removeComments: true,
        collapseWhitespace: true,
        removeAttributeQuotes: true
      },
    }),
    new webpack.optimize.UglifyJsPlugin(),
    new OptimizeCSSPlugin()
  ]
});

module.exports = config;
```

`webpack.optimize.UglifyJsPlugin` es un plugin del propio Webpack que sirve para optimizar JS.

## Marcar paquete como producción

Es buena práctica que marquemos el paquete como producción. Esto es así porque cuando la variable de entorno se encuentra en producción hace que se desconecten mecanismos que no deberían verse.

Por ejemplo, en vue, cuando el compilado se encuentra en producción, las Developers Tools especiales de vue se desactivan para que nadie pueda depurar las rutas o los flujos. También, muchas librerías usan este mecanismo para esconder trazas de log. Con ello ganamos en seguridad y en velocidad.

Para incluir esto, mete este nuevo plugin de esta manera:

```
// webpack.pro.config.js

new webpack.DefinePlugin({
  'process.env.NODE_ENV': JSON.stringify('production')
});
```

## Conclusiones

Con todo esto, tenemos mucho avanzado. Nos hemos dejado mucho por el camino. No hemos explicado como incluir un linter que nos evalúe la homogeneidad de nuestro código, ni hemos incluido plugins que ejecuten tests. No hemos añadido plugins para cambiar las cadenas de i18n, ni hemos transpilado nuestros ficheros SASS para convertirlos en CSS.

Sin embargo, creo que lo más difícil está conseguido que es saber cómo empezar y cómo continuar en una herramienta con Webpack. El resto de cosas tendrán que irse incluyendo según necesidad. Puede que en mi proyecto ahora mismo la internacionalización no sea importante, pero dentro de 3 meses sí. Quién sabe.

En el último post de la serie comprobaremos los conocimientos que hemos adquirido, analizando la build que nos genera vue-cli por defecto. Si entendemos un build tan complejo como ese, estaremos preparado para seguir nuestro camino.

Nos leemos :)

Os dejo el repositorio por aquí por si queréis ver lo que hemos hecho en este post.

## Capítulo 16. Caching, Shimming & Splitting

Con lo trabajado hasta ahora podríamos tirar sin problemas. Sin embargo, Webpack presentan un potencial mayor al de solo crear empaquetados. Webpack es una herramienta que se preocupa mucho por la optimización de nuestros estáticos.

Está bastante sensibilizado con aquellos proyectos que necesitan reducir hasta el más mínimo byte para hacer que la aplicación cargue en nuestro navegador lo más rápido posible. No olvidemos que los desarrolladores Web hemos creado un sistema sobre un protocolo (HTTP) que no estaba pensado para este envío masivo de información.

Por tanto, es bueno saber que con una sola herramienta vamos a poder hacer estas pequeñas mejoras que nos distanciarán con el resto de webs en cuanto a tamaño de ficheros y tiempo de carga.

Hoy, para terminar la serie de Webpack, hablaremos de 3 conceptos avanzados muy orientados a mejorar estos aspectos de optimización: el Caching, el Shimming y el Splitting.

Veamos:

### Caching

O acumular. ¿De qué forma puede ayudarnos Webpack a cachear, si los ficheros van a distribuirse a los navegadores de nuestros usuarios? Como sabemos, los navegadores modernos cuentan con sistemas de caché automáticos por defecto.

Esto hace, que si un usuario vuelve a visitar nuestra web, el navegador compruebe si ya tiene copias guardadas de los ficheros que queremos pedir a un servidor determinado.

Esto nos ha provocado ciertos quebraderos de cabeza a los desarrolladores, ya que cuando se han realizado cambios en nuestros empaquetados, a veces los cambios no han podido ser repercutidos en el navegador porque los ficheros cacheados y cambiados se llamaban igual.

La forma que hemos tenido para engañar al navegador ha sido añadiendo una variable en el queryString, que provoque esta actualización por diferencias en la URI del estático.

Por ejemplo, podemos hacer peticiones de los script así:

```
application.js?build=1
application.css?build=1
```

De esta forma, cuando vamos creando una nueva build, aumentaremos el número y el navegador sabrá que es un nuevo fichero. Para hacer esto hemos usado muchas tretas. Pero con Webpack podemos hacerlo por defecto. En Webpack contamos con una serie de patrones para nombrar a los empaquetados de salida. Por ejemplo, podemos hacer esto:

```
// webpack.config.js
const path = require("path");

module.exports = {
  entry: {
    vendor: "./src/vendor.js",
    main: "./src/index.js"
  },
  output: {
    path: path.join(__dirname, "build"),
    filename: "[name].[hash].js"
  }
};
```

Si nos fijamos, en lo que pone en `output.filename`, estamos creando un patrón con palabras reservadas que solo Webpack sabe interpretar cuando se encuentran entre corchetes. Lo que estamos diciendo es que a los empaquetados que se genere les ponga el nombre (`[name]`) indicado en el `entry`, que la extensión final sea `.js` y que en el medio lleve un hash.

Cada vez que se ejecuta un proceso de empaquetado por parte de Webpack se genera un hash. Este hash se puede utilizar para indicar la build que acabamos de construir de esta forma.

Si ejecutamos la configuración anterior, obtendremos algo como esto:

```
Hash: 2a6c1fee4b5b0d2c9285
Version: webpack 2.2.0
Time: 62ms

          Asset      Size  Chunks             Chunk Names
vendor.2a6c1fee4b5b0d2c9285.js  2.58 kB      0 [emitted]  vendor
main.2a6c1fee4b5b0d2c9285.js  2.57 kB      1 [emitted]  main
   [0] ./src/index.js 63 bytes {1} [built]
   [1] ./src/vendor.js 63 bytes {0} [built]
```

Con esto, cada vez que se haga un cambio en nuestro código, se actualizarán las copias del navegador. Sin embargo, seguimos teniendo un problema. Enlazar todos los assets al mismo hash, provoca que todos los assets tengan que ser actualizados por el navegador.

Esto no es un comportamiento real ya que en muchas ocasiones nuestro código cambiará, pero el código de las dependencias ( `vendor` ), por ejemplo, no lo hará tan asiduamente. Necesitamos un mecanismo que nos desacople este funcionamiento.

La solución es generar un hash por cada paquete creado. Este hash se genera a partir del contenido del fichero. De esta forma, si el fichero no cambia, el hash sigue siendo el mismo.

Lo que hacemos es, no indicar la palabra reservada `hash`, y sí la palabra reservada

`chunkhash` .

```
module.exports = {
  /* ... */
  output: {
    /* ... */
    filename: "[name].[chunkhash].js"
  }
};
```

De esta manera, si ejecuto mi configuración, observaremos que los hashes no se comparten. Cada fichero es independiente en el sistema de caché.

```
Hash: cfba4af36e2b11ef15db
Version: webpack 2.2.0
Time: 66ms
```

	Asset	Size	Chunks		Chunk Names
	vendor.50cfb8f89ce2262e5325.js	2.58 kB	0	[emitted]	vendor
	main.70b594fe8b07bcedaa98.js	2.57 kB	1	[emitted]	main
	[0] ./src/index.js	63 bytes	{1}	[built]	
	[1] ./src/vendor.js	63 bytes	{0}	[built]	

## Shimming

O calzar. Webpack sabe trabajar con módulos escritos con ES6, CommonJS o AMD entre otros. Sin embargo, hay librerías de terceros que no se encuentran escritas con estos sistemas de módulo y que, o siguen otro sistema, o tienen incluido todo lo que necesitan para funcionar en el ámbito global.

Webpack nos aporta mecanismos para insertar estas librerías en los paquetes y evitar que se rompa la construcción de nuestro paquete. Al final, lo que Webpack hace es crear una serie de envoltorios sobre estas librerías para que sepan adaptarse bien a nuestras

necesidades.

Por ejemplo, aunque Webpack sabe buscar dependencias dentro de `node_modules` para incluirlas en el paquete, muchas veces, podremos ser nosotros los que indiquemos a por dónde tiene que ir a buscarla. Si yo hago esto:

```
// webpack.config.js

module.exports = {
  ...
  resolve: {
    alias: {
      jquery: "jquery/src/jquery"
    }
  }
};
```

Le estoy diciendo a Webpack que cada vez que encuentre una dependencia global que ponga `jquery`, no vaya a la carpeta `node_modules` y que en su lugar vaya a la ruta que hemos indicado, `jquery/src/jquery`.

Esto es una buena práctica de optimización. En vez de coger el fichero minificado por defecto, hacemos que Webpack vaya a por el código normal y que sea él el encargado de hacer el proceso.

Otro problema viene cuando una librería está haciendo uso de una variable global de otra librería. Por ejemplo, imaginemos en un plugin desarrollado con jQuery, lógicamente tiene una dependencia de una variable global, pero queremos seguir haciendo uso del sistema de módulos.

Si empaquetamos con Webpack sin hacer nada, cuando usemos la aplicación, nos dirá que esa variable no existe en módulo correspondiente. Webpack nos proporciona un sistema por el cual el es capaz de inyectarnos los `imports` que necesitamos en cada momento. Si hago esto:

```
module.exports = {
  plugins: [
    new webpack.ProvidePlugin({
      $: 'jquery',
      jQuery: 'jquery'
    })
  ]
};
```

Cada vez que Webpack encuentre el uso de \$ o jQuery en un módulo, incluirá al principio del mismo un `var $ = require('jquery')`. De esta manera todo funcionará correctamente.

Puede darse el caso también, que una de estas librerías que se encuentran en el ámbito global, hagan uso de `this`. Cuando se usa `this` dentro de un sistema de módulos como Webpack, el puntero o apunta a `window` sino a `module.exports`. Esto se puede solucionar gracias al loader `import-loader`:

```
module.exports = {
  module: {
    rules: [{
      test: require.resolve("some-module"),
      use: 'imports-loader?this=>window'
    }]
  }
};
```

Cuando una librería sufra este mal, ejecutaremos este loader para que sus `this` apunten a `window`.

## Splitting

O dividir. Lo más seguro es que no queramos que todo nuestro código de aplicación se encuentre en un único paquete. Por ello, Webpack nos proporciona formas de dividir nuestros empaquetados en diferentes trozos para cumplir con ciertos aspectos claves de la optimización.

Dentro esta opción de división, contamos con 3 tipos:

### Separación del CSS

El primero trata sobre el hecho de poder separar nuestro CSS de nuestro JavaScript. Esta opción ya la comentamos en el post anterior y no vamos a profundizar, pero recuerda que existe un plugin llamado `ExtractTextWebpackPlugin` que nos ayudará en esta posibilidad de aislar el CSS en un empaquetado por separado.

### Separación de librerías externas

El segundo aborda la separación de código propio con el de código de librerías de terceros. Aunque lo hemos ido comentando en la serie, no hemos profundizado en este hecho. Poder separar las librerías externas en un compilado aparte, nos va a ayudar en procesos de

optimización.

Si podemos separar este código externo, que por lo general va a cambiar menos que el nuestro propio, podemos hacer que un buen porcentaje de código sea cacheado por los navegadores sin que nosotros tengamos que preocuparnos de él.

Para conseguir esto, lo primero que tenemos que hacer es indicar que queremos que se generen diferentes compilados. Lo hacemos como explicamos en el primer post de Webpack. Indicamos dos puntos de entrada diferentes. Imaginemos que usamos la librería `moment` como librería externa y queremos separarla del paquete principal:

```
var path = require('path');

module.exports = function(env) {
  return {
    entry: {
      main: './index.js',
      vendor: 'moment'
    },
    output: {
      filename: '[name].[chunkhash].js',
      path: path.resolve(__dirname, 'dist')
    }
  }
}
```

Con esto se generan dos paquetes dentro de `dist` uno para `main` y otro para el `vendor`.

La configuración anterior presenta varios problemas. El primero es que aunque estemos separando `moment` en un paquete aparte dentro de `vendor`, no significa que no se esté incluyendo también en `main`. Estamos duplicando código. El segundo es un problema de flexibilidad. Estamos haciendo que en nuestro paquete de `vendor` solo se incluya la librería `moment`. Esto no es real pues muchas dependencias se deberán incluir aquí.

Para solucionar el primer problema, usamos el plugin `CommonsChunkPlugin` que lo que hace es extraer todas las dependencias repetidas de los paquetes generados a donde nosotros le digamos. Por lo tanto hacemos esto:

```
var webpack = require('webpack');
var path = require('path');

module.exports = function(env) {
  return {
    entry: {
      main: './index.js',
      vendor: 'moment'
    },
    output: {
      filename: '[name].[chunkhash].js',
      path: path.resolve(__dirname, 'dist')
    },
    plugins: [
      new webpack.optimize.CommonsChunkPlugin({
        name: 'vendor'
      })
    ]
  }
}
```

Incluimos el plugin e indicamos dónde queremos guardar los módulos comunes. En este caso en `vendor`.

El segundo caso va muy relacionado con el primero ya que lo que vamos a hacer es quitar el punto de entrada de `vendor` y vamos a dejar que el plugin de `CommonsChunkPlugin` se encargue de generar el paquete de forma implícita.

```
var webpack = require('webpack');
var path = require('path');

module.exports = function() {
  return {
    entry: {
      main: './index.js'
    },
    output: {
      filename: '[name].[chunkhash].js',
      path: path.resolve(__dirname, 'dist')
    },
    plugins: [
      new webpack.optimize.CommonsChunkPlugin({
        name: 'vendor',
        minChunks: function (module) {
          return module.context && module.context.indexOf('node_modules') !==
-1;
        }
      })
    ]
  };
}
```

Lo que hemos hecho es decirle a Webpack que saque los módulos comunes en el paquete `vendor` de todo lo que se encuentra dentro de la carpeta `node_modules`. Como todas nuestras dependencias externas se encontrarán en esta carpeta, ya tenemos la separación en un paquete independiente solucionado.

Seguimos teniendo un último problema. Webpack incluye en todos los empaquetados un motor de ejecución para gestionar estos módulos en los paquetes que realiza. Si hemos creado dos paquetes dentro de nuestra aplicación, hemos hecho que Webpack incluya este runtime en cada uno de ellos. Para optimizar esto del todo tenemos una solución: Sacar este motor a un nuevo paquete denominado por Webpack como `manifest`. Lo hacemos así:

```

var webpack = require('webpack');
var path = require('path');

module.exports = function() {
  return {
    entry: {
      main: './index.js' //Notice that we do not have an explicit vendor entry here
    },
    output: {
      filename: '[name].[chunkhash].js',
      path: path.resolve(__dirname, 'dist')
    },
    plugins: [
      new webpack.optimize.CommonsChunkPlugin({
        name: 'vendor',
        minChunks: function (module) {
          // this assumes your vendor imports exist in the node_modules directory
          return module.context && module.context.indexOf('node_modules') !== -1;
        }
      }),
      //CommonChunksPlugin will now extract all the common modules from vendor and main bundles
      new webpack.optimize.CommonsChunkPlugin({
        name: 'manifest' //But since there are no more common modules between them we end up with just the runtime code included in the manifest file
      })
    ]
  };
}

```

Con volver a incluir el plugin para que se vuelva a ejecutar el proceso de separación de porciones comunes e indicarle que el paquete se llamará manifest, Webpack ya sabe que tiene que extraer el runtime en un nuevo fichero.

Con esto tenemos una forma optima de empaquetado de nuestra aplicación.

## Separación bajo demanda o carga perezosa

El último caso que nos queda es la posibilidad de cargar zonas de nuestro código bajo demanda del usuario. Puede que nuestra aplicación sea tan grande y el número de funcionalidades tan profunda, que no merezca la pena hacer que el usuario se descargue todo al principio.

Con Webpack, podemos indicar que se genere un paquete mínimo con lo que más se suele usar y después ir cargando otros módulos cuando sean usados.

La forma de conseguir esto no es por medio de la configuración de nuestro Webpack, sino que nos supone un cambio a la hora de que desarrollemos. Dentro de la especificación de JavaScript ya existe un apartado dedicado a la carga de módulos bajo demanda. La forma de usarlo es por medio de la nueva palabra reservada `import()`.

Este método es asíncrono y permite ser usado por medio de promesas y funciones asíncronas. Por ejemplo, si quisiéramos cargar `moment` bajo demanda, podría hacer algo como esto:

```
function determineDate() {
  import('moment').then(function(moment) {
    console.log(moment().format());
  }).catch(function(err) {
    console.log('Failed to load moment', err);
  });
}

determineDate();
```

Webpack entiende perfectamente este código y se encargará de cargar `moment` cuando le hayamos indicado. Vue hará mucho uso de esta funcionalidad también, por lo que es importante tenerla clara.

Con funciones asíncronas sería de la siguiente manera:

```
async function determineDate() {
  const moment = await import('moment');
  return moment().format('LLLL');
}

determineDate().then(str => console.log(str));
```

Hay mucha documentación al respecto sobre casos específicos de esta funcionalidad. Por ahora no creo que sea preciso que entremos en más detalle, pero os dejo la documentación por aquí por si alguien lo ve oportuno profundizar.

## Conclusión

Con lo aprendido en el post tenemos un conocimiento mucho más profundo de cómo podemos exprimir Webpack para que nos permita configurar y optimizar los paquetes creados de la mejor manera posible.

Conocer una herramienta como Webpack es difícil sino se llega a probar y trastear. Aunque en la serie hemos podido explicar muchos conceptos, en vuestros proyectos podrán salir nuevos casos de uso que tendréis que mirar y analizar. Como todo proyecto no es igual, ni tiene el mismo nivel de complejidad, solo puedo deciros eso.

Con Webpack va a ser posible hacer casi todo y si no se puede, siempre podréis optar por pedir ayuda a la comunidad o por ampliar la funcionalidad de Webpack.

A lo largo de la serie no he podido hablar de cómo de rápido o lento llega Webpack a trabajar. Tengo constancia de que las primeras versiones tenían unos tiempos de trabajo bastante grandes como para tener la herramienta en cuenta, pero si es cierto que todo lo que llega de la comunidad de Webpack últimamente son buenas noticias en cuanto al rendimiento de las nuevas versiones.

Si algún día empiezo a usar Webpack en producción, os comentaré la jugada, por el momento es todo.

A partir de ahora, volveremos a vue para terminar con la serie. Nos dedicaremos a estudiar cómo podemos renderizar nuestros componentes y vistas en servidor para ganar en tiempos y en posicionamiento para los buscadores. Mucho trabajo todavía :)

Nos leemos :)

# Capítulo 17. Introducción a Server-Side Rendering

Tenemos todas las herramientas necesarias para desarrollar aplicaciones con VueJS. Hemos aprendido a crear componentes, a renderizarlos e interaccionar con ellos. Hemos aprendido a escalar nuestras aplicaciones mediante vue-router y vuex e incluso hemos aprendido a crear una build con Webpack para empaquetar nuestras aplicaciones.

Puede que todo esto sea suficiente para la mayoría de proyectos y puede que haya un número concreto de proyectos, en los que los tiempos de respuesta sean críticos para conseguir una masa suficiente de clientes, y que necesitemos nuevas herramientas para conseguir este plus de carga y eficiencia.

VueJS no abandona a estos desarrolladores y ha creado una librería que se basa en lo que llamamos "Renderizado en la parte servidor". A lo largo de los próximos capítulos estudiaremos este nuevo concepto que se está poniendo tan de moda en frameworks como Vue o React, aunque nos sean harto conocidos para los que hemos trabajado en proyectos con ASPX o JSP.

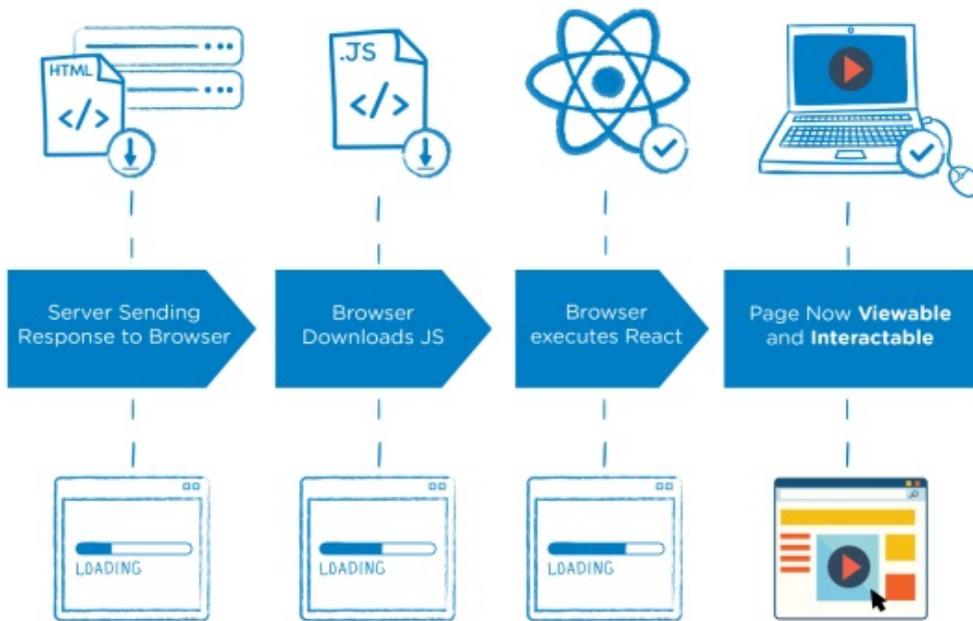
Veamos:

## ¿Qué es?

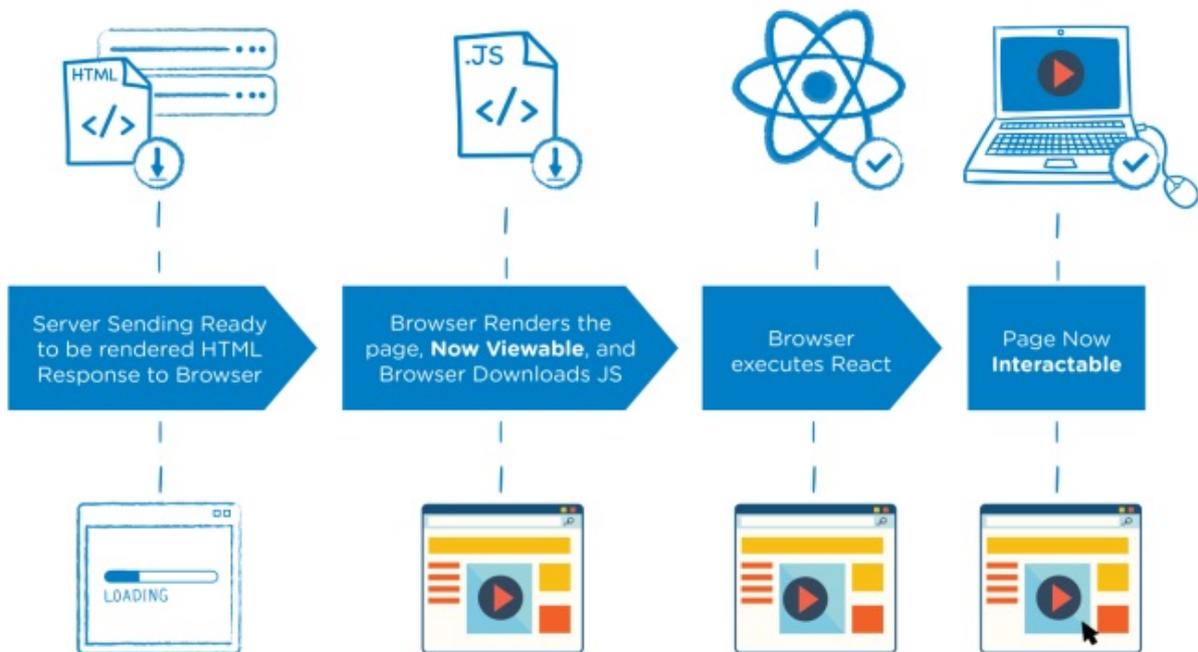
El Server-side rendering (o renderizado en la parte servidor) se basa en la posibilidad de poder renderizar el HTML de nuestros componentes en cadenas de texto en la parte servidor, en vez de la parte cliente. Estas cadenas serán las respuestas que nuestros servidores de NodeJS devolverán a las peticiones principales de nuestra Web. En vez de funciones que manipulan DOM en el navegador, delegamos este renderizado a una fase anterior en el servidor.

Los siguientes diagramas explican perfectamente la diferencia entre CSR (Client-side rendering) y SSR (Server-side rendering):

# CSR



# SSR



Es decir, en vez de que un usuario obtenga un `index.html` vacío en la primera petición de nuestra Web, como suele pasar en una SPA convencional, el sistema renderiza un primer esqueleto de la aplicación, para que el usuario tenga sensación inmediata de progreso. El contenido puede ser hidratado posteriormente si se desea.

Delegamos las primeras fases de renderizado a nuestro servidor en vez de a nuestro cliente. Esto tecnológicamente es muy potente porque consigue que componentes que se renderizaban en cliente, ahora también se rendericen en servidor. Hemos conseguido que nuestros componentes sean isomórficos o universales y nos permite evitar dos problemas típicos de las SPA.

## ¿Cuándo es útil?

Pensarás ¿para qué nos puede ser útil un mecanismo como este? Si bien hemos dicho siempre, una aplicación SPA ha tenido como misión permitir escalar mejor nuestros servidores ¿No es esto un regresar al pasado y volver a la época en que renderizábamos nuestras plantillas de PHP o Jade para NodeJS?

Pues podríamos decir que sí, con la diferencia de que ahora nuestra aplicación tiene la posibilidad de renderizarse donde nosotros decidamos. Tenemos que ser prácticos y serios y darnos cuenta de que existen una serie de proyectos donde las aplicaciones SPA no están cumpliendo su cometido. Por ejemplo:

## Posicionamiento Web

Necesitamos una aplicación que se encuentre posicionada en los buscadores más importantes del mercado, como Google o Bing, en las primeras posiciones. Los robots de estos buscadores se encuentran muy optimizados y empiezan a trabajar muy bien con aplicaciones que tienen un uso intensivo de JavaScript.

Ahora bien, trabajan muy bien con carga de JavaScript síncrono. Del asíncrono no quieren saber nada hasta el momento. Como nuestra aplicación empieza con un spinner de carga, nos vamos olvidando de que el buscador indexe algo más que tenga que ver con nuestra estructura de página o con nuestro contenido.

Es por esto que el renderizado en servidor es necesario. Necesitamos sistemas que permitan renderizar estos componentes en servidor para que cuando lleguen a los robots de Google o Bing, contenga un mayor número de matices en el HTML con el que nos puedan indexar.

## Prioridad al tiempo de carga frente al contenido

Los usuarios son muy impacientes. Y no vamos a negarlo, si estoy haciendo que un usuario se descargue una webapp entera y que se renderice, puede que lo hayamos perdido por el camino por tiempos excesivos de carga. La sensación de progreso y de que algo esté pasando es muy importante para mantener a nuestros usuarios.

Es por eso que si el propio servidor es capaz de agilizar estas primeras fases y de servir la estructura base, el usuario tendrá una sensación de carga menor. Renderizar e hidratar en servidor puede ayudarnos mucho a mantener a nuestros potenciales clientes.

## ¿Qué impacto en desarrollo me puede suponer usar SSR?

Tenemos que tener claro que hacer uso de SSR no va a ser útil para todos y que su uso tiene que estar muy justificado en nuestra aplicación, pues podemos tener 3 imprevistos con los que no habíamos contado:

1. Contar con SSR nos va a suponer tener un aumento en gastos por infraestructura. Nuestras aplicaciones SPA, cuando se encuentran empaquetadas, se comportan como estáticos que pueden ser servidos desde un CDN que tengamos contratado. Si necesitamos SSR, mínimo necesitamos una máquina que ejecute NodeJS para que se realice el sistema de renderizado en servidor.
2. Delegar este renderizado a servidor nos va a suponer una mayor carga en recursos. Lo que quizá provoque que nuestro sistema escale peor y que necesitemos más CPU o memoria en casos de un uso intensivo del sistema.
3. Algunas librerías de cliente que vayamos a usar, puede que no se lleven bien con el SSR y que tengamos que adaptarlas para que funcionen con este sistema. O incluso puede que ya tengamos una aplicación en vue servida como una SPA, que esté haciendo uso de hooks del ciclo de vida del componente, y que en SSR no vayan a funcionar por su propia naturaleza, teniendo que hacer adaptaciones.

## ¿Y si estos imprevistos me son insalvables a corto plazo?

Puede que estos imprevistos nos supongan un mayor esfuerzo de lo que podamos ganar con el propio SSR, por lo tanto tengámoslo en cuenta. La propia comunidad de vue pide cautela a la hora de usar estos sistemas y son partícipes de hacer uso, en primeras fases de desarrollo prerendering.

El prerendering es un proceso que se realiza en tiempo de construcción de la aplicación y que permite renderizar ciertos componentes que nosotros indiquemos. La idea subyace en usar Webpack para renderizar algunos componentes o vistas que nosotros veamos clave. Por ejemplo, sería buena opción renderizar las pantallas principales que serán las que más usuarios usen y las que mayor impacto puedan tener.

Para conseguir esto, haremos uso del plugin de Webpack [prerender-spa-plugin](#). Con este plugin, podemos conseguir no hacer sobreingeniería, pero sí cumplir con ciertos requisitos de SEO y carga inmediata. Puede que en el futuro se nos quede algo corto, pero para primeras fases de un proyecto, nos puede bastar. Esto dependerá de nuestra experiencia y nuestro contexto.

## ¿Cómo puedo empezar?

Como siempre, lo primero que hacemos es instalar la librería que nos proporciona la comunidad de vue:

```
npm install vue vue-server-renderer --save
```

Esta librería está creada para NodeJS. Tendremos que tener en cuenta esto. Si necesitamos que el renderizado se produzca en servidores PHP, por ejemplo, tendremos que buscar otra alternativa para el renderizado.

El funcionamiento de la librería es fácil:

1. Se instancia la aplicación de vue que se quiere renderizar,
2. Se instancia la librería de renderizado,
3. Y se pasa la instancia de vue para que sea renderizada en formato cadena de HTML.

Veamos los pasos con código:

```
// 1: Se crea la instancia de vue
const Vue = require('vue');
const app = new Vue({
  template: '<div>Hello World</div>'
});

// 2. Se instancia la librería de SSR
const renderer = require('vue-server-renderer').createRenderer();

// 3. Se renderiza a cadena
renderer.renderToString(app, (err, html) => {
  if (err) throw err;
  console.log(html);
});
```

Este ejemplo sirve para poco. Solo demuestra cómo funciona la librería. No deja de ser el típico funcionamiento de un motor de plantillas. Tu le ofreces una plantilla y el la renderiza en String HTML según las reglas de vue.

Lo importante ahora es ver cómo podemos satisfacer peticiones de los usuarios con SSR. Vamos a usar [express](#) como librería de infraestructura. Veamos el ejemplo:

```
const Vue = require('vue');
const server = require('express')();
const renderer = require('vue-server-renderer').createRenderer();

server.get('*', (req, res) => {
  const app = new Vue({
    data: { url: req.url },
    template: '<div>The visited URL is: {{ url }}</div>'
  });

  renderer.renderToString(app, (err, html) => {
    if (err) {
      res.status(500).end('Internal Server Error');
      return;
    }

    res.end(`
      <!DOCTYPE html>
      <html lang="en">
        <head>
          <title>Hello</title>
        </head>
        <body>${html}</body>
      </html>
    `);
  });
});

server.listen(8080);
```

Nuestro ejemplo escucha peticiones en el puerto 8080. Cualquier petición que recibamos, ejecuta un manejador que se encarga de renderizar nuestra instancia de vue y enviarlo dentro de un `index.html` típico.

Bastante artesanal, pero no muy diferente de lo que queremos hacer en el mundo real.

Podemos mejorar el ejemplo anterior extrayendo el el HTML genérico a un template.

Podemos guardar el siguiente HTML dentro de un fichero `index.template.html` :

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Hello</title>
  </head>
  <body>
    <!--vue-ssr-outlet-->
  </body>
</html>
```

El comentario `<!--vue-ssr-outlet-->` es interpretado por la librería de rendering para inyectar el resultado final del renderizado. En nuestro código solo tendríamos que poner esto:

```
const renderer = createRenderer({
  template: require('fs').readFileSync('./index.template.html', 'utf-8')
});

renderer.renderToString(app, (err, html) => {
  console.log(html)
});
```

De esta manera, la propia librería se encarga de hacer todo el trabajo. Queda mucho más limpio y nos da mayor flexibilidad que la plantilla base hardcodeada.

## Conclusión

Lo dejamos aquí por ahora. Hemos visto qué es esto del SSR y en que se diferencia con lo que hacen las SPAs convencionales. Hemos visto su utilidad y que inconvenientes tiene. Hemos estudiado que nos proporciona vue para cumplir con un SSR de calidad y por último hemos aprendido a cómo empezar un proyecto con SSR.

La librería tienen tal nivel de configuración que esto es solo la punta del iceberg de todo lo que podremos hacer. Como inicio está bien, pero aprender todo su potencial nos dará una mayor flexibilidad en el futuro.

Recordemos eso sí, que SSR no es apto para todo tipo de proyecto y que decidimos por este modelo tendrá que ser una decisión muy bien meditada. Podríamos por ejemplo, empezar a usar pre-rendering de vistas principales y quizá en el futuro hacer uso SSR.

Nos leemos :)

# Capítulo 18. Configurando Webpack para SSR

Una vez que tenemos claros los conceptos básicos, es hora de trabajar para que nuestro proyecto se adapte a las necesidades del SSR. Empezaremos la adaptación configurando la build.

La forma en la que vamos a empaquetar nuestra aplicación variará debido a las necesidades de ejecutar código de vue en servidor y en cliente de forma indistinta. Necesitamos que el código de nuestra aplicación sea universal y para ello vamos a apoyarnos en Webpack para conseguir este tipo de empaquetados.

Más tarde veremos cómo adaptar el resto del proyecto, pero por ahora centrémonos en este punto: Configurar Webpack a esta nueva situación:

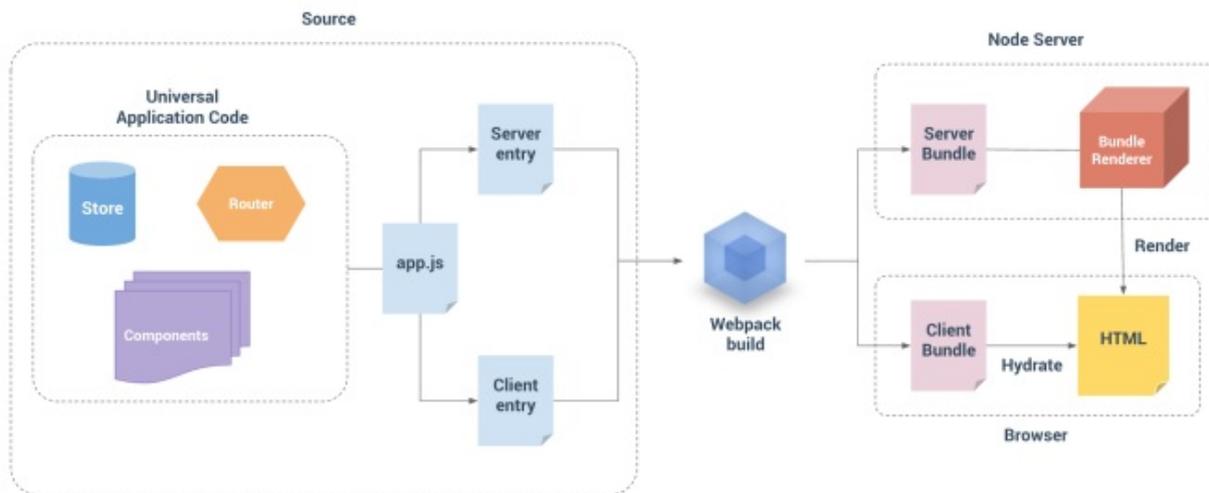
## La estructura de ficheros

Mantener un sistema universal puede ayudarnos a adaptarnos mejor a cambios. Contar con Webpack nos va a dar mucha versatilidad a la hora de poder trabajar en un escenario SSR y/o CSR.

No es una mala idea dejar el sistema adaptado para ambos. Un cambio muy pequeño en la configuración de Webpack y la estructura de ficheros, puede aportarnos mucho.

Además, en muchos escenarios un trabajo mixto suele estar más alineado con la realidad. Esto es, un sistema donde rendericemos vistas en servidor, pero que puedan ser hidratadas con contenido dinámico desde cliente. Por tanto, crear aplicaciones universales es obligatorio si nos acercamos a esta aproximación.

Nos gustaría conseguir un flujo de construcción parecido al de la imagen:



Una app con componentes universales, que cuente con dos entradas para que Webpack nos genere dos paquetes. Estos paquetes serán usados dependiendo de la necesidad.

Estructuremos nuestra aplicación de esta forma:

```

src
├── components
│   ├── Foo.vue
│   ├── Bar.vue
│   └── Baz.vue
├── App.vue
├── app.js # entrada universal
├── entry-client.js # solo se ejecuta en navegador
└── entry-server.js # solo se ejecuta en servidor
    
```

Lo único que varía de la estructura típica de un proyecto de vue son los tres ficheros del final. Veamos qué forma tienen:

```

// app.js
import Vue from 'vue';
import App from './App.vue';

export function createApp () {
  const app = new Vue({
    render: h => h(App)
  });

  return { app };
}
    
```

Nada nuevo. Una factoría que renderiza el componente raíz de la app. Lo bueno es que es universal. Este código - pasado por Webpack - funciona tanto en NodeJS como en navegador.

Fijaos que la instancia de Vue no se pasa directamente al módulo. Se usa una factoría para que se instancie un contexto para cada usuario. De esa manera no contaminaremos las respuestas de otros usuarios y cada uno tendrá una copia de vue propia al renderizar. La clave está en los dos ficheros `entry-`. En ellos se ven las particularidades de cada sistema.

```
// entry-server.js
import { createApp } from './app';

export default context => {
  const { app } = createApp();
  return app
}
```

Si nos encontramos en el servidor, solo necesitamos la instancia a renderizar. Nada más.

```
// entry-client.js
import { createApp } from './app';

const { app } = createApp();
app.$mount('#app');
```

En cambio, si nos encontramos en el cliente y alguien hace uso de esta paquete, se monta la instancia en el DOM.

## Hidratación desde cliente

Un pensamiento que te puede venir a la cabeza con este sistema mixto es el siguiente:

Si ya he renderizado mi vista en servidor y mi `index.html` obtiene el empaquetado de cliente ( `entry-client.js` ), ¿vue intentará crear la instancia de nuevo y volver a montar el componente que ya está renderizado desde servidor?

Lo bueno es que en el equipo de vue ya ha pensado en ello y el sistema no produce un conflicto entre el SSR y el CSR. Cuando renderizamos en servidor, la librería `vue-server-renderer` añade esta etiqueta:

```
<div id="app" data-server-rendered="true">
```

Esta etiqueta provoca, que el paquete de cliente no monte su instancia, ni se vuelva a crear todo el DOM de nuevo. Vue se pone en modo hidratación.

Cuando se ejecuta `app.$mount('#app');`, simplemente se crea la estructura del Virtual DOM en memoria. Hace esto para poder hacer todos los cambios dinámicos que sean precisos por interacción del usuario. De esta manera, evitamos el conflicto: el servidor renderiza vistas y el cliente hace cambios dinámicos.

## La build

Lo siguiente es cambiar los ficheros de configuración para que acepte estas dos entradas y cumpla con el dibujo que hemos puesto anteriormente.

Para simplificar el ejemplo, vamos a separar la build en tres ficheros: el fichero base, que contiene toda la configuración común del proyecto, la configuración cliente, que contiene toda la configuración para el paquete de cliente, y la configuración de servidor que contiene toda la configuración para el paquete de renderizado en servidor.

De la configuración base, nos vamos a olvidar, pues en nuestro caso solo indica donde se encontrarán las salidas. Como suele ser normal, guardaremos todos los resultados en `/dist`. También incluiremos loaders comunes con el vue-loader y el babel-loader.

Para la parte de cliente, contaremos con una configuración como esta:

```
// webpack.client.config.js
const webpack = require('webpack');
const merge = require('webpack-merge');
const baseConfig = require('./webpack.base.config.js');
const VueSSRClientPlugin = require('vue-server-renderer/client-plugin');

module.exports = merge(baseConfig, {
  entry: './src/entry-client.js',
  plugins: [
    new webpack.optimize.CommonsChunkPlugin({
      name: "manifest",
      minChunks: Infinity
    }),
    new VueSSRClientPlugin()
  ]
});
```

Ponemos como entrada el fichero `entry-client` que hemos creado y que se encarga de montar la aplicación en el navegador. Después, añadimos dos plugins:

El primer plugin se encarga de extraer el core de Webpack llamado manifest se encarga de las tareas de inyección de descubrimiento de dependencias. De esta manera, como iremos dividiendo nuestra aplicación en módulos que se cargarán de manera dinámica, solo se incluirá una vez este motor en lo más alto de la cadena de dependencias.

El segundo plugin es el propio de la librería `vue-server-renderer` y se encarga de generar un fichero en formato JSON llamado `vue-ssr-client-manifest.json` que contiene metainformación del paquete de cliente y que nos será útil a la hora de servir peticiones por medio de la funcionalidad Bundle Renderer que explicaremos en un momento.

Con esto tendríamos el paquete de cliente.

La configuración de servidor es la siguiente:

```
const merge = require('webpack-merge');
const nodeExternals = require('webpack-node-externals');
const baseConfig = require('./webpack.base.config.js');
const VueSSRServerPlugin = require('vue-server-renderer/server-plugin');

module.exports = merge(baseConfig, {
  entry: './src/entry-server.js',
  target: 'node',
  devtool: 'source-map',
  output: { libraryTarget: 'commonjs2' },
  externals: nodeExternals({
    whitelist: /\.css$/
  }),
  plugins: [ new VueSSRServerPlugin() ]
});
```

En esta configuración tenemos más cosas. Lo primero, lo de siempre, indicar el fichero de entrada que creamos específicamente. Seguidamente, indicamos que el contexto de ejecución será para NodeJS. Esto hace que los loaders se optimicen y orienten para un funcionamiento en servidor.

El atributo `devtool` lo indicamos con `source-map`. De esta manera, se nos generará un sourcemap del paquete de servidor. Indicamos también que el sistema de módulos a usarse debería ser CommonJS para que sea utilizable desde NodeJS. El atributo `externals` nos ayuda a quitar todo aquello superfluo para el empaquetado de servidor. Por ejemplo, el CSS no será necesario procesarlo en servidor, por tanto, evitamos su procesamiento y aligeramos los tiempos.

El último atributo es usar el plugin de `vue-server-renderer` pero para la parte de servidor. Lo que nos crea es un JSON llamado `vue-ssr-server-bundle.json` con la metainformación del paquete que estamos creando, pero para procesos del renderizado en servidor. Este fichero será interpretado por la funcionalidad Bundle Renderer que explicamos a continuación.

## El Bundle Renderer

Como hemos visto en el apartado anterior, Webpack y los plugins de `vue-renderer-server` nos van a añadir en `/dist` dos JSON que contienen metainformación de los paquetes generados para cliente y servidor. Es una forma de desacoplar los paquetes creados y su uso en cliente y servidor. Esto tiene su sentido:

Hay algo molesto al desarrollar para NodeJS y es que cada vez que se realiza un cambio, tenemos que parar el servidor y volver a lanzarlo. Existen procesos para que no sea así cuando trabajamos para desarrollos de NodeJS específicos, pero cuando estamos desarrollando una aplicación vue, el proceso tiene algo de fricción.

La gente de vue ya ha pensado en ello y ha creado una funcionalidad denominada Bundle Renderer. Esta funcionalidad se encarga de escuchar en los JSON que hemos creado con Webpack y renderizar a partir de los cambios, sin tener que parar e iniciar servidor.

De esta manera, podemos desarrollar nuestra webapp, ejecutar webpack y generar los nuevos JSON. El servidor encargado del renderizado ya sabrá que tiene que renderizar con el nuevo paquete generado porque los cambios han sido reflejados en el JSON.

Lo mismo ocurre para el paquete de cliente. La funcionalidad tiene sistema de carga en caliente (HRM) por lo que si existe un cambio, el paquete de cliente sabe cambiarlo en tiempo de ejecución.

Para usar esta funcionalidad, deberemos cambiar nuestro servidor a algo parecido a esto:

```
// ./index.js
const express = require('express');
const server = express();
const template = require('fs').readFileSync(__dirname + '/index.html', 'utf-8');
const serverBundle = require('./dist/vue-ssr-server-bundle.json');
const clientManifest = require('./dist/vue-ssr-client-manifest.json');
const { createBundleRenderer } = require('vue-server-renderer');

const renderer = createBundleRenderer(serverBundle, {
  template,
  clientManifest
});

server.get('*', (req, res) => {
  const context = { url: req.url }

  renderer.renderToString(context, (err, html) => {
    // se quita el manejo de error para simplificar...
    res.end(html)
  });
});
```

Lo único que cambia es cómo generamos el renderer. Tenemos que indicarle que lo genere a partir de los dos JSON.

## Conclusión

Con esto, tenemos un buen porcentaje de nuestro proyecto adaptado. Como veis, el proceso de renderizado en servidor, supone una fe ciega hacia un sistema poco intuitivo y que hace cosas como por arte de magia.

Un sistema como estos nos está suponiendo un nivel de fontanería elevado. No hemos desarrollado nada de nuestro negocio y todo lo que hacemos es configurar y configurar para que todo vaya de manera óptima. Las opciones de vue-server-renderer + webpack pueden llegar a abrumar, así que, si no nos queda más remedio que hacer uso de un sistema como este, tendremos que seguir practicando y estudiando para asimilar los conceptos.

En el próximo post, nos centraremos en adaptar el router y el store de la aplicación.

Nos leemos :)

## Capítulo 19. Adaptando tu proyecto a SSR

Con lo visto en el post anterior, tenemos una primera aproximación de cómo dejar configurado nuestro primer proyecto con SSR. Además de esto, necesitamos adaptar ciertas partes de nuestro código para que todo funcione como debe.

Tenemos que pensar que la forma de trabajar en cliente y servidor no es la misma y conseguir código universal necesita de una serie de adaptaciones.

En el post de hoy hablaremos de cómo y por qué realizar estas adaptaciones. Aunque existirán seguramente más partes que adaptar, nos vamos a centrar en los cores del framewrok: la instancia de vue, los routers y los stores.

### Adaptando la instancia de vue

Monohilo. Recuérdalo. NodeJS es monohilo y esto en muchos casos, nos puede traer problemas. Cuando envías una aplicación vue a un navegador, la instancia que creas es única por usuario.

Como cada usuario tiene su propia copia de vue y de sus datos, y los refrescos continuos de navegador van a hacer que se renueve según la necesidad, no tenemos problemas. Podríamos decir que la arquitectura cliente-servidor con webapp SPA se comporta como una aplicación multihilo, ya que la gestión de memoria se hace en el cliente y no en el servidor.

Ahora bien, si estamos renderizando la instancia principal de vue en servidor, podemos llegar a tener problemas de compartición de datos entre usuarios por crear una única instancia del objeto. Se puede crear una contaminación de contexto que deberíamos evitar.

Lo normal es que el código del post anterior (ese get enorme que creaba instancias, renderizaba y servía peticiones), lo refactorices y que la instanciación de la aplicación vue la lleves a un módulo separado. De esta manera, el módulo sería universal.

Pero ten en cuenta que cuando NodeJS usa un módulo, lo cachea, haciendo que su funcionamiento se asemeje al de una especie de Singleton. Así que te recomiendo que uses una factoría que instancie el objeto raíz de tu app vue.

Podría ser de esta forma:

```
// app.js
const Vue = require('vue');

module.exports = function createApp(context) {
  return new Vue({
    data: { url: context.url },
    template: `<div>The visited URL is: {{ url }}</div>`
  });
}
```

```
// server.js
const express = require('express');
const server = express();
const createApp = require('./app');

server.get('*', (req, res) => {
  const context = { url: req.url };
  const app = createApp(context);

  render.renderToString(app, (err, html) => {
    // handle error...
    res.end(html);
  });
});
```

Con esto, creamos una copia de la aplicación diferente para cada uno de los usuarios que quieren utilizar la aplicación. Este será un patrón que usaremos posteriormente en la instanciación de routers y stores.

## Adaptando el router

Si miramos el manejador de express que escribimos, encontramos que dejamos un get con un asterisco (\*). Esto significa que cualquier ruta que enviemos a nuestro servidor, será manejado por este código.

Por tanto, no es mala idea pensar en utilizar nuestro router (vue-router, lógicamente) en la parte servidor para que el sistema también se encargue de renderizar el resto de vistas de nuestra aplicación. Para conseguir esto, tendremos que hacer 3 cambios en nuestro proyecto:

1. La instancia del router ( `./src/router/index.js` ) tendremos que devolverla con una función factoría.
2. La entrada de servidor ( `entry-server.js` ) tendrá una pequeña gestión para la posible carga perezosa de componentes vista.

3. En el manejador de la parte servidor ( `server.js` ) gestionaremos lo que nos devuelva `entry-server.js` .

Escribamos el código. Primero empecemos con la configuración del router:

```
// ./router
import Vue from 'vue';
import Router from 'vue-router';

Vue.use(Router);

export function createRouter () {
  return new Router({
    mode: 'history',
    routes: [ // ... ]
  });
}
```

Poco nuevo en este código. Se devuelve la instancia en una función factoría y ponemos como modo de enrutado `history` para que se envíen las peticiones a servidor y no sean gestionadas por la parte cliente.

Ahora configuramos el `app.js` :

```
// app.js
import Vue from 'vue';
import App from './App.vue';
import { createRouter } from './router';

export function createApp () {
  const router = createRouter()
  const app = new Vue({
    router,
    render: h => h(App)
  });

  return { app, router };
}
```

Instanciamos el router y lo incluimos en la instancia de vue. Devolvemos tanto la app como el router porque nos será útil en siguientes pasos.

Detengámonos en `entry-server.js` que tiene más complejidad. Lo comentamos en el código:

```
// entry-server.js
import { createApp } from './app';

export default context => {
  // Devolvemos una promesa porque puede que nuestras
  // rutas se gestionen de manera asíncrona, así el servidor
  // esperará
  return new Promise((resolve, reject) => {
    const { app, router } = createApp();

    // Registramos la url que nos viene desde cliente
    // para que vue-router decida más tarde si tiene alguna
    // ruta registrada
    router.push(context.url)

    // Nos quedamos esperando hasta que las rutas se encuentran
    // listas y cargadas
    router.onReady(() => {
      // Vemos si hay una coincidencia con la ruta enviada
      const matchedComponents = router.getMatchedComponents();

      // Si no existe, devolvemos un 404
      if (!matchedComponents.length) {
        return reject({ code: 404 })
      }

      // Si existe, indicamos la instancia de vue
      resolve(app);
    }, reject);
  });
}
```

Lo último que queda es gestionar en `server.js` la decisión de si se ha encontrado una ruta o no. Analizamos en los comentarios de nuevo:

```
// server.js
const express = require('express');
const server = express();
const createApp = require('/path/to/built-server-bundle.js');

// Escuchamos en todas las rutas (*)
server.get('*', (req, res) => {
  // Obtenemos la url de la petición
  const context = { url: req.url };

  // Cuando la promesa sea resuelta
  createApp(context).then(app => {
    // Renderizamos esa ruta
    renderer.renderToString(app, (err, html) => {
      if (err) {
        if (err.code === 404) {
          res.status(404).end('Page not found');
        } else {
          res.status(500).end('Internal Server Error');
        }
      } else {
        // Devolvemos la vista renderizada
        res.end(html);
      }
    });
  });
});
```

## Adaptando el store

Si lo pensamos detenidamente, SSR no es otra cosa que la renderización de un 'snapshot' específico del sistema en servidor. Esto significa que si el usuario necesita renderizar datos del usuario, tengamos que tener algún mecanismo para cargarlos. Ya no nos vale solo el sistema de: renderizo en cliente y hago una obtención de datos.

necesitamos algo más sofisticado que nos permita alternar este sistema en cliente, con un sistema de precarga de datos. Lo primero que tenemos que pensar es en obligarnos a incluir vuex. No podemos cargar datos en componentes, pues nos dificulta mucho el trabajo de hacer cargas y renderizados.

Lo ideal es hacer una precarga de datos en el componente vista. Ese componente que se instancia cuando existe una ruta relacionada. Por tanto, vamos a modificar nuestro fichero para que acepten la precarga de datos antes del renderizado.

Lo primero que hacemos es convertir nuestro store en una factoría de la siguiente manera:

```
// ./store/index.js
import Vue from 'vue';
import Vuex from 'vuex';
Vue.use(Vuex);

// Como no nos importa la forma en la que obtenemos
// los datos asumimos que tenemos una librería universal
// como axios por ejemplo
import { fetchItem } from './api';

export function createStore () {
  return new Vuex.Store({
    state: {
      items: {}
    },
    actions: {
      fetchItem ({ commit }, id) {
        return fetchItem(id).then(item => {
          commit('setItem', { id, item });
        });
      }
    },
    mutations: {
      setItem (state, { id, item }) {
        Vue.set(state.items, id, item);
      }
    }
  });
}
```

Lo siguiente es adaptar el fichero `app.js` :

```
// app.js
import Vue from 'vue';
import App from './App.vue';

import { createRouter } from './router';
import { createStore } from './store';
import { sync } from 'vuex-router-sync';

export function createApp () {
  const router = createRouter();
  const store = createStore();

  // sincronizamos router y store comunes
  sync(store, router);

  const app = new Vue({
    router, store,
    render: h => h(App)
  });

  // Exponemos todo para su uso posterior
  return { app, router, store };
}
```

Ahora, tenemos que ver en qué componentes de vista se necesita cargar datos antes del renderizado. Para conseguir esto, añadimos un nuevo método al componente llamado `asyncData`.

Recuerda que este método será estático. No podrás hacer uso de propiedad de la instancia (todo aquello que este en `this`), ya que es un método que será ejecutado antes de la instanciación del componente y del renderizado.

Un ejemplo de componente con `asyncData` sería este:

```
<!-- Item.vue -->
<template>
  <div>{{ item.title }}</div>
</template>

<script>
export default {
  asyncData ({ store, route }) {
    // devolvemos la promesa para gestionarla
    // en servidor
    return store.dispatch('fetchItem', route.params.id)
  },
  computed: {
    item () {
      return this.$store.state.items[this.$route.params.id];
    }
  }
}
</script>
```

Una vez que tenemos esto preparado, tenemos que cambiar las dos posibilidades que tenemos:

## Precarga en servidor

La precarga en servidor, se tiene que hacer antes del renderizado, pero después de que la ruta haya sido relacionada, por tanto, el mejor sitio para hacerlo es en `entry-server.js` que es donde estamos haciendo estos apañíos:

```

// entry-server.js
import { createApp } from './app';

export default context => {
  return new Promise((resolve, reject) => {
    const { app, router, store } = createApp();

    router.push(context.url);
    router.onReady(() => {
      const matchedComponents = router.getMatchedComponents();

      if (!matchedComponents.length) {
        return reject({ code: 404 });
      }

      // Ejecutamos los asyncData de todos los componentes
      // vista relacionados
      Promise.all(matchedComponents.map(Component => {
        if (Component.asyncData) {
          return Component.asyncData({ store, route: router.currentRoute });
        }
      })).then(() => {
        // LA CLAVE ESTA AQUÍ, LO EXPLICAMOS MÁS ABAJO
        context.state = store.state;
        resolve(app);
      }).catch(reject);
    }, reject)
  })
}

```

Lo que hacemos es recorrer todos los componentes vista relacionados y ejecutar `asyncData` si existe en el componente.

Guardamos los estados obtenidos en el contexto. De esta forma, `vue-server-renderer` sabrá obtener los datos y automáticamente serializará los datos en el HTML para que sean accedidos por el cliente antes del montaje y no se produzca una desincronización entre DOM real y virtual.

Esta serialización se guarda en `window.__INITIAL_STATE__` por lo que tenemos que tocar un poco el fichero `entry-client.js` para que todo funcione como debe:

```

// entry-client.js
const { app, router, store } = createApp();

if (window.__INITIAL_STATE__) {
  store.replaceState(window.__INITIAL_STATE__);
}

```

## Precarga en cliente

Ahora bien, puede que en ciertos escenarios, la carga de datos tenga que darse en cliente. En este contexto tenemos dos aproximaciones:

Que queramos cargar los datos antes del renderizado de la vista. Con lo que haríamos esto:

```
// entry-client.js

// ...

router.onReady(() => {
  router.beforeResolve((to, from, next) => {
    const matched = router.getMatchedComponents(to);
    const prevMatched = router.getMatchedComponents(from);

    // Hacemos lo siguiente para comprobar si ya se ha hecho
    // renderizado en servidor. De esa manera evitamos una
    // doble precarga
    let diffed = false;
    const activated = matched.filter((c, i) => {
      return diffed || (diffed = (prevMatched[i] !== c));
    });

    if (!activated.length) {
      return next();
    }

    Promise.all(activated.map(c => {
      if (c.asyncData) {
        return c.asyncData({ store, route: to });
      }
    })).then(() => {
      next();
    }).catch(next);
  });

  app.$mount('#app');
});
```

O que la carga de los datos se realice después del renderizado y lo que tengamos que hacer (como posible solución) sea registrar un mixin global que lo gestione:

```
Vue.mixin({
  beforeMount () {
    const { asyncData } = this.$options;

    if (asyncData) {
      this.dataPromise = asyncData({
        store: this.$store,
        route: this.$route
      });
    }
  }
});
```

Muchos aspectos cubiertos, para según el escenario que necesitemos.

## Conclusión

Puede parecer que incluir SSR en nuestro proyecto se basa en incluir una fontanería que lo que hace es irnos más del foco de solución y del dominio. Yo opino igual. Sin embargo, si decidimos hacer esto por nuestra cuenta o si ya contábamos con un proyecto en CSR que queremos migrar a SSR no tendremos más solución que arremangarnos y empezar a adaptar poco a poco.

Como iremos viendo, existen alternativas un poco más rápidas si empezamos un proyecto con SSR desde cero. Esto lo veremos en el post dedicado a nuxt que veremos a continuación. Por ahora, será mejor que asimilemos estos cambios y que sigamos valorando si el esfuerzo nos va a compensar.

Nos leemos :)

## Capítulo 20. Aplicaciones universales con Nuxt

Si has llegado hasta aquí, ¡Enhorabuena! Estamos al final de camino. Hemos aprendido muchas cosas sobre Vue en general y sobre SSR en particular durante estas últimas semanas.

La sensación que nos dejaba `vue-server-renderer` era la de ser una librería con la que tener que lidiar con demasiada configuración engorrosa, que no nos ayudaba a centrarnos en lo importante: desarrollar aplicaciones.

Como os he ido prometiéndolo, esto, a día de hoy, tiene solución: el proyecto Nuxt ha nacido para encargarse de toda esa fontanería y ayudarnos en que focalicemos y aportemos valor lo antes posible. Por todo lo probado y leído, Nuxt me parece un proyecto con mucho futuro, por tanto creo que este post será un buen punto y final a la serie de VueJS.

Terminemos lo empezado:

### ¿Qué es?

Es curioso, pero si tengo que explicar qué es Nuxt, lo describiría como un framework de un framework. La idea detrás de Nuxt es coger toda la potencia que tienen VueJS, Webpack y SSR, y permitir crear aplicaciones universales de una forma fácil, rápida y flexible, con menos configuración.

Si recordamos, las aplicaciones universales son aquellas que contienen módulos de código capaces de ser ejecutados tanto en un navegador como en un servidor. Por tanto, con Nuxt, vamos a poder tener aplicaciones que cumplan con los requisitos que nos habíamos autoimpuesto de SEO y de servir contenido al usuario lo antes posible, sin tener que renunciar a todo el dinamismo en cliente de las SPAs convencionales.

Si has seguido la serie en sus últimos capítulos, comprenderás que Nuxt no resuelve nada nuevo de lo que ya hemos explicado. Sin embargo, lo bueno de este framework es que ya hace todo el trabajo engorroso de configuración y construcción por nosotros.

La idea de Nuxt es permitirnos generar un template de un proyecto base, como hacíamos con `vue-cli`, pero con una estructura específica para trabajar con todos los elementos de vue - el router, los stores, los componentes... - de una forma más uniforme, más intuitiva y con un sistema basado en convenciones y no tanto en configuraciones.

Por ahora, el proyecto se encuentra en versión alpha, pero las expectativas de la comunidad vue puesta en él son altísimas.

## ¿Qué funcionalidades tiene?

Puede que hasta ahora Nuxt nos pueda parecer confuso, pero las funcionalidades que promete son como para darle una oportunidad. Entre ellas encontramos lo siguiente:

- **Escritura de ficheros vue:** Nada innovador. Una aplicación desarrollada en Nuxt, es una aplicación desarrollada en Vue. Por tanto, el mecanismo de desarrollar nuestros componentes en ficheros de tipo Vue sigue igual. Puede que deseemos migrar un proyecto anterior de Vue a la estructura de Nuxt y la migración será asequible. Al final Nuxt, lo único que sabe es interpretar componentes Vue de una forma más limpia.
- **Separación de código en paquetes de forma automática:** Hemos aprendido a lo largo de la serie cómo usar Webpack y Vue para generar paquetes más pequeños y que se carguen bajo demanda, para disminuir el paquete principal y hacer nuestra aplicación más rápida. En Nuxt no tendremos que hacer nada para activar este sistema. Las vistas ya contienen esta separación en módulos dinámicos y bajo demanda.
- **Renderizado en la parte de servidor:** Sin configuraciones, sin tener que bajar al barro. Todas las vistas de tu aplicación son renderizadas en servidor. Todos los cambios dinámicos son interceptados por Vue en cliente. Nuxt nos hace el SSR transparente.
- **Sistema de rutas y sincronismo de datos avanzado:** Olvídate de configurar vue-router. Nuxt sabe exactamente que rutas generar dependiendo de cómo estructuras tus vistas dentro del proyecto. Además, existe funcionalidad extra para la sincronización de datos y componentes.
- **Servir ficheros estáticos:** Tu propio proyecto va a ser capaz de servir estáticos. Ya sea de vistas, imágenes o fuentes. Todo con un servidor integrado por defecto.
- **Transpilación de ES6/ES7, preprocesamiento de SASS/LESS/Stylus y empaquetado/minificado de JS y CSS:** Todo tu Webpack configurado para que no tengamos que preocuparnos de ello, solo de escribir código de negocio.
- **Carga en caliente en desarrollo:** Nuxt configura nuestro proyecto para que en desarrollo se acepte la carga caliente de cambios, con lo que supone en tiempos a la hora de desarrollar.

- **Generación de vistas a formato estático:** la funcionalidad más importante de Nuxt es esta. Poder generar todo un proyecto Vue de manera estática. Renderizar todo en ficheros HTML para que cualquier CDN pueda hospedarlo y sea servido de una forma rápida y optimizada al máximo.

## ¿Cómo empezar?

Nada nuevo: instala NodeJS en tu equipo y ten disponible vue-cli:

```
$ npm install -g vue
```

Nuxt se instala como un nuevo template de Vue de la siguiente manera:

```
$ vue init nuxt/starter <project-name>
```

Nuxt tiene varios templates propios dependiendo de lo que necesites y tu contexto, por lo que te recomiendo que los analices antes de empezar un proyecto. Nosotros con el proyecto base tenemos suficiente.

Esto nos generará una serie de carpetas y ficheros que nos permitirán trabajar en Nuxt. Lo siguientes en hacer es obvio:

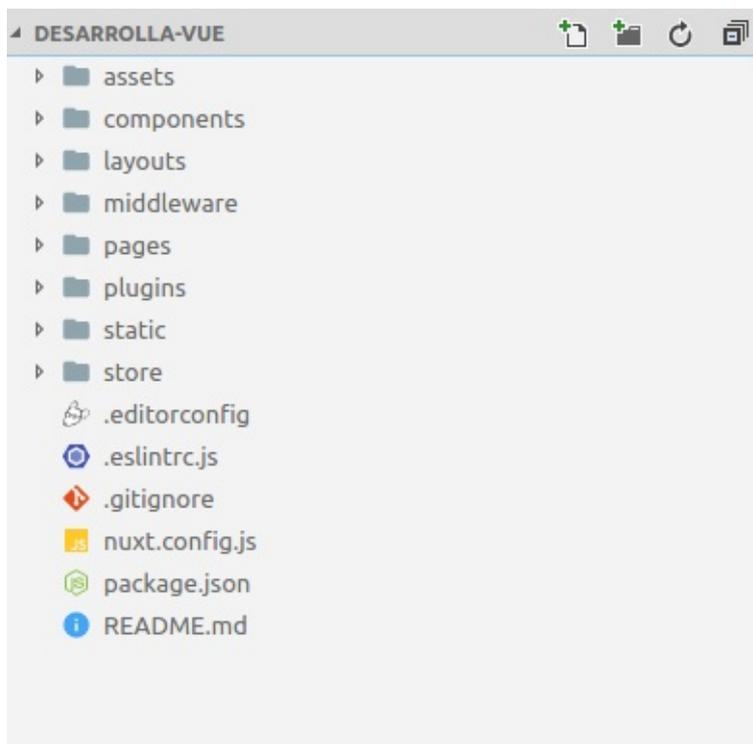
```
$ cd <project-name>
$ npm install
```

Nos descargará todas las dependencias de nuestro proyecto, entre ellas la nueva utilidad de terminal llamada nuxt.

Si queremos ejecutar el proyecto de ejemplo, lanzamos `npm run dev` y listo.

## ¿Cuál es la estructura de un proyecto Nuxt?

De lo que se nos ha generado podemos aprender mucho. Nuxt me parece tan intuitivo que sin mucho estudio, sabremos qué debemos guardar en cada carpeta. El proyecto será parecido a este:



Explicaremos cada elemento y qué deberemos guardar dentro:

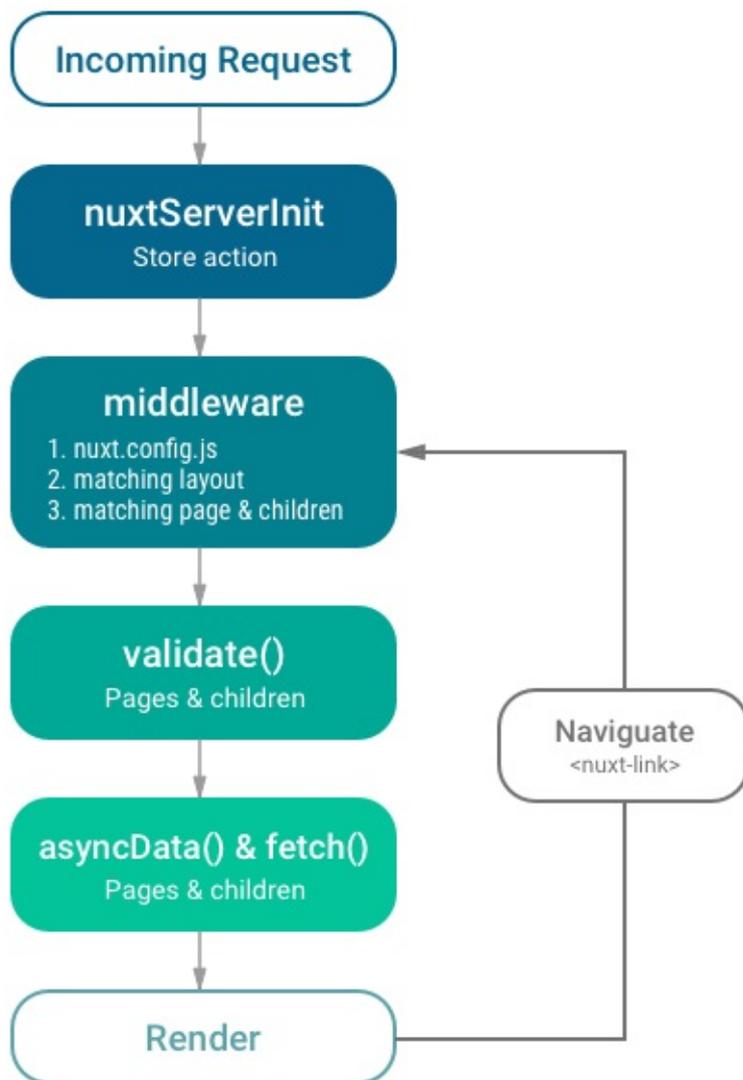
- **assets**: guardaremos todos aquellos ficheros estáticos de los cuales nuestra aplicación depende: imágenes, fuentes, css principales, librerías legadas que no siguen el sistema actual, tienen cabida aquí.
- **components**: todos aquellos componentes visuales, muy core, muy genéricos que pueden ser reutilizados se guardarán en components.
- **layouts**: Componentes de tipo layout que conforman nuestra aplicación. Por lo general, solo se contará con una layout, pero el sistema permite mucha versatilidad en cuanto a layouts y sublayouts se trata.
- **middleware**: son funciones o pequeñas librerías que deben ejecutarse justamente antes del renderizado de páginas (o grupo de páginas) en servidor.
- **pages**: compuesto por todos aquellos componentes de tipo página. Todas las pantallas de tu aplicación se encontrarán aquí. La forma en cómo organizamos esta carpeta, supondrá la forma en la que se configurará el vue-router, por tanto hay que mimar mucho su organización.
- **plugins**: contiene toda aquella funcionalidad que extiende el funcionamiento de Vue. Por ejemplo, podemos añadir plugins de filtros, o plugins para la internacionalización de la aplicación (i18n).
- **statics**: parecido a assets. Con la diferencia de que aquí se almacenan estáticos que no sean dependencia directa de nuestros componentes o vistas. Los ficheros que se suelen guardar aquí son: el favicon, el robot.txt o el sitemap.xml.

- **store:** todos nuestros módulos de stores se guardan aquí para que Nuxt los ponga a disposición de los componentes.
- **nuxt.config.js:** contiene configuraciones de nuestra aplicación y del funcionamiento de Nuxt. No todos los proyectos son iguales y puede que ciertos procesos tengan que ser diferentes. Puede que el Webpack que usa Nuxt no sea todo lo que necesitamos y que tengamos que extenderlo. Este fichero nos da la versatilidad para que podamos hacerlo.

Como vemos, solo hay un fichero de configuración en toda la aplicación, no hay nada de builds, ni webpacs. No hay routers, no hay componente inicial, ni puntos de entrada diferentes para el paquete de servidor o de cliente. Todo es automático. Es la gran ventaja de Nuxt.

## ¿Cómo funciona?

Una aplicación creada con Nuxt tiene un ciclo de funcionamiento muy específico. Es el siguiente:



1. Un usuario realiza una petición de una ruta determinada a servidor.
2. El servidor ejecuta la acción `nuxtServerInit` del store principal si la tiene implementada. Esta acción nos permite cargar datos iniciales (prefetching de datos globales).
3. Se ejecutan todos aquellos middlewares que se encuentren en el fichero de configuración `nuxt.config.js` y los relacionados con el layout, la página raíz y las páginas hijas coincidentes que se hayan implementado.
4. Si existe un validador, se ejecuta. Si se resuelve con un `true` se sigue el proceso, si no se devuelve un 404.
5. Se obtienen aquellos datos de la página para que sean renderizados.
6. Se renderiza en servidor y se sirve al usuario.
7. Si el usuario navega por la aplicación hacia otra ruta, se repite el ciclo.

## Conclusión

No vamos a profundizar más en Nuxt por ahora. Lo dejamos aquí ya que hay mucho que asimilar y mucho de lo que profundizar en el ecosistema Vue. El proceso para llegar aquí será largo para el lector que decida practicar con Vue, pero creo que la recompensa merecerá la pena.

No recomiendo empezar la casa por el tejado, esto es, empezar a desarrollar en Nuxt sin saber bien sobre Vue, Webpack o SSR. Los principios, como en todo, serían fáciles y amigables, pero Nuxt sigue ocultando mucho trabajo internamente, muchos procesos, muchos conceptos que si no se han asimilado anteriormente, pueden explotarnos en la cara en momento en los que Nuxt no nos esté funcionando como debiera.

Si te ves con fuerzas para seguir, Nuxt no aporta muchos más conceptos o conocimientos sobre Vue. Solo es una herramienta para ser más productivos con Vue. Aprender los entresijos de Nuxt nos va a suponer un par de días de estudio y práctica y quizá lo único en lo que pensaremos es en cómo no lo habíamos descubierto antes.

Nos leemos :)

## Capítulo 21. Introducción

Buenas! Os tenía muy abandonados y abandonadas! Perdonadme, entre una cosas y otras, no he sacado ni el tiempo ni las ganas suficientes para escribir algo que me apeteciera de verdad por aquí. Vengo con las pilas cargadas y nuevos temas de los que hablar.

Os había comentado en post anteriores que nos íbamos a alejar una temporada de VueJS, pero me está siendo difícil... Como sabréis la mayoría, la serie de artículos que escribí aquí en El Abismo sobre VueJS se hizo bastante viral y eso ha hecho que mucha gente preguntase sobre qué pasó con ese gran olvidado de siempre: El testing.

Tengo que contaros que llegué un poco flojo de fuerzas hacia el final de la serie y sacrifiqué la parte de cómo testear una aplicación VueJS. Estaba en la planificación, lo prometo, pero se quedó por el camino.

Como bien me comentó [Alberto Moratilla](#) (principal precursor y cabeza pensante del manual de GitBook), no incluir esta parte en la serie da una sensación 'mala'. Olvidándonos del testing, estamos dando a entender que la calidad del código que estamos desarrollando no nos es importante. Sin embargo, sabemos que esto no es así; testear y probar nuestro código de una manera automática es la clave para que nuestra aplicación resista mejor al tiempo y a los cambios.

Por otra parte, he ido posponiendo la escritura de esta parte porque la comunidad ya cuenta con dos recursos buenísimos sobre testing en VueJS. Estos son, el libro '[Testing VueJS components with Jest](#)' de [Alex Jover](#) y el Curso de Codely '[Testing con VueJS y Jest](#)' de [Javi Rubio](#) y [Alberto Gualis](#) que puede que sean bastante mejores y amplios que esta serie por el material y la experiencia que tienen los tres en esta materia.

Pero bueno, como las cosas hay que terminarlas y hay que intentar terminarlas bien y con una cobertura aceptable, empecemos, con muchas ganas e ilusión, a hablar de testing y VueJS:

### ¿Por qué necesito testear?

Un test es una serie de pasos que hay que seguir en una aplicación para que se de un determinado resultado esperado. Hay muchos tipos de test según lo que queramos probar ya sea de manera manual (una persona se encarga de ejecutar los pasos) o automática (el propio ordenador es programado para ejecutar estos pasos).

Nosotros en todo momento nos centraremos en explicar mecanismos automáticos. De esta manera, podremos agilizar procesos y evitar realizar tareas tediosas que un ordenador puede hacer mejor que nosotros.

El tipo de test en el que nos centraremos es el denominado test unitario. Los test unitarios son los tests encargados de probar, de manera aislada, cada una de las piezas y sus posibles configuraciones de las que está compuesta una unidad de código. Por unidad de código podemos pensar en un componente, una clase, una función o un procedimiento. Dependerá de la prueba que queramos llevar a cabo.

Los tests nos ayudan en muchas cosas importantes. Las que a mi más me gustan y me aportan son estas:

- **Nos dan feedback:** de una manera muy rápida de cuál es el estado de mi código ante un cambio que yo haya podido realizar. Es decir, que si yo tengo una batería de pruebas incluida en mi aplicación, puedo saber en todo momento si he roto algo o no con cada uno de mis cambios inmediatos.
- **Nos alertan de posibles bugs.** Son una buena red sobre la que hacer equilibrismo. Pensemos. Estamos en JavaScript, un lenguaje débilmente tipado y dinámico. Es un lenguaje muy propicio para cometer errores. Qué bueno sería tener un sistema automatizado que se encargue de comprobar que dadas unas entradas de configuración de mis unidades lógicas, siempre se reciban las mismas salidas como respuesta.
- **Nos ayudan a escribir mejor código.** Escribir tests es algo complejo que necesita cierta disciplina y orden. El cómo nombremos las cosas, el cómo modularicemos, dividamos y aislemos es clave para testear y conseguir piezas reutilizables. El testing nos ayuda a evitar acoplamientos y rápido nos señala si estamos cometiendo alguna mala decisión de diseño.
- **Nos generan una documentación viva** de lo que puede y de lo que no puede hacer nuestro código. Cuando un desarrollador o desarrolladora entra nuevo en un equipo de front, lo primero que suele hacer es mirar el `package.json` para saber con qué dependencias va a trabajar y mirar la batería de test para empezar a asimilar los diferentes caminos y bifurcaciones por las que el código posiblemente le pueda llevar.

Hay más razones, pero creo que son suficientes cómo para que tomemos las técnicas de pruebas automáticas como una tarea importante. Así que, veamos que necesitamos para probar nuestra aplicación de manera eficiente.

## ¿Qué necesito para testear?

Estamos en un ecosistema bastante peculiar. Hablamos de front, un front suele necesitar un navegador para renderizar elementos. En este caso, elementos Web. Además, nos hará falta alguna herramienta que esté pendiente de los cambios y de cuando ejecutar los test.

Nos vendría bien algún tipo de framework que nos facilite el trabajo a la hora de probar cosas... y no solo eso. Trabajamos con VueJs, con ES6, con Webpack... necesitamos herramientas que se integren bien en nuestro flujo de trabajo y que sepan trabajar juntas.

Así que bueno... empecemos a ver qué necesitamos en realidad:

## Necesitamos un test runner

Un test runner es una herramienta encargada de ejecutar una batería de pruebas de manera automática y según una configuración previa. Los test runner suelen encargarse de orquestar todas las piezas para que podamos ejecutar test en un entorno aislado.

Hay muchos. Los más comunes Karma y Jest. En nuestro caso en particular, tenemos que elegir un test runner que cumpla una serie de funcionalidades mínimas.

1. Lo primero es **que sepa trabajar con JSDOM**. JSDOM es una forma de virtualizar el mecanismo del DOM de los navegadores pero sin los navegadores. JSDOM es una librería de NodeJS que implementa el estándar [WHATWG DOM](#). De esta manera, si quiero ejecutar tests en un sistema de integración continua, no necesito tener instalados navegadores como Chrome o Firefox para probar partes muy acopladas al API DOM gracias a esta herramienta.
2. Lo segundo es **que se lleve bien con los Single File Component**. Los SFC son los ficheros `.vue` donde se encuentra todo lo necesario de un componente Vue (todo el HTML, CSS y JS) para que funcione. Webpack y vue-loader son los encargados de romper estos ficheros y dividirlos en sus unidades de responsabilidad y los tests tiene que ser capaces. Por tanto, nos tiene que permitir esto.
3. Luego están las funcionalidades de performance. Pensemos que nuestro proyecto puede tener una batería de pruebas bastante grande y que necesitaremos que los test se ejecuten periódicamente y que me den feedback en cada cambio que realice, por lo tanto, necesitamos algo que compile, renderice y ejecute test de la manera más rápida posible.

De todos los test runners que hay en el mercado, **Jest es el que más o menos cumple con todas ellas**. Jest es el test runner + test framework (ya veremos luego esto) creado por Facebook, sobre todo pensado para probar aplicaciones React, pero que es tan agnóstico que otros ecosistemas como VueJS lo están incluyendo en sus flujos de trabajo.

Jest se caracteriza por ser un test runner rápido, con una CLI muy intuitiva y clara que permite filtrar y ordenar la ejecución de test según nuestras necesidades, así como la de cachear y estar atento a cuándo y cómo se ha cambiado un tests o el código que se prueba. Además, es un test runner que está configurado por defecto con JSDOM y que por lo general, lleva a muy poca configuración, como veremos a continuación.

Para trabajar con Jest en nuestro proyecto de VueJS tenemos que hacer lo siguiente:

NOTA: Doy por hecho que trabajaremos en un proyecto donde Webpack y vue-loader se encuentran en el flujo de trabajo. A lo largo de la serie hemos explicado todos estos conceptos

En un terminal, que se encuentre en la raíz de nuestro proyecto, ejecutamos lo siguiente:

```
$ npm install --save-dev jest
```

Con esto hemos instalado Jest y ya podemos trabajar con él. Lo siguiente es crear una tarea de npm para que podamos ejecutar los tests. Es tan sencillo como esto:

```
// package.json

{
  "scripts": {
    "test": "jest"
  }
}
```

Y bueno... a partir de aquí está casi todo. Como te digo, Jest lleva muy poca configuración y todo va por convención. Si ejecutamos ese comando, Jest lo que va a hacer es buscar todos los ficheros que acaben en `.spec.js` o `.test.js` dentro del proyecto y ejecutar su código.

[Este patrón puede ser cambiado](#). No es obligatorio. ¿Dónde colocamos estos ficheros de testing? Pues depende dónde más te guste. La convención indica que todos los tests se encuentren bajo una carpeta llamada `__tests__` en cada una de las carpetas importantes del proyecto. Es decir:

```
/app
  /src
    /components
      /__tests__
    /stores
      /__tests__
    /routes
      /__tests__
```

Otros equipos colocan todos los tests a nivel de `src`. Es decir:

```
/app
  /src
  /tests
```

Esto ya es más a gusto de consumidor. Si usas la `vue-cli` e indicas que quieres tests unitarios con Jest, te lo incluirá de esta última manera.

Vale, hasta aquí guay, pero dónde configuro otro tipo de cosas. Jest es poco intrusivo, pero como se ve en su documentación, tiene una [gran cantidad de configuraciones](#). Para configurar funcionalidades diferentes a las de por defecto, añadiremos un nuevo apartado dentro de `package.json` de esta manera:

```
// package.json
{
  "jest": {
    // ...
    "collectCoverage": true,
    "collectCoverageFrom": [
      "**/*.{js,vue}",
      "!**/node_modules/**"
    ],
    "coverageReporters": ["html", "text-summary"],
    "moduleNameMapper": {
      "^@/(.*)$": "<rootDir>/src/$1"
    }
  }
}
```

Esta configuración está indicando que se active la opción de comprobar la cobertura de test ( `collectCoverage` ), que se recopile toda la cobertura de los ficheros `.js` y `.vue` que no se encuentren dentro de `node_modules` ( `collectCoverageFrom` ) y que la forma de mostrarlo sea por el terminal y en html ( `coverageReporters` ).

La última configuración lo que hace es permitirnos usar un alias (en este caso `@`) para poder hacer referencias absolutas a la raíz del proyecto ( `moduleNameMapper` ). Esto es algo que ya hacíamos en Webpack y a que nos va a ser útil ahora para importar módulos de una manera más sencilla.

NOTA: Si no te gusta ensuciar el `package.json` con configuraciones extra, podemos llevarnos toda la configuración de Jest a un fichero separado.

A lo largo del post veremos nuevas configuraciones extra que tendremos que incluir por estar trabajando con VueJS.

## Necesitamos un traductor SFC-ES6

Los SFC son bastante peculiares. Necesitamos que de alguna manera Jest sepa cómo tratarlos para que luego podamos importar estos componentes dentro de nuestros tests y Jest no tenga problemas al ejecutarlo. Jest no tiene por defecto un convertidor de código `.vue` a código `.js`, pero sí nos deja incluir transformadores. Algo parecido a lo que hace Webpack con sus loaders.

Jest no tiene por defecto este transformador, pero el core de VueJS ha creado `vue-jest`, una librería encargada de transformar estos SFC a código ES6.

Para incluir esto en nuestro proyecto, tenemos que hacer lo siguiente:

Instalamos la librería `vue-jest`:

```
$ npm install --save-dev vue-jest
```

Y ahora incluimos la siguiente configuración en nuestro `package.json` o en nuestro fichero de configuración Jest:

```
// package.json
{
  // ...
  "jest": {
    "moduleFileExtensions": [ "js", "json", "vue" ],
    "transform": {
      ".*\\.vue$": "<rootDir>/node_modules/vue-jest"
    }
  }
}
```

Con esto le decimos a Jest que los módulos que se podrán importar en nuestros ficheros de tests tienen la posible extensión `js`, `json` y `vue` (`moduleFileExtensions`) y que cuando encuentre una importación con un fichero que termine en `.vue` que lo transforme con la librería `vue-jest` a ficheros `.js`.

Sencillo.

## Necesitamos un traductor ES6-ES5

Nos podría valer esta transformación, pero si nos ponemos un poco paranoicos, hay que tener en cuenta que nuestros componentes están escritos en ES6, por lo general.

Vale que Jest se ejecuta sobre NodeJS y que NodeJS ya soporta casi todas las funcionalidades de ES6, pero qué necesidad hay de preocuparnos cuando podemos convertir nuestro código a ES5.

Lo que vamos a hacer es incluir otra librería que nos permite usar Babel en Jest. Para ello hacemos lo siguiente:

Instalamos `babel-jest` como de costumbre:

```
$ npm install --save-dev babel-jest
```

E incluimos un nuevo transformador en Jest:

```
{
  // ...
  "jest": {
    // ...
    "transform": {
      // ...
      "^.+\\.jsx?$": "<rootDir>/node_modules/babel-jest"
    },
    // ...
  }
}
```

Al igual que antes, lo que hace Jest es ejecutar un transformador sobre todos los ficheros con extensión `.js`

## Necesitamos una herramienta que manipule componentes Vue

Por parte del runner y de los preprocesamiento tenemos todo. ¿Ahora qué? Pensemos que un componente VueJS tiene un ciclo de vida y unas peculiaridades visuales. Testear elementos tan arraigados al DOM suele ser complejo.

Necesitamos un mecanismo que nos permita renderizar componentes, montarlos, crearlos, destruirlos, lanzar sus eventos nativos y personalizados a nuestro gusto o mockear partes si fuera necesario.

Para ello, la gente de Vue ha pensado en cómo hacerlo y han creado una librería que nos permite realizar todo esto. Se llama `vue-test-utils`. Es una librería agnóstica al framework de testing que uses y al test runner. Por lo que si Jest no te convence para probar componentes de Vue, siempre podrás seguir usando `vue-test-utils` con tu ecosistema favorito.

Para incluir la librería en nuestro proyecto, hacemos lo siguiente:

```
$ npm install --save-dev @vue/test-utils
```

Por ahora solo la instalamos. En el siguiente capítulo veremos todo su potencial.

## Necesitamos un framework de testing

Todo test suele tener un sistema para modularizar y jerarquizar los tests por medio de suites. Estas suites a su vez cuenta con casos de uso que se encargan de confirmar un comportamiento específico de un elemento.

Necesitamos también una serie de matcher o checkers que se encarguen de confirmar de una manera más semántica si las cosas son correctas o si no lo son. En definitiva, necesitamos un framework JavaScript de tests que nos permita escribir casos de una manera clara y rápida.

En este caso, no pasa como otros test runners que solo ejecutan test, si no que Jest además incluye toda la librería de pruebas de manera global para trabajar en el desarrollo de tests. No tenemos que instalar nada, con tener Jest, lo tenemos.

Aunque lo veremos a lo largo de la serie, [es bueno que eches un vistazo a la sintaxis de esta parte de Jest](#).

## Nuestro primer test en Jest

Si toda la configuración ha ido bien, ya estaríamos preparados para escribir nuestro primer test. Como en todo el mundo del desarrollo, nuestro primer test va a ser para probar que hemos hecho bien un `Hello World`. En nuestro caso un `Hello Name`.

Así que lo primero que hacemos es crear un fichero `hello-name.js` en nuestro proyecto con el siguiente código:

```
// ./src/hello-name.js
export default function (name) {
  return 'Hello ' + name;
}
```

Ahora lo que hacemos es crear un fichero de test. Lo llamamos `hello-name.spec.js` y escribimos los siguiente:

```
// ./test/hello-name.spec.js
import helloName from '@hello-name'

describe('Tests File hello-name.js', () => {
  it('Should get 'Hello Jose', () => {
    const name = 'Jose'
    const message = helloName(name)
    expect(message).toEqual('Hello ' + name)
  })
})
```

Todos los test, más largos, más pequeños, más complejos, más simples tienen una estructura parecida a esta.

Lo primero que se hace es importar el módulo que se quiere probar en una batería completa de tests. En este caso importamos 'hello-name'. Lo siguiente es generar una suite ( `describe` ) con una descripción clara de qué unidad lógica vamos a probar. En este caso vamos a probar casos posibles de 'hello-name'.

Los siguiente es incluir casos de uso ( `it` ). Cada uno de los casos de uso suelen tener un mecanismo parecido: Configuración-ejecución-comprobación. En este caso la configuración es la variable `name` . La ejecución es la ejecución de la función del módulo `helloName` y la comprobación es el matcher final con el `expect` y el `toEqual` .

Con esto, si ejecutamos en el terminal `npm test` , debería de funcionar y de conseguir nuestro primer test correcto. Un test mu simple, pero que ya demuestra toda la potencia de la automatización.

Esta suite no está acabada. ¿Qué ocurre si yo no indico un parámetro a la función? ¿Cómo se comporta nuestro módulo e este caso? Recuerda que crear tests unitarios es una forma de forzar todas las posibles salidas de un sistema. Se imaginativo y piensa en usos de caso que podrían darse. Usa mucho los informes de cobertura para comprobar si un módulo tiene partes sin cubrir con un test.

## Conclusión

Al final, la instalación y los primeros conceptos suele ser algo laborioso. Pero creedme si os digo que Jest es de las herramientas JavaScript que menos configuración tiene. En este caso las peculiaridades de VueJs ha hecho que se complique un poco más. Pero ya está listo para trabajar con VueJS.

Como he dicho en los primeros apartados. Este bloque de testing en VueJS quiero que sirva para concienciar de la importancia de incluir test. Quiero concienciar sobre los tiempos que nos puede ahorrar, sobre las decisiones que nos puede hacer tomar.

Muchos cargos y mucha gente con la que me he encontrado en estos años suele ver el testing automático como una pérdida de tiempo o como un tiempo que no genera los beneficios esperados. Yo os digo que aprender cuesta y que cambiar actitudes y hábitos es complicado.

Yo mismo estoy en pleno proceso de aprendizaje y cambio de mentalidad, pero creedme que cuando los tests salen, empiezan a dar feedback y empiezas a ver un mejor código, vuestra mentalidad también cambiará. Hay que tener mucha paciencia y mucha constancia.

En el próximo capítulo empezaremos a explicar en detalle cómo empezar a testear componentes y veremos las facilidades que nos dará `vue-test-utils` para crear test más legibles y fáciles de desarrollar. Veremos cuales serán las mejores partes de testear los componentes e intentaremos entender mejor todo el flujo de trabajo con Jest.

Por ahora lo dejamos aquí

Nos leemos :)

## Capítulo 22. Testeando nuestros componentes

En el post anterior, explicamos de manera superficial cómo crear pruebas con Jest. Conseguimos configurar el entorno e incluimos todas las librerías necesarias para poder trabajar en un entorno típico de VueJS.

En el post de hoy, entraremos en más detalle de cómo nos va ayudar `vue-test-utils` a crear test para VueJS, qué elementos de un componente nos interesa probar y qué peculiaridades vamos a tener a la hora de crear tests para nuestras pequeñas vistas.

Como hemos dicho hasta ahora, por lo general, un test intentará probar unidades de código como si fuesen cajas negras. Creados unos datos de entrada, nuestro código tiene que devolver unos datos de salida, el cómo lo consiga es un problema de implementación.

En nuestro caso, nuestra unidad de código mínima será un componente, un componente visual, un componente con su HTML, CSS y JavaScript. Lo que tendremos que hacer en las pruebas unitarias de componentes es inyectar unos datos de entrada determinados y obtener aquel HTML renderizado de la manera que nosotros deseamos.

También será necesario que lancemos todos aquellos eventos para los que nuestro componentes tienen comportamientos implementados y vigilaremos que el estado se encuentre correcto según nuestras especificaciones de negocio.

Pero bueno... como es mejor verlo que contarlo, empecemos a trabajar con `vue-test-utils` :

### ¿Cómo renderizamos un componente para testearlo?

Lo primero que necesitamos es una forma de renderizar en memoria un componente y contar con alguna herramienta que nos permita confirmar que ciertos elementos se encuentran donde deben.

Vale, no contamos con navegador, no podemos renderizar en un entorno real un componente. ¿Cómo lo vamos a hacer? `vue-test-utils` cuenta con dos funcionalidades muy útiles para renderizar un componente en memoria: `mount` y `shallow` .

Ahora veremos qué diferencia hay entre estas dos funcionalidades por ahora tenemos que saber que nos permite renderizar componentes VueJS. Pensemos en el siguiente componente de VueJS:

```
<template>
  <div>
    <span class="count">{{ count }}</span>
    <button @click="increment">Increment</button>
  </div>
</template>

<script>
export default {
  data () {
    return { count: 0 }
  },
  methods: {
    increment () {
      this.count++
    }
  }
}
</script>
```

Este componente lo único que hace es gestionar un contador que el usuario puede ir aumentando. Este componente se caracteriza por no tener datos de entrada ( `props` ) y no tener como dependencia a otros componentes o librerías.

Para poder hacer pruebas sobre este componente, creamos un fichero llamado `counter.spec.js` e incluimos su primer test:

```
// counter.spec.js
import { mount } from '@vue/test-utils'
import Counter from './counter'

describe('Component Counter', () => {
  it('should render the correct markup', () => {
    const wrapper = mount(Counter)
    expect(wrapper.html()).toContain('<span class="count">0</span>')
  })
})
```

Este primer test (nada útil, pero sí muy didáctico), lo único que hace es comprobar que el contenido renderizado del componente es el correcto. ¿Cómo hacemos esto?

- Primero importamos la utilidad `mount` que nos permite renderizar componentes en memoria

- Importamos el componente ( `Counter` ) a probar.
- Creamos una nueva suite para el componente con `describe` .
- Creamos un caso de uso con `it` .
- Internamente del caso de uso, empieza el test. Primero montamos y renderizamos el componente `Counter` con `mount` . Esto nos devuelve un envoltorio o `wrapper` que tiene un montón de métodos que nos permite jugar con facilidad con el elemento renderizado en memoria. Los que hayáis trabajado con jQuery, vais a disfrutar mucho con este envoltorio [ya que nos permite hacer pequeñas consultas sobre este DOM renderizado en memoria](#).
- Por último esperamos que dentro del html renderizado ( `wrapper.html()` ), exista el contenido indicado en `toContain`.

Fácil ¿no?

Como habéis visto, en el ejemplo hemos usado `mount` . La diferencia entre `mount` y `shallow` es que `mount` te renderiza y monta el componente en su totalidad y `shallow` no, Esto significa que si nuestro componente tuviese en su interior otros componentes hijos, `mount` también te los renderizaría, mientras `shallow` los ignoraría.

Que exista ambos sistemas tiene sentido. Puede que muchas veces queramos probar un componente de manera aislada, sin contar con sus dependencias ( `shallow` ) para hacer pruebas de ciclo de vida, de entradas y salidas, de gestión de eventos. Renderizar el resto de elementos solo va a aumentar tiempos de ejecución al test y además, arrastrarnos dependencias que no necesitamos.

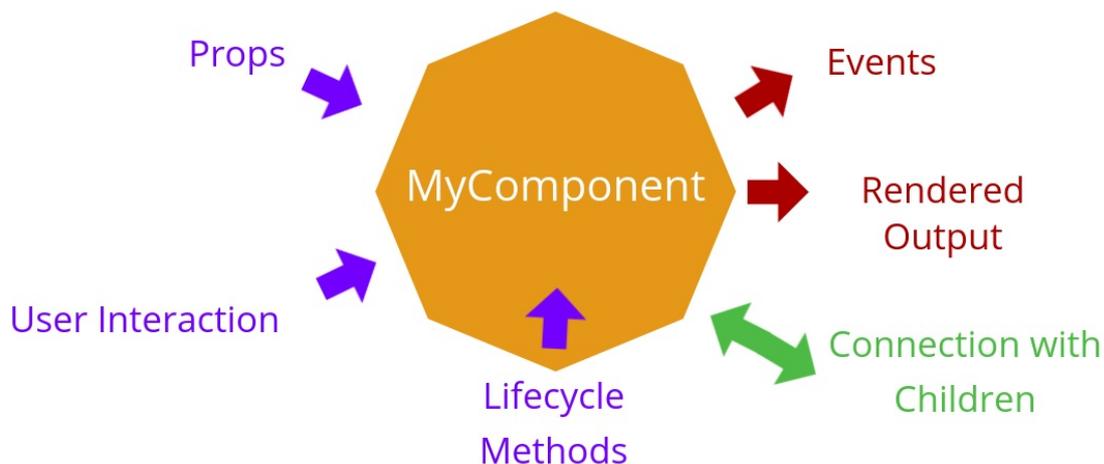
Pero puede darse el caso que nuestro tests se comporte más como un test de integración y que lo que nos interese sea ver cómo un componente orquesta a componentes hijos, ver cómo obtengo información de un componente y ver si la inyección en otro funciona correctamente.

Saber cuando elegir uno y otro será la clave para el tester.

## ¿Qué partes deberíamos testear de un componente y cómo?

Un componente de VueJS tiene estas posibles entradas y salida:

## Test the Public Interface!



Dependiendo de cómo configuremos un componente ( `props` ), cómo sea la interacción del usuario y cómo sea el ciclo de vida, se darán unas salidas u otras. Además, hay que pensar que un componente puede tener en su interior otras componentes de manera estática o dinámica (slots).

## Manipulando el estado del componente

Un componente tiene un estado externo o propiedades de configuración llamadas `props` y tiene un estado interno llamado `data` .

En todo momento un componente montado localmente cuenta con dos métodos para realizar renderizados reactivos. Estos son los siguientes:

```
wrapper.setData({ count: 10 })  
wrapper.setProps({ foo: 'bar' })
```

También podemos 'mockear' las propiedades de un componente. Tenemos el mismo componente `Counter` de antes, solo que ahora permitimos que se pueda configurar por defecto

```
<template>
  <div>
    <span class="count">{{ count }}</span>
    <button @click="increment">Increment</button>
  </div>
</template>

<script>
export default {
  props: {
    count: {
      type: Number,
      default: 0
    }
  },
  methods: {
    increment () {
      this.count++
    }
  }
}
</script>
```

Para este componente con la prop numérica `count` tengo dos posibles casos de uso: La propiedad al ser opcional, me va a permitir indicarle una propiedad o no. Por tanto hagamos los dos casos de uso:

El primero es sencillo:

```
// counter.spec.js
import { mount } from '@vue/test-utils'
import Counter from './counter'

describe('Component Counter', () => {
  it('should set count with data default', () => {
    const wrapper = mount(Counter)
    expect(wrapper.find('.count').text()).toBe(0)
  })
  // ...
})
```

Lo que hacemos es montar el componente `Counter` sin pasar ninguna propiedad 'mockeada' y comprobar que el texto que hay en el elemento con clase `count` es el valor por defecto (en este caso 0).

El segundo caso es en el que el desarrollador sí le va a indicar un valor al componente. Lo hacemos así:

```
// counter.spec.js
import { mount } from '@vue/test-utils'
import Counter from './counter'

describe('Component Counter', () => {
  //...
  it('should set count with data default', () => {
    const countDefault = 10
    const wrapper = mount(Counter, {
      propsData: { count: 10 }
    })
    expect(wrapper.find('.count').text()).toBe(countDefault)
  })
  // ...
})
```

Montamos el componente con lo que se indique en `propsData`. El ejemplo es tontísimo, pero se ve la potencia de cómo se comporta el componente internamente dependiendo del valor indicado.

Si ahora un desarrollador decide que la propiedad por defecto es otra o que ya no tiene que tener valor por defecto, el test nos lo chivará.

## Simulando las interacciones del usuario

Estamos tan cerca del usuario, que la mayoría de los cambios de estado los va a provocar el propio usuario. Es por eso que necesitamos un mecanismo para lanzar esos eventos y que el componente reaccione a los cambios registrados.

Siguiendo con el componente `Counter`, hagamos un caso de uso dónde el usuario decida incrementar el contador en uno. El test podría ser este:

```
//...
it('button click should increment the count', () => {
  const wrapper = mount(Counter)
  expect(wrapper.vm.count).toBe(0)
  const button = wrapper.find('button')
  button.trigger('click')
  expect(wrapper.vm.count).toBe(1)
})
//...
```

Lo que hacemos es comprobar primero que, al instanciarse el componente, el valor sea el esperado por defecto (0). Lo siguiente es acceder al botón del componente y luego lanzar un evento `click` por medio del método que tiene el `wrapper` llamado `trigger`.

Esto ejecutará funciones internas (o no). El caso esperado es que el valor de `count` se haya incrementado en 1.

Con `trigger` podemos lanzar todos los eventos nativos y podemos usar los modificadores de eventos.

## Confirmando eventos personalizados

Cuando un componente hijo quiere transmitir información acciones a un componente padre lo hace por medio de eventos personalizados. ¿Pero cómo confirmo si un componente ha emitido un evento personalizado?

El `wrapper` es nuestro mejor amigo y tiene métodos que nos confirman si estas cosas han ocurrido o no.

Volvemos a tener el componente contador. Solo que en este caso además de incrementar el valor, emite el nuevo valor para que cada padre haga con el valor lo que desee:

```
<template>
  <div>
    <span class="count">{{ count }}</span>
    <button @click="increment">Increment</button>
  </div>
</template>

<script>
export default {
  props: {
    count: {
      type: Number,
      default: 0
    }
  },
  methods: {
    increment () {
      this.count++
      this.$emit('increment', this.count)
    }
  }
}
</script>
```

El caso de uso, sería de esta manera:

```
import { mount } from '@vue/test-utils'
import Counter from './Counter'

describe('Component Counter', () => {
  //...
  it('should emit event increment', () => {
    const wrapper = mount(Counter)
    const button = wrapper.find('button')
    button.trigger('click')

    expect(wrapper.emitted().increment).toBeTruthy()
    expect(wrapper.emitted().increment[0]).toEqual([1])
  })
  // ...
})
```

Vuelvo a ejecutar el `click` que es el que provoca la emisión de `increment`. Lo que hacemos luego es comprobar que se ha registrado un evento de tipo `increment` con el método del `wrapper` llamado `emitted`.

## ¿Cómo testeamos un componente que tiene dependencias?

Nos va a pasar que algunos componentes tengamos dependencias de otros plugins de VueJS. Imagina que tu componente usa `vuex` o `vue-router`. Como los componentes se renderizan de manera aislada y fuera del propio framework en una instancia de VueJS local, todas estas dependencias que usa, van a provocar comportamientos inesperados. Son efectos laterales que deberíamos evitar.

Para conseguir que nuestro plugin tenga estas dependencias, contamos con `createLocalVue`. Esta pieza nos permite renderizar y montar nuestro componente bajo una copia local de VueJS donde ya sí podemos incluir plugins.

Lo hacemos de la siguiente forma:

```
import { createLocalVue } from '@vue/test-utils'
import { MyPlugin } from './my-plugin'

const localVue = createLocalVue()
localVue.use(MyPlugin)

mount(Component, { localVue })
```

Lo único que hacemos es importar `createLocalVue` y nuestro plugin. Creamos una instancia de VueJS y le instalamos el plugin con `localVue.use()`. Lo siguiente es decirle a `mount` o `shallow` que use nuestra instancia local de Vue porque es la que cuenta con las dependencias globales necesarias.

Puede que a veces incluir toda una librería global sea matar moscas a cañonazos. Puede darse el caso que en un componente se use algo ínfimo y que no merezca la pena. En este caso podemos inyectar un mock para que se hagan pruebas con él.

Imaginemos que queremos mockear `vue-router` en un componente. Podríamos hacer esto:

```
import { mount } from '@vue/test-utils'

const $route = {
  path: '/',
  hash: '',
  params: { id: '123' },
  query: { q: 'hello' }
}

mount(Component, {
  mocks: {
    $route
  }
})
```

Cuidado con lo explicado en este apartado ¿vale? Que tengamos esta facilidad no significa que estemos usando bien el framework. Existirán casos donde nuestro componentes estén tan acoplados al proyecto, que hacer esto no suponga un problema. Es decir, un componente muy acoplado al proyecto, que no sea posible reutilizarlo, no tiene problemas a que tenga como dependencias elementos globales del framework.

Ahora bien ¿Qué pasa si necesitamos componentes muy reutilizables y que se puedan usar en varios proyectos? Que tu tengas un plugin instalado en un proyecto, no tiene que significar que en otro sí. Por lo que si queremos usar ese componente, también nos tendremos que llevar el plugin. Abusar de `createLocalVue` y la inyección de plugins puede ser un buen 'Code Smell'.

Intenta que solo los componentes orquestadores (también llamados controladores o tipo page) tengan estas dependencias. Evita que componentes muy visuales puedan realizar uso de piezas tan globales o las estarás acoplando.

Tenlo en cuenta.

## ¿Cómo pruebo comportamientos asíncronos?

Tenemos el siguiente componente:

```
<template>
  <button @click="fetchResults" />
</template>

<script>
import axios from 'axios'
export default {
  data () {
    return {
      value: null
    }
  },
  methods: {
    async fetchResults () {
      const response = await axios.get('mock/service')
      this.value = response.data
    }
  }
}
</script>
```

Testear este componente nos va a dar problemas por dos razones:

1. Tenemos una dependencia super hardcodeada en el componente como es el uso de `axios`
2. Tenemos un método con comportamiento asíncrono, por tanto puede dar problemas en lo resultado esperados.

### Dependencias hardcodeadas y los mocks de Jest

El primer problema se soluciona de una manera simple: Mockeando la parte de `axios` que nos interesa y que devuelva los datos que deseamos. Para hacer esto, lo vamos a hacer de manera genérica. Jest nos permite mockear métodos de librerías.

Para ello incluimos una carpeta de test que tenga este nombre `__mocks__`. La nomenclatura es la típica de Jest, así que nada que objetar. Lo siguiente que hacemos es incluir el siguiente fichero `axios.js` dentro:

```
// ./test/__mocks__/axios.js

export default {
  get: () => new Promise(resolve => {
    resolve({ data: 'value' })
  })
}
```

De esta manera, Jest lo que hará será usar este código cuando decidamos usar `axios` como un mock.

Para indicarle a Jest dentro de una suite que haremos uso del mocks, solo tenemos que indicar lo siguiente:

```
import { shallow } from '@vue/test-utils'
import Foo from './Foo'

jest.mock('axios')

describe('Foo', () => { })
```

Con `jest.mock('axios')`, Jest ya sabe que tiene que ir a por un mock y usarlo en nuestro código. Lo inyecta de manera automática, no tengo que preocuparme de nada más.

Volvemos a lo mismo de antes, habrá casos en lo que no nos quedará más remedio que hardmockear de esta manera las dependencias, pero por lo general indica un 'Code Smell' de que no se están inyectando las dependencias correctamente.

Usa las propiedades para pasar funciones a su componentes hijos, emite eventos para que componentes más acoplados se encarguen de estas llamadas, envuelve las librerías en piezas que sean fácil de intercambiar por mocks. Todo con tal de evitar estos mecanismos tan extremos.

## Las llamadas asíncronas

Vale, ya sabemos que `axios.get` nos devolverá una promesa a nuestro gusto. Pues hagamos su test:

```
import { shallow } from '@vue/test-utils'
import Foo from './Foo'

jest.mock('axios')

describe('Foo', () => {
  it('fetches async when a button is clicked', () => {
    const wrapper = shallow(Foo)
    wrapper.find('button').trigger('click')
    expect(wrapper.vm.value).toBe('value')
  })
})
```

Vaya... el test ha fallado. No hemos gestionado la promesa correctamente y no hemos dado tiempo a que `wrapper.vm.value` tenga el valor correcto.

Para esto, `vue-test-utils` ya ha pensado en ello y nos permite esperar hasta que la promesa está resuelta. Veamos:

```
import { shallow } from '@vue/test-utils'
import Foo from './Foo'

jest.mock('axios')

describe('Foo', () => {
  it('fetches async when a button is clicked', () => {
    const wrapper = shallow(Foo)
    wrapper.find('button').trigger('click')
    wrapper.vm.$nextTick(() => {
      expect(wrapper.vm.value).toBe('value')
    })
  })
})
```

Lo que hacemos es hacer uso de `$nextTick` de VueJS. Esta función se ejecuta cuando el DOM se vuelve a ejecutar. Como la promesa va a provocar cambios, esperamos a esos cambios.

El test no es correcto todavía porque bueno... el componente se comporta de manera asíncrona, pero el caso de uso, el test, es secuencial, no entiende de asincronismo tal y como está planteado, por tanto el test acabará antes de esperar a las confirmaciones ( `expect` ).

Para arreglarlo, Jest ya ha pensado en ello y nos permite una función de `callback` que nos permite indicar cuándo un test se tiene que dar por terminado. Lo hacemos así:

```
import { shallow } from '@vue/test-utils'
import Foo from './Foo'

jest.mock('axios')

describe('Foo', () => {
  it('fetches async when a button is clicked', (done) => {
    const wrapper = shallow(Foo)
    wrapper.find('button').trigger('click')
    wrapper.vm.$nextTick(() => {
      expect(wrapper.vm.value).toBe('value')
      done()
    })
  })
})
```

¿Veis ese `done` ? Hasta que no es ejecutado, no se da por terminado el test. Esto tiene un tiempo máximo, si pasado ese tiempo no se ha ejecutado `done` , Jest sigue con el resto de tests.

Puede que el sistema genere demasiados anidamientos o que sea lioso gestionar asincronía para un test. Podemos simplificar el código si usamos la librería `flush-promise` . Esta librería nos devuelve una promesa que no es resuelta hasta que no se han resuelto el resto de manejadores. El código ahora quedaría muy limpio:

```
import { shallow } from '@vue/test-utils'
import flushPromises from 'flush-promises'
import Foo from './Foo'

jest.mock('axios')

describe('Foo', () => {
  it('fetches async when a button is clicked', async () => {
    const wrapper = shallow(Foo)
    wrapper.find('button').trigger('click')

    await flushPromises()
    expect(wrapper.vm.value).toBe('value')
  })
})
```

Sin `done` , sin `$nextTick` . Lo que hacemos es una `async function` y esperar a que todos los manejadores se den por terminados. En ese momento es cuando estamos preparado para hacer las confirmaciones.

## Conclusiones

Parece que hemos explicado mucho, pero solo hemos explicado la superficie del testeo de componentes. Sí, sabemos lo principal, pero queda mucho trabajo de cómo inyectar mocks, stubs y demás mecanismos de testing.

¿Qué pasa cuando hay mucho código repetido en una suite? ¿Qué pasa con otros elementos como los `slots` o los `watcher`? ¿Qué pasa si quiero mockear un CSS o un PNG? Creo que son temas interesantes que por ahora se van a quedar fuera de este material.

Al igual que pasaba con todo el manual, estamos en una serie introductoria, hay mucho trabajo de práctica y profundización que tendremos que hacer después, pero creo que esto es un buen comienzo para conocer `vue-test-utils`.

Una cosa que queda patente en el posts, es que con la inclusión de tests en nuestro proyecto, no es suficiente. Los tests nos dan feedback, pero en muchas ocasiones sin un buen diseño de componentes, será difícil trabajar bien y hacer desarrollos de calidad (término bastante subjetivo y que siempre lleva a debate).

Evitemos poner dependencias a fuego. Hagamos piezas pequeñas, con una única responsabilidad. Encapsulemos bien los estados, definamos buenas entradas y salidas. Tengamos en mente siempre SOLID, KISS y YAGNI y nos haremos la vida más fácil.

Por el momento, es todo.

Nos leemos :)

## Capítulo 23. Testeando nuestros stores

Hemos visto cómo configurar nuestro proyecto para que pueda ser testeado, hemos creado nuestro primeros test sobre componentes de VueJS y hemos jugado con diferentes tipos de componentes y complejidades, desde componentes que tenían dependencias hasta componentes que solo eran visuales.

Nos queda para terminar esta sería, ver cómo somos capaces de testear nuestros stores. Si recordamos, los stores son aquellas piezas de código que configuraban vuex y que nos permitían gestionar el estado global de nuestra aplicación en pequeños estancos bien modularizados y abstraídos.

Tenemos que ver las aproximaciones que podemos llevar a cabo y cómo probar cada uno de los elementos. Vayamos al lío:

### ¿Qué aproximación seguir a la hora de probar stores?

Tenemos dos formas de probar un store:

- **Separando las diferentes partes de un store (getters, mutations y actions) y probarlos de manera independiente:** Lo bueno de vuex es que cada uno de su elementos son funciones normales que pueden ser testadas de manera separada. Si una función trabaja bien por separado, cuando sea incluida en otra pieza, seguirá teniendo ese buen comportamiento. Es la magia de la modularización. En este caso es más fácil ver qué parte del store ha fallado. Pueden venir muy bien como test unitarios al uso.
- **Juntando todos los elementos y hacerles trabajar como un todo dentro de un store real. Se usan los mecanismos que existen en vuex y se comprueba su comportamiento y el estado que han dejado.** En este caso se realizan los tests tal y como luego los componentes los van a usar. Pueden venir muy bien como test de integración. Ambas son válidas y pueden ayudarnos en diferentes contextos dependiendo de cómo nos sintamos mas cómodos.

A continuación veremos cómo probar los elementos por separado de un store:

### ¿Cómo pruebo un getter?

Tenemos el siguiente getter de un posible store:

```
// getters.js
export default {
  evenOrOdd: state => state.count % 2 === 0 ? 'even' : 'odd'
}
```

Si recordamos, un getter no era más que una posible consulta sobre un estado determinado de un store. En este caso, la consulta indica si el contador es impar o par.

Bueno, probar esto es fácil, tenemos una función llamada `evenOrOdd` que tiene como parámetro un objeto estado y que contiene un flujo interno. Por tanto, tendremos dos tests: uno para comprobar que nos devuelve `even` cuando hay un número par en el contador y `odd` cuando hay un número impar en el contador:

```
// getters.spec.js

import getters from './getters'

describe('getters store', () => {
  it('evenOrOdd returns even if state.count is even', () => {
    const state = { count: 2 }
    expect(getters.evenOrOdd(state)).toBe('even')
  })

  it('evenOrOdd returns odd if state.count is even', () => {
    const state = { count: 1 }
    expect(getters.evenOrOdd(state)).toBe('odd')
  })
})
```

Iniciamos el estado con el contador como deseamos y comprobados que el getter devuelve lo esperado.

## ¿Cómo mockeo un getter en un componente?

Ahora bien, si un componente hace uso de uno de nuestros getter, ¿cómo podemos mapearlo? Tenemos el siguiente componente:

```
<template>
  <div>
    <p v-if="inputValue">{{inputValue}}</p>
    <p v-if="clicks">{{clicks}}</p>
  </div>
</template>

<script>
import { mapGetters } from 'vuex'
export default{
  computed: mapGetters([
    'clicks',
    'inputValue'
  ])
}
</script>
```

Este componente muestra por pantalla los datos de los getters clicks y inputValue. Para hacer un tests de esto, lo que hago es mockear los getters. Esto lo consigo:

- Añadiendo vuex a nuestra instancia de VueJS local al test.
- Creando un mock de nuestro getters
- Incluyendo este store a nuestro componente a probar.

El test sería así:

```
import { shallow, createLocalVue } from '@vue/test-utils'
import Vuex from 'vuex'
import Actions from '@components/Getters'

const localVue = createLocalVue()
localVue.use(Vuex)

describe('Getters.vue', () => {
  let getters
  let store

  beforeEach(() => {
    getters = {
      clicks: () => 2,
      inputValue: () => 'input'
    }
    store = new Vuex.Store({ getters })
  })

  it('Renders "state.inputValue" in first p tag', () => {
    const wrapper = shallow(Actions, { store, localVue })
    const p = wrapper.find('p')
    expect(p.text()).toBe(getters.inputValue())
  })

  it('Renders "state.clicks" in second p tag', () => {
    const wrapper = shallow(Actions, { store, localVue })
    const p = wrapper.findAll('p').at(1)
    expect(p.text()).toBe(getters.clicks().toString())
  })
})
```

El `beforeEach` nos ayuda a iniciar el store para cada tests. Tenemos que tener en cuenta que cada test tiene que ser determinista y aislado. Esto quiere decir que siempre debe darnos el mismo resultado dado el mismo estado de entrada y que si yo ordeno los tests de otra manera, todos siguen siendo correctos. Ningún test debe depender de otro test.

Luego usamos ese store, para que los métodos internos del componente trabajen correctamente. Desta manera hemos aislado el componente de vuex y podemos probar comportamientos internos.

Aunque lo explicaremos. este sistema de mockeo será igual para todas las partes de vuex.

## ¿Cómo pruebo una mutation?

El resto de elementos se testeara y se mockeara en un componente de manera muy parecida. Las mutations son funciones que permiten cambiar o mutar el estado de un store. Son funciones de ejecución síncrona por lo que se comportan parecido que los getters. Podemos ver a los getter como funciones para obtener el estado del store y los mutations como funciones para setear el estado del store.

Entonces, dada la siguiente mutation que permite incrementar un contador:

```
// mutations.js
export default {
  increment (state) {
    state.count++
  }
}
```

Podemos testarlo de la siguiente forma:

```
// mutations.spec.js
import mutations from './mutations'

test('increment increments state.count by 1', () => {
  const state = { count: 0 }
  mutations.increment(state)
  expect(state.count).toBe(1)
})
```

Bastante sencillo: inicio el state, ejecuto la mutation y compruebo que el estado tiene el valor esperado. Como paso el estado por referencia, la mutación se hace sobre el mismo objeto todo el rato.

## ¿Cómo mockeo un mutation en un componente?

Para mockear las mutations en un componente, lo hacemos igual que en el caso de los getters.

## ¿Cómo pruebo un action?

Los actions se caracterizan por permitarnos crear lógica que dependa de comportamientos asíncronos. En nuestro caso, el comportamiento asíncrono no es determinante ya que mockearemos todo aquello de lo que tengan alguna dependencia, por tanto, se comporta al

final parecido a una mutación.

Tenemos el siguiente store:

```
// mutations.js
export default {
  SET_DATA (state, data) {
    state.data = data
  }
}

// actions.js
export default {
  getAsync ({ commit }) {
    return axios.get('https://jsonplaceholder.typicode.com/posts/1')
      .then(response => commit('SET_DATA', response.data))
      .catch(err => console.log(err))
  }
}
```

Vemos que tenemos varias complicaciones para probar la acción `getAsync` de manera aislada. Vemos que el elemento dependn de una llamada AJAX con `axios` y que una vez que tenemos el dato, debemos mockear la mutación.

Esto lo hacemos de la siguiente manera:

```
// actions.spec.js
import actions from './actions'
import flushPromises from 'flush-promises'

jest.mock('axios')

describe('actions', () => {
  it('tests with a mock commit', async () => {
    let count = 0
    let data
    let mockCommit = (state, payload) => {
      data = payload
      count += 1
    }

    actions.getAsync({ commit: mockCommit })

    await flushPromises()

    expect(count).toBe(1)
    expect(data).toEqual({ title: 'Mock with Jest' })
  })
})
```

- Con `jest.mock`, mockeamos la llamada `get` de `axios` como hicimos en el post anterior.
- Luego creamos una `mutation` a nuestro gusto que nos permitirá jugar con los datos obtenidos por la `action`
- Ejecutamos la `action getAsync`. Le pasamos el `mockCommit` para que cuando termine ejecute esa `mutation`.
- Hacemos las comprobaciones sobre los datos obtenidos.

Bueno... pues eso, que entre probar esto y nada, lo mismo nos da XD. Hay tanto mockeo que el código me parece demasiado adulterado. Es por etso que quizá para este caso tan pequeño, el `action` sea mejor probarlo con un test sobre un store completo y ver el mecanismo completo de `action->mutation->getter`.

## ¿Cómo mockeo un action en un componente?

El mockeo de un `action` en un componente es igual que en el caso de los `getters` y los `mutations`. Se trata de incluir `vuex` en la instancia local y mockear los métodos de los que depende el componente.

Dado el siguiente componente:

```
<template>
  <div class="text-align-center">
    <input type="text" @input="actionInputIfTrue" />
    <button @click="actionClick()">Click</button>
  </div>
</template>

<script>
import { mapActions } from 'vuex'
export default{
  methods: {
    ...mapActions([
      'actionClick'
    ]),
    actionInputIfTrue: function actionInputIfTrue (event) {
      const inputValue = event.target.value
      if (inputValue === 'input') {
        this.$store.dispatch('actionInput', { inputValue })
      }
    }
  }
}
</script>
```

Lo testeamos de esta forma:

En el fichero `action-mock.spec.js` importamos las dependencias necesarias. En este caso `vue-test-utils`, `vuex` y el propio componente:

```
import { shallow, createLocalVue } from '@vue/test-utils'
import Vuex from 'vuex'
import Actions from '@components/ActionMock'
```

Instanciamos una copia local de Vue y le incluimos vuex:

```
const localVue = createLocalVue()
localVue.use(Vuex)
```

Dentro del test, definimos las actions a mockear y generamos un store como mockeo:

```
describe('ActionsMock.vue', () => {
  let actions
  let store

  beforeEach(() => {
    actions = {
      actionClick: jest.fn(),
      actionInput: jest.fn()
    }
    store = new Vuex.Store({ state: {}, actions })
  })
})
```

Ahora ya podemos crear los diferentes casos de uso sobre el componente. Simplemente, cuando montemos un componente, deberemos indicarle el store del que debe tirar:

```
it('calls store action "actionInput" when input value is "input" and an "input" event is fired', () => {
  const wrapper = shallow(Actions, { store, localVue })
  const input = wrapper.find('input')
  input.element.value = 'input'
  input.trigger('input')
  expect(actions.actionInput).toHaveBeenCalled()
})
```

Podemos ver el ejemplo todo junto:

```
import { shallow, createLocalVue } from '@vue/test-utils'
import Vuex from 'vuex'
import Actions from '@components/ActionMock'

const localVue = createLocalVue()
localVue.use(Vuex)

describe('ActionsMock.vue', () => {
  let actions
  let store

  beforeEach(() => {
    actions = {
      actionClick: jest.fn(),
      actionInput: jest.fn()
    }
    store = new Vuex.Store({ state: {}, actions })
  })

  it('calls store action "actionInput" when input value is "input" and an "input" event is fired', () => {
    const wrapper = shallow(Actions, { store, localVue })
    const input = wrapper.find('input')
    input.element.value = 'input'
    input.trigger('input')
    expect(actions.actionInput).toHaveBeenCalled()
  })
})
```

Se trata de añadir un poco de fontanería para que trabajemos sobre lo que nos interesa.

## Conclusión

Como observamos, hacer pruebas en vuex tiene más que ver con probar piezas JavaScript separadas que con algo que tenga que ver con el propio framework. Hay poca interacción de las librerías y habrá que tener cuidado, como siempre, con las dependencias de cada pieza y con qué llamadas mockeamos.

Por ahora, con que tengamos estos conocimientos sobre testing en VueJS podemos empezar a trabajar. Con el tiempo necesitaremos profundizar y mejorar en técnicas, pero las peculiaridades a las que nos lleva el frameworks las tenemos cubiertas.

Si os habéis quedado con ganas de más, os vuelvo a recomendar el libro [‘Testing VueJS components with Jest’](#) de [Alex Jover](#) y el Curso de Codely [‘Testing con VueJS y Jest’](#) de [Javi Rubio](#) y [Alberto Gualis](#) que os van ayudar a profundizar y a tener una sensibilidad mayor a cómo deben probarse los elementos en VueJS.

Nos leemos :)