Contenido

1.- INTRODUCCIÓN

1.1.- ¿Qué es Vue?

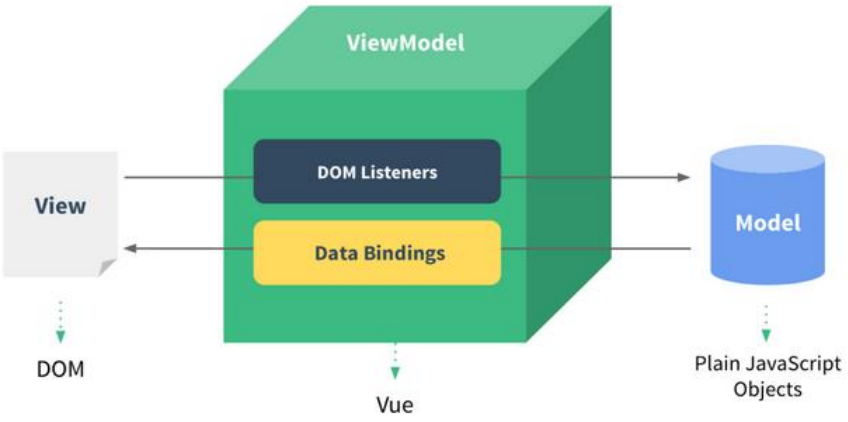
Vue (pronunciado /vju:/, como view en inglés) es un framework progresivo para la creación de interfaces de usuario.
A diferencia de otros frameworks monolíticos, Vue está diseñado desde la base para ser incrementalmente adaptable.
La librería básica se centra solamente en la capa de vista y es muy fácil de integrar con otras librerías o proyectos existentes.
Por otra parte, Vue también es perfectamente capaz de soportar aplicaciones sofisticadas de una página –Single-Page Applications (SPA)– cuando se utiliza con herramientas modernas y librerías de apoyo.

1.2.- Conocimientos previos

La guía oficial asume conocimiento nivel intermedio de HTML, CSS y JavaScript.
Si eres totalmente nuevo en el desarrollo frontend, no es lo mejor empezar a user un framework como primer paso - aprende lo básico y vuelve!
Tener experiencia previa con otros frameworks ayuda, pero no es necesario.
La manera mas fácil de probar Vue.js es usando el ejemplo Hello World. (ver Anexo I)

2.- PRINCIPALES CARACTERÍSTICAS

Entre las características principales de Vue, encontramos que es un **framework "reactivo"** que implementa "two way data-binding" o en español: enlace de datos en dos direcciones (entre la vista y el modelo) de una manera muy eficiente y rápida.



Vue.js es un framework de JavaScript nuevo, si lo comparamos con otros frameworks como Backbone o Ember. Sin embargo, su facilidad de aprendizaje y uso con respecto a otros frameworks y libraries como ReactJS, su rendimiento comparado con AngularJS y la facilidad para usarlo y adaptarlo a proyectos tanto grandes como pequeños, ha hecho que Vue gane cada vez más popularidad.
Vue.js es el framwork "favorito" de los dearrolladores con Laravel.
Vue.js, está más enfocado hacia la vista, y puede ser implementado en el HTML de cualquier proyecto web sin requerir cambios drásticos en el marcado. Por otro lado, también existen componentes que permiten manejar rutas, peticiones AJAX etc. con Vue.js; por ende puedes usarlo tanto para crear pequeños widgets interactivos como para crear "Single Page Applications" (SPA) complejas.
Como acabamos de comentar Vue.js es un framework de JavaScript que se enfoca principalmente en construir **interfaces de usuario**. Ya que solo se encarga de 'manipular' la capa de la vista puede ser integrado fácilmente con otras librerías.

Vue.js es bastante ligero (17 kb) lo que hace que añadirlo a nuestros proyectos no vaya a tener un costo de velocidad o peso. Pero aún así, viene con una gran funcionalidad para crear aplicaciones de página simple.
Algunas de las funcionalidades que nos ofrece son:

- Filtros
- Directivas
- Enlace de datos (data binding)
- Componentes
- Manejo de Eventos (event handling)
- Propiedades computadas
- Render declarativo
- Animaciones
- Transiciones
- Lógica en la plantilla

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 del tema.
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.

2.1.- 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. 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 apartados de este tema, 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.

2.1.1.- EDITORES

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'.

3.- CICLO DE VIDA DE LA INSTANCIA DEL OBJETO VUE (y de cada componente de Vue):

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 de este apartado 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:

3.1.- 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, el resto, debido a su naturaleza, sólo pueden ser usados en el navegador.

3.1.1.- 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', []);
     }
};
```

3.1.2.- 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 hooks 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));
   }
};
```

3.2.- 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).

3.2.1.- 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 lo más recomendable es delegar esto al hook created.

3.2.2.- 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({});
   }
};
```

3.3.- 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.

3.3.1.- 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.

3.3.2.- 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");
       }
};
```

3.4.- 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.

3.4.1.- 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);
      }
};
```

3.4.2.- 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:

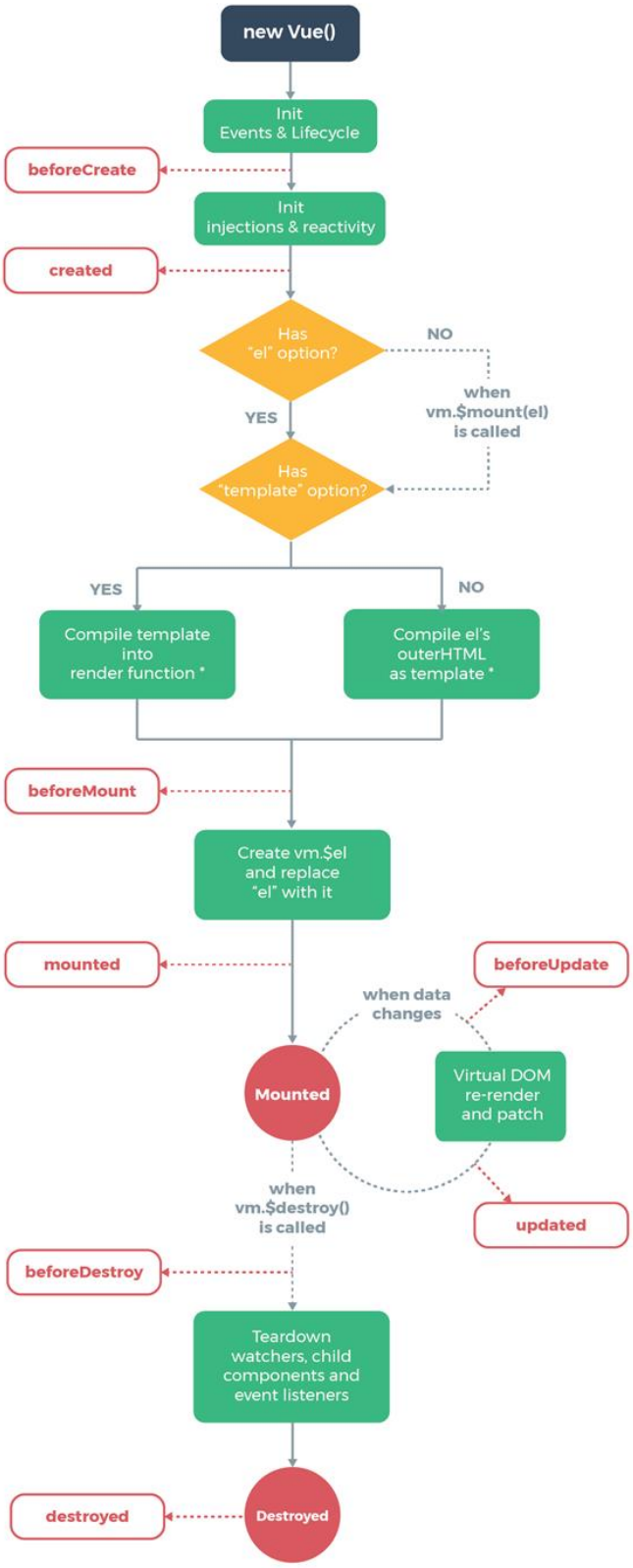
```
const component = {
       destroyed: function () {
             localStorage.removeItem('tasks');
      }
};
```

3.5.- 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 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.

3.6.- En resumen

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. Lo bueno es la posibilidad de contar con ello y poder registrarnos en estos métodos y no depender tanto de la magia interna de un framework.
Os dejo el diagrama que resume el ciclo de vida de un componente:



* template compilation is performed ahead-of-time if using a build step, e.g. single-file components

5.- 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:

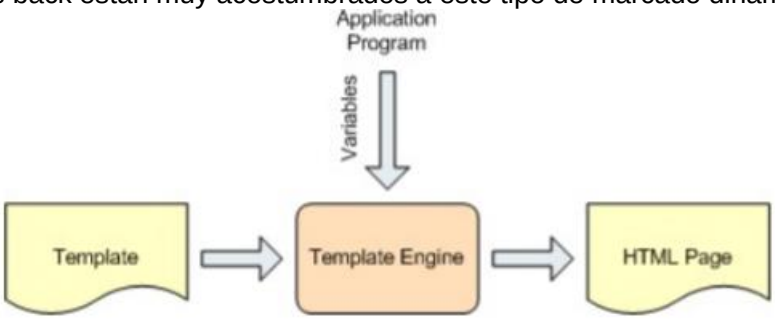
- Podemos optar por guardar una serie de HTMLs estáticos que ya cuentan con la información del usuario necesaria incrustada y almacenados dentro de base de datos. Este es el modelo que suelen desarrollar los CMS de muchos e-commerces o blogs personales.
- Otra opción sería la de renderizar o pintar estos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

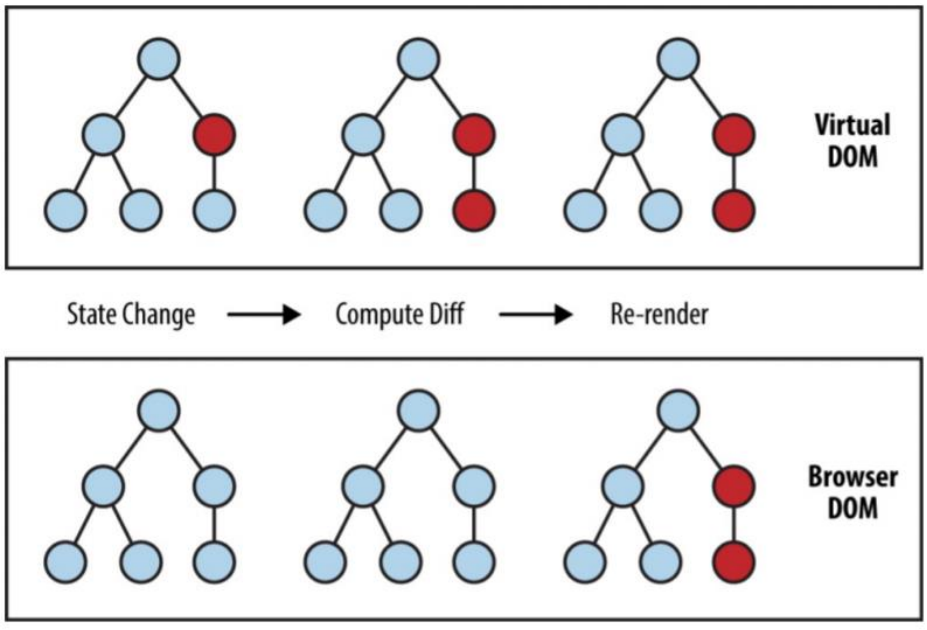
VueJS nos permite desarrollar plantillas HTML que se pueden renderizar tanto en tiempo de back como de front de manera dinámica.

En este apartado vamos a tratar de explicar toda esta sintaxis mediante la explicación los conceptos de interpolación, directiva y filtro.

VueJS cuenta con un motor de plantillas que nos permite 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 maquetadores, 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.



5.1.- 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>
```

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.

5.1.1.- 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="isFormEmpty">Entrar</button>
```

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

5.1.2.- 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 } }}
```

5.2.- 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.

5.2.1.- 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.

5.2.2.- 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 muchas más.

5.3.- Los 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:

https://github.com/freearhey/vue2-filters

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.

Ver ejemplo completo de uso en Anexo III

6.- DETALLES SOBRE SINTAXIS

6.1- Representación declarativa(Declarative Rendering)

En el núcleo de Vue.js se encuentra un sistema que nos permite representar de forma declarativa los datos en el DOM mediante una sintaxis de plantilla (template) sencilla:

.HTML

```
<div id="app" mark="crwd-mark">
 {{ message }}
</div>
```

.JS

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

Es muy similar a haber procesado un string template, pero Vue ha hecho un montón de cosas por debajo. Los datos y el DOM están ahora ligados, y ahora todo es **reactivo**.
¿Como lo sabemos?
Abre la console JavaScript de tu navegador (ahora, en esta misma página) y dale un valor distinto a `app.message`.
Deberías ver el que el ejemplo representado anteriormente se actualiza en consecuencia.

Además de la interpolación de texto, también podemos vincular atributos del elemento como este:

.HTML

```
<div id="app-2" mark="crwd-mark">
 <span v-bind:title="message">
 Hover your mouse over me for a few seconds to see my dynamically bound title!
 </span>
</div>
```

.JS

```
var app2 = new Vue({
 el: '#app-2',
 data: {
 message: 'You loaded this page on ' + new Date()
 }
})
```

Aqui vemos alguna novedad.
AL atributo `v-bind` se le denomina **directiva**.
Las directivas vienen prefijadas con `v-` para indicar que son atributos especiales provistos por Vue, y como debes de haber supuesto, aplican comportamiento especial al procesar el DOM.
Básicamente le estamos diciendo "mantén el atributo `title` de este elemento actualizado con la propiedad `message` de la instancia de Vue"
Si vuelves a abrir tu consola JavaScript e introduces `app2.message = 'some new message'`, verás otra vez que el HTML ligado - en este caso el atributo `title` - ha sido actualizado.

6.2.- Condicionales y bucles

También en muy sencillo alternar la presencia de un elemento:
.HTML

```
<div id="app-3" mark="crwd-mark">
 <p v-if="seen">Now you see me</p>
</div>
```

.JS

```
var app3 = new Vue({
 el: '#app-3',
 data: {
 seen: true
 }
})
```

Introduce `app3.seen = false` en la consola. Deberías de ver como el mensaje desaparece.
Este ejemplo demuestra que podemos ligar datos no solo a texto y atributos, sino también a la **estructura** del DOM.
Además, Vue también provee un sistema muy potente de efectos de transición que puede aplicar transiciones cuando Vue inserte/actualice/elimine elementos de la pagina.
Hay bastantes otras directivas, cada una con sus funcionalidad especiales. Por ejemplo, `v-for` se puede usar para mostrar un listado de elementos usando datos de un Array:
.HTML

```
<div id="app-4" mark="crwd-mark">
 <ol>
 <li v-for="todo in todos">
 {{ todo.text }}
 </li>
 </ol>
</div>
```

.JS

```
var app4 = new Vue({
 el: '#app-4',
 data: {
 todos: [
 { text: 'Learn JavaScript' },
 { text: 'Learn Vue' },
 { text: 'Build something awesome' }
 ]
 }
})
```

Inserta `app4.todos.push({ text: 'New item' })` en la consola. Deberías de ver un elemento añadirse a la lista.

6.2.1.- Renderizado condicional

- **v-if**

Debido a que **v-if** es una directiva, debe estar asociada a un solo elemento.
Pero, ¿y si queremos cambiar más de un elemento?

En este caso, podemos usar v-if en un elemento <template>, que sirve como un contenedor invisible. El resultado final afectará a todo el elemento <template>.

```
<template v-if="ok">
  <h1>Title</h1>
  <p>Paragraph 1</p>
  <p>Paragraph 2</p>
</template>
```

- **v-else**

Se puede usar la directiva v-else para indicar un "bloque else" después de una directiva v-if

```
<div v-if="Math.random() > 0.5">
  Now you see me
</div>
<div v-else>
  Now you don't
</div>
```

- **v-else-if**

El v-else-if, como su nombre indica, sirve como un "else if block" para v-if. También se puede encadenar varias veces

```
<div v-if="type === 'A'">
  A
</div>
<div v-else-if="type === 'B'">
  B
</div>
<div v-else-if="type === 'C'">
  C
</div>
<div v-else>
  Not A/B/C
</div>
```

NOTA: Un elemento v-else debe seguir inmediatamente a un elemento v-if o v-else-if; de lo contrario, no será reconocido. De manera similar a v-else, un elemento v-else-if debe seguir inmediatamente a un elemento v-if o a v-else-if

- Controlar elementos reutilizables con **key**

Vue intenta representar elementos de la manera más eficiente posible, a menudo reutilizándolos en lugar de renderizarlos desde cero.
Más allá de ayudar a Vue a ser muy rápido, esto puede tener algunas ventajas útiles. Por ejemplo, si permite a los usuarios alternar entre varios tipos de inicio de sesión:

```
<template v-if="loginType === 'username'">
  <label>Username</label>
  <input placeholder="Enter your username">
</template>
```

```
<template v-else>
  <label>Email</label>
  <input placeholder="Enter your email address">
</template>
```

Así, al cambiar el loginType en el código anterior no se borrará lo que el usuario ya ha ingresado. Como ambas plantillas usan los mismos elementos, el <input> no se reemplaza, solo su marcador de posición.

Sin embargo, esto no siempre es deseable, por lo que Vue le ofrece una forma de decir: "Estos dos elementos están completamente separados, no los reutilice".
Consiste en agregar un atributo **key** con valores únicos:

```
<template v-if="loginType === 'username'">
  <label>Username</label>
  <input placeholder="Enter your username" key="username-input">
</template>
<template v-else>
  <label>Email</label>
  <input placeholder="Enter your email address" key="email-input">
</template>
```

Ahora esas entradas se renderizarán desde cero cada vez que se alterna

Tenga en cuenta que los elementos <label> aún se reutilizan de manera eficiente, ya que no tienen atributos clave.

- **v-show**

Otra opción para mostrar condicionalmente un elemento es la directiva v-show. El uso es básicamente el mismo:

```
<h1 v-show="ok">Hello!</h1>
```

La diferencia es que un elemento con **v-show** siempre se representará y permanecerá en el DOM; v-show solo alterna la propiedad CSS de visualización del elemento.

Tenga en cuenta que **v-show** no admite el elemento <template>, ni funciona con v-else.

COMPARATIVA v-if VS v-show
v-if es una representación condicional "real" porque garantiza que los detectores(listener) de eventos y los componentes secundarios dentro del bloque condicional se destruyan y recreen correctamente durante las alternancias(toggles).

v-if también es “perezoso” (lazy): si la condición es falsa en el procesamiento inicial, no hará nada, el bloque condicional no se representará hasta que la condición se vuelva verdadera por primera vez.

En comparación, **v-show** es mucho más simple: el elemento siempre se representa independientemente de la condición inicial, con alternancia basada en CSS.

En términos generales, **v-if** tiene mayores costos de alternancia, mientras **que v-show** tiene costos iniciales de procesamiento más altos. Por lo tanto, prefiera v-show si necesita alternar algo con mucha frecuencia, y prefiera **v-if** si es poco probable que la condición cambie en el tiempo de ejecución.

- **v-if con v-for**

Al utilizar juntos v-if y f-for, v-for tiene mayor prioridad .

6.3.- Gestionar input del usuario

Para que los usuarios puedan interactuar con tu aplicación, podemos utilizar la directiva `v-on` para conectar capturadores de eventos que invocan métodos en nuestras instancias de Vue:

.HTML

```
<div id="app-5" mark="crwd-mark">
 <p>{{ message }}</p>
 <button v-on:click="reverseMessage">Reverse Message</button>
</div>
```

.JS

```
var app5 = new Vue({
 el: '#app-5',
 data: {
 message: 'Hello Vue.js!'
 },
 methods: {
 reverseMessage: function () {
 this.message = this.message.split('').reverse().join('')
 }
 }
})
```

Tener en cuenta que en el método únicamente actualizamos el estado de nuestra aplicación sin tocar el DOM - Vue gestiona todas las manipulaciones del DOM, permitiéndote escribir código centrado únicamente en la capa de lógica subyacente.

Vue también ofrece la directiva `v-model` que hace muy sencillo el enlace bidireccional entre el input del formulario y el estado de la aplicación:

.HTML

```
<div id="app-6" mark="crwd-mark">
 <p>{{ message }}</p>
 <input v-model="message">
</div>
```

.JS

```
var app6 = new Vue({
 el: '#app-6',
 data: {
 message: 'Hello Vue!'
 }
})
```

6.4. Propiedades computadas

Las expresiones en las plantillas (templates) son muy prácticas, pero están pensadas para operaciones simples. Poner demasiada lógica en sus plantillas puede hacerlas demasiado grandes y difíciles de mantener. Por ejemplo:

```
<div id="example">
  {{ message.split('').reverse().join('') }}
</div>
```

En este punto, la plantilla ya no es simple y declarativa. Tienes que mirarlo por un segundo antes de darte cuenta de que muestra un mensaje al revés. El problema empeora cuando desea incluir el mensaje revertido en su plantilla más de una vez.

Es por eso que para cualquier lógica compleja, debe usar una propiedad calculada.

```
var vm = new Vue({
  el: '#example',
  data: {
    message: 'Hello'
  },
  computed: {
    // a computed getter
    reversedMessage: function () {
      // `this` points to the vm instance
      return this.message.split('').reverse().join('')
    }
  }
})
```

Aquí hemos declarado una propiedad calculada `reversedMessage`. La función que proporcionamos se usará como la función de obtención para la propiedad `vm.reversedMessage`

```
console.log(vm.reversedMessage) // => 'olleH'
vm.message = 'Goodbye'
console.log(vm.reversedMessage) // => 'eybdooG'
```

Puede enlazar datos a propiedades calculadas en plantillas como una propiedad normal.
Vue es consciente de que vm.reversedMessage depende de vm.message, por lo que actualizará cualquier enlace que dependa de vm.reversedMessage cuando vm.message cambie. Y la mejor parte es que hemos creado esta relación de dependencia de manera declarativa: la función de obtención calculada no tiene efectos secundarios, lo que hace que sea más fácil de probar y comprender.

6.4.1.- MÉTODOS VS PROPIEDADES COMPUTADAS

Habrás notado que podemos lograr el mismo resultado al invocar un método en la expresión:

```
<p>Reversed message: "{{ reverseMessage() }}"</p>
```

```
// in component
methods: {
  reverseMessage: function () {
    return this.message.split('').reverse().join('')
  }
}
```

En lugar de una propiedad calculada, podemos definir la misma función como un método.

Para el resultado final, los dos enfoques son exactamente los mismos. Sin embargo, la diferencia es que las propiedades calculadas se almacenan en caché en función de sus dependencias. Una propiedad calculada solo volverá a ser evaluada cuando algunas de sus dependencias hayan cambiado. Esto significa que mientras el mensaje no haya cambiado, el acceso múltiple a la propiedad calculada de reversedMessage devolverá inmediatamente el resultado calculado previamente sin tener que ejecutar la función nuevamente.

Esto también significa que la siguiente propiedad calculada nunca se actualizará, porque Date.now () no es una dependencia reactiva:

```
computed: {
  now: function () {
    return Date.now()
  }
}
```

En comparación, una invocación a un método siempre ejecutará la función siempre que ocurra una re-renderización.

¿Por qué necesitamos el almacenamiento en caché? Imagina que tenemos una propiedad calculada A costosa, que requiere un bucle a través de una gran array y hacer una gran cantidad de cálculos. Entonces podemos tener otras propiedades calculadas que a su vez dependen de A. ¡Sin almacenamiento en caché, estaríamos ejecutando el método para obtener A muchas más veces de las necesarias! En los casos en que no desee el almacenamiento en caché, utilice un método (method) en su lugar.

6.4.2.-Propiedad Computada vs Observada

Vue proporciona una forma más genérica de observar y reaccionar a los cambios de datos en una instancia de Vue: las propiedades observadas (watch).

Cuando tiene algunos datos que deben cambiar en función de algunos otros datos, es tentador abusar de watch. Sin embargo, a menudo es mejor idea usar una propiedad calculada. Considera este ejemplo:

```
<div id="demo">{{ fullName }}</div>
```

```
var vm = new Vue({
  el: '#demo',
  data: {
    firstName: 'Foo',
    lastName: 'Bar',
    fullName: 'Foo Bar'
  },
  watch: {
    firstName: function (val) {
      this.fullName = val + ' ' + this.lastName
    },
    lastName: function (val) {
      this.fullName = this.firstName + ' ' + val
    }
  }
})
```

El código anterior es imperativo y repetitivo. Si lo comparamos con una versión de propiedad calculada:

```
var vm = new Vue({
  el: '#demo',
  data: {
    firstName: 'Foo',
    lastName: 'Bar'
  },
  computed: {
    fullName: function () {
      return this.firstName + ' ' + this.lastName
    }
  }
})
```

Es mucho mejor

6.4.3.- Computed setter

Las propiedades calculadas son, por defecto, sólo para obtención (getter), pero también puede proporcionar el establecimiento de un valor (setter) cuando lo necesite:

```
// ...
computed: {
  fullName: {
    // getter
    get: function () {
      return this.firstName + ' ' + this.lastName
    },
    // setter
    set: function (newValue) {
      var names = newValue.split(' ')
      this.firstName = names[0]
      this.lastName = names[names.length - 1]
    }
  }
}
// ...
```

Ahora cuando ejecuta vm.fullName = 'John Doe', se invocará al setter y se actualizarán vm.firstName y vm.lastName en consecuencia.

6.4.4.- Propiedades observadas (watchers)

Si bien las propiedades calculadas son más apropiadas en la mayoría de los casos, hay ocasiones en que es necesario un observador personalizado.

Es por eso que Vue proporciona una forma más genérica de reaccionar a los cambios de datos a través de la opción de **watch**.

Esto es más útil cuando desea realizar operaciones asíncronas o costosas en respuesta a datos cambiantes. Por ejemplo:

```
<div id="watch-example">
  <p>
    Ask a yes/no question:
    <input v-model="question">
  </p>
  <p>{{ answer }}</p>
</div>
```

```
<!-- Since there is already a rich ecosystem of ajax libraries    -->
<!-- and collections of general-purpose utility methods, Vue core -->
<!-- is able to remain small by not reinventing them. This also   -->
<!-- gives you the freedom to use what you're familiar with. -->
<script src="https://cdn.jsdelivr.net/npm/axios@0.12.0/dist/axios.min.js"></script>
<script src="https://cdn.jsdelivr.net/npm/lodash@4.13.1/lodash.min.js"></script>
<script>
var watchExampleVM = new Vue({
  el: '#watch-example',
  data: {
    question: '',
    answer: 'I cannot give you an answer until you ask a question!'
  },
  watch: {
    // whenever question changes, this function will run
    question: function (newQuestion, oldQuestion) {
      this.answer = 'Waiting for you to stop typing...'
      this.getAnswer()
    }
  },
  methods: {
    // _.debounce is a function provided by lodash to limit how
    // often a particularly expensive operation can be run.
    // In this case, we want to limit how often we access
    // yesno.wtf/api, waiting until the user has completely
    // finished typing before making the ajax request. To learn
    // more about the _.debounce function (and its cousin
    // _.throttle), visit: https://lodash.com/docs#debounce
    getAnswer: _.debounce(
      function () {
        if (this.question.indexOf('?') === -1) {
```

```
          this.answer = 'Questions usually contain a question mark. ;-)'
          return
        }
        this.answer = 'Thinking...'
        var vm = this
        axios.get('https://yesno.wtf/api')
          .then(function (response) {
            vm.answer = _.capitalize(response.data.answer)
          })
          .catch(function (error) {
            vm.answer = 'Error! Could not reach the API. ' + error
          })
      },
      // This is the number of milliseconds we wait for the
      // user to stop typing.
      500
    )
  }
})
</script>
```

En este caso, usar la opción watch nos permite realizar una operación asincrónica (acceder a una API), limitar la frecuencia con la que realizamos esa operación y establecer estados intermedios hasta que obtengamos una respuesta final. Nada de eso sería posible con una propiedad calculada.

6.5.- Renderizado de listas (v-for)

6.5.1.- Mapeo de un array a Elements con v-for

Podemos usar la directiva v-for para generar una lista de elementos basada en un array. La directiva v-for requiere una sintaxis especial en forma de elemento en los elementos, donde los elementos son el array de datos de origen y el elemento es un alias para el elemento del array que en el que se itera:

```
<ul id="example-1">
  <li v-for="item in items">
    {{ item.message }}
  </li>
</ul>
```

```
var example1 = new Vue({
  el: '#example-1',
  data: {
    items: [
      { message: 'Foo' },
      { message: 'Bar' }
    ]
```

```
  }
})
```

Dentro de los bloques v-for tenemos acceso completo a las propiedades de ámbito del padre. v-for también admite un segundo argumento opcional para el índice del elemento actual

```
<ul id="example-2">
  <li v-for="(item, index) in items">
    {{ parentMessage }} - {{ index }} - {{ item.message }}
  </li>
</ul>
```

```
var example2 = new Vue({
  el: '#example-2',
  data: {
    parentMessage: 'Parent',
    items: [
      { message: 'Foo' },
      { message: 'Bar' }
    ]
  }
})
```

También puede usar of como delimitador en lugar de in, para que esté más cerca de la sintaxis de JavaScript para los iteradores:

```
<div v-for="item of items"></div>
```

6.5.2.- v-for con un Objeto

También puede usar v-for para recorrer las propiedades de un objeto.

```
<ul id="v-for-object" class="demo">
  <li v-for="value in object">
    {{ value }}
  </li>
</ul>
```

```
new Vue({
  el: '#v-for-object',
  data: {
```

```
    object: {
      firstName: 'John',
      lastName: 'Doe',
      age: 30
    }
  }
})
```

También puede proporcionar un segundo argumento para la clave (key):

```
<div v-for="(value, key) in object">
  {{ key }}: {{ value }}
</div>
```

```
<div v-for="(value, key, index) in object">
  {{ index }}. {{ key }}: {{ value }}
</div>
```

NOTA:

Al iterar sobre un objeto, el orden se basa en el orden de enumeración de claves de Object.keys (), que no se garantiza que sea coherente en todas las implementaciones del motor de JavaScript.

6.5.3.- Utilizando key

Cuando Vue está actualizando una lista de elementos representados con v-for, de manera predeterminada usa una estrategia de "in place patch". Si el orden de los elementos de datos ha cambiado, en lugar de mover los elementos DOM para que coincidan con el orden de los elementos(items), Vue corregirá cada elemento en su lugar y se asegurará de que refleje lo que se debe representar en ese índice en particular.

Este modo predeterminado es eficiente, pero solo es adecuado cuando su resultado de representación de lista no depende del estado del componente hijo o del estado temporal deDOM (por ejemplo, valores de entrada del formulario).

Para darle a Vue una pista para que pueda rastrear la identidad de cada nodo, y así reutilizar y reordenar los elementos existentes, debe proporcionar un atributo clave único (**key**) para cada elemento. Un valor ideal para la clave sería la identificación única de cada elemento:

```
<div v-for="item in items" :key="item.id">
  <!-- content -->
</div>
```

Se recomienda proporcionar una clave(key) con v-for siempre que sea posible, a menos que el contenido iterado de DOM sea simple o dependa intencionalmente del comportamiento predeterminado para obtener mejoras en el rendimiento.

6.5.4.- Detección de cambios en el array

MÉTODOS DE MUTACIÓN

Vue incluye los métodos de mutación de un array observado para que también activen actualizaciones de vista. Los métodos incluidos son:

- push( )
- pop()
- Shift()
- unshift()
- splice()
- sort()
- reverse()

Los métodos de mutación, como su nombre indica, mutan la array original en la que se invocaron. En comparación, también hay métodos no mutantes, por ejemplo. filter (), concat () y slice(), que no modifican la array original pero siempre devuelven una nueva array. Al trabajar con métodos no mutantes, puede reemplazar la array anterior por la nueva

```
example1.items = example1.items.filter(function (item) {
    return item.message.match(/Foo/)
})
```

Podría pensar que esto causará que Vue se deshaga del DOM existente y vuelva a procesar la lista completa; afortunadamente, ese no es el caso. Vue implementa algunas heurísticas inteligentes para maximizar la reutilización de elementos DOM, por lo que reemplazar un array con otro array que contenga objetos superpuestos es una operación muy eficiente.
Debido a las limitaciones en JavaScript, Vue no puede detectar los siguientes cambios en una array:

1.- Cuando establece directamente un elemento con el índice, por ejemplo:

```
vm.items [indexOfItem] = newValue
```

2.- Cuando modifica la longitud de la array, por ejemplo

```
vm.items.length = newLength
```

Para superar la advertencia 1, ambos siguientes harán lo mismo que
vm.items [indexOfItem] = newValue,
pero también activarán las actualizaciones de estado en el sistema de reactividad:

```
/ Vue.set
Vue.set(example1.items, indexOfItem, newValue)

// Array.prototype.splice
example1.items.splice(indexOfItem, 1, newValue)
```

Para manejar la advertencia 2, podemos usar splice:

```
example1.items.splice(newLength)
```

6.5.5.- Advertencias(caveats) de detección de cambios en Objetos

Nuevamente debido a las limitaciones del JavaScript moderno, Vue no puede detectar la adición o eliminación de propiedades. Por ejemplo:

```
var vm = new Vue({
    data: {
        a: 1
    }
})
// `vm.a` is now reactive
vm.b = 2
// `vm.b` is NOT reactive
```

Vue no permite agregar dinámicamente nuevas propiedades reactivas de nivel de raíz a una instancia ya creada. Sin embargo, es posible agregar propiedades reactivas a un objeto anidado utilizando el método `Vue.set (objeto, clave, valor).` Por ejemplo, dado:

```
var vm = new Vue({
    data: {
        userProfile: {
            name: 'Anika'
        }
    }
})
```

Puede agregar una nueva propiedad de edad al objeto anidado userProfile con:

```
Vue.set(vm.userProfile, 'age', 27)
```

También puede usar el método de instancia vm.$ Set, que es un alias para el Vue.set global:

```
vm.$set(vm.userProfile, 'age', 27)
```

A veces puede querer asignar un número de propiedades nuevas a un objeto existente, por ejemplo, usando Object.assign () o _.extend (). En tales casos, debe crear un objeto nuevo con propiedades de ambos objetos. Entonces, en lugar de

```
Object.assign(vm.userProfile, {
    age: 27,
    favoriteColor: 'Vue Green'
})
```

emplearás:

```
vm.userProfile = Object.assign({}, vm.userProfile, {
    age: 27,
    favoriteColor: 'Vue Green'
})
```

6.5.6.- Aplicación de filtros y ordenación de resultados

A veces queremos mostrar una versión filtrada u ordenada de una array sin realmente mutar o restablecer los datos originales. En este caso, puede crear una propiedad calculada que devuelva el array filtrado u ordenado. Por ejemplo:

```
<li v-for="n in evenNumbers">{{ n }}</li>
```

```
data: {
    numbers: [ 1, 2, 3, 4, 5 ]
},
computed: {
    evenNumbers: function () {
        return this.numbers.filter(function (number) {
            return number % 2 === 0
        })
    }
}
```

En situaciones en las que las propiedades calculadas no son factibles (por ejemplo, dentro de bucles v-for anidados), puede usar un método:

```
<li v-for="n in even(numbers)">{{ n }}</li>
data: {
    numbers: [ 1, 2, 3, 4, 5 ]
},
methods: {
    even: function (numbers) {
        return numbers.filter(function (number) {
            return number % 2 === 0
        })
    }
}
```

6.5.7.- v-for en un rango

v-for también puede tomar un número entero. En este caso, repetirá la plantilla muchas veces.

```
<div>
    <span v-for="n in 10">{{ n }} </span>
</div>
```

Resultado:

1 2 3 4 5 6 7 8 9 10

6.5.8.- v-for en un `<template>`

Similar a la plantilla v-if, también puede usar una etiqueta <template> con v-for para representar un bloque de elementos múltiples. Por ejemplo:

```
<ul>
    <template v-for="item in items">
        <li>{{ item.msg }}</li>
        <li class="divider"></li>
    </template>
</ul>
```

6.5.9.- v-for con v-if

Cuando existen en el mismo nodo, v-for tiene una prioridad más alta que v-if. Eso significa que v-if se ejecutará en cada iteración del ciclo por separado. Esto puede ser útil cuando desea generar nodos para solo algunos elementos, como a continuación:

```
<li v-for="todo in todos" v-if="!todo.isComplete">
    {{ todo }}
</li>
```

Lo anterior solo representa los que no están completos. Si, en cambio, su intención es omitir condicionalmente la ejecución del ciclo, puede colocar el v-if en un elemento contenedor (o <plantilla>). Por ejemplo:

```
<ul v-if="todos.length">
    <li v-for="todo in todos">
        {{ todo }}
    </li>
</ul>
<p v-else>No todos left!</p>
```

6.5.10.- v-for con un Componente

Puede usar directamente v-for en un componente personalizado, como cualquier elemento normal:

```
<my-component v-for="item in items" :key="item.id"></my-component>
```

Sin embargo, esto no pasará automáticamente ningún dato al componente, porque los componentes tienen sus propios alcances aislados. Para pasar los datos iterados al componente, también deberíamos usar props:

```
 <my-component
    v-for="(item, index) in items"
    v-bind:item="item"
    v-bind:index="index"
    v-bind:key="item.id"
></my-component>
```

La razón por la que no se inyecta automáticamente un elemento en el componente es porque eso hace que el componente esté estrechamente relacionado con el funcionamiento de v-for.

Ser explícito acerca de dónde provienen sus datos hace que el componente sea reutilizable en otras situaciones. Aquí hay un ejemplo completo de una lista de tareas pendientes simple:

.html

```
<div id="todo-list-example">
    <input
        v-model="newTodoText"
        v-on:keyup.enter="addNewTodo"
        placeholder="Add a todo"
    >
    <ul>
        <li
            is="todo-item"
            v-for="(todo, index) in todos"
            v-bind:key="todo.id"
            v-bind:title="todo.title"
            v-on:remove="todos.splice(index, 1)"
        ></li>
    </ul>
</div>
```

Tenga en cuenta el atributo is = "todo-item". Esto es necesario en las plantillas DOM, porque solo un elemento <li> es válido dentro de un <ul>. Hace lo mismo que <todo-item>, pero soluciona un posible error de análisis del navegador.

.js

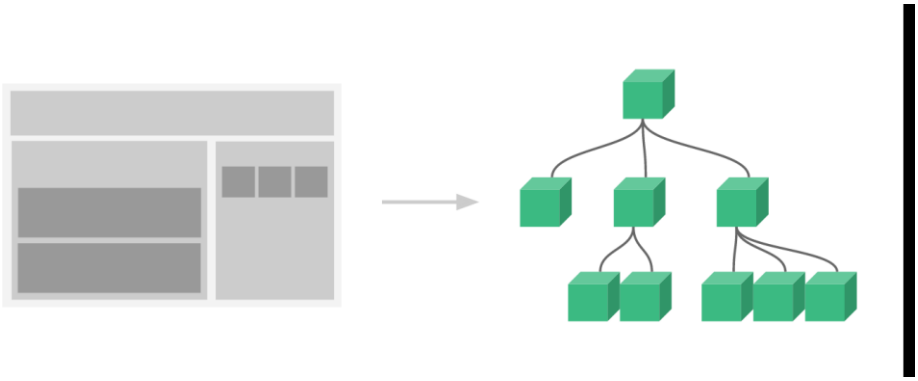
```
Vue.component('todo-item', {
    template: '\
        <li>\
            {{ title }}\
            <button v-on:click="$emit(\'remove\')">X</button>\
        </li>\
    ',
    props: ['title']
})
new Vue({
    el: '#todo-list-example',
    data: {
        newTodoText: '',
        todos: [
            {
```

```
                id: 1,
                title: 'Do the dishes',
            },
            {
                id: 2,
                title: 'Take out the trash',
            },
            {
                id: 3,
                title: 'Mow the lawn'
            }
        ],
        nextTodoId: 4
    },
    methods: {
        addNewTodo: function () {
            this.todos.push({
                id: this.nextTodoId++,
                title: this.newTodoText
            })
            this.newTodoText = ''
        }
    }
})
```

6.6.- Componentes

El sistema de componentes es otro concepto importante en Vue, porque es una abstracción que nos permite construir aplicaciones a gran escala compuestas de componentes pequeños, autocontenidos y, a menudo, reutilizables.

Si lo pensamos bien, casi cualquier tipo de interfaz de aplicación se puede resumir en un árbol de componentes:



En Vue, un componente es esencialmente una instancia de Vue con opciones predefinidas. El registro de un componente en Vue es sencillo.

.HTML

```
// Define a new component called todo-item
Vue.component('todo-item', {
 template: '<li>This is a todo</li>'
})
```

Ahora puede componerlo en la plantilla de otro componente

```
<ol mark="crwd-mark">
 <!-- Create an instance of the todo-item component -->
 <todo-item></todo-item>
```

```
</ol>
```

Pero esto representaría el mismo texto para cada tarea, lo cual no es muy interesante.
Deberíamos poder pasar datos desde el alcance principal a los componentes secundarios.
Modifiquemos la definición del componente para que acepte una propo:

.JS

```
Vue.component('todo-item', {
 // The todo-item component now accepts a
 // "prop", which is like a custom attribute.
 // This prop is called todo.
 props: ['todo'],
 template: '<li>{{ todo.text }}</li>'
})
```

Ahora podemos pasar el "todo" a cada componente repetido usando v-bind:

.HTML

```
<div id="app-7" mark="crwd-mark">
 <ol>
 <!-- Now we provide each todo-item with the todo object -->
 <!-- it's representing, so that its content can be dynamic -->
 <todo-item v-for="item in groceryList" v-bind:todo="item"></todo-item>
 </ol>
</div>
```

.JS

```
Vue.component('todo-item', {
 props: ['todo'],
 template: '<li>{{ todo.text }}</li>'
})

var app7 = new Vue({
 el: '#app-7',
 data: {
 groceryList: [
 { text: 'Vegetables' },
 { text: 'Cheese' },
 { text: 'Whatever else humans are supposed to eat' }
 ]
 }
})
```

Este es solo un ejemplo artificial, pero hemos logrado separar nuestra aplicación en dos unidades más pequeñas, y el "hijo" está razonablemente bien desacoplado del "padre" a través de la interfaz props.
Ahora podemos mejorar aún más nuestro componente <todo-item> con plantillas y lógica más complejas sin afectar la aplicación principal (parent app).
En una aplicación grande, es necesario dividir la aplicación en componentes para que el desarrollo sea más manejable.
Hablaremos mucho más acerca de componentes más adelante, pero dejamos aquí un ejemplo (imaginario) de plantilla de lo que una aplicación podría parecer con componentes:

.HTML

```
<div id="app" mark="crwd-mark">
 <app-nav></app-nav>
 <app-view>
 <app-sidebar></app-sidebar>
 <app-content></app-content>
 </app-view>
</div>
```

6.6.1.-Relación con elementos personalizados

Puede haber notado que los componentes Vue son muy similares a los Elementos personalizados (**Custom Elements**), que son parte de la Especificación de componentes web (Web Components Spec).

Esto se debe a que la sintaxis de los componentes de Vue está modelada de manera flexible después de la especificación.
Por ejemplo, los componentes Vue implementan el Slot API y el atributo es especial.
Sin embargo, hay algunas diferencias clave:
1. La especificación de componentes web todavía está en borrador, y no se implementa de forma nativa en cada navegador.
2. En comparación, los componentes Vue no requieren ningún polyfills y funcionan de manera consistente en todos los navegadores soportados (IE9 y superior).
3. Cuando sea necesario, los componentes Vue también pueden envolverse dentro de un elemento personalizado nativo.
4. Los componentes de Vue proporcionan características importantes que no están disponibles en elementos personalizados simples, más notablemente flujo de datos de componentes cruzados, comunicación personalizada de eventos e integraciones de herramientas de compilación.

Si quieres ampliar más sobre componentes tienes un ejemplo en ANEXO II

7.- CLASES Y ESTILOS

Este apartado vamos a dedicarlo a explicar cómo podemos incluir o quitar clases o estilos dependiendo del estado de nuestra aplicación.

7.1.- 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:

7.1.1.- 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.

7.1.2.-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>
`
});
```

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.

7.2.- 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.

7.2.1.- 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

Ver un ejemplo de aplicación en Anexo IV

ANEXO I.- HOLA MUNDO

A) SIN cli

Para incluir Vue en este caso tendremos que crear un archivo .html e incluir Vue así:
<script src="https://unpkg.com/vue/dist/vue.js" mark="crwd-mark"></script>

B) Con cli

Con este comando podemos inicializar un nuevo proyecto:

```
vue init webpack miaplicacion
```

Este comando tiene 4 partes

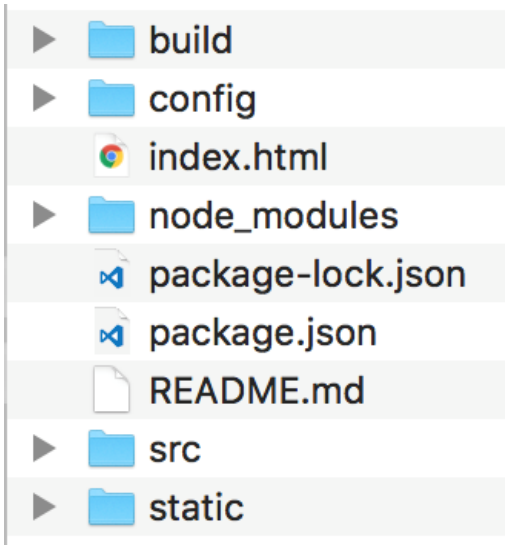
- El comando **vue,** lo usamos para llamar el CLI
- **init**, estamos diciéndole al CLI que queremos inicializar un nuevo proyecto.
- **webpack**, es la plantilla que queremos usar. Hay varias plantillas, más información al respecto más adelante
- **miaplicacion**, es el nombre que le queremos dar a nuestro proyecto

Para completar la creación del proyecto tenemos que contestar unas preguntas:
http://vuejs-templates.github.io/webpack

```
~: vue init webpack miaplicacion

? Project name miaplicacion
? Project description A Vue.js project
? Author Jonathan
? Vue build standalone
? Install vue-router? No
? Use ESLint to lint your code? No
? Setup unit tests with Karma + Mocha? No
? Setup e2e tests with Nightwatch? No

   vue-cli · Generated "miaplicacion".

   To get started:

     cd miaplicacion
     npm install
     npm run dev

   Documentation can be found at https://vuejs-templates.github.io/webpack
```

B.1.- Estructura de una aplicación Vue

La siguiente es la estructura de carpetas y archivos que podemos ver dentro de nuestra carpeta 'miaplicacion':



B.2.- Código

Los archivos importantes son los siguientes:

B.2.1.- index.html

Este archivo es la entrada a nuestra aplicación y tiene el siguiente código

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>miaplicacion</title>
  </head>
  <body>
    <div id="app"></div>
    <!-- built files will be auto injected -->
  </body>
</html>
```

En el código podemos ver un div con un `id="app"`. En este div es donde vamos a ver todo el código que Vue JS va a generar.

B.2.2.- src/main.js

En este archivo es donde nuestra aplicación se inicializa, este es el código:

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

Vue.config.productionTip = false

/* eslint-disable no-new */
new Vue({
  el: '#app',
  template: '<App/>',
  components: { App }
})
```

en la parte superior podemos ver

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

Vue es el nombre de la clase principal del framework
App es el nombre del componente raíz de nuestra aplicación
Luego creamos una nueva instancia de la clase Vue (usando new) y la inicializamos pasándole un objeto al constructor. Este objeto tiene 4 propiedades:

1. el: es el elemento (la etiqueta html) donde queremos mostrar el contenido de nuestra aplicación. En este caso estamos pasando '#app' que es el id del div que tenemos en index.html.
2. template: Es el código HTML que define nuestra aplicación (lo que queremos mostrar al usuario). En este caso solo vemos <App/> que no es un elemento de HTML. Este elemento está indicando que vamos a usar la plantilla (template) del componente App.

3. components: La lista de componentes que son necesarios en la plantilla (template).

B.2.3.-src/App.vue

En este archivo vemos la implementación del componente App, este es el código:

```
<template>
  <div id="app">
    <img src="./assets/logo.png">
    <hello></hello>
  </div>
</template>

<script>
export default {
  name: 'app',
  components: {
    Hello
  }
}
</script>

<style>
#app {
  font-family: 'Avenir', Helvetica, Arial, sans-serif;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
  text-align: center;
  color: #2c3e50;
  margin-top: 60px;
}
</style>
```

Este tipo de archivo se conoce como un componente de archivo simple (single-file component) porque definimos el html, js y css en el mismo archivo en las secciones template, script y style respectivamente. En la sección template estamos definiendo el código HTML que queremos usar en nuestra página. Luego definimos la sección script, en la cual tenemos un objeto, default, el cual estamos exportando. Este objeto tiene (por ahora) dos propiedades:
name: el nombre del id sobre el cual este componente va a actuar. En este caso es 'app' y en la sección template podemos ver como tenemos un div con el mismo id. El componente va a tener acceso a todo el div.
components: los componentes que el componente actual (en este caso el componente raíz) necesita (Dependencias).
De la misma forma que lo vimos con el archivo main.js, en App.vue estamos diciendo que necesitamos el componente Hello dentro del componente App. Este subcomponente lo podemos encontrar en la carpeta components:

B.2.4.-Hello.vue

```
<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>
```

Este archivo también es una plantilla de archivo simple con las tres secciones que vimos antes; template, script y style.
Y de la misma forma estamos exportando, en la sección script, el objeto de configuración del componente. En el componente Hello vemos esta vez un método llamado data. Este método retorna un objeto el cual representa el modelo del componente. Propiedades definidas en el modelo pueden ser usadas en el template del componente usando interpolación ({{ }}). Este modelo solo tiene una propiedad, llamada msg. Y estamos mostrando el contenido de esta variable en el template usando interpolación:

```
<h1>{{ msg }}</h1>
```

La interpolación requiere el uso de corchetes dobles con el nombre de la propiedad que queremos mostrar en nuestro template.

ANEXO II .-CURSOS

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.

A) 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.

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 documentot contábamos con 5 maneras:

- browserify: nos genera una plantilla con todo lo necesario para que el empaquetado e 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 ẁebpack 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:

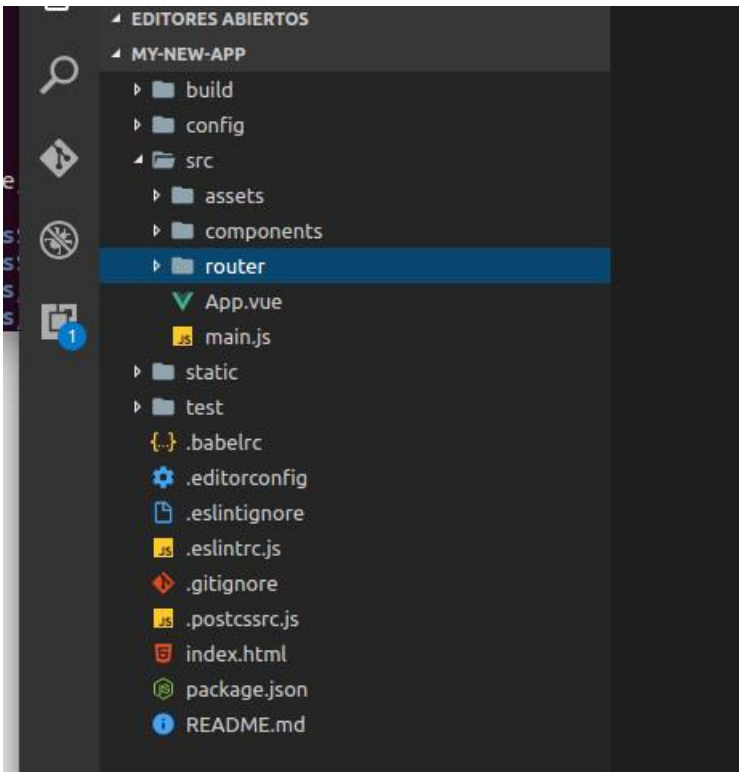
- ? Project name: nos deja elegir un nombre para el proyecto, podemos coger el que hemos indicado por defecto.
- ? Project description: una descripción que será incluida en el package.json del proyecto.
- ? Author: El auto a incluir en el package.json
- ? Runtime + Compiler or Runtime-only: nos deja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

B) 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.

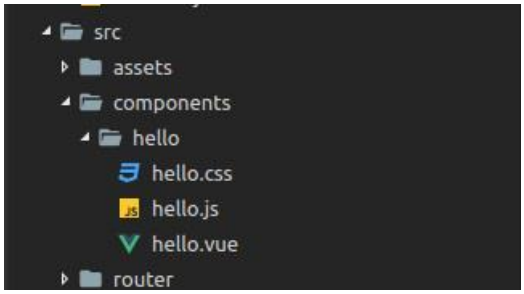
Una buena característica es que el loader de VueJS (la pieza encargada de compilar el código en webpack) tiene compatibilidad con otros motores de plantillas como pug, con preprocesadores css como SASS o LESS y con transpiladores como Babel o TypeScript, con lo que no estamos limitados por el framework.

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 maquetador, 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.

C) 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.

C.1.- Crear 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.

C.2.-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
});
```

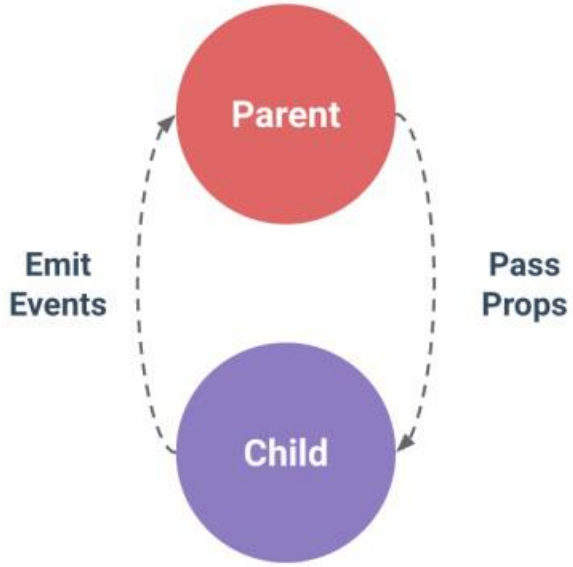
Estamos indicando, dentro del atributo `props` del objeto `options` , las propiedades de entrada que queremos que tenga nuestro componente, en nuestro caso son 3: title , subitle y description , al igual que en la función.
Estas propiedades, ahora, pueden ser usadas en su template. Es buena práctica 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.

C.3.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.

C.4.- 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>
                <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>
    `
};
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.

C.5.- 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>
    `
};
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>
    `
};
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>
    `,
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.

C.6.- 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>
    `
});
```

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 .

C.7.- Todo junto

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

```
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;
}
```

app.js

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

index.html

```
<!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>
```

ANEXO III.- LOGIN

Lo que hemos hecho es desarrollar una plantilla en VueJS sobre una vista típica de login. El marcado es el siguiente:

index.html

```
<!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>
```

App.js

```
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 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();
}
}
});
```

ANEXO VI.- COMBATE DE POKEMON

Hemos creado un pequeño juego “chorra” 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.
El ejemplo consta de estos tres ficheros:

app.css

```
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;
	}
```

app.js

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

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

index.html

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