Asco de Codigo

"Asquerosamente Remediable"

Introducción a VueJS

Julio 11, 2018

¿Qué es VueJS?

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

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

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

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

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

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

¿Qué caracteriza a VueJS?

Pero ¿qué define a VueJS? ¿Qué lo diferencia o lo asemeja al resto de alternativas? ¿Por qué se está poniendo tan de moda?

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

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

¿Cómo empezamos?

Para empezar, lo único que tendremos que hacer es incluir la dependencia de VueJS a

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

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

Hemos añadido la librería VueJS de desarrollo y no la minificada. Esto es así porque la librería de desarrollo nos lanzará un montón de advertencias y errores que nos ayudarán a aprender y trabajar con VueJS.

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

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

Primer ejemplo

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

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

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

<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
    <script src="app.js"></script>
</body>
</html>
```

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

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

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

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

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

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

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

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

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

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

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

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

Teniendo esto, tenemos el 50% hecho de nuestra "aplicación". Lo siguiente que vamos a ver es la definición de los tres componentes visuales en los que he dividido la interfaz. Empecemos por el componente `game-header` :

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

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

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

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

Miremos un poco en detalle:

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

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

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

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

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

El componente `game-item` recibe un modelo y lo pinta.
El sistema es reactivo, es decir que si yo inserto un nuevo elemento en el array de juegos, VueJS es lo suficientemente inteligente para saber que tiene que renderizar los elementos precisos.

En el siguiente ejemplo podemos ver todo junto:

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

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

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

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

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

Trabajando con templates

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

Podemos optar por guardar una serie de HTMLs estáticos que ya cuentan con la

información del usuario necesaria incrustada y almacenados dentro de base de datos. Este es el modelo que suelen desarrollar los CMS de muchos e-commerces o blogs personales.

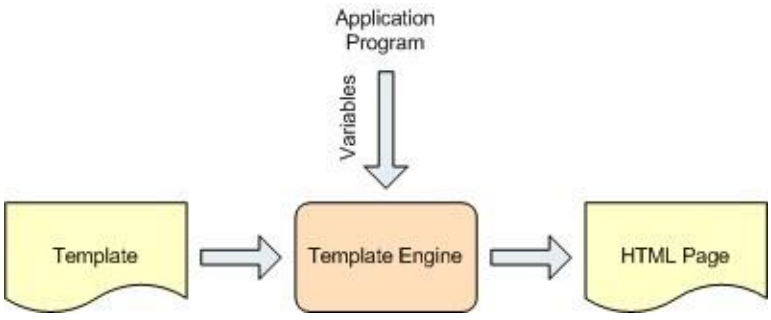
Otra opción sería la de renderizar o pintar estos datos dentro del HTML de manera dinámica. La idea subyace en crear una plantilla del HTML que queremos mostrar al usuario, dejando marcados aquellos huecos donde queremos que se pinten variables, modelos o entidades determinadas. Este es el sistema que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

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

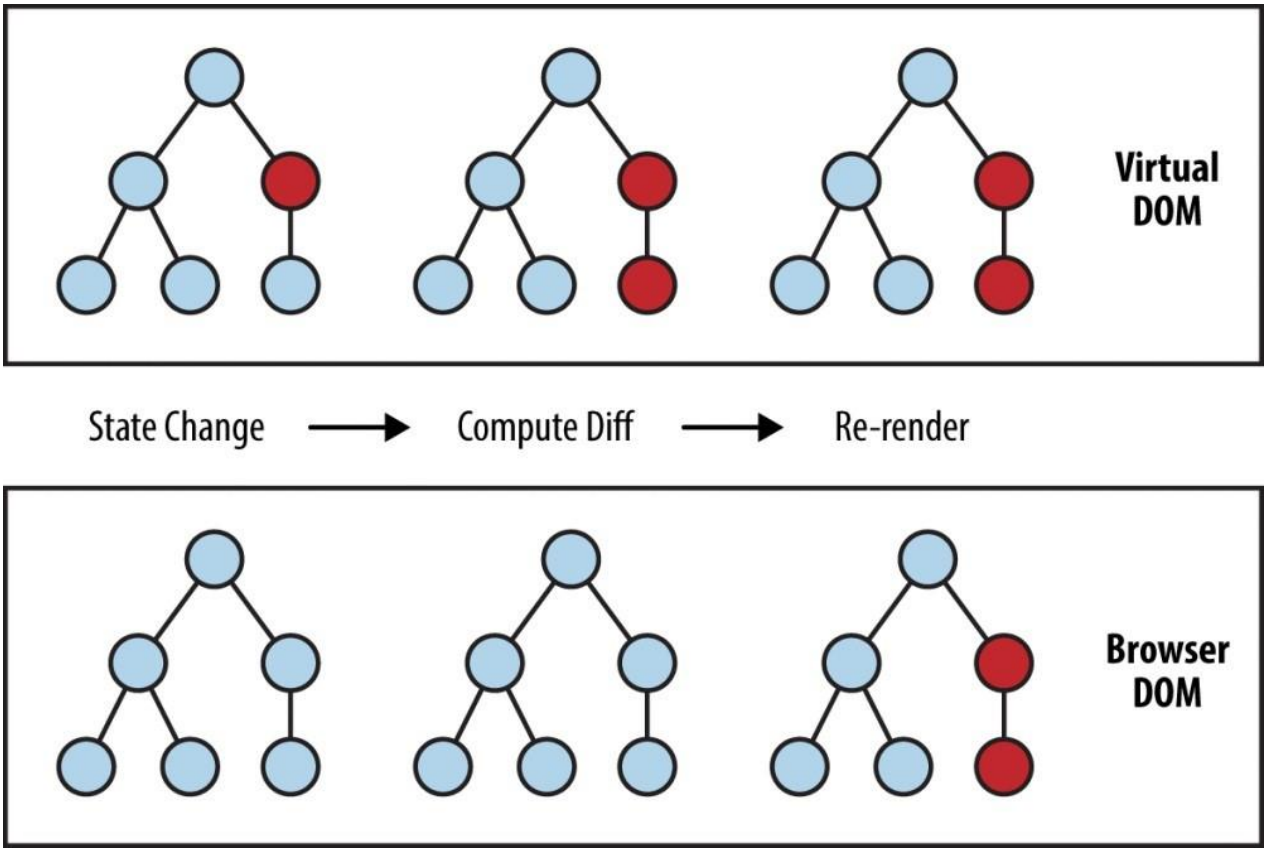
El post de hoy trata de explicar toda esta sintaxis mediante la explicación los conceptos de interpolación, directiva y filtro. Espero que os guste:

¿Cómo funciona?

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



La interpolación

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

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

Interpolando HTML

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

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

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

Interpolando atributos

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

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

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

Interpolando por medio de expresiones

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

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

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

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

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