ASCO DE CODIGO

"Asquerosamente Remediable"

Vue CLI

Julio 12, 2018

Definiendo componentes en un único fichero

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

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

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

Creando un proyecto con `vue-cli`

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

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

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

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

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

```
$ vue -V
```

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

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

```
$ vue list
```

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

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

Nosotros nos vamos a basar en la plantilla de 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.

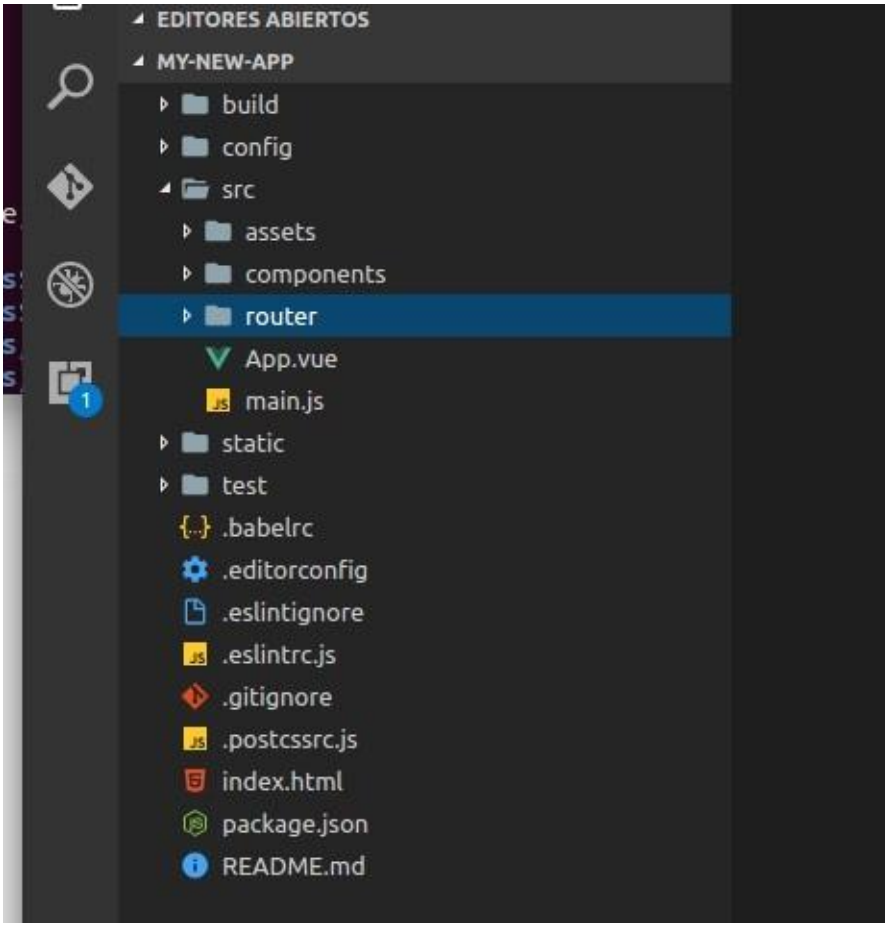
- **? Install vue-router**: Nos incluye un router por defecto en la solución y las dependencias necesarias.
- **? Use ESLint to lint your code**: Nos permite incluir un linter con la plantilla que deseemos para las reglas genéricas.
- **? Setup unit tests with Karma + Mocha**: Nos incluye las dependencias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

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.

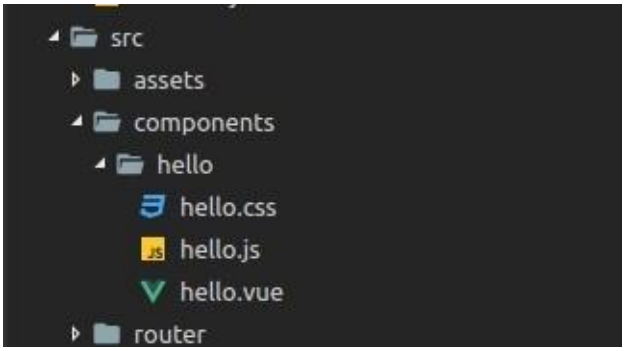
Una buena característica es que el loader de VueJS (la pieza encargada de compilar el código en webpack) tiene compatibilidad con otros motores de plantillas como pug, con preprocesadores css como SASS o LESS y con transpiladores como Babel o TypeScript, con lo que no estamos limitados por el framework.

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.

Conclusión

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

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

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

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