1.- INTRODUCCIÓN

1.1.- ¿Qué es Vue?

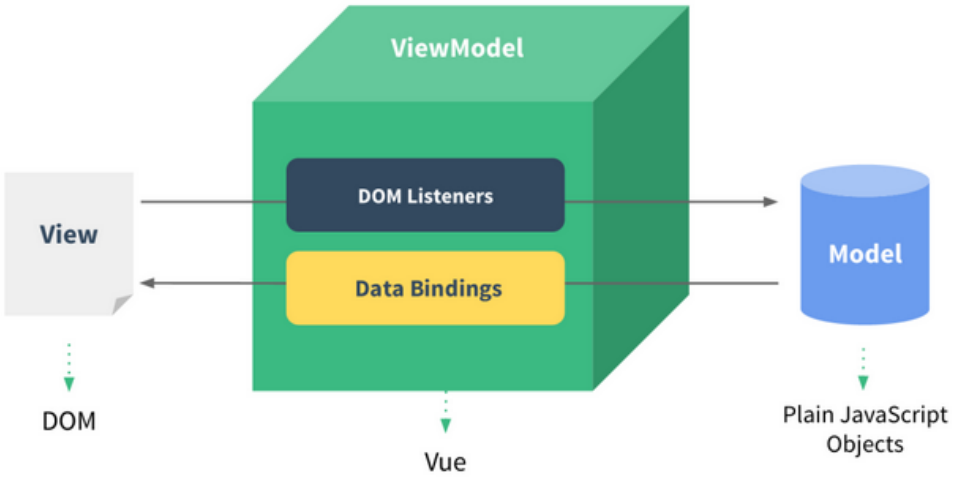
Vue (pronunciado /vju:/, como view en inglés) es un framework progresivo para la creación de interfaces de usuario.
A diferencia de otros frameworks monolíticos, Vue está diseñado desde la base para ser incrementalmente adaptable.
La librería básica se centra solamente en la capa de vista y es muy fácil 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.



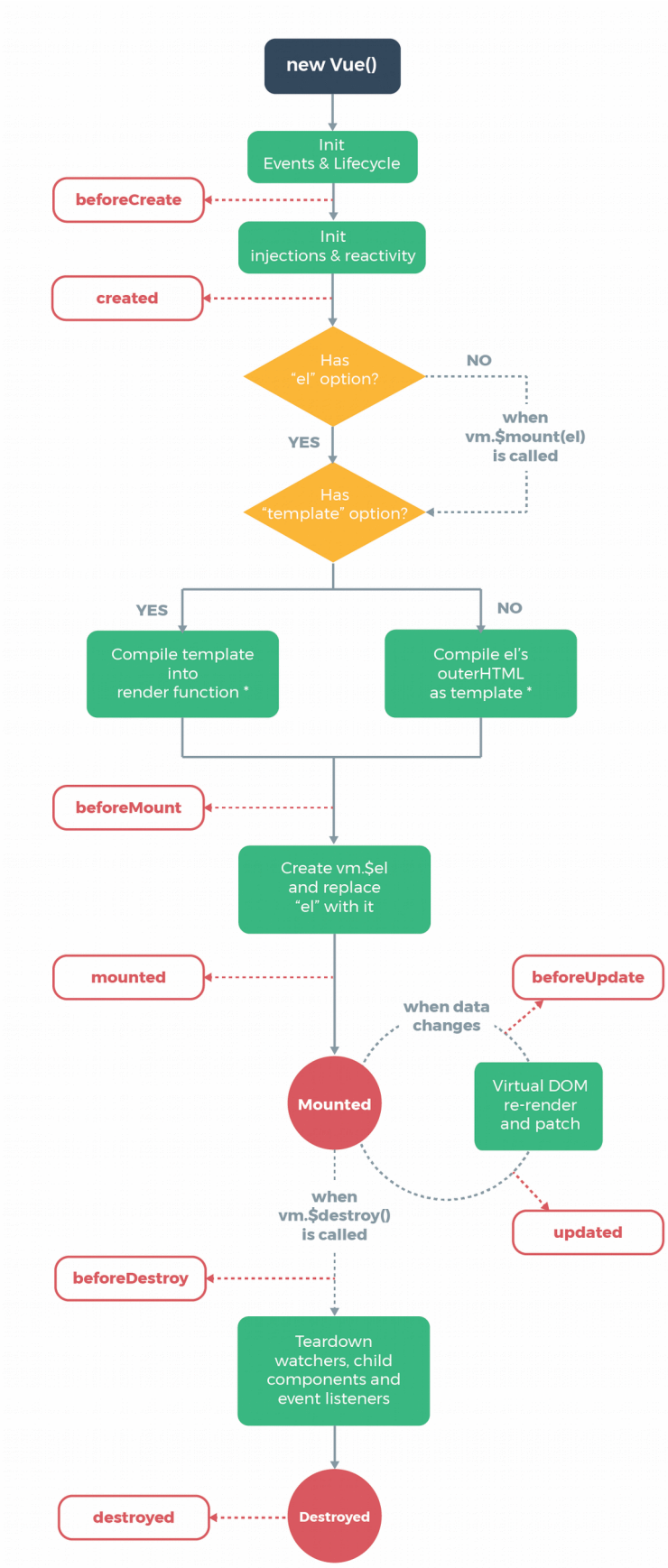
Vue.js es un framework de JavaScript nuevo, si lo comparamos con otros frameworks como Backbone o Ember. Sin embargo, su facilidad de aprendizaje y uso con respecto a otros frameworks y libraries como ReactJS, su rendimiento comparado con AngularJS y la facilidad para usarlo y adaptarlo a proyectos tanto 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

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

3.1.- EVENTOS (hooks):

before create: podemos acceder al objeto pero no al DOM, ya que todavía no está cargado
created
before mount
mounted
Before update
Upddated
Before destroy
Destroyed



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

4.-DESARROLLO DE APLICACIONES CON CLI

1.- El CLI lo podemos obtener de Github:
https://github.com/vuejs/vue-cli

FUENTE: https://github.com/vuejs/vue-cli/blob/dev/docs/cli.md

4.1.- CLI

4.1.1.- INSTALACIÓN

```
npm install -g @vue/cli
vue create my-project
```

4.1.2.- UTILIZACIÓN

Uso: `vue <command> [options]`

Commandos:

`create [options] <app-name>`	para crear un nuevo proyecto impulsado por vue-cli-service
`invoke <plugin> [pluginOptions]`	invocar el generador de un complemento en un proyecto ya creado
`inspect [options] [paths...]`	inspeccionar la configuración del paquete web en un proyecto con vue-cli-service
`serve [options] [entry]`	servir un archivo .js o .vue en modo de desarrollo con “zero config”
`build [options] [entry]`	construir un archivo .js o .vue en modo de producción con “zero config”
`init <template> <app-name>`	generar un proyecto a partir de una plantilla remota (API heredada, requiere @ vue / cli-init)

Para cada comando también podemos usar:
`vue <command> --help`
para ver un uso más detallado.

4.1.3.-Crear un nuevo proyecto

Uso: `create [options] <app-name>`
crear un nuevo proyecto impulsado por vue-cli-service

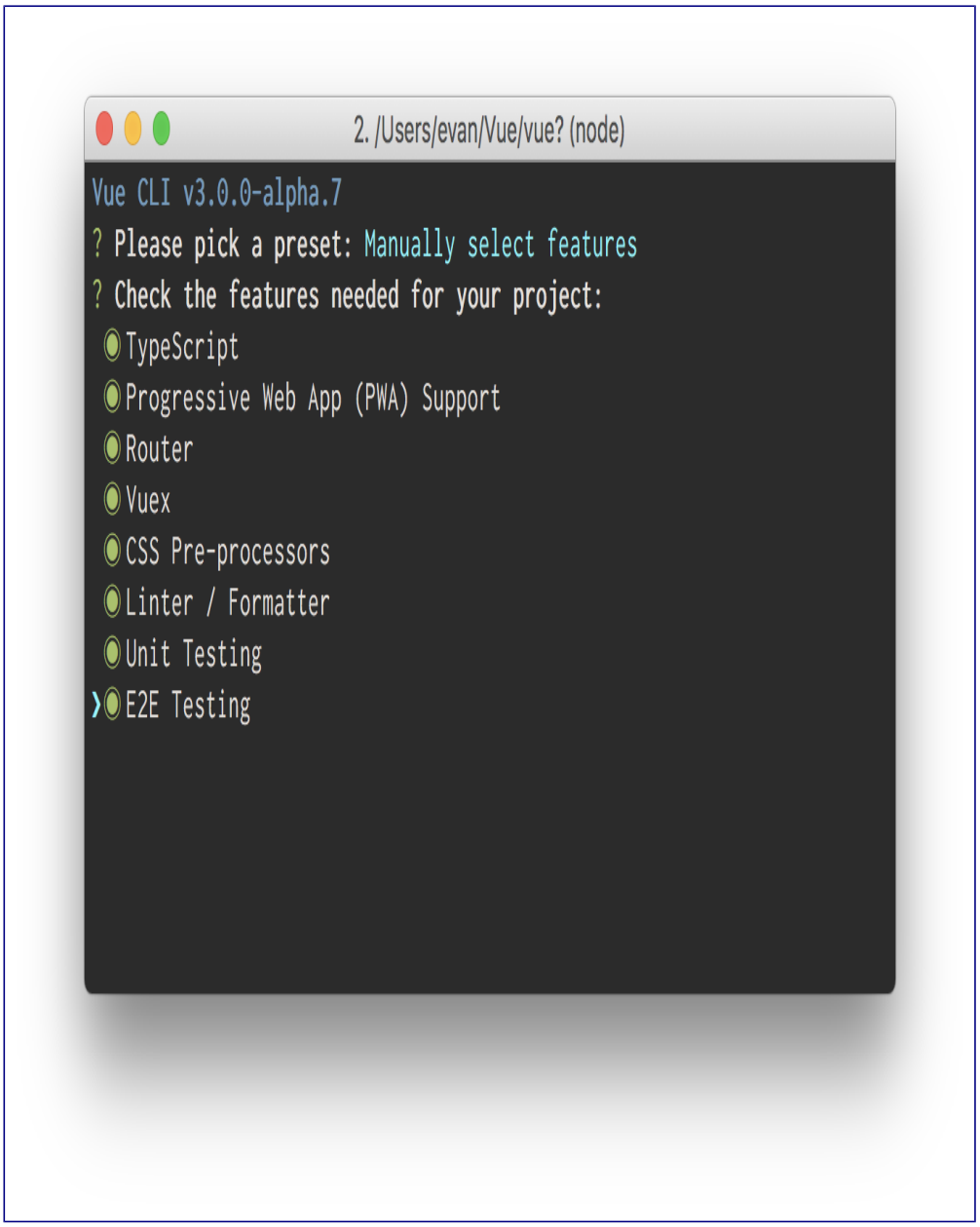
Options:

`-p, --preset <presetName>`	Omite las indicaciones y usa el preajuste guardado
`-d, --default`	Omitir indicaciones y usar preajuste predeterminado(por defecto)
`-i, --inlinePreset <json>`	Omite las indicaciones y usa cadena JSON que se le pasa como parámetro como preestablecido
`-m, --packageManager <command>`	Usar el cliente npm especificado al instalar

	dependencias.
`-r, --registry <url>`	Use el registro npm especificado al instalar dependencias (solo para npm)
`-f, --force`	Sobreescribe el directorio destino si existe
`-h, --help`	información de uso de salida

Ejemplo:

```
vue create my-project
```



4.1.3.1.-Preconfiguraciones o Presets

Después de que haya seleccionado las características, puede guardarlas opcionalmente como un ajuste preestablecido para que pueda reutilizarlo para proyectos futuros. Si desea eliminar un ajuste preestablecido guardado, puede hacerlo editando `~ / .vuerc.`

4.1.4.-Prototipado Zero-config

Puede prototipar rápidamente con un solo archivo * .vue con los comandos vue serve y vue build, pero requieren instalar primero un complemento global adicional:

```
npm install -g @vue/cli-service-global
```

El inconveniente de vue serve es que depende de dependencias instaladas globalmente que pueden ser inconsistentes en diferentes máquinas. Por lo tanto, esto solo se recomienda para la creación rápida de prototipos.

4.1.4.1.- vue serve

Uso: `serve [options] [entry]`

Proporciona un archivo .js o .vue en modo de desarrollo con zero config

Opciones:

```
 -o, --open Abre en navegador
 -h, --help Información de uso de la salida (ayuda para el comando)

All you need is a *.vue file:
```

```
echo '<template><h1>Hello!</h1></template>' > App.vue
vue serve
```

`vue serve` utiliza la misma configuración predeterminada (webpack, babel, postcss & eslint) que los proyectos creados por **vue create**. Automáticamente infiere el archivo de entrada en el directorio actual; la entrada puede ser una de `main.js, index.js, App.vue` o `app.vue`. También puede especificar explícitamente el archivo de entrada:

```
vue serve MyComponent.vue
```

Si es necesario, también puede proporcionar un **`index.html, package.json,`** instalar y usar dependencias locales, o incluso configurar babel, postcss & eslint con los archivos de configuración correspondientes.

4.1.4.2.-vue build

Uso:: build [options] [entry]

Proporciona un archivo .js o .vue en modo de producción con zero config

```
Opciones

  -t, --target <target> compila en un destino especificado (app | lib | wc |
wc-async, default: app)
  -n, --name <name> para nombrar un lib o un componente web (por defecto:
introducir nombre del archivo)
 -d, --dest <dir> directorio de salida (por defecto: dist)
 -h, --help  Información de uso de la salida (ayuda para el comando)
```

También puede construir el archivo de destino en un paquete de producción para la implementación con la versión vue:

```
vue build MyComponent.vue
```

vue build también proporciona la capacidad de construir el componente como una biblioteca o un componente web.

4.1.5.-Instalación de Plugins en un proyecto existente

Cada plugin CLI se envía con un generador (que crea archivos) y un complemento de tiempo de ejecución (que ajusta la configuración del paquete web principal e inyecta comandos). Cuando use `vue create` para crear un nuevo proyecto, algunos complementos se preinstalarán en función de su selección de funciones.

Cuando queramos instalar un plugin en un proyecto ya creado simplemente lo instalaremos primero:

```
npm install -D @vue/cli-plugin-eslint
```

Después podemos invocar al generador de plugin para que genere los ficheros pertinentes a nuestro proyecto:

```
# the @vue/cli-plugin- prefix can be omitted
vue invoke eslint
```

Además, podemos pasarle opciones al plugin:

```
vue invoke eslint --config airbnb --lintOn save
```

Se recomienda confirmar el estado actual de su proyecto antes de ejecutar vue invoke, para que después de la generación de archivos pueda revisar los cambios y revertir si es necesario.

Ver Anexo II

5.- DETALLES SOBRE SINTAXIS (INCOMPLETO)

5.1- 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.

5.2.- 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.

5.3.- 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.

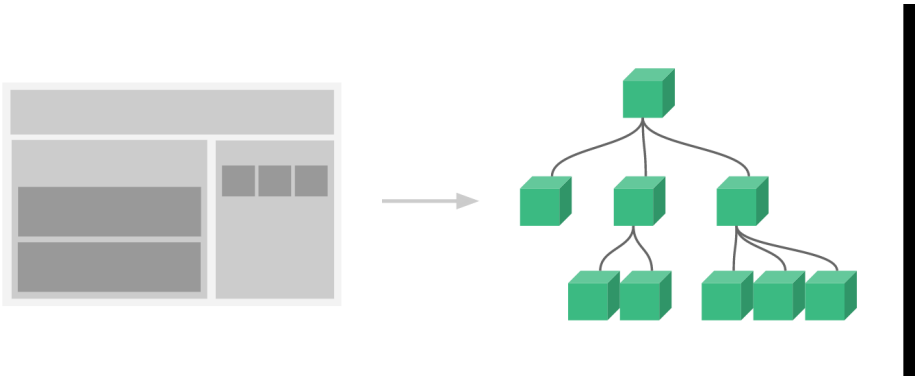
Vue también ofrece la directiva `v-model` que hace muy sencillo el enlace bidireccional entre el input del formulario y el estado de la aplicación:

.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!'
 }
})
```

5.4.- 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>
```

5.4.1.-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

6.- CLASES Y ESTILOS

Una necesidad común del enlazado de datos es la manipulación de las clases de los elementos y sus estilos en linea.

Puesto que ambos son atributos, se puede utilizar `v-bind` para gestionarlos: únicamente tenemos que calcular la cadena final con nuestras expresiones.

Sin embargo, lidiar con la concatenación de cadenas es molesto y propenso a errores.

Por esta razón, Vue ofrece mejoras especiales cuando se usa `v-bind` con `class` y `style`.

Además de cadenas, las expresiones pueden también evaluar objetos o arrays.

Enlazar clases HTML

Sintaxis de objeto

Se puede pasar un objeto a `v-bind:class` para alternar las clases dinámicamente:

```
<div v-bind:class="{ active: isActive }"></div>
```

La sintaxis anterior significa que la presencia de la clase `active` se determinará por la veracidad de la propiedad `isActive`.

Se pueden tener varias clases alternas al mismo tiempo teniendo varios campos en el objeto.

Además, la directiva `v-bind:class` puede co-existir con el atributo plano `class`.

Así que dado el siguiente ejemplo:

```
<div class="static"
 v-bind:class="{ active: isActive, 'text-danger': hasError }">
</div>
```

Y los siguientes datos:

```
data: {
```

```
 isActive: true,
 hasError: false
}
```

Se presentará:

```
<div class="static active" mark="crwd-mark"></div>
```

Cuando `isActive` o `hasError` cambia, la lista de clases se actualizará en consecuencia.

Por ejemplo, si `hasError` es `true`, la lista de clases será `"static active text-danger"`.

El objeto enlazado no tiene que estar en línea:

```
<div v-bind:class="classObject"></div>


data: {
 classObject: {
 active: true,
 'text-danger': false
 }
}
```

Esto generará el mismo resultado. También podemos enlazar una propiedad computada (computed property) que retorne un objeto. Esto es un patrón muy común y potente:

```
<div v-bind:class="classObject"></div>


data: {
 isActive: true,
 error: null
},
computed: {
 classObject: function () {
 return {
 active: this.isActive && !this.error,
 'text-danger': this.error && this.error.type === 'fatal',
 }
 }
}
```

Sintaxis de array

Se puede pasar un array a `v-bind:class` para aplicar a una lista de clases:

```
<div v-bind:class="[activeClass, errorClass]">


data: {
 activeClass: 'active',
 errorClass: 'text-danger'
```

```
}
```

Que generará:

```
<div class="active text-danger" mark="crwd-mark"></div>
```

Para alternar una clase de la lista de forma condicional, se puede hacer con una expresión ternaria:

```
<div v-bind:class="[isActive ? activeClass : '', errorClass]">
```

Esto siempre aplicará `errorClass`, pero únicamente aplicara `activeClass` cuando `isActive` sea `true`.

Sin embargo, esto puede ser demasiado extenso si se poseen varias clases condicionales. Es por ello que también es posible utilizar la sintaxis de objeto dentro de la sintaxis de array:

```
<div v-bind:class="[{ active: isActive }, errorClass]">
```

Con componentes

> Esta sección asume conocimiento de Componentes Vue. No dudes en saltar a este apartado y volver aquí más tarde.

Cuando se utiliza el atributo `class` en un componente personalizado, las clases se agregarán a la raíz del componente. Las clases existentes de este elemento no se sobrescribirán.

Por ejemplo, si se declara este componente:

```
Vue.component('my-component', {
 template: '<p class="foo bar">Hi</p>'
})
```

Luego añadir algunas clases al usarlo:

```
<my-component class="baz boo"></my-component>
```

El código HTML será:

```
<p class="foo bar baz boo" mark="crwd-mark">Hi</p>
```

Lo mismo vale para los enlaces de clases:

```
<my-component v-bind:class="{ active: isActive }"></my-component>
```

Cuando `isActive` es verdadero, el HTML generado será:

```
<p class="foo bar active" mark="crwd-mark">Hi</p>
```

Enlazar estilos en línea

Sintaxis de objeto

La sintaxis de objetos para `v-bind:style` es bastante sencilla - parece casi CSS, excepto que es un objeto JavaScript.

Se puede utilizar camelCase o kebab-case (con comillas en este último caso) para los nombres de las propiedades CSS:

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

```
data: {
 activeColor: 'red',
 fontSize: 30
}
```

Es a menudo buena idea enlazar a un objeto de estilo directamente para que la plantilla quede más limpia:

```
<div v-bind:style="styleObject"></div>
```

```
data: {
 styleObject: {
 color: 'red',
 fontSize: '13px'
 }
}
```

De nuevo, la sintaxis de objeto se utiliza en conjunto con las propiedades computadas que retornan objetos.

Sintaxis de array

La sintaxis de array para `v-bind:style` permite aplicar múltiples objetos de estilo al mismo elemento:

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

Auto-prefijo

Cuando se utiliza una propiedad CSS que requiere prefijos de otro proveedor en `v-bind:style`, por ejemplo `transform`, Vue lo detectará automáticamente y añadirá los prefijos apropiados a estos estilos.

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

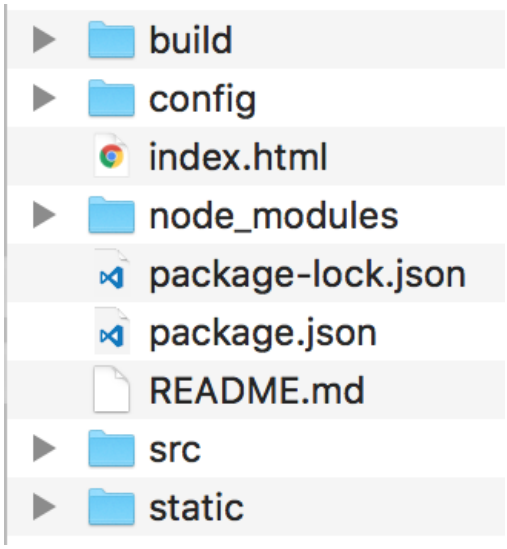
- El comando **vue,** lo usamos para llamar el CLI
- **init**, estamos diciéndole al CLI que queremos inicializar un nuevo proyecto.
- **webpack**, es la plantilla que queremos usar. Hay varias plantillas, más información al respecto más adelante
- **miaplicacion**, es el nombre que le queremos dar a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

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
```

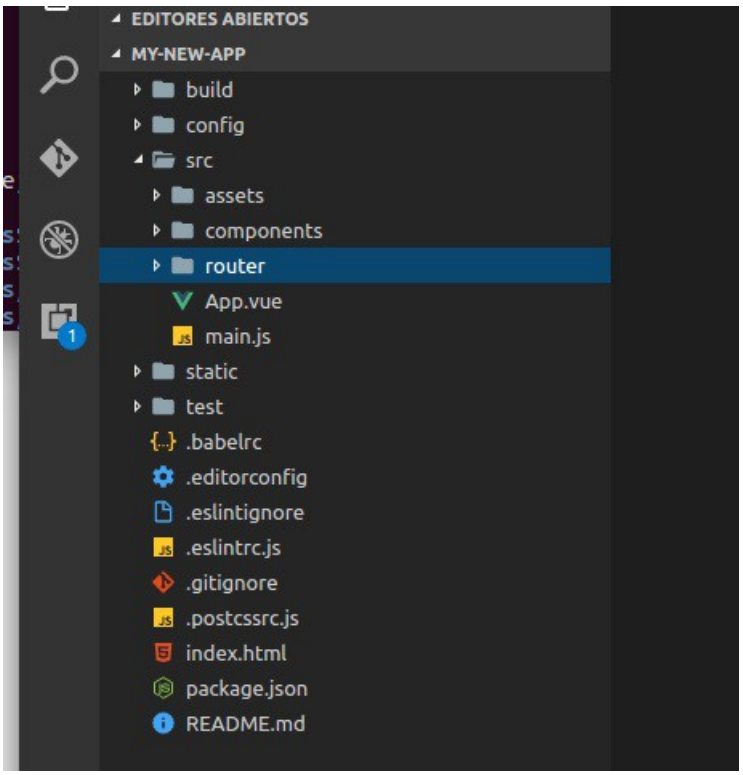
Lo que este comando hace es generar una aplicación llamada my-new-app con la plantilla de webpack.
Lo que hará vue-cli es una serie de preguntas para que configuremos ciertos aspectos a nuestro gusto. En el momento de creación del post, eran las siguientes:

- ? 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.

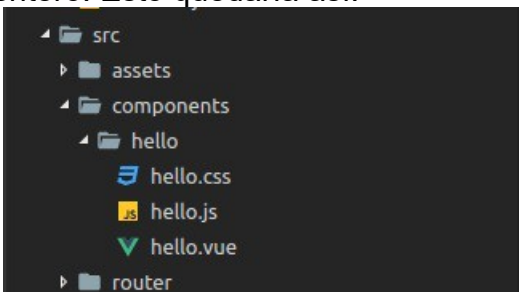
El loader de VueJS es capaz de entender estas etiquetas y de incluir cada porción en sus paquetes correspondientes. Nosotros no tenemos que preocuparnos de ellos.

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:

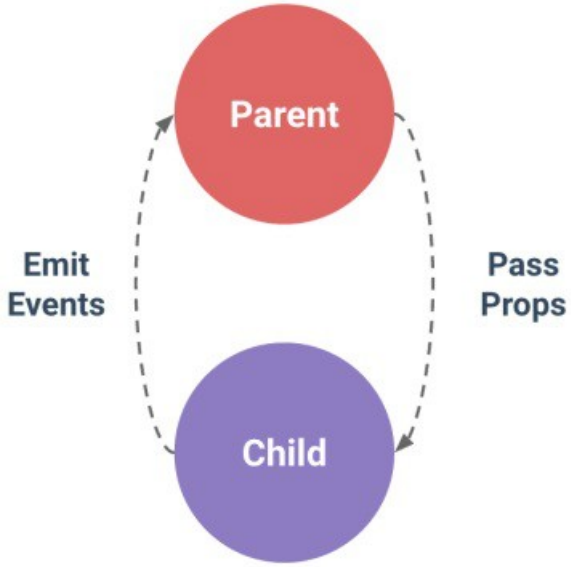
```
<course
  title="Curso JavaScript"
  subtitle="Curso Avanzado"
  description="Esto es un nuevo curso para aprender">
</course>
```

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>
```