ASCO DE CODIGO

"Asquerosamente Remediable"

Directivas, Eventos y Componentes

Julio 11, 2018

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` querremos evitar que nos haga un post contra el servidor para así realizar la lógica que necesitemos 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 elemento 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: {
         isFormEmpty: 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 cara
cteres');
            }

            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();
        }
    }
});
```

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` y `removeClass` - 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 incluímos 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.

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. Cómo 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`, `subitle` 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.

Personalizando eventos

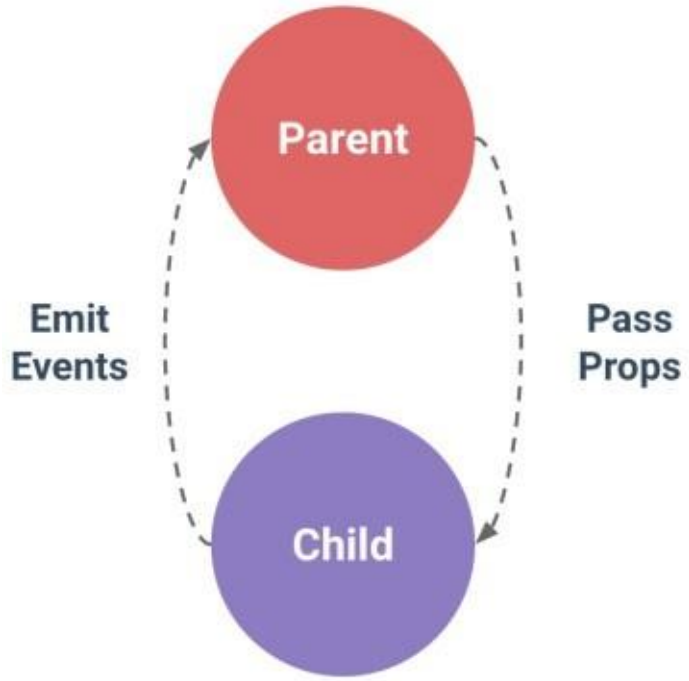
Hemos visto cómo el componente padre consigue comunicar datos a un componente hijo por medio de las propiedades, pero ¿Qué ocurre cuando un hijo necesita informar de ciertos datos o acciones que están ocurriendo en su interior? ¿Cómo sería el return de un

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` .

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

```
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 });
        }
    },
```

VueJS, tenemos que hacer lo siguiente:

```
    template: [
        '<div :class="["course", styleClass]">',
            '<header class="course-header" v-once>',
                '<img :src="header.image" :alt="header.title">',
                '<h2>{{ header.title }}</h2>',
            '</header>',
            '<main class="course-content">',
                '<img src="http://lorempixel.com/300/150/" alt="">',
                '<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 sobreescribir. 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 Introductorio"
  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.

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>
                <img :src="header.image" :alt="header.title">
                <h2>{{ header.title }}</h2>
            </header>
            <main class="course-content">
                <img src="http://lorempixel.com/300/150/" alt="">
                <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="descriptio
n"></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>',
            '<img :src="image" :alt="title">',
            '<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">',
            '<img src="http://lorempixel.com/300/150/" :alt="title">',
            '<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>',
            '<img :src="image" :alt="title">',
            '<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">',
            '<img src="http://lorempixel.com/300/150/" :alt="title">',
            '<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-heade
r>',
            '<course-content :title="title" :subtitle="subtitle" :description="descrip
tion"></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: 'Curse 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 pro
fesores 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.

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 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 cómputos, 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 `actived` y `deactived` , 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.

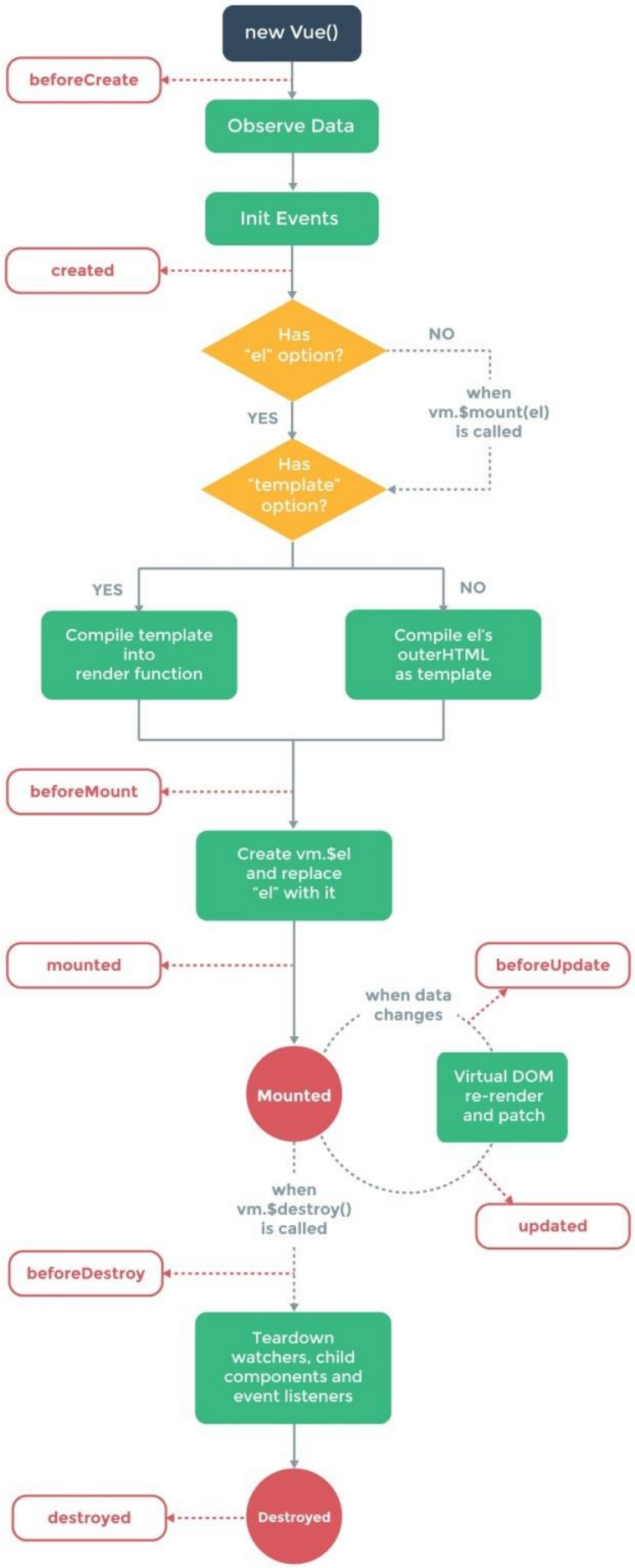
Nos permite saber cómo funciona todo y en qué orden.

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:

Curso ADC


new Vue()
beforeCreate
Observe Data
Init Events
created
Has "el" option?
NO
when vm.$mount(el) is called
YES
Has "template" option?
YES
NO
Compile template into render function
Compile el's outerHTML as template
beforeMount
Create vm.$el and replace "el" with it
mounted
beforeUpdate
when data changes
Mounted
Virtual DOM re-render and patch
when vm.$destroy() is called
updated
beforeDestroy
Teardown watchers, child components and event listeners
destroyed
Destroyed