



# La documentación oficial

## 2.0

# Guía

---

Inicio

# Instalación

## # Notas de compatibilidad

Vue **no** está soportado en IE8 ni versiones anteriores, porque utiliza características de ECMAScript 5 que son irreproducibles en ellos. Sin embargo soporta todos los **navegadores compatibles con ECMAScript 5**.

## # Notas de lanzamiento

Se pueden encontrar notas de lanzamiento detalladas para cada versión en **GitHub**.

## Versión independiente

---

Simplemente descargala e incluyela con una etiqueta script. **Vue** se registrará como una variable global.



No utilices la versión minificada durante el desarrollo. ¡Perderás todas las advertencias para los errores comunes!

Versión de desarrollo

Con todas las advertencias y el modo depuración.

Versión de producción

Advertencias eliminadas, 24.72kb min+gzip

## # CDN

Recomendación: **unpkg**, el cual contendrá la última versión apenas haya sido publicada en npm. También puedes explorar el código fuente del paquete npm en **unpkg.com/vue/**.

También se encuentra disponible en [jsdelivr](#) o [cdnjs](#), pero estos dos servicios pueden tardar algún tiempo en sincronizar, por lo que la última versión puede no estar disponible todavía.

## NPM

---

NPM es el método de instalación recomendado cuando se construyen aplicaciones de gran escala con Vue. Se integra bien con empaquetadores de módulos como [Webpack](#) o [Browserify](#). Vue también provee herramientas complementarias para la creación de **componentes de un solo archivo**.

Shell

```
# última versión estable
$ npm install vue
```

### # Versión independiente vs. versión *Runtime-only*

Hay dos versiones disponibles, la independiente y la *runtime-only*. La diferencia es que la primera incluye un **compilador de plantillas** y la última no.

El compilador de plantillas es responsable de compilar plantillas de Vue en funciones de renderizado de JavaScript puro. Si deseas utilizar la opción `template`, entonces necesitas el compilador.

- La versión independiente incluye el compilador y soporta la opción `template`. **También depende de la presencia de APIs del navegador, por lo que no puedes usarlo para renderizado del lado servidor.**
- La versión *runtime-only* no incluye el compilador de plantillas y no soporta la opción `template`. Solo puedes utilizar la opción `render` cuando estás utilizando esta versión, pero funciona con componentes de un solo archivo, porque las plantillas de los componentes de un solo archivo son pre-compiladas en funciones `render` durante la etapa de construcción. La versión *runtime-only* es aproximadamente 30% más liviana que la versión independiente, ocupando solo 17.14kb min+gzip.

Por defecto, el paquete NPM exporta la versión **runtime-only**. Para utilizar la versión independiente, añade el siguiente alias en tu archivo de configuración de Webpack:

```
resolve: {
  alias: {
    'vue$': 'vue/dist/vue.common.js'
  }
}
```

Para Browserify, puedes añadir un alias a tu archivo package.json:

JS

```
"browser": {
  "vue": "vue/dist/vue.common"
},
```



No realices un `import Vue from 'vue/dist/vue.js'` - dado que algunas herramientas en bibliotecas de terceros pueden también importar vue y podría causar que la aplicación intente cargar ambas versiones al mismo tiempo, conduciendo a errores.

## # Ambientes CSP

Algunos ambientes, como las aplicaciones de Google Chrome, imponen las Políticas de Seguridad de Contenido (CSP por sus siglas en inglés), las cuales prohíben el uso de `new Function()` para la evaluación de expresiones. La versión independiente depende de esta característica para compilar plantillas, por lo que no es posible utilizarla en estos ambientes.

Por otro lado, la versión *runtime-only* es completamente compatible con CSP. Cuando utilices la versión *runtime-only* **Webpack + vue-loader** o **Browserify + vueify**, tus plantillas serán pre-compiladas en funciones `render` las cuales funcionan perfectamente en ambientes CSP.

## CLI

---

Vue.js provee una **CLI oficial** para estructurar rápidamente Aplicaciones de una Sola Página (SPA por sus siglas en inglés). Provee configuraciones *todo-en-uno* para un flujo de trabajo frontend moderno. Solo toma unos minutos estar preparado para el desarrollo con: recarga en caliente, *lint-on-save* y versiones listas para producción:

Shell

```
# Instala vue-cli
$ npm install --global vue-cli
```

```
# Crea un nuevo proyecto usando la plantilla "webpack"
$ vue init webpack my-project
# Instala las dependencias y ¡listo!
$ cd my-project
$ npm install
$ npm run dev
```

!

La *CLI* asume un conocimiento previo de Node.js y las herramientas de trabajo asociadas. Si eres principiante con Vue o las herramientas de desarrollo front-end, te recomendamos fervientemente leer **la guía** sin ninguna herramienta de desarrollo previo a usar la *CLI*.

## Versión desarrollo

---

**Importante:** los archivos contruidos dentro de la carpeta `/dist` de GitHub son compiladas solo durante lanzamientos. Para utilizar el código fuente más reciente de Vue en GitHub, ¡tendrás que construirlo tú mismo!

Shell

```
git clone https://github.com/vuejs/vue.git node_modules/vue
cd node_modules/vue
npm install
npm run build
```

## Bower

---

Shell

```
# Última versión estable
$ bower install vue
```

## Gestores de módulos AMD

---

La versión independiente o las instaladas a través de Bower están encapsuladas con UMD, por lo que pueden ser usadas directamente como módulos AMD.



# Introducción

## ¿Qué es Vue.js?

---

Vue (pronunciado /vjuː/ en inglés, como **view**) es un **framework progresivo** para construir interfaces de usuario. A diferencia de otros *frameworks* monolíticos, Vue está diseñado desde el inicio para ser adoptado incrementalmente. La biblioteca principal se enfoca solo en la capa de la vista, y es muy simple de utilizar e integrar con otros proyectos o bibliotecas existentes. Por otro lado, Vue también es perfectamente capaz de soportar aplicaciones sofisticadas de una sola página (en inglés *single-page-application* o SPA) cuando se utiliza en combinación con **herramientas modernas y librerías compatibles**.

Si eres un desarrollador de *frontend* con experiencia y quieres saber como Vue se compara con otras bibliotecas/frameworks, revisa **esta comparación**.

## Empezando

---

!

La guía oficial asume un conocimiento intermedio de HTML, CSS y JavaScript. Si eres totalmente nuevo en el desarrollo de *frontend*, puede no ser la mejor idea empezar a utilizar un *framework* - ¡aprende los conceptos básicos y luego regresa aquí! La experiencia previa con otros *frameworks* ayuda, pero no es obligatoria.

La manera más sencilla de probar Vue.js es usando el **ejemplo “hola mundo” en JSFiddle**. Siéntete libre de abrirlo en otra pestaña y revisarlo a medida que avanzamos con ejemplos básicos. Si no, puedes crear un archivo `.html` e incluir Vue con:

HTML

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

La **página de instalación** provee más opciones para instalar Vue. Nota que **no** recomendamos a los principiantes comenzar con `vue-cli`, especialmente si no estás familiarizado con las herramientas de trabajo basadas en Node.js.



# Renderizado declarativo

---

En el corazón de Vue.js se encuentra un sistema que nos permite renderizar declarativamente datos en el DOM utilizando una sintaxis de plantillas directa:

HTML

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

JS

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

Hello Vue!

¡Ya hemos creado nuestra primera aplicación Vue! Esto parece bastante similar a renderizar una plantilla de texto, pero internamente Vue ha hecho muchas cosas. Los datos y el DOM están enlazados, y todo es **reactivo**. ¿Cómo lo sabemos? Abre la consola de JavaScript en tu navegador (ahora mismo, en esta página) y cambia el valor de `app.message`. Deberías ver el ejemplo renderizado actualizarse acorde a lo que has ingresado.

Además de interpolación de texto, también podemos enlazar atributos de un elemento, por ejemplo:

HTML

```
<div id="app-2">
  <span v-bind:title="message">
    ¡Deja tu mouse sobre este mensaje unos segundos para ver el atributo `title` en
  </span>
</div>
```

JS

```
var app2 = new Vue({
```

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

¡Deja tu mouse sobre este mensaje unos segundos para ver el atributo `title` enlazado dinámicamente!

Aquí nos encontramos con algo nuevo. El atributo `v-bind` que estás viendo es conocido como una **directiva**. Las directivas llevan el prefijo `v-` para indicar que son atributos especiales provistos por Vue y, como debes haber adivinado, aplican un comportamiento reactivo especial al DOM renderizado. En este caso, básicamente está diciendo “mantén el atributo `title` de este elemento enlazado con la propiedad `message` en la instancia de Vue”.

Si abres nuevamente tu consola JavaScript y escribes `app2.message = 'some new message'`, verás que el HTML enlazado (en este caso, el atributo `title`) ha sido actualizado.

## Condicionales y bucles

Es bastante sencillo alternar la presencia de un elemento:

HTML

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

JS

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

Now you see me

Adelante, escribe `app3.seen = false` en la consola. Deberías ver desaparecer el mensaje.

Este ejemplo demuestra que no solo podemos enlazar datos con texto y atributos, sino también con la **estructura** del DOM. Además, Vue provee un sistema de transiciones muy poderoso que puede aplicar automáticamente **efectos de transición** cuando los elementos son agregados/actualizados/removidos por Vue.

Hay unas cuantas otras directivas, cada una con una funcionalidad especial. Por ejemplo, la directiva `v-for` puede ser utilizada para mostrar una lista de elementos usando los datos de un array:

HTML

```
<div id="app-4">
  <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' }
    ]
  }
})
```

1. Learn JavaScript
2. Learn Vue
3. Build something awesome

En la consola, escribe `app4.todos.push({ text: 'New item' })` . Deberías ver un nuevo elemento agregado a la lista.

## Manejando entradas de usuario

---

Para permitir a los usuarios interactuar con tu aplicación, podemos usar la directiva `v-on` para añadir *listeners* de eventos que invocan métodos en nuestras instancias de Vue:

HTML

```
<div id="app-5">
  <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('')
    }
  }
})
```

Hello Vue.js!

Reverse Message

Nota que en el método simplemente actualizamos el estado de nuestra aplicación sin modificar del DOM - todas las manipulaciones del mismo son manejadas por Vue, y el código que escribes se enfoca en la lógica subyacente.

Vue también provee la directiva `v-model` que hace muy sencillo el enlace de dos vías entre un `input` de un formulario y el estado de la aplicación:

HTML

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

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

Hello Vue!

## Componentes

---

El sistema de componentes es otro concepto importante en Vue, porque es una abstracción que nos permite construir aplicaciones de gran escala compuestas por componentes pequeños, autocontenidos y, normalmente, reutilizables. Si lo pensamos, casi cualquier tipo de interfaz gráfica de una aplicación puede ser representada de manera abstracta como un árbol de componentes:



En Vue, un componente es esencialmente una instancia de Vue con opciones predefinidas. Registrar un componente en Vue es directo y sencillo:

```
// Define un nuevo componente llamado todo-item  
Vue.component('todo-item', {  
  template: '<li>This is a todo</li>'  
})
```

Ahora puedes utilizarlo en la plantilla de otro componente:

```
<ol>  
  <!-- Crea una instancia del componente todo-item -->  
  <todo-item></todo-item>  
</ol>
```

Pero esto renderizaría el mismo texto para cada *todo*, lo cual no es muy interesante. Deberíamos ser capaces de pasar datos desde el padre a los componentes hijo. Vamos a modificar la definición del componente para aceptar **propiedades**:

JS

```
Vue.component('todo-item', {
  // El componente todo-item ahora acepta
  // "prop", el cual es similar a un atributo personalizado
  // La propiedad se llama _todo_.
  props: ['todo'],
  template: '<li>{{ todo.text }}</li>'
})
```

Ahora podemos pasar *todo* a cada componente repetido utilizando **v-bind** :

HTML

```
<div id="app-7">
  <ol>
    <!-- Ahora le pasamos a cada todo-item with el objeto todo -->
    <!-- que representa, para que su contenido pueda ser dinámico -->
    <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' }
    ]
  }
})
```

1. Vegetables
2. Cheese
3. Whatever else humans are supposed to eat

Este es simplemente un ejemplo imaginario, pero hemos logrado separar nuestra aplicación en porciones más pequeñas, y el hijo está razonablemente desacoplado del padre a través de la interfaz de propiedades. Podemos mejorar aún más nuestro componente `<todo-item>` con una plantilla más compleja o diferente lógica sin afectar a la aplicación padre.

En aplicaciones grandes, es necesario dividir la aplicación entera en componentes para un desarrollo manejable. Hablaremos mucho más acerca de los componentes **más adelante en la guía**, pero aquí tienes un ejemplo (imaginario) de como luciría una plantilla de aplicación utilizando componentes:

HTML

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

## # Relación con los elementos personalizados

Puedes haber notado que los componentes de Vue son muy similares a los **Elementos Personalizados** (*Custom Elements*), los cuales son parte de la **especificación de Componentes Web** (*Web Components*). Esto es porque la sintaxis de componente de Vue está modelada basándose en ideas de la especificación. Por ejemplo, los componentes de Vue implementan la **API de Slot** y el atributo especial `is`. Sin embargo, hay unas cuantas diferencias clave:

1. La especificación de Componentes Web todavía está en un estado de borrador, y no está implementada nativamente en todos los navegadores. En comparación, los componentes de Vue no requieren ningún *polyfill* y funcionan consistentemente en todos los navegadores soportados (IE9 y superiores). Cuando se necesite, los componentes de Vue pueden ser envueltos dentro de un elemento personalizado nativo.
2. Los componentes de Vue proveen características importantes que no están disponibles en los elementos personalizados, siendo las más notables el flujo de datos entre componentes, la comunicación con eventos personalizados, y la integración con herramientas de desarrollo.

## ¿Listo para más?

---

Hemos introducido brevemente las características más básicas del corazón de Vue.js - el resto de esta guía cubrirá estas y otras características avanzadas con mucho más detalle, ¡asegúrate de leerla entera!



# La instancia de Vue

## Constructor

---

Cada vm Vue es iniciada creando una **instancia raíz de Vue** con la función constructora **Vue** :

JS

```
var vm = new Vue({  
  // opciones  
})
```

A pesar de no estar estrictamente asociado con el **patrón MVVM**, el diseño de Vue estuvo parcialmente inspirado en él. Como convención, normalmente utilizamos la variable **vm** (abreviación de ViewModel) para referirnos a instancias de Vue.

Cuando crees una nueva instancia de Vue, necesitas pasar un **objeto de opciones** el cual puede contener opciones para datos, una plantilla, el elemento donde montarla, métodos, *callbacks* para el ciclo de vida, etc. Puedes encontrar la lista completa de opciones en la **documentación de referencia de la API**.

El constructor de **Vue** puede ser extendido para crear **constructores de componentes** reutilizables con opciones predefinidas:

JS

```
var MyComponent = Vue.extend({  
  // opciones de extensión  
})  
  
// todas las instancias de `MyComponent` son creadas con  
// las opciones de extensión predefinidas  
var myComponentInstance = new MyComponent()
```

Aunque es posible crear instancias extendidas imperativamente, la mayor parte del tiempo es recomendable componerlas declarativamente in plantillas como elementos personalizados. Hablaremos de ello en detalle en **el sistema de componentes**. Por ahora, solo necesitas saber que todos los componentes de Vue son, esencialmente, instancias de Vue extendidas.

# Propiedades y métodos

---

Cada instancia de Vue **proxies** todas las propiedades que se encuentran en su objeto **data** :

JS

```
var data = { a: 1 }
var vm = new Vue({
  data: data
})

vm.a === data.a // -> verdadero

// modificar el valor de la propiedad también afecta a los datos originales
vm.a = 2
data.a // -> 2

// ... y viceversa
data.a = 3
vm.a // -> 3
```

Debe tenerse en cuenta

It should be noted that only these proxied properties are **reactive**. If you attach a new property to the instance after it has been created, it will not trigger any view updates. We will discuss the reactivity system in detail later.

In addition to data properties, Vue instances expose a number of useful instance properties and methods. These properties and methods are prefixed with **\$** to differentiate them from proxied data properties. For example:

JS

```
var data = { a: 1 }
var vm = new Vue({
  el: '#example',
  data: data
})

vm.$data === data // -> true
vm.$el === document.getElementById('example') // -> true

// $watch is an instance method
vm.$watch('a', function (newVal, oldVal) {
  // this callback will be called when `vm.a` changes
})
```

! Don't use **arrow functions** on an instance property or callback (e.g. `vm.$watch('a', newVal => this.myMethod())`). As arrow functions are bound to the parent context, `this` will not be the Vue instance as you'd expect and `this.myMethod` will be undefined.

Consult the **API reference** for the full list of instance properties and methods.

## Instance Lifecycle Hooks

---

Each Vue instance goes through a series of initialization steps when it is created - for example, it needs to set up data observation, compile the template, mount the instance to the DOM, and update the DOM when data changes. Along the way, it will also invoke some **lifecycle hooks**, which give us the opportunity to execute custom logic. For example, the **created** hook is called after the instance is created:

JS

```
var vm = new Vue({
  data: {
    a: 1
  },
  created: function () {
    // `this` points to the vm instance
    console.log('a is: ' + this.a)
  }
})
// -> "a is: 1"
```

There are also other hooks which will be called at different stages of the instance's lifecycle, for example **mounted**, **updated**, and **destroyed**. All lifecycle hooks are called with their `this` context pointing to the Vue instance invoking it. You may have been wondering where the concept of "controllers" lives in the Vue world and the answer is: there are no controllers. Your custom logic for a component would be split among these lifecycle hooks.

## Lifecycle Diagram

---

Below is a diagram for the instance lifecycle. You don't need to fully understand everything going on right now, but this diagram will be helpful in the future.



# Sintaxis de plantillas

Vue.js utiliza una sintaxis de plantilla basada en HTML lo que te permite enlazar declarativamente el DOM con los datos de la instancia de Vue subyacente. Todas las plantillas de Vue.js están compuestas por HTML válido que puede ser analizadas por navegadores compatibles con las especificaciones o analizadores HTML.

Internamente, Vue compila las plantillas a funciones de renderizado de DOM Virtual. En combinación con el sistema de reactividad, Vue es capaz de descifrar inteligentemente cual es la cantidad mínima de componentes a re-renderizar y aplicar la menor cantidad posible de manipulaciones al DOM cuando el estado de la aplicación cambia.

Si estas familiarizado con los conceptos del DOM Virtual y prefieres el poder de JavaScript puro, puedes también **escribir directamente funciones de renderizado** en lugar de plantillas, con soporte opcional para JSX.

## Interpolations

---

### # Text

The most basic form of data binding is text interpolation using the “Mustache” syntax (double curly braces):

```
<span>Message: {{ msg }}</span>
```

HTML

The mustache tag will be replaced with the value of the `msg` property on the corresponding data object. It will also be updated whenever the data object’s `msg` property changes.

You can also perform one-time interpolations that do not update on data change by using the **v-once directive**, but keep in mind this will also affect any binding on the same node:

```
<span v-once>This will never change: {{ msg }}</span>
```

HTML

## # HTML Puro

Las llaves dobles interpretan los datos como texto plano, no HTML. Si deseas mostrar HTML real, necesitarás usar la directiva `v-html` :

HTML

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

El contenido es insertado como HTML puro - los enlaces de datos son ignorados. Nota que no puedes utilizar `v-html` para componer plantillas parciales, porque Vue no es un motor de plantillas basado en cadenas de texto. En su lugar, se prefiere utilizar a los componentes como unidad fundamental para la reutilización de UI y la composición.

!

Renderizar dinámicamente HTML arbitrario en tu sitio web puede ser muy peligroso ya que conduce a **vulnerabilidades XSS**. Utiliza interpolación HTML solo con contenido de confianza y **nunca** con contenido provisto por el usuario.

## # Atributos

Las llaves no deben ser utilizadas dentro de atributos HTML, en su lugar utiliza la **directiva v-bind**:

HTML

```
<div v-bind:id="dynamicId"></div>
```

También funciona para atributos booleanos - el atributo sera quitado si la condición se evalúa como falsa:

HTML

```
<button v-bind:disabled="someDynamicCondition">Button</button>
```

## # Utilizando expresiones JavaScript

Hasta ahora, solo hemos estado enlazando a propiedades simples en nuestras plantillas. Pero Vue.js en realidad soporta todo el poder de las expresiones JavaScript dentro de cualquier enlace con los datos:

```

{{ number + 1 }}

{{ ok ? 'YES' : 'NO' }}

{{ message.split('').reverse().join('') }}

<div v-bind:id="'list-' + id"></div>

```

Estas expresiones serán evaluadas como JavaScript en el ámbito de datos de la instancia de Vue. Una restricción es que cada enlace puede contener solo **una expresión simple**, por lo que lo siguiente **NO** funcionará:

```

<!-- esto es una declaración, no una expresión: -->
{{ var a = 1 }}

<!-- el flujo de control tampoco funcionará, utiliza expresiones ternarias -->
{{ if (ok) return message }}

```

!

Las expresiones en las plantillas y tiene acceso restringido a una lista blanca de variables globales como `Math` y `Date`. No debes intentar acceder a variables globales definidas por el usuario dentro de expresiones en las plantillas.

## Directivas

Las directivas son atributos especiales identificadas con el prefijo `v-`. Los valores de los atributos de directivas deben ser **una sola expresión JavaScript** (con la excepción de `v-for`, el cual discutiremos luego). El trabajo de una directiva es aplicar reactivamente efectos secundarios al DOM cuando el valor de su expresión cambia. Veamos el ejemplo que utilizamos en la introducción:

```
<p v-if="seen">Now you see me</p>
```

Aquí, la directiva `v-if` removería/insertaría el elemento `<p>` basada en la veracidad del valor de la expresión `seen`.

## # Argumentos

Algunas directivas pueden recibir un “argumento”, indentificado con dos puntos luego del nombre de la directiva. Por ejemplo, la directiva `v-bind` se utiliza para actualizar reactivamente un atributo HTML:

HTML

```
<a v-bind:href="url"></a>
```

Aquí `href` es el argumento, el cual le indica a la directiva `v-bind` que enlace el atributo `href` del elemento con el valor de la expresión `url`.

Otro ejemplo es la directiva `v-on`, la cual escucha eventos del DOM:

HTML

```
<a v-on:click="doSomething">
```

Aquí el argumento es el nombre del evento que debe escuchar. Hablaremos acerca del manejo de eventos en detalle también.

## # Modificadores

Los modificadores son sufijos especiales identificados con un punto, los cuales indican que la directiva debe ser enlazada de alguna forma especial. Por ejemplo, el modificador `.prevent` indica a la directiva `v-on` que ejecute `event.preventDefault()` en el evento disparado:

HTML

```
<form v-on:submit.prevent="onSubmit"></form>
```

Veremos más usos de los modificadores cuando hablemos en detalle de `v-on` y `v-model`.

## Filtros

---

Vue.js te permite definir filtros que pueden ser usados para aplicar formatos de texto comunes. Pueden ser utilizados en dos lugares: **en la interpolación con llaves y las expresiones** `v-bind`.



Los filtros deben ser agregados al final de las expresiones JavaScript, luego de un símbolo de tubería:

HTML

```
<!-- en interpolación de texto -->
{{ message | capitalize }}

<!-- en v-bind -->
<div v-bind:id="rawId | formatId"></div>
```

! Los filtros de Vue 2.x solo pueden ser usados dentro de interpolaciones con llaves y expresiones `v-bind` (esto último soportado desde la versión 2.1.0) porque están diseñados principalmente para transformar texto. Para transformaciones de datos más complejas en otras directivas, deberías utilizar en su lugar **propiedades computadas**.

Los filtros siempre reciben el valor de la expresión como primer parámetro.

JS

```
new Vue({
  // ...
  filters: {
    capitalize: function (value) {
      if (!value) return ''
      value = value.toString()
      return value.charAt(0).toUpperCase() + value.slice(1)
    }
  }
})
```

Pueden ser encadenados:

HTML

```
{{ message | filterA | filterB }}
```

Son funciones JavaScript, por lo que pueden recibir parámetros:

HTML

```
{{ message | filterA('arg1', arg2) }}
```

Aquí, la cadena de texto `'arg1'` será pasada al filtro como segundo parámetro, y el valor de la expresión `arg2` será evaluado y pasado como tercer parámetro.

# Atajos

---

El prefijo `v-` sirve como ayuda visual para identificar atributos específicos de Vue en tus plantillas. Esto es útil cuando estás utilizando Vue.js para añadir comportamiento dinámico a una estructura existente, pero puede tornarse repetitivo para algunas directivas utilizadas frecuentemente. A la vez, la necesidad del prefijo `v-` se vuelve menos importante cuando estás construyendo una **SPA** donde Vue.js controla todas las plantillas. Por lo tanto, Vue.js provee atajos especiales para dos de las directivas más utilizadas, `v-bind` y `v-on` :

## # Atajo para `v-bind`

HTML

```
<!-- sintaxis completa -->
<a v-bind:href="url"></a>

<!-- atajo -->
<a :href="url"></a>
```

## # Atajo para `v-on`

HTML

```
<!-- sintaxis completa -->
<a v-on:click="doSomething"></a>

<!-- atajo -->
<a @click="doSomething"></a>
```

Pueden parecer un poco diferentes al HTML normal, pero `:` y `@` son caracteres válidos para nombres de atributo y todos los navegadores soportados por Vue.js los pueden analizar correctamente. Además, no aparecen en la estructura renderizada final. La sintaxis corta es totalmente opcional, pero seguramente te gustará cuando aprendas más acerca de su uso.

# Propiedades computadas y observadores

## Propiedades computadas

---

Las expresiones dentro de las plantillas son muy cómodas, pero están pensadas solo para operaciones simples. Agregar demasiada lógica en tus plantillas puede hacerlas engorrosas y difíciles de mantener. Por ejemplo:

HTML

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

En este punto, la plantilla ya no es más sencilla y declarativa. Tienes que observarla durante un momento antes de entender que muestra el valor de `message` invertido. El problema es peor cuando quieres incluir el mensaje invertido en más de un lugar dentro de tu plantilla.

Por esto es que para cualquier lógica compleja, deberías utilizar una **propiedad computada**.

## # Ejemplo básico

HTML

```
<div id="example">
  <p>Original message: "{{ message }}"</p>
  <p>Computed reversed message: "{{ reversedMessage }}"</p>
</div>
```

JS

```
var vm = new Vue({
  el: '#example',
  data: {
    message: 'Hello'
  },
  computed: {
    // un getter computado
    reversedMessage: function () {
      // `this` apunta a la instancia de vm
      return this.message.split('').reverse().join('')
    }
  }
})
```

```
}  
})
```

Resultado:

```
Original message: "Hello"  
Computed reversed message: "olleH"
```

Aquí hemos declarado una propiedad computada `reversedMessage` . La función que codificamos será usada como función *getter* para la propiedad `vm.reversedMessage` :

JS

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

Puedes abrir la consola y jugar con el ejemplo. El valor de `vm.reversedMessage` siempre depende del valor de `vm.message` .

Puedes crear enlaces de datos a propiedades computadas en las plantillas como harías con una propiedad normal. Vue está al tanto que `vm.reversedMessage` depende de `vm.message` , por lo que actualizará cualquier enlace que dependa de `vm.reversedMessage` cuando `vm.message` cambie. Y la mejor parte es que hemos creado esta relación de dependencias declarativamente: la función *getter* computada no tiene efectos secundarios, lo cual hace sencillo entenderla y probarla.

## # Cacheo computado vs Métodos

Puede que hayas notado que podemos lograr el mismo resultado invocando un método en la expresión:

HTML

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

JS

```
// dentro del componente
```

```

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

```

En lugar de una propiedad computada, podemos definir la misma función como un método. Desde el punto de vista del resultado, ambos enfoques son exactamente iguales. Sin embargo, la diferencia es que las **propiedades computadas son cacheadas basándose en sus dependencias**. Una propiedad computada solo se reevaluará cuando alguna de sus dependencias haya cambiado. Esto significa que mientras que `message` no haya cambiado, acceder reiteradamente a la propiedad computada `reversedMessage` devolverá inmediatamente el resultado computado previamente sin tener que ejecutar la función de nuevo.

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

JS

```

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

```

En comparación, la invocación a un método **siempre** ejecutará la función cuando haya un re-renderizado.

¿Para que necesitamos el cacheo? Imagina que tenemos una propiedad computada **A** costosa en términos de rendimiento, la cual requiere iterar a través de un arreglo enorme y hacer muchos cálculos. Además podemos tener otras propiedades computadas que dependan de **A**. Sin cacheo, ¡estaríamos ejecutando la función *getter* de **A** muchas más veces de las necesarias! En casos donde no quieras cacheo, utiliza un método.

## # Propiedades computadas vs Propiedades observadas

Vue provee una forma más generica de observar cambios de datos en una instancia de Vue y reaccionar frente a ellos: **observar propiedades**. Cuando tienes algunos datos que deben cambiar basándose en otros datos, es tentador utilizar excesivamente `watch` - especialmente si tienes experiencia con AngularJS. De cualquier manera, normalmente es una mejor idea utilizar una propiedad computada en lugar de una llamada imperativa a la función *callback* `watch`. Por ejemplo:

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

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

El código anterior es imperativo y repetitivo. Compáralo con la versión con propiedades computadas:

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

Mucho mejor, ¿no?

## # Función *setter* computada

Las propiedades computadas solo tienen una función *getter* por defecto, pero puedes agregarles una función *setter* cuando lo necesites:

JS

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

Ahora cuando ejecutes `vm.fullName = 'John Doe'`, la función *setter* será ejecutada y `vm.firstName` y `vm.lastName` serán actualizadas según corresponda.

## Observadores

---

Mientras que las propiedades computadas son más apropiadas en la mayoría de los casos, hay veces que un observador personalizado es necesario. Es por eso que Vue provee una forma más genérica de reaccionar a los cambios en los datos a través de la opción `watch`. Esto es mayormente útil cuando quieres realizar operaciones asíncronas o costosas en respuesta a los cambios de los datos.

Por ejemplo:

HTML

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

```

<!-- Dado que ya existe un ecosistema rico de bibliotecas AJAX
<!-- y colecciones de métodos utilitarios de propósito general, el núcleo de Vue
<!-- es capaz de mantenerse pequeño porque no crea los suyos propios. Esto también
<!-- te da la libertad de utilizar cualquier cosa con la que ya estes familiarizado
<script src="https://unpkg.com/axios@0.12.0/dist/axios.min.js"></script>
<script src="https://unpkg.com/lodash@4.13.1/lodash.min.js"></script>
<script>
var watchExampleVM = new Vue({
  el: '#watch-example',
  data: {
    question: '',
    answer: 'I cannot give you an answer until you ask a question!'
  },
  watch: {
    // cuando 'question' cambie, se ejecutará esta función
    question: function (newQuestion) {
      this.answer = 'Waiting for you to stop typing...'
      this.getAnswer()
    }
  },
  methods: {
    // _.debounce es una función probista por lodash para limitar cuan
    // seguido una operación particularmente costosa puede ejecutarse.
    // En este caso, queremos limitar las peticiones a
    // yesno.wtf/api, esperando a que el usuario haya finalizado
    // de tipear antes de hacer la petición ajax. Para aprender
    // más acerca de la función _.debounce ( y su prima
    // _.throttle), visita: https://lodash.com/docs#debounce
    getAnswer: _.debounce(
      function () {
        if (this.question.indexOf('?') === -1) {
          this.answer = 'Questions usually contain a question mark. ;-)'
          return
        }
        this.answer = 'Thinking...'
        var vm = this
        axios.get('https://yesno.wtf/api')
          .then(function (response) {
            vm.answer = _.capitalize(response.data.answer)
          })
          .catch(function (error) {
            vm.answer = 'Error! Could not reach the API. ' + error
          })
      },
      // Este es el número de milisegundos que esperamos
      // a que el usuario termine de tipear.
      500
    )
  }
})
</script>

```



Resultado:

Ask a yes/no question:

I cannot give you an answer until you ask a question!

En este caso, utilizar la opción `watch` nos permite realizar operaciones asíncronas (acceder a una API), limitando cuan seguido ejecutamos esa operación, y establece estados intermediarios hasta que tengamos una respuesta final. Nada de esto sería posible sin una propiedad computada.

Además de la opción `watch` , puedes utilizar la **API de `vm.$watch`**.

# Enlace de estilos y clases

Una necesidad común cuando se enlazan datos es manipular la lista de clases de un elemento y sus estilos en línea. Dado que ambos son atributos, podemos utilizar `v-bind` para manejarlos: solo necesitamos calcular la cadena de texto final con nuestras expresiones. Sin embargo, lidiar con la concatenación de texto es molesto y propenso a errores. Por este motivo, Vue provee mejoras especiales cuando `v-bind` se utiliza en conjunto con `class` y `style`. Además de cadenas de texto, las expresiones pueden evaluar también objetos o arreglos:

## Enlazando clases HTML

---

### # Sintaxis de objeto

Podemos pasar un objeto a `v-bind:class` para intercambiar clases dinámicamente:

HTML

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

La sintaxis anterior indica que la presencia de la clase `active` será determinada por **el valor de verdad** de la propiedad `isActive`.

Puedes intercambiar múltiples clases agregando más campos al objeto. Además, la directiva `v-bind:class` puede co-existir con el atributo `class`. Entonces dada la siguiente plantilla:

HTML

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

Y los siguientes datos:

JS

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

Renderizará:

HTML

```
<div class="static active"></div>
```

Cuando `isActive` o `hasError` cambien, la lista de clases será actualizada en consecuencia.

Por ejemplo, si `hasError` pasa a valer `true`, la lista de clases se convertirá en

`"static active text-danger"`.

El objeto enlazado no tiene que ser declarado en línea:

HTML

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

JS

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

Esto renderizará el mismo resultado. Podemos también enlazar a una **propiedad computada** que devuelve un objeto. Este es un patrón muy común y poderoso:

HTML

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

JS

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

## # Sintaxis de arreglo

Podemos pasar un arreglo a `v-bind:class` para aplicar una lista de clases:

HTML

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

JS

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

Lo cual renderizará:

HTML

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

Si quisieras alternar una clase en la lista condicionalmente, puedes hacerlo con una expresión ternaria:

HTML

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

Esto siempre aplicará la clase `errorClass`, pero solo `activeClass` cuando `isActive` valga `true`.

Sin embargo, esto puede resultar engorroso si tienes múltiples clases condicionales. Es por eso que también es posible utilizar la sintaxis de objetos dentro de la sintaxis de arreglos:

HTML

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

## # Dentro de componentes

Esta sección asume un conocimiento previo de los **componentes de Vue**. Siéntete libre de saltarla y volver luego.

Cuando utilizas el atributo `class` en un componente personalizado, esas clases serán agregadas al elemento raíz del componente. Las clases existentes en el elemento no serán sobre escritas.

Por ejemplo, si declaras este componente:

JS

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

Y luego agregas algunas clases cuando lo utilizas:

HTML

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

El HTML renderizado será:

HTML

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

Lo mismo aplica a clases enlazadas:

HTML

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

Cuando `isActive` sea verdadero, el HTML renderizado será:

HTML

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

## Enlazando estilos en línea

---

### # Sintaxis de objeto

La sintaxis de objeto para `v-bind:style` es bastante directa - es similar al CSS puro, excepto que es un objeto JavaScript. Puedes utilizar tanto *camelCase* como *kebab-case* (utiliza comillas con *kebab-case*) para los nombres de las propiedades CSS:

HTML

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

JS

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

Normalmente es una buena idea enlazar a un objeto de estilo directamente para que la plantilla sea más clara:

HTML

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

JS

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

Nuevamente, la sintaxis de objeto es utilizada normalmente en conjunto con propiedades computadas que devuelven objetos.

## # Sintaxis de arreglo

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

HTML

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

## # Prefijos automáticos

Cuando utilizas una propiedad CSS en `v-bind:style` que requiere prefijos, por ejemplo `transform`, Vue lo detectará automáticamente y agregará los prefijos apropiados a los estilos.

aplicados.

# Renderizado condicional

## v-if

---

En plantillas de cadena de texto, por ejemplo Handlebars, escribiríamos un bloque condicional de la siguiente manera:

```
<!-- Plantilla de Handlebars -->
{{#if ok}}
  <h1>Yes</h1>
{{/if}}
```

HTML

En Vue, utilizamos la directiva `v-if` para lograr el mismo resultado:

```
<h1 v-if="ok">Yes</h1>
```

HTML

También es posible agregar un bloque “else” con `v-else` :

```
<h1 v-if="ok">Yes</h1>
<h1 v-else>No</h1>
```

HTML

## # Grupos condicionales con `v-if` dentro de `<template>`

Debido a que `v-if` es una directiva, tiene que ser añadida a un elemento en particular. Pero, ¿qué sucede si queremos mostrar/esconder más de un elemento? En este caso utilizamos `v-if` en un elemento `<template>` , el cual sirve como elemento envolvente invisible. El resultado del renderizado final no lo incluíra.

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

HTML



## # v-else

Puedes utilizar la directiva `v-else` para indicar un “bloque else” para `v-if` :

HTML

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

Un elemento `v-else` debe encontrarse inmediatamente después de un elemento `v-if` o `v-else-if` - de otra forma, no será reconocido.

## # v-else-if

Nuevo en 2.1.0

`v-else-if` , como su nombre lo indica, sirve como un “bloque else if” para `v-if` . Puede ser encadenado varias veces seguidas:

HTML

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

Similar a `v-else` , un elemento `v-else-if` debe ubicarse inmediatamente luego de un elemento `v-if` o `v-else-if` .

## # Controlando elementos reusables con **key**

Vue intenta renderizar elementos de la manera más eficiente posible, normalmente reutilizándolos en lugar de renderizarlos de cero. Más allá de ayudar a Vue a ser muy rápido, esto puede tener algunas ventajas muy útiles. Por ejemplo, si un usuario intenta alternar entre diferentes tipos de inicio de sesión:

HTML

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

Intercambiando **loginType** en el código anterior no borrará lo que el usuario ya haya escrito. Dado que ambas plantillas utilizan los mismos elementos, **<input>** no es reemplazado, solo su **placeholder** .

Verifícalo tu mismo escribiendo algo en el campo de texto y luego presionando el botón “Toggle”:

Username

Enter your username

Toggle login type

Sin embargo, este no siempre es el comportamiento deseado, por lo que Vue te ofrece una manera de decir: “Estos dos elementos están completamente separados, no los reutilices”. Simplemente agrega el atributo **key** con un valor único:

HTML

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

Ahora esos campos de texto serán renderizados desde cero cada vez que los intercambies. Verifícalo:

Username

Nota que los elementos `<label>` están siendo reutilizados eficientemente, porque no tienen el atributo `key`.

## v-show

---

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

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

HTML

La diferencia es que un elemento con `v-show` siempre será renderizado y permanecerá en el DOM. `v-show` simplemente alterna el valor de la propiedad CSS `display` del elemento.

! Nota que `v-show` no soporta la sintaxis `<template>` ni funciona con `v-else`.

## v-if vs v-show

---

`v-if` es renderizado condicional “real” porque se asegura que los *listeners* de eventos y componentes hijo dentro del bloque condicional sean destruidos y recreados apropiadamente durante los cambios de condición.

`v-if` es también **lazy**: si la condición es falsa durante el renderizado inicial, no hará nada. El bloque condicional no será renderizado hasta que la condición sea verdadera por primera vez.

En comparación, `v-show` es mucho más sencillo: el elemento siempre es renderizado sin importar el estado inicial de la condición, con una alternancia basada en CSS.

Generalmente, `v-if` tiene un costo de alternancia mayor mientras que `v-show` tiene un costo de renderizado inicial mayor. Entonces escoge `v-show` si necesitas alternar algo muy frecuentemente o `v-if` si es poco probable que la condición cambie durante la ejecución.

## `v-if` with `v-for`

---

Cuando se utiliza en conjunto con `v-for`, `v-for` tiene una prioridad mayor que `v-if`. Lee la [guía de renderizado de listas](#) para más detalles.

# Renderizado de listas

## v-for

---

Podemos utilizar la directiva `v-for` para renderizar una lista de elementos basándonos en un arreglo. La directiva `v-for` requiere una sintaxis especial de la forma `item in items`, donde `items` es el arreglo de datos fuente e `item` es un **alias** para el elemento sobre el que se está iterando:

### # uso básico

HTML

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

JS

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

Resultado:

- Foo
- Bar

Dentro de los bloques `v-for` tenemos acceso total a las propiedades del ámbito del padre. `v-for` también soporta un segundo parámetro opcional para indicar el índice el elemento actual.

HTML

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

JS

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

Resultado:

- Parent - 0 - Foo
- Parent - 1 - Bar

Puedes utilizar `of` como delimitador en lugar de `in` , para que la sintaxis sea más parecida a la utilizada en JavaScript para iteradores:

HTML

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

## # `v-for` en

Similar a `v-if` , puedes utilizar una etiqueta `<template>` con `v-for` para renderizar un bloque de múltiples elementos. Por ejemplo:

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

## # v-for con objetos

Puedes utilizar `v-for` para iterar a través de las propiedades de un objeto.

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

```
new Vue({
  el: '#repeat-object',
  data: {
    object: {
      firstName: 'John',
      lastName: 'Doe',
      age: 30
    }
  }
})
```

Resultado:

- John
- Doe
- 30

También puedes proveer un segundo argumento para la llave:

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

Y otro para el índice:

HTML

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

! Cuando iteres sobre un objeto, el orden está basado en el orden de enumeración de `Object.keys()`, el cual **no** garantizamos que sea consistente en todas las implementaciones de motores JavaScript.

## # `v-for` con rangos

`v-for` puede recibir un entero. En este caso, repetirá la plantilla esa cantidad de veces.

HTML

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

Resultado:

1 2 3 4 5 6 7 8 9 10

## # Componentes y `v-for`

Esta sección asume un conocimiento previo de los **componentes de Vue**. Siéntete libre de saltarla y volver luego.

Puedes utilizar `v-for` directamente en un componente personalizado, como cualquier otro elemento normal:



```
<my-component v-for="item in items"></my-component>
```

Sin embargo, no pasará automáticamente ningún dato al componente, porque los componentes tienen ámbitos aislados propios. Para pasar los datos de la iteración al componente, debes utilizar propiedades:

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

La razón para no inyectar automáticamente `item` al componente es que genera un acoplamiento estrecho entre el componente y el funcionamiento de `v-for`. Siendo explícito acerca del origen de los datos hace que el componente sea reutilizable en otras situaciones.

Aquí tienes un ejemplo completo de una lista de tareas pendientes:

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

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

```

    })

    new Vue({
      el: '#todo-list-example',
      data: {
        newTodoText: '',
        todos: [
          'Do the dishes',
          'Take out the trash',
          'Mow the lawn'
        ]
      },
      methods: {
        addNewTodo: function () {
          this.todos.push(this.newTodoText)
          this.newTodoText = ''
        }
      }
    })

```

Add a todo

- Do the dishes ☐
- Take out the trash ☐
- Mow the lawn ☐

## # v-for con v-if

Cuando existen en el mismo nodo, `v-for` tiene mayor prioridad que `v-if`. Esto significa que `v-if` será ejecutado en cada iteración del bucle separadamente. Esto es muy útil cuando quieres renderizar nodos solo para *algunos* elementos, como:

HTML

```

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

```

Lo anterior solo renderizará los *todos* que no hayan sido completados.

Si, en cambio, tu intención es saltarte condicionalmente la ejecución del bucle, puedes utilizar `v-if` en un elemento envolvente (o `<template>`). Por ejemplo:

```
<ul v-if="shouldRenderTodos">
  <li v-for="todo in todos">
    {{ todo }}
  </li>
</ul>
```

## key

---

Cuando Vue está actualizando una lista de elementos renderizados con `v-for`, por defecto utiliza una estrategia “in-place patch”. Si el orden de los elementos en los datos ha cambiado, en lugar de mover los elementos del DOM para coincidir con el orden de los elementos en los datos, Vue simplemente actualizará cada elemento en su lugar y se asegurará que refleje lo que debería ser renderizado en ese índice en particular. Esto es similar al comportamiento de `track-by="$index"` en Vue 1.x.

Este modo por defecto es eficiente, pero solo conveniente **cuando la lista renderizada no depende del estado de componentes hijos o de estado temporario del DOM (por ejemplo, campos de texto de un formulario)**.

Para darle una pista a Vue de que debe llevar un seguimiento de la identidad de cada nodo, y entonces reusar y reordenar elementos existentes, necesitas proveer un atributo `key` único para cada elemento. Un valor ideal para `key` sería un id único. Este atributo especial es un equivalente aproximado a `track-by` en 1.x, pero funciona como un atributo, por lo que necesitas utilizar `v-bind` para enlazarlo con valores dinámicos (utilizamos la forma corta aquí):

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

Es recomendable proveer un `key` para `v-for` cuando sea posible, a menos que el contenido del DOM iterado sea simple, o estarás confiando en el comportamiento por defecto para las ganancias de rendimiento.

Dado que es un mecanismo genérico para identificar nodos, `key` también tiene otros usos que no están relacionados específicamente con `v-for`, como veremos más adelante en la guía.

# Detección de cambios en un arreglo

---

## # Métodos de mutación

Vue envuelve los métodos de mutación de los arreglos observados por lo que también disparará actualizaciones de las vistas. Los métodos envueltos son:

- `push()`
- `pop()`
- `shift()`
- `unshift()`
- `splice()`
- `sort()`
- `reverse()`

Puedes abrir la consola y jugar con el arreglo `items` de los ejemplos anteriores ejecutando sus métodos de mutación. Por ejemplo: `example1.items.push({ message: 'Baz' })`.

## # Reemplazando un arreglo

Los métodos de mutación, como su nombre sugiere, modifican el arreglo original sobre el cual actúan. En comparación, hay también métodos no-mutantes, por ejemplo: `filter()`, `concat()` y `slice()`, los cuales no modifican el arreglo original sino que **siempre devuelven un nuevo arreglo**. Cuando trabajes con métodos no mutantes, puedes simplemente reemplazar el arreglo anterior por el nuevo:

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

JS

Puedes pensar que esto hará que Vue tire todo el DOM existente y re-renderice la lista entera. Por suerte, no es el caso. Vue implementa algunas heurísticas inteligentes para reutilizar al máximo los elementos del DOM, por lo que reemplazar un arreglo existente por otro que contiene objetos solapados es una operación muy eficiente.

## # Advertencias

Debido a las limitaciones en JavaScript, Vue **no puede** detectar los siguientes cambios en un arreglo:

1. Cuando intentas establecer el valor de un elemento a través del índice:

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

2. Cuando intentas modificar el largo de un arreglo: `vm.items.length = newLength`

Para solucionar el problema 1, las siguientes dos opciones lograrán el mismo resultado que `vm.items[indexOfItem] = newValue` , pero también dispararán actualizaciones de estado en el sistema de reactividad:

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

JS

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

JS

Para solucionar el problema 2, puedes utilizar también `splice` :

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

JS

## Mostrando resultados filtrados/ordenados

---

En ocasiones queremos mostrar una versión filtrada u ordenada de un arreglo sin tener que modificar o restablecer los datos originales. En este caso, puedes crear una propiedad computada que devuelva el arreglo filtrado u ordenado:

Por ejemplo:

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

HTML

```
data: {
```

JS

```

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

```

Como alternativa, puedes utilizar un método donde las propiedades computadas no son factibles (por ejemplo, dentro de bucles `v-for` anidados):

HTML

```

<li v-for="n in even(numbers)">{{ n }}</li>

```

JS

```

data: {
  numbers: [ 1, 2, 3, 4, 5 ]
},
methods: {
  even: function (numbers) {
    return numbers.filter(function (number) {
      return number % 2 === 0
    })
  }
}

```

# Manejo de eventos

## Escuchando eventos

---

Podemos utilizar la directiva `v-on` para escuchar eventos del DOM y ejecutar algo de JavaScript cuando son disparados.

Por ejemplo:

```
<div id="example-1">
  <button v-on:click="counter += 1">Add 1</button>
  <p>The button above has been clicked {{ counter }} times.</p>
</div>
```

HTML

```
var example1 = new Vue({
  el: '#example-1',
  data: {
    counter: 0
  }
})
```

JS

Resultado:

Add 1

The button above has been clicked 0 times.

## Métodos para controladoras de eventos

---

La lógica para muchas funciones controladoras de eventos puede ser un tanto compleja, por lo que mantener el código JavaScript en el valor del atributo `v-on` simplemente no es factible. Por esto es que `v-on` acepta el nombre del método que te gustaría ejecutar.

Por ejemplo:

```
<div id="example-2">
  <!-- `greet` is the name of a method defined below -->
  <button v-on:click="greet">Greet</button>
</div>
```

```
var example2 = new Vue({
  el: '#example-2',
  data: {
    name: 'Vue.js'
  },
  // define métodos dentro del objeto `methods`
  methods: {
    greet: function (event) {
      // `this` dentro de los métodos apunta a la instancia de Vue
      alert('Hello ' + this.name + '!')
      // `event` es el evento nativo del DOM
      if (event) {
        alert(event.target.tagName)
      }
    }
  }
})

// puedes invocar métodos en JavaScript también
example2.greet() // -> 'Hello Vue.js!'
```

Resultado:

Greet

## Métodos dentro de controladoras en línea

En lugar de enlazar directamente con el nombre de un método, podemos utilizar métodos en una declaración JavaScript en línea:

```
<div id="example-3">
  <button v-on:click="say('hi')">Say hi</button>
  <button v-on:click="say('what')">Say what</button>
</div>
```

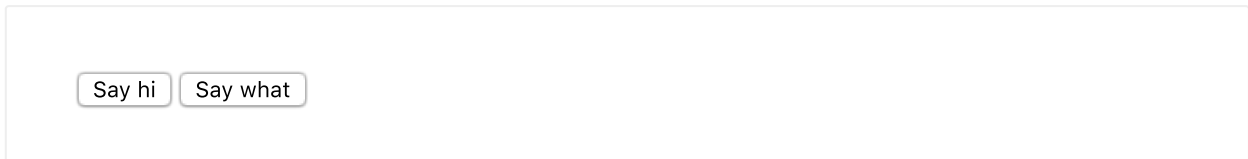


```

new Vue({
  el: '#example-3',
  methods: {
    say: function (message) {
      alert(message)
    }
  }
})

```

Resultado:



A veces necesitamos acceder al evento original del DOM en una declaración JavaScript en línea. Puedes pasarlo al método utilizando la variable especial `$event` :

HTML

```

<button v-on:click="warn('Form cannot be submitted yet.', $event)">Submit</button>

```

JS

```

// ...
methods: {
  warn: function (message, event) {
    // ahora tenemos acceso al evento nativo
    if (event) event.preventDefault()
    alert(message)
  }
}

```

## Modificadores de eventos

Es una necesidad muy común llamar a `event.preventDefault()` o `event.stopPropagation()` dentro de las funciones controladoras. Aunque podemos hacerlo fácilmente dentro de los métodos, sería mejor si estos tuvieran solo la lógica acerca de los datos en lugar de lidiar con detalles del evento del DOM.

Para solucionar este problema, Vue provee **modificadores de eventos** para `v-on`. Recuerda que esos modificadores son sufijos en las directivas indicados con un punto.

- `.stop`
- `.prevent`
- `.capture`
- `.self`
- `.once`

HTML

```
<!-- la propagación del evento 'click' será detenida -->
<a v-on:click.stop="doThis"></a>

<!-- el evento 'submit' no recargará la página -->
<form v-on:submit.prevent="onSubmit"></form>

<!-- los modificadores pueden encadenarse -->
<a v-on:click.stop.prevent="doThat"></a>

<!-- solo el modificador -->
<form v-on:submit.prevent></form>

<!-- usa el modo captura cuando agregas _listener_ de eventos -->
<div v-on:click.capture="doThis">...</div>

<!-- solo dispara la función controladora si event.target es este elemento -->
<!-- quiere decir, no un elemento hijo -->
<div v-on:click.self="doThat">...</div>
```

### Nuevo en 2.1.4

HTML

```
<!-- el evento 'click' será disparado una vez como máximo -->
<a v-on:click.once="doThis"></a>
```

A diferencia de otros modificadores, los cuales son exclusivos de los eventos nativos del DOM, el modificador `.once` puede ser usado también en **eventos de componentes**. Si todavía no has leído acerca de los componentes, por ahora no te preocupes por esta característica.

## Modificadores de teclas

---

Cuando escuchamos eventos del teclado, muchas veces necesitamos verificar códigos de teclas comunes. Vue también permite modificadores de teclas para `v-on` cuando escuchamos eventos de teclas:

HTML

```
<!-- solo ejecuta vm.submit() cuando keyCode es 13 -->
<input v-on:keyup.13="submit">
```

Recordar todos los códigos de teclas es una molestia, por lo que Vue provee alias para los más utilizados:

HTML

```
<!-- lo mismo que el ejemplo anterior -->
<input v-on:keyup.enter="submit">

<!-- también funciona en la forma corta -->
<input @keyup.enter="submit">
```

Aquí hay una lista completa de alias de modificadores de teclas:

- `.enter`
- `.tab`
- `.delete` (captura tanto las teclas “Delete” como “Backspace”)
- `.esc`
- `.space`
- `.up`
- `.down`
- `.left`
- `.right`

Puedes definir **alias personalizados para modificadores de teclas** a través del objeto global `config.keyCodes` :

JS

```
// habilita v-on:keyup.f1
Vue.config.keyCodes.f1 = 112
```

## Teclas modificadoras

---

Puedes utilizar los siguientes modificadores para disparar *listener* de eventos del teclado o el ratón solo cuando la correspondiente tecla modificadora es presionada:

- `.ctrl`
- `.alt`
- `.shift`
- `.meta`

**Nota:** En los teclados Macintosh, meta es la tecla *comando* (⌘). En teclados Windows, meta es la tecla windows (⊞). En teclados Sun Microsystems, meta es la tecla identificada con un diamante sólido (◆). En algunos teclados, específicamente en máquinas Lisp y MIT o sucesoras, como el teclado *Knight* o *space-cadet*, meta está etiquetada como “META”. En teclados Symbolics, meta está etiquetada como “META” o “Meta”.

Por ejemplo:

HTML

```
<!-- Alt + C -->
<input @keyup.alt.67="clear">

<!-- Ctrl + Click -->
<div @click.ctrl="doSomething">Do something</div>
```

## ¿Por qué *listeners* en el HTML?

Debes estar preocupado porque todo este enfoque de escucha de eventos viola las viejas y buenas reglas acerca de la “separación de intereses”. Ten la garantía: dado que todas las funciones controladoras y expresiones en Vue están estrictamente enlazadas con el *ViewModel* que controla la vista actual, no causará ningún problema de mantenimiento. De hecho, hay varios beneficios por utilizar `v-on` :

1. Es más sencillo localizar las implementaciones de las funciones controladoras dentro de tu código JS simplemente echando un vistado a la plantilla HTML.
2. Dado que no tienes que agregar manualmente *listeners* de eventos en JS, el código de tu *ViewModel* puede ser lógica pura y estar libre del DOM. Esto lo hace más sencillo de probar.
3. Cuando un *ViewModel* es destruido, todos los *listeners* de eventos son removidos automáticamente. No necesitas preocuparte por quitarlos tú mismo.



# Enlaces con campos de formulario

## Uso básico

---

Puedes utilizar la directiva `v-model` para crear enlaces de datos de dos vías en los campos de un formulario. Esta directiva elige automáticamente la forma correcta de actualizar el elemento basado en el tipo de campo. A pesar de parecer algo mágico, `v-model` es esencialmente azúcar sintáctico para actualizar datos debido a eventos de entrada de los usuarios, además de algunos cuidados para casos extremos.

! `v-model` descarta los valores iniciales provistos por los campos de un formulario. Siempre tratará a los datos en la instancia de Vue como fuente de verdad.

! Para lenguajes que requieran un **IME** (Chinese, Japanese, Korean etc.), verás que `v-model` no se actualiza durante la composición IME. Si deseas realizar estas actualizaciones también, utiliza el evento `instead` en su lugar.

## # Texto

HTML

```
<input v-model="message" placeholder="edit me">
<p>Message is: {{ message }}</p>
```

Message is:

## # Texto multilinea

HTML

```
<span>Multiline message is:</span>
<n style="white-space: pre">{{ message }}</n>
```

```

<p style="white-space: pre">{{ message }}</p>
<br>
<textarea v-model="message" placeholder="add multiple lines"></textarea>

```

Multiline message is:

add multiple lines

! La interpolación en textareas ( `<textarea>{{text}}</textarea>` ) no funcionará. Utiliza `v-model` en su lugar.

## # Checkbox

Checkbox simple, valor booleano:

HTML

```

<input type="checkbox" id="checkbox" v-model="checked">
<label for="checkbox">{{ checked }}</label>

```

☐ false

Checkbox múltiples, enlazados al mismo arreglo:

HTML

```

<input type="checkbox" id="jack" value="Jack" v-model="checkedNames">
<label for="jack">Jack</label>
<input type="checkbox" id="john" value="John" v-model="checkedNames">
<label for="john">John</label>
<input type="checkbox" id="mike" value="Mike" v-model="checkedNames">
<label for="mike">Mike</label>
<br>
<span>Checked names: {{ checkedNames }}</span>

```

```

new Vue({
  el: '...',
  data: {
    checkedNames: []
  }
})

```

☐ Jack ☐ John ☐ Mike

Checked names: []

## # Radio

HTML

```

<input type="radio" id="one" value="One" v-model="picked">
<label for="one">One</label>
<br>
<input type="radio" id="two" value="Two" v-model="picked">
<label for="two">Two</label>
<br>
<span>Picked: {{ picked }}</span>

```

☐ One

☐ Two

Picked:

## # Select

Select simple:

HTML

```

<select v-model="selected">
  <option>A</option>
  <option>B</option>
  <option>C</option>
</select>
<span>Selected: {{ selected }}</span>

```

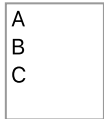


A Selected:

Select múltiple (enlazado a un arreglo):

HTML

```
<select v-model="selected" multiple>
  <option>A</option>
  <option>B</option>
  <option>C</option>
</select>
<br>
<span>Selected: {{ selected }}</span>
```



Selected: []

Opciones dinámicas renderizadas con `v-for` :

HTML

```
<select v-model="selected">
  <option v-for="option in options" v-bind:value="option.value">
    {{ option.text }}
  </option>
</select>
<span>Selected: {{ selected }}</span>
```

JS

```
new Vue({
  el: '...',
  data: {
    selected: 'A',
    options: [
      { text: 'One', value: 'A' },
      { text: 'Two', value: 'B' },
      { text: 'Three', value: 'C' }
    ]
  }
})
```

One Selected: A

## Enlace de valores

Para *radio*, *checkbox* y opciones de *select*, los valores del enlace de `v-model` son normalmente cadenas de texto estáticas (o booleanos en caso del *checkbox*):

HTML

```
<!-- `picked` es la cadena de texto "a" cuando se tilda -->
<input type="radio" v-model="picked" value="a">

<!-- `toggle` es 'true' o 'false' -->
<input type="checkbox" v-model="toggle">

<!-- `selected` es la cadena de texto "abc" cuando se selecciona -->
<select v-model="selected">
  <option value="abc">ABC</option>
</select>
```

Pero a veces podemos querer enlazar el valor a una propiedad dinámica en la instancia de Vue. Podemos utilizar `v-bind` para lograr esto. Además, utilizar `v-bind` nos permite enlazar el valor del campo a valores de otros tipos:

### # Checkbox

HTML

```
<input
  type="checkbox"
  v-model="toggle"
  v-bind:true-value="a"
  v-bind:false-value="b"
>
```

JS

```
// Cuando se tilda:
vm.toggle === vm.a
// cuando se destila:
```

```
vm.toggle === vm.b
```

## # Radio

HTML

```
<input type="radio" v-model="pick" v-bind:value="a">
```

JS

```
// cuando se tilda:  
vm.pick === vm.a
```

## # opciones de Select

HTML

```
<select v-model="selected">  
  <!-- objeto literal en línea -->  
  <option v-bind:value="{ number: 123 }">123</option>  
</select>
```

JS

```
// cuando se selecciona:  
typeof vm.selected // -> 'object'  
vm.selected.number // -> 123
```

# Modificadores

---

## # .lazy

Por defecto, `v-model` sincroniza el campo con los datos luego de cada evento `input` (con la excepción de las composiciones IME **como se mencionó anteriormente**). Puedes agregar el modificador `lazy` para sincronizar luego de eventos `change` en su lugar:

HTML

```
<!-- sincronizado luego de "change" en lugar de "input" -->  
<input v-model.lazy="msg" >
```

## # .number

Si deseas que un campo de usuario sea convertido automáticamente a un número, puedes agregar el modificador `number` a tus campos controlados por `v-model` :

HTML

```
<input v-model.number="age" type="number">
```

Esto es útil, debido a que incluso con `type="number"` , el valor de los elementos HTML `<input>` siempre devuelven una cadena de texto.

## # .trim

Si deseas que a los datos de usuario se le aplique `trim` automáticamente, puedes agregar el modificador `trim` a tus campos controlados por `v-model` :

HTML

```
<input v-model.trim="msg">
```

## v-model con componentes

---

Si todavía no estas familiarizado con los componentes de Vue, saltea esto por ahora.

Los tipos de campos de formularios nativos de HTML no siempre se ajustarán a tus necesidades. Afortunadamente, los componentes de Vue te permiten construir campos de entrada reutilizables con comportamiento completamente personalizado. ¡Estos campos de entrada incluso funcionan con `v-model` ! Para aprender más, lee acerca de los **campos de entrada personalizados** en la guía de componentes.

# Componentes

## ¿Qué son los componentes?

---

Los componentes son una de las características más poderosas de Vue. Te permiten extender elementos HTML básicos para encapsular código reutilizable. En un nivel alto, los componentes son elementos personalizados a los que el compilador de Vue les añade comportamiento. En algunos casos, pueden aparecer como elementos HTML nativos extendidos con el atributo especial `is`.

## Utilizando componentes

---

### # Registro

Hemos aprendido en las secciones anteriores que podemos crear una nueva instancia de Vue con:

JS

```
new Vue({  
  el: '#some-element',  
  // opciones  
})
```

Para registrar un componente global, puedes utilizar `Vue.component(tagName, options)`. Por ejemplo:

JS

```
Vue.component('my-component', {  
  // opciones  
})
```

!

Nota que Vue no te obliga a utilizar las **reglas de W3C** para nombres de etiquetas personalizadas (todo en minúscula, con un guión medio) aunque seguir esta convención es considerado una buena práctica.

Una vez registrado, un componente puede ser utilizado en la plantilla de una instancia como un elemento personalizado `<my-component></my-component>` . Asegúrate que el componente este registrado **antes** de crear la instancia principal de Vue. Aquí hay un ejemplo completo:

HTML

```
<div id="example">
  <my-component></my-component>
</div>
```

JS

```
// registro
Vue.component('my-component', {
  template: '<div>A custom component!</div>'
})

// crear la instancia principal
new Vue({
  el: '#example'
})
```

Lo cual renderizará:

HTML

```
<div id="example">
  <div>A custom component!</div>
</div>
```

A custom component!

## # Registro local

No tienes que registrar cada componente globalmente. Puede hacer que un componente este disponible solo en el ámbito de otro componente/instancia registrándolo en la opción

`components` :

JS

```
var Child = {
  template: '<div>A custom component!</div>'
}

new Vue({
```

```
// ...
components: {
  // <my-component> solo estará disponible en la plantilla del padre
  'my-component': Child
}
})
```

El mismo encapsulamiento aplica para otras características registrables de Vue, como las directivas.

## # Advertencias en el análisis de plantillas del DOM

Cuando utilizas el DOM como tu plantilla (por ejemplo, utilizando la opción `el` para montar un elemento con contenido existente), estarás sujeto a algunas restricciones que son inherentes a como trabaja el HTML, porque Vue solo puede recuperar el contenido de la plantilla **luego** que el navegador lo haya analizado y normalizado. Incluso, algunos elementos como `<ul>` , `<ol>` , `<table>` y `<select>` tienen restricciones acerca de que otros elementos pueden aparecer dentro de ellos, y otros como `<option>` solo pueden aparecer dentro de ciertos otros.

Esto conducirá a problemas cuando se utilicen componentes personalizados con elementos que tengas esas restricciones, por ejemplo:

```
HTML
<table>
  <my-row>...</my-row>
</table>
```

El componente personalizado `<my-row>` será marcado como contenido inválido, causando por ende errores en la salida renderizada. Una solución alternativa es utilizar el atributo especial `is` :

```
HTML
<table>
  <tr is="my-row"></tr>
</table>
```

Debe notarse que estas limitaciones no aplican si estás utilizando plantillas de texto de alguna de las siguientes fuentes:

- `<script type="text/x-template">`
- Plantillas de texto JavaScript en línea
- Componentes `.vue`

Por lo tanto, prefiere utilizar plantillas de texto siempre que sea posible.

## # **data** debe ser una función

La mayoría de las opciones que pueden ser pasadas a un constructor de Vue pueden ser utilizadas en un componente, con un caso especial: **data** debe ser una función. De hecho, si intentas esto:

JS

```
Vue.component('my-component', {
  template: '<span>{{ message }}</span>',
  data: {
    message: 'hello'
  }
})
```

Vue se detendrá y emitirá advertencias en la consola, diciéndote que **data** debe ser una función para instancias de componentes. Es bueno entender por qué existe la regla, así que hagamos algo de trampa:

HTML

```
<div id="example-2">
  <simple-counter></simple-counter>
  <simple-counter></simple-counter>
  <simple-counter></simple-counter>
</div>
```

JS

```
var data = { counter: 0 }

Vue.component('simple-counter', {
  template: '<button v-on:click="counter += 1">{{ counter }}</button>',
  // 'data' es técnicamente una función, por lo que Vue no
  // se quejará, pero estamos devolviendo la misma referencia
  // de objeto en cada instancia de componente
  data: function () {
    return data
  }
})

new Vue({
  el: '#example-2'
})
```



0 0 0

Dado que las tres instancias del componente comparten el mismo objeto `data` , ¡incrementar un contador los incrementa a todos! Ouch. Arreglemos esto retornando en su lugar un objeto de datos nuevo:

JS

```
data: function () {  
  return {  
    counter: 0  
  }  
}
```

Ahora todos nuestros contadores tienen su propio estado interno:

0 0 0

## # Componiendo componentes

Los componentes están pensados para ser utilizados en conjunto, comunmente en relaciones padre-hijo: el componente A puede utilizar al componente B en su propia plantilla. Necesitarán inevitablemente comunicarse entre ellos: el padre puede necesitar pasar datos hacia el hijo, mientras que el hijo puede necesitar informar de algo que ha ocurrido al padre. Sin embargo, también es muy importante mantener lo más posiblemente desacopados al padre e hijo a través de una interface definida claramente. Esto asegura que el código de cada componente puede ser escrito y se puede razonar aisladamente acerca de él, haciéndolos más mantenibles y potencialmente fáciles de reutilizar.

In Vue.js, la relación padre-hijo entre componentes puede ser resumida como **propiedades hacia abajo, eventos hacia arriba**. El padre pasa datos al hijo a través de **propiedades** y el hijo envía mensajes al padre a través de **eventos**. Veamos como trabajan.

 props down, events up

# Propiedades

---

## # Pasando datos a través de propiedades

Cada instancia de componente tiene su propio **ámbito aislado**. Esto significa que no puedes (y no debes) referenciar directamente datos del padre en la plantilla del componente hijo. Los datos pueden ser pasados hacia el hijo utilizando **propiedades**.

Una propiedad es un atributo personalizado para pasar información desde componentes padres. Un componente hijo necesita declarar explícitamente las propiedades que espera recibir utilizando la **opción** **props** :

JS

```
Vue.component('child', {  
  // declara las propiedades  
  props: ['message'],  
  // como 'data', las propiedades pueden ser utilizadas dentro de las  
  // plantillas y está disponibles en la vm como this.message  
  template: '<span>{{ message }}</span>'  
})
```

Entonces, puedes pasar una cadena de texto plana como:

HTML

```
<child message="hello!"></child>
```

Resultado:

hello!

## # camelCase vs. kebab-case

Los atributos HTML no distinguen entre mayúsculas y minúsculas, por lo que cuando utilices plantillas que no sean de texto, los nombres de propiedades en formato *camelCase* necesitan ser escritas con su equivalente *kebab-case*:

```
Vue.component('child', {
  // camelCase en JavaScript
  props: ['myMessage'],
  template: '<span>{{ myMessage }}</span>'
})
```

HTML

```
<!-- kebab-case en HTML -->
<child my-message="hello!"></child>
```

De nuevo, si estás utilizando plantillas de texto, entonces esta limitación no aplica.

## # Propiedades dinámicas

Similar a enlazar atributos normales a una expresión, podemos utilizar `v-bind` para enlazar dinámicamente propiedades con datos en el padre. Cuando los datos sean actualizados en el padre, los cambios fluirán hacia el hijo:

HTML

```
<div>
  <input v-model="parentMsg">
  <br>
  <child v-bind:my-message="parentMsg"></child>
</div>
```

Normalmente es más sencillo utilizar la forma corta de `v-bind` :

HTML

```
<child :my-message="parentMsg"></child>
```

Resultado:

Message from parent

## # Literal vs dinámico

Un error común que los principiantes suelen cometer es tratar de pasar un número utilizando la sintaxis literal:

HTML

```
<!-- esto pasa la cadena de texto "1" -->
<comp some-prop="1"></comp>
```

Sin embargo, dado que esta es una propiedad literal, su valor se pasa como la cadena de texto `"1"` en lugar de un número. Si en realidad queremos pasar un número JavaScript, necesitamos utilizar `v-bind` para que su valor sea evaluado como una expresión JavaScript:

HTML

```
<!-- esto pasa el numero 1 -->
<comp v-bind:some-prop="1"></comp>
```

## # Flujo de datos en un solo sentido

Todas las propiedades establecen un enlace de **un solo sentido** entre la propiedad del hijo y la del padre: cuando la propiedad del padre cambia, fluirá hacia el hijo, pero no en el sentido inverso. Esto previene que los componentes hijo modifiquen accidentalmente el estado del padre, lo cual puede hacer que el flujo de tu aplicación sea difícil de razonar.

Además, cada vez que el componente padre es actualizado, todas las propiedades en el componente hijo serán refrescadas con el último valor. Esto significa que **no** deberías modificar una propiedad en un componente hijo. Si lo haces, Vue te advertirá en la consola.

Hay usualmente dos casos en los que puede ser tentador modificar una propiedad:

1. La propiedad es utilizada solo para dar un valor inicial, el componente hijo solo quiere utilizarla como una propiedad de datos local luego;
2. La propiedad es pasada como un valor crudo que luego necesita ser transformado.

Las respuestas apropiadas a estos casos de uso son:

1. Define una propiedad de datos local que utilice el valor inicial de la propiedad como su valor inicial:

JS

```
props: ['initialCounter'],
data: function () {
  return { counter: this.initialCounter }
```

```
}
```

2. Define una propiedad computada que sea calculada en base al valor de la propiedad pasada:

JS

```
props: ['size'],
computed: {
  normalizedSize: function () {
    return this.size.trim().toLowerCase()
  }
}
```



Nota que los objetos y arreglos en JavaScript son pasados por referencia, por lo que si la propiedad es un arreglo u objeto, modificarlos dentro del hijo **afectará** al estado del padre.

## # Validación de propiedades

Es posible especificar en un componente requerimientos para las propiedades que recibe. Si no se cumple con un requerimiento, Vue emitirá advertencias. Esto es especialmente útil cuando estás creando componentes con la intención que sean utilizados por otros.

En lugar de definir las propiedades como un arreglo de cadenas de texto, puedes utilizar un objeto con los requerimientos de validación:

JS

```
Vue.component('example', {
  props: {
    // verificación de tipo básica (`null` significa que acepta cualquier tipo)
    propA: Number,
    // múltiples tipos posibles
    propB: [String, Number],
    // cadena de texto requerida
    propC: {
      type: String,
      required: true
    },
    // un número con valor por defecto
    propD: {
      type: Number,
      default: 100
    },
    // Los valores por defecto de objetos/arreglos deben ser devueltos
```

```

// desde una función fábrica
propE: {
  type: Object,
  default: function () {
    return { message: 'hello' }
  }
},
// función de validación personalizada
propF: {
  validator: function (value) {
    return value > 10
  }
}
}
})

```

El `type` puede ser uno de los siguientes constructores nativos:

- String
- Number
- Boolean
- Function
- Object
- Array

Además, `type` puede ser una función constructora personalizada y la verificación será hecha con `instanceof`.

Cuando una validación de propiedad falla, Vue producirá una advertencia en la consola (si estás utilizando la versión de desarrollo).

## Eventos personalizados

---

Hemos aprendido que los padres pueden pasar datos hacia los hijos utilizando propiedades. Pero, ¿cómo nos comunicamos con el padre cuando algo sucede? Aquí es donde el sistema de eventos personalizados de Vue entra en juego.

### # Utilizando `v-on` con eventos personalizados

Cada instancia de Vue implementa una **interface de eventos**, lo cual significa que puede

- Escuchar un evento utilizando `$on(eventName)`
- Emitir un evento utilizando `$emit(eventName)`

! Nota que el sistema de eventos de Vue está separado de la **API EventTarget API** de los navegadores. Aunque funcionan similarmente, `$on` y `$emit` **not** son alias para `addEventListener` y `dispatchEvent`.

Además, un componente padre puede escuchar los eventos emitidos por un componente hijo utilizando `v-on` directamente en la plantilla donde el componente hijo está siendo utilizado.

! No puedes utilizar `$on` para escuchar eventos emitidos por hijos. Debes utilizar `v-on` directamente en la plantilla, como en el ejemplo debajo.

Aquí hay un ejemplo:

HTML

```
<div id="counter-event-example">
  <p>{{ total }}</p>
  <button-counter v-on:increment="incrementTotal"></button-counter>
  <button-counter v-on:increment="incrementTotal"></button-counter>
</div>
```

JS

```
Vue.component('button-counter', {
  template: '<button v-on:click="increment">{{ counter }}</button>',
  data: function () {
    return {
      counter: 0
    }
  },
  methods: {
    increment: function () {
      this.counter += 1
      this.$emit('increment')
    }
  },
})

new Vue({
  el: '#counter-event-example',
  data: {
    total: 0
  },
})
```

```

methods: {
  incrementTotal: function () {
    this.total += 1
  }
}
})

```

0

0 0

En este ejemplo, es importante notar que el componente hijo todavía está completamente desacoplado de lo que pasa fuera de él. Todo lo que hace es reportar información acerca de su propia actividad, en caso de que a un componente padre pueda importarle.

### Enlazando eventos nativos a componentes

Puede haber momentos en los que quieras escuchar un evento nativo en el elemento raíz de un componente. En estos casos, puedes utilizar el modificador `.native` para `v-on`. Por ejemplo:

HTML

```
<my-component v-on:click.native="doTheThing"></my-component>
```

## # Componentes de campos de formularios utilizando eventos personalizados

Los eventos personalizados también pueden ser utilizados para crear campos personalizados que funcionen con `v-model`. Recuerda:

HTML

```
<input v-model="something">
```

es simplemente azúcar sintáctico para:

HTML

```
<input v-bind:value="something" v-on:input="something = $event.target.value">
```

Cuando se utiliza con un componente, se simplifica a:



```
HTML
<custom-input v-bind:value="something" v-on:input="something = arguments[0]"></custom-input>
```

Entonces, para que un componente trabaje con `v-model`, debe:

- aceptar una propiedad `value`
- emitir un evento `input` con el nuevo valor

Veámoslo en acción con un ejemplo muy sencillo de un campo de entrada de dinero:

```
HTML
<currency-input v-model="price"></currency-input>
```

```
JS
Vue.component('currency-input', {
  template: '\
    <span>\
      $\
      <input\
        ref="input"\
        v-bind:value="value"\
        v-on:input="updateValue($event.target.value)"\
      >\
    </span>\
  ',
  props: ['value'],
  methods: {
    // En lugar de actualizar el valor directamente, este
    // método es utilizado para dar formato y aplicar restricciones
    // al valor de la entrada
    updateValue: function (value) {
      var formattedValue = value
      // Remueve espacios en blanco de ambos lados
      .trim()
      // Acorta a dos decimales
      .slice(0, value.indexOf('.') + 3)
      // Si el valor no estaba normalizado aún,
      // lo sobrescribimos manualmente
      if (formattedValue !== value) {
        this.$refs.input.value = formattedValue
      }
      // Emite el valor numérico a través del evento 'input'
      this.$emit('input', Number(formattedValue))
    }
  }
})
```

\$

La implementación de arriba es bastante inocente. Por ejemplo, los usuarios pueden ingresar múltiples puntos e incluso letras algunas veces, ¡Aagh! Para aquellos que quieran ver un ejemplo no tan trivial, aquí hay un ejemplo más robusto de un campo de entrada de dinero:

Result   HTML   JavaScript

[Edit in JSFiddle](#)



Price \$   
Shipping \$   
Handling \$   
Discount \$

Total: \$0.00

La interface de eventos puede ser usada para crear campos de entrada poco comunes. Por ejemplo, imagina estas posibilidades:

HTML

```
<voice-recognizer v-model="question"></voice-recognizer>  
<webcam-gesture-reader v-model="gesture"></webcam-gesture-reader>  
<webcam-retinal-scanner v-model="retinalImage"></webcam-retinal-scanner>
```

## # Comunicación entre componentes sin relación padre/hijo

En ocasiones dos componentes pueden necesitar comunicarse entre ellos pero no son padre/hijo. En escenarios simples, puedes utilizar una instancia de Vue vacía como un bus central de eventos:

JS

```
var bus = new Vue()
```

JS

```
// en un método del componente A
bus.$emit('id-selected', 1)
```

JS

```
// en el _hook created_ del componente B
bus.$on('id-selected', function (id) {
  // ...
})
```

En escenarios más complejos, deberías considerar emplear un **patrón de manejo de estado dedicado**.

## Distribución de contenido con slots

---

Cuando se utilizan componentes, es usual querer componerlos como:

HTML

```
<app>
  <app-header></app-header>
  <app-footer></app-footer>
</app>
```

Hay dos cosas a notar aquí:

1. El componente `<app>` no sabe que contenido puede ser necesario en el elemento donde será montado. Eso lo decide cualquier componente que esté utilizando `<app>`.
2. El componente `<app>` probablemente tenga su propia plantilla.

Para lograr que la composición funcione, necesitamos una manera de entrelazar el “contenido” del padre y la propia plantilla del componente. Este es un proceso llamado **distribución de contenido** (o “transclusión” si estás familiarizado con Angular). Vue.js implementa una API de distribución de contenido que está modelada basada en el **borrador actual de la especificación de componentes web**, utilizando el elemento especial `<slot>` para trabajar como contenedor de distribución para el contenido original.

## # Ámbito de compilación

Antes de sumergirnos en la API, clarifiquemos primero en que ámbito son compilados los contenidos. Imagina una plantilla como esta:

HTML

```
<child-component>
  {{ message }}
</child-component>
```

¿ `message` debería estar enlazado a los datos del padre o del hijo? La respuesta es al padre. Suna The answer is the parent. Una regla sencilla para recordar el ámbito de los componentes es:

**Todo lo que se encuentre en la plantilla del padre es compilado en el ámbito del padre; todo lo que se encuentra en la plantilla del hijo, es compilado en el ámbito del hijo.**

Un error común es tratar de enlazar una directiva en la plantilla del padre a un método/propiedad del hijo:

HTML

```
<!-- NO funciona -->
<child-component v-show="someChildProperty"></child-component>
```

Asumiendo que `someChildProperty` es una propiedad en el componente hijo, los ejemplos anteriores no funcionarán. La plantilla del padre no tiene conocimiento del estado de un componente hijo.

Si necesitas enlazar directivas en el ámbito de un hijo en el nodo raíz de un componente, debes hacerlo en la plantilla propia de ese componente:

JS

```
Vue.component('child-component', {
  // esto funciona, porque estamos en el ámbito correcto
  template: '<div v-show="someChildProperty">Child</div>',
  data: function () {
    return {
      someChildProperty: true
    }
  }
})
```

De manera similar, el contenido distribuido será compilado en el ámbito padre.

## # Slot único

El contenido del padre será **descartado** a menos que la plantilla del componente hijo contenga por lo menos un contenedor `<slot>`. Cuando haya solo un *slot* sin atributos, el contenido completo será insertado en su posición en el DOM, reemplazándolo.

Cualquier contenido originalmente dentro de la etiqueta `<slot>` es considerado **por defecto**. El contenido por defecto es compilado en el ámbito del hijo y solo será mostrado si el elemento de alojamiento está vacío y no tiene contenido para ser insertado.

Imagina que tenemos un componente llamado `my-component` con la siguiente plantilla:

HTML

```
<div>
  <h2>I'm the child title</h2>
  <slot>
    This will only be displayed if there is no content
    to be distributed.
  </slot>
</div>
```

Y el padre que utiliza el componente:

HTML

```
<div>
  <h1>I'm the parent title</h1>
  <my-component>
    <p>This is some original content</p>
    <p>This is some more original content</p>
  </my-component>
</div>
```

El resultado renderizado será:

HTML

```
<div>
  <h1>I'm the parent title</h1>
  <div>
    <h2>I'm the child title</h2>
    <p>This is some original content</p>
    <p>This is some more original content</p>
  </div>
</div>
```

## # Slots con nombre

Los elementos `<slot>` tienen un atributo especial, `name`, el cual puede ser utilizado para personalizar aún más como debe ser distribuido el contenido. Puedes tener múltiples *slots* con diferentes nombres. Un *slot* con nombre se emparejará con cualquier elemento que tenga su correspondiente atributo `slot` en el fragmento de contenido.

Todavía puede haber un *slot* sin nombre, el cual es el **slot por defecto** que captura cualquier contenido que no haya coincidido anteriormente. Si no hay un *slot* por defecto, el contenido que no haya coincidido será descartado.

Por ejemplo, imagina que tenemos un componente `app-layout` con la siguiente plantilla:

HTML

```
<div class="container">
  <header>
    <slot name="header"></slot>
  </header>
  <main>
    <slot></slot>
  </main>
  <footer>
    <slot name="footer"></slot>
  </footer>
</div>
```

La estructura del padre:

HTML

```
<app-layout>
  <h1 slot="header">Here might be a page title</h1>

  <p>A paragraph for the main content.</p>
  <p>And another one.</p>

  <p slot="footer">Here's some contact info</p>
</app-layout>
```

El renderizado resultante será:

HTML

```
<div class="container">
  <header>
    <h1>Here might be a page title</h1>
  </header>
  <main>
    <p>A paragraph for the main content.</p>
```

```
<p>And another one.</p>
</main>
<footer>
  <p>Here's some contact info</p>
</footer>
</div>
```

La API de distribución de contenido es un mecanismo muy útil cuando se diseñan componentes pensados para utilizarse compuestos con otros.

## # Slots con ámbito

### Nuevo en 2.1.0

Un *slot* con ámbito es un tipo especial de *slot* que funciona como una plantilla reusable (a la que se pueden pasar datos) en lugar de elementos-ya-renderizados.

En un componente hijo, simplemente pasa datos a un *slot* como si estuvieses pasando propiedades a un componente:

HTML

```
<div class="child">
  <slot text="hello from child"></slot>
</div>
```

En el padre, un elemento `<template>` con el atributo especial `scope` indica que es una plantilla para un *slot* con ámbito. El valor de `scope` es el nombre de una variable temporaria que mantiene el objeto de propiedades pasado desde el hijo:

HTML

```
<div class="parent">
  <child>
    <template scope="props">
      <span>hello from parent</span>
      <span>{{ props.text }}</span>
    </template>
  </child>
</div>
```

Si renderizamos lo anterior, la salida será:

HTML

```
<div class="parent">
  <div class="child">
    <span>hello from parent</span>
    <span>hello from child</span>
  </div>
</div>
```

Un caso de uso típico para *slots* con ámbito sería un componente lista que permita al consumidor del componente personalizar como debería ser renderizado cada elemento en la lista:

HTML

```
<my-awesome-list :items="items">
  <!-- los 'slots' con ámbito pueden también tener nombre -->
  <template slot="item" scope="props">
    <li class="my-fancy-item">{{ props.text }}</li>
  </template>
</my-awesome-list>
```

Y la plantilla para el componente lista:

HTML

```
<ul>
  <slot name="item"
    v-for="item in items"
    :text="item.text">
    <!-- contenido por defecto aquí -->
  </slot>
</ul>
```

## Componentes dinámicos

---

Puedes utilizar el mismo punto de montaje e intercambiar dinámicamente múltiples componentes utilizando el elemento reservado `<component>` y enlazarlo dinámicamente a su atributo `is` :

JS

```
var vm = new Vue({
  el: '#example',
  data: {
    currentView: 'home'
  },
  components: {
    home: { /* ... */ },
    posts: { /* ... */ },
```



```

        archive: { /* ... */ }
    }
})

```

HTML

```

<component v-bind:is="currentView">
  <!-- ¡el componente cambia cuando vm.currentView cambia! -->
</component>

```

Si prefieres, puedes enlazar directamente a objetos de componente:

JS

```

var Home = {
  template: '<p>Welcome home!</p>'
}

var vm = new Vue({
  el: '#example',
  data: {
    currentView: Home
  }
})

```

## # keep-alive

Si quieres mantener en memoria los componentes que han sido sacados para preservar su estado o evitar un re-renderizado, puedes envolver un componente dinámico con un elemento

`<keep-alive>` :

HTML

```

<keep-alive>
  <component :is="currentView">
    <!-- ¡los componentes inactivos serán guardados en una memoria caché! -->
  </component>
</keep-alive>

```

Más detalles acerca de `<keep-alive>` en la [referencia de la API](#).

## Misc

---

## # Creando componentes reusables

Cuando crees componentes, es bueno tener en cuenta si tu intención es reutilizarlo en algún otro lugar luego. Está bien que componentes de un solo uso estén estrechamente acoplados, pero los componentes reusables deben definir una interface pública limpia y no suponer nada acerca del contexto en el que está siendo utilizado.

La API para un componente de Vue se divide en tres partes: propiedades, eventos y *slots*:

- Las **propiedades** permiten al ambiente externo pasar datos al componente
- Los **eventos** permiten al componente disparar efectos secundarios en el ambiente externo
- Los **Slots** permiten al ambiente externo componer al componente con contenido extra

Con la sintaxis corta para `v-bind` y `v-on`, las intenciones pueden ser comunicadas clara y exitosamente a la plantilla:

HTML

```
<my-component
  :foo="baz"
  :bar="qux"
  @event-a="doThis"
  @event-b="doThat"
>
  
  <p slot="main-text">Hello!</p>
</my-component>
```

## # Referencia a componentes hijo

A pesar de la existencia de propiedades y eventos, a veces puede que necesites acceder directamente a un componente hijo en JavaScript. Para lograr esto tienes que asignarle un ID de referencia utilizando `ref`. Por ejemplo:

HTML

```
<div id="parent">
  <user-profile ref="profile"></user-profile>
</div>
```

JS

```
var parent = new Vue({ el: '#parent' })
```

```
// accede a la instancia del componente hijo
var child = parent.$refs.profile
```

Cuando `ref` se utiliza en conjunto con `v-for`, la referencia que obtendrás será un arreglo de objetos que contienen componentes hijos espejados con la fuente de datos.

! `$refs` son asignadas luego de que el componente haya sido renderizado, y no es reactivo. Está pensado como una vía de escape para la manipulación directa de hijos. Debes evitar utilizar `$refs` en las plantillas o propiedades computadas.

## # Componentes asíncronos

En aplicaciones grandes, puede que necesitemos dividir la aplicación en porciones mas chicas y cargar un componente desde el servidor solo cuando es en realidad utilizado. Para hacer esto más sencillo, Vue te permite definir tus componentes como una función fabrica que resuelve asincrónicamente la definición de ese componente. Vue ejecutará la función fábrica solo cuando el componente necesite ser renderizado y guardará en memoria caché el resultado para futuras re-renderizaciones. Por ejemplo:

JS

```
Vue.component('async-example', function (resolve, reject) {
  setTimeout(function () {
    // Pasa la definición del componente a la función callback de resolución
    resolve({
      template: '<div>I am async!</div>'
    })
  }, 1000)
})
```

La función fábrica recibe una función *callback* `resolve`, la cual debe ser llamada cuando hayas recuperado la definición del componente desde tu servidor. Puedes también ejecutar `reject(reason)` para indicar que la carga ha fallado. `setTimeout` aquí está simplemente como demostración. Como se recupera el componente es una decisión totalmente a tu criterio. Un enfoque recomendado es utilizar los componentes asíncronos junto con la **característica de división de código de Webpack**:

JS

```
Vue.component('async-webpack-example', function (resolve) {
  // Esta sintaxis 'require' especial indicará a Webpack que
  // automáticamente divida el código de tu módulos en diferentes módulos
  // los cuales serán cargados a través de peticiones Ajax.
```

```
require(['./my-async-component'], resolve)
})
```

Puedes devolver también un **Promise** en la función fábrica, por lo que con la sintaxis de Webpack 2 + ES2015 puedes hacer:

JS

```
Vue.component(
  'async-webpack-example',
  () => import('./my-async-component')
)
```

!

Si eres un usuario **Browserify** y te gustaría utilizar componentes asíncronos, desafortunadamente su creador **ha dejado muy claro** que la carga asíncrona “no es algo que Browserify soportará nunca”. Oficialmente al menos. La comunidad ha encontrado **algunas soluciones alternativas**, las cuales pueden ser útiles para aplicaciones complejas ya existentes. Para cualquier otro escenario, recomendamos utilizar Webpack para tener soporte asíncrono de primera clase incorporado.

## # Convención de nombres de componentes

Cuando registres componentes (o propiedades), puedes utilizar kebab-case, camelCase, o TitleCase. A Vue no le interesa.

JS

```
// en una definición de componente
components: {
  // registra utilizando kebab-case
  'kebab-cased-component': { /* ... */ },
  // registra utilizando camelCase
  'camelCasedComponent': { /* ... */ },
  // registra utilizando TitleCase
  'TitleCasedComponent': { /* ... */ }
}
```

Sin embargo, dentro de las plantillas HTML, debes utilizar kebab-case:

HTML

```
<!-- siempre utiliza kebab-case en plantillas HTML -->
<kebab-cased-component></kebab-cased-component>
<camel-cased-component></camel-cased-component>
<title-cased-component></title-cased-component>
```

Sin embargo, cuando utilices plantillas de texto, no estás limitado por las restricciones de HTML. Eso significa que incluso en las plantillas, puedes referenciar a tus componentes y propiedades utilizando camelCase, TitleCase, o kebab-case:

HTML

```
<!-- ¡utiliza lo que quieras en plantillas de texto! -->
<my-component></my-component>
<myComponent></myComponent>
<MyComponent></MyComponent>
```

Si tu componente no recibe contenido a través de elementos `slot`, puedes incluso hacer que se auto-cierre con una `/` luego del nombre:

HTML

```
<my-component/>
```

De nuevo, esto *solo* funciona con plantillas de texto, dado que los elementos personalizados auto-cerrados no son HTML válido y el analizador nativo de tu navegador no los entenderá.

## # Componentes recursivos

Los componentes pueden invocarse recursivamente en su propia plantilla. Sin embargo, solo pueden hacerlo con la opción `name`:

JS

```
name: 'unique-name-of-my-component'
```

Cuando registres un componente globalmente utilizando `Vue.component`, el ID global es asignado automáticamente como la opción `name` del componente.

JS

```
Vue.component('unique-name-of-my-component', {
  // ...
})
```

Si no eres cuidadoso, los componentes recursivos pueden conducir a bucles infinitos:

```
name: 'stack-overflow',
template: '<div><stack-overflow></stack-overflow></div>'
```

Un componente como el anterior resultará en un error “tamaño máximo de pila excedido”, así que asegúrate que la invocación recursiva sea condicional (quiere decir, utiliza un `v-if` que eventualmente será `false` ).

## # Referencias circulares entre componentes

Digamos que estas construyendo un árbol de directorios, como en Finder o Explorador de archivos. Puede que tengas un componente `tree-folder` con esta plantilla:

HTML

```
<p>
  <span>{{ folder.name }}</span>
  <tree-folder-contents :children="folder.children"/>
</p>
```

Luego un componente `tree-folder-contents` con esta otra plantilla:

HTML

```
<ul>
  <li v-for="child in children">
    <tree-folder v-if="child.children" :folder="child"/>
    <span v-else>{{ child.name }}</span>
  </li>
</ul>
```

Cuando miras detenidamente, verás que estos componentes serán en realidad descendientes y ancestros uno del otro en el árbol, ¡una paradoja! Cuando registras componentes globalmente con `Vue.component` , esta paradoja se resuelve automáticamente por ti. Si es tu caso, deja de leer aquí.

Sin embargo, si estas requiriendo/importando componentes utilizando un **sistema de módulos**, por ejemplo Webpack o Browserify, obtendrás un error:

```
Failed to mount component: template or render function not defined.
```

Para explicar lo que está sucediendo, llamaré a nuestros componentes A y B. El sistema de módulos ve que necesita a A, pero primero A necesita a B, pero B necesita a A, pero A necesita a B, etc. Queda trabado en un bucle, no sabiendo como resolver adecuadamente cada componente sin resolver primero el otro. Para arreglar esto, necesitamos indicarle al sistema de módulos un punto en el cual pueda decir: “A necesitará a B *en algún momento*, pero no es necesario resolver B primero”.

En nuestro caso, haré que ese punto sea el componente `tree-folder`. Sabemos que el hijo que crea la paradoja es el componente `tree-folder-contents`, por lo que esperaremos al *hook* del ciclo de vida `beforeCreate` para registrarlo:

JS

```
beforeCreate: function () {  
  this.$options.components.TreeFolderContents = require('./tree-folder-contents.vue')  
}
```

¡Problema resuelto!

## # Plantillas en línea

Cuando el atributo especial `inline-template` está presente en un componente hijo, el componente utilizará su propio contenido interno como su plantilla, en lugar de tratarlo como contenido distribuido. Esto permite una mayor flexibilidad en la creación de plantillas.

HTML

```
<my-component inline-template>  
  <div>  
    <p>These are compiled as the component's own template.</p>  
    <p>Not parent's transclusion content.</p>  
  </div>  
</my-component>
```

Sin embargo, `inline-template` hace más difícil razonar acerca del ámbito de nuestra plantilla. Como una buena práctica, es preferible definir plantillas dentro de componentes utilizando la opción `template` o en un elemento `template` dentro de un archivo `.vue`.

## # X-Templates

Otra forma de definir plantillas es dentro de un elemento `script` con el tipo `text/x-template`, y luego referenciando la plantilla con un id. Por ejemplo:

HTML

```
<script type="text/x-template" id="hello-world-template">
  <p>Hello hello hello</p>
</script>
```

JS

```
Vue.component('hello-world', {
  template: '#hello-world-template'
})
```

Esto puede ser útil para demostraciones con plantillas grandes o en aplicaciones extremadamente pequeñas, pero debe ser evitado en otro caso, porque separan las plantillas del resto de la definición del componente.

## # Componentes estáticos baratos con `v-once`

Renderizar HTML plano es muy rápido en Vue, pero en ocasiones puede que tengas un componente que contiene **mucho** contenido estático. En estos casos, puedes asegurarte que sea evaluado solo una vez y agregado a la memoria caché añadiéndole la directiva `v-once` al elemento raíz, de la siguiente forma:

JS

```
Vue.component('terms-of-service', {
  template: '\
    <div v-once>\
      <h1>Terms of Service</h1>\
      ... a lot of static content ...\
    </div>\
  ',
})
```



Avanzado

# Reactividad en profundidad

Ya hemos cubierto la mayoría de los aspectos básicos de Vue, ¡es hora de sumergirnos en las profundidades! Una de las características más distintivas de Vue es un sistema de reactividad poco intrusivo. Los modelos son simples objetos planos de JavaScript. Cuando los modificas, la vista se actualiza. Esto hace al manejo de estado algo muy simple e intuitivo, pero también es importante entender como trabaja para evitar algunas trampas comunes. En esta sección, vamos a introducirnos en algunos de los detalles de bajo nivel del sistema de reactividad de Vue.

## Cómo son registrados los cambios

---

Cuando pasas un objeto plano de JavaScript a una instancia de Vue como su opción `data`, Vue recorrerá todas sus propiedades y las convertirá en *getters/setters* usando `Object.defineProperty`. Esto funciona solo con ES5 y es una característica irreproducible, razón por la cual Vue no soporta IE8 ni versiones anteriores.

Los *getters/setters* son invisibles para el usuario, pero internamente permiten a Vue realizar seguimiento de dependencias y notificación de cambios cuando las propiedades son accedidas o modificadas. Una contra es que la consola del navegador da un formato diferente a los *getters/setters* cuando se guarda un registro de los objetos de datos convertidos, por lo que pueda que quieras instalar `vue-devtools` para obtener una interface de inspección más amigable.

Cada instancia de un componente tiene su correspondiente instancia de **observador**, la cual registra cualquier propiedad “tocada” como dependencia durante la renderización del componente. Luego, cuando un *setter* de una dependencia es disparado, notifica al observador, lo cual genera que el componente se renderice nuevamente.



## Advertencias para la detección de cambios

---

Debido a las limitaciones de JavaScript moderno (y el abandono de `Object.observe`), Vue **no puede detectar cuando se agrega o quita una propiedad**. Dado que Vue realiza la el proceso de

conversión a *getters/setters* durante la inicialización de la instancia, las propiedades deben estar presentes en el objeto `data` para que puedan convertirse y ser reactivos. Por ejemplo:

JS

```
var vm = new Vue({
  data: {
    a: 1
  }
})
// `vm.a` es reactiva ahora

vm.b = 2
// `vm.b` NO es reactiva
```

Vue no permite añadir dinámicamente propiedades reactivas a la raíz de una instancia ya creada. Sin embargo, es posible añadir propiedades reactivas a objetos anidados utilizando el método `Vue.set(object, key, value)` :

JS

```
Vue.set(vm.someObject, 'b', 2)
```

También puedes utilizar el método de instancia `vm.$set` , el cual es simplemente un alias para el método global `Vue.set` :

JS

```
this.$set(this.someObject, 'b', 2)
```

En ocasiones puede que quieras asignar algunas propiedades a un objeto existente, por ejemplo, utilizando `Object.assign()` o `_.extend()` . Sin embargo, nuevas propiedades agregadas al objeto no dispararan cambios. En esos casos, crea un nuevo objeto con las propiedades del objeto original y el que estás intentando fusionar:

JS

```
// en lugar de `Object.assign(this.someObject, { a: 1, b: 2 })`
this.someObject = Object.assign({}, this.someObject, { a: 1, b: 2 })
```

También hay algunas advertencias relacionadas a los arreglos, las cuales fueron discutidas anteriormente en la [sección de renderización de listas](#).

## Declarando propiedades reactivas

---

Dado que Vue no permite añadir dinámicamente propiedades reactivas en la raíz , debes inicializar las instancias de Vue declarando por adelantado todas las propiedades de datos, incluso si solo contienen un valor vacío:

JS

```
var vm = new Vue({
  data: {
    // declara "message" con un valor vacío
    message: ''
  },
  template: '<div>{{ message }}</div>'
})
// luego dale un valor a `message`
vm.message = 'Hello!'
```

Si no declaras `message` en la opción `data` , Vue te advertirá que la función de renderizado está tratando de acceder a una propiedad que no existe.

Hay razones técnicas detrás de esta restricción: elimina un tipo de caso extremo en el sistema de seguimiento de dependencias y también hace que las instancias de Vue trabajen mejor en conjunto a sistemas de verificación de tipos. Pero hay otra consideración importante en términos de mantenibilidad de código: el objeto `data` es como un esquema del estado de tu componente. Declarar todas las propiedades reactivas por adelantado hace que el componente sea más simple de entender cuando tengas que revisar el código u otro desarrollador tenga que leerlo.

## Cola de actualizaciones asíncronas

---

En caso que no lo hayas notado todavía, Vue realiza actualizaciones del DOM **asíncronicamente**. Cuando se observa un cambio en los datos, iniciará una cola y pondrá en espera todos los cambios en los datos que sucedan en el mismo ciclo de eventos. Si el mismo observador se dispara varias veces, será colocado en la cola solo una vez. Esta deduplicación de los cambios almacenados es importante para evitar cálculos innecesarios y manipulaciones del DOM. Luego, en el siguiente *tick* del ciclo de eventos, Vue vacía la cola y realiza el trabajo real (el cual ya se encuentra deduplicado). Internamente, Vue intenta utilizar versiones nativas de `Promise.then` y `MutationObserver` para las colas asíncronas o recurre a `setTimeout(fn, 0)` en otro caso.

Por ejemplo, cuando estableces `vm.someData = 'new value'` , el componente no se volverá a renderizar inmediatamente. Se actualizará en el siguiente *tick*, cuando la cola sea vaciada. La mayor parte del tiempo no necesitamos preocuparnos por esto pero puede ser un tanto

engorroso cuando intentas hacer algo que depende del estado del DOM luego de la actualización. Aunque Vue generalmente alienta a los desarrolladores a pensar de una forma “orientada a los datos” y evitar manipular el DOM directamente, a veces es necesario ensuciarse las manos. Para esperar a que Vue haya finalizado de actualizar el DOM luego de un cambio en los datos, puedes utilizar `Vue.nextTick(callback)` inmediatamente después de que hayan sido modificados. La *callback* será ejecutada luego de que haya sido actualizado en DOM. Por ejemplo:

HTML

```
<div id="example">{{ message }}</div>
```

JS

```
var vm = new Vue({
  el: '#example',
  data: {
    message: '123'
  }
})
vm.message = 'new message' // cambia los datos
vm.$el.textContent === 'new message' // falso
Vue.nextTick(function () {
  vm.$el.textContent === 'new message' // verdadero
})
```

Existe también el método de instancia `vm.$nextTick()`, el cual es especialmente útil dentro de componentes, porque no necesita la variable global `Vue` y el contexto `this` de la *callback* estará enlazado con la instancia Vue actual:

JS

```
Vue.component('example', {
  template: '<span>{{ message }}</span>',
  data: function () {
    return {
      message: 'not updated'
    }
  },
  methods: {
    updateMessage: function () {
      this.message = 'updated'
      console.log(this.$el.textContent) // => 'not updated'
      this.$nextTick(function () {
        console.log(this.$el.textContent) // => 'updated'
      })
    }
  }
})
```



# Efectos de transición

## Vistazo general

---

Vue provee una amplia variedad de maneras de aplicar efectos de transición cuando nuevos elementos son insertados, actualizados o quitados del DOM. Esto incluye herramientas para:

- aplicar automáticamente clases para transiciones y animaciones CSS
- integrar bibliotecas de animación CSS de terceros, como Animate.css
- utilizar JavaScript para manipular directamente el DOM durante los *hooks* de transición
- integrar bibliotecas de animación de terceros de JavaScript, como Velocity.js

En esta página, solo cubriremos las transiciones de entrada, salida o listas, pero puedes ver la siguiente sección para el **manejo de transiciones de estado**.

## Transiciones para componentes/elementos únicos

---

Vue provee un componente envolvente `transition`, permitiéndote agregar transiciones de entrada/salida para cualquier elemento o componente en los siguientes contextos:

- Renderización condicional (utilizando `v-if`)
- Visualización condicional (utilizando `v-show`)
- Componentes dinámicos
- Nodos raíz de componentes

Así es como luce un ejemplo muy simple:

```
<div id="demo">
  <button v-on:click="show = !show">
    Toggle
  </button>
  <transition name="fade">
    <p v-if="show">hello</p>
  </transition>
</div>
```

HTML

JS

```
new Vue({
  el: '#demo',
  data: {
    show: true
  }
})
```

CSS

```
.fade-enter-active, .fade-leave-active {
  transition: opacity .5s
}
.fade-enter, .fade-leave-to /* .fade-leave-active in <2.1.8 */ {
  opacity: 0
}
```

A screenshot of a web application. It features a light gray background. In the upper left, there is a small, rounded rectangular button with the text 'Toggle' in a dark gray font. Below the button, the word 'hello' is displayed in a large, bold, dark gray sans-serif font.

Cuando un elemento encerrado en un componente `transition` es insertado o removido, esto es lo que sucede:

1. Vue verificará automáticamente si el elemento objetivo tiene transiciones CSS aplicadas o no. Si tiene, las clases CSS para estas transiciones serán agregadas/quitadas en el momento oportuno.
2. Si el componente de transición provee **hooks de JavaScript**, estos *hooks* serán llamados en el momento oportuno.
3. Si no se detectan transiciones/animaciones CSS ni se proveen *hooks* de JavaScript, las operaciones para la inserción y/o remoción serán ejecutadas inmediatamente en el siguiente marco (Nota: nos referimos a un marco de animación del navegador, diferente al concepto `nextTick` de Vue).

## # Clases para la transición

Hay seis clases aplicadas a las transiciones de entrada/salida.



1. **v-enter** : Estado inicial para la transición de entrada. Se agrega antes que el elemento sea insertado, se quita un marco luego que el elemento fue insertado.
2. **v-enter-active** : Estado activo para la entrada. Es aplicada durante la fase de entrada completa. Se agrega antes que el elemento sea insertado, se quita cuando la transición/animación finaliza. Esta clase puede ser usada para definir la duración, retraso y la curva de alivio para la transición de entrada.
3. **v-enter-to** : **Solo disponible en versiones >=2.1.8.** Estado final para la entrada. Se agrega un marco después de haber insertado el elemento (al mismo tiempo que **v-enter** es removida), se quita cuando la transición/animación finaliza.
4. **v-leave** : Estado inicial para la salida. Se agrega inmediatamente cuando se dispara una transición de salida, se quita luego de un marco.
5. **v-leave-active** : Estado activo para la salida. Es aplicada durante la fase de salida completa. Se agrega inmediatamente cuando una transición de salida es disparada, se quita cuando la transición/animación finaliza. Esta clase puede ser usada para definir la duración, retraso y la curva de alivio para la transición de salida.
6. **v-leave-to** : **Solo disponible en versiones >=2.1.8.** Estado final para la salida. Se agrega un marco después de que se dispare la transición de salida (al mismo tiempo que **v-leave** es removida), se quita cuando la transición/animación finaliza.

### Diagrama de transición

A cada una de estas clases se le agregará un prefijo con el nombre de la transición. Aquí el prefijo **v-** es el valor por defecto cuando utilizas un elemento `<transition>` sin nombre. Si utilizas `<transition name="my-transition">` por ejemplo, entonces la clase **v-enter** sería **my-transition-enter**.

**v-enter-active** y **v-leave-active** te dan la habilidad de especificar diferentes curvas de alivio para transiciones de entrada/salida, como podrás ver en un ejemplo en la siguiente sección.

## # Transiciones CSS

Una de las transiciones más comunes utiliza transiciones CSS. Aquí va un pequeño ejemplo:

HTML

```
<div id="example-1">
  <button @click="show = !show">
    Toggle render
```

```

</button>
<transition name="slide-fade">
  <p v-if="show">hello</p>
</transition>
</div>

```

JS

```

new Vue({
  el: '#example-1',
  data: {
    show: true
  }
})

```

CSS

```

/* Las animaciones de entrada y salida pueden utilizar */
/* diferentes funciones y duraciones */
.slide-fade-enter-active {
  transition: all .3s ease;
}
.slide-fade-leave-active {
  transition: all .8s cubic-bezier(1.0, 0.5, 0.8, 1.0);
}
.slide-fade-enter, .slide-fade-leave-to
/* .slide-fade-leave-active for <2.1.8 */ {
  transform: translateX(10px);
  opacity: 0;
}

```

Toggle

hello

## # Animaciones CSS

Las animaciones CSS son aplicadas de la misma manera que las transiciones CSS, con la diferencia que `v-enter` no se remueve inmediatamente luego de que el elemento sea insertado, sino en el evento `animationend`.

Aquí hay un ejemplo, omitiendo las reglas de prefijos para acortar el código:

HTML

```

<div id="example-2">

```

```
<button @click="show = !show">Toggle show</button>
<transition name="bounce">
  <p v-if="show">Look at me!</p>
</transition>
</div>
```

JS

```
new Vue({
  el: '#example-2',
  data: {
    show: true
  }
})
```

CSS

```
.bounce-enter-active {
  animation: bounce-in .5s;
}
.bounce-leave-active {
  animation: bounce-out .5s;
}
@keyframes bounce-in {
  0% {
    transform: scale(0);
  }
  50% {
    transform: scale(1.5);
  }
  100% {
    transform: scale(1);
  }
}
@keyframes bounce-out {
  0% {
    transform: scale(1);
  }
  50% {
    transform: scale(1.5);
  }
  100% {
    transform: scale(0);
  }
}
```

Toggle show

Look at me!

## # Clases de transición personalizadas

Puedes especificar clases personalizadas agregando los siguientes atributos:

- `enter-class`
- `enter-active-class`
- `enter-to-class` (solo  $\geq 2.1.8$ )
- `leave-class`
- `leave-active-class`
- `leave-to-class` (solo  $\geq 2.1.8$ )

Estos valores reemplazarán a los nombres de clase por defecto. Es especialmente útil cuando quieres combinar el sistema de transición de Vue con una biblioteca de animación CSS existente, como **Animate.css**.

Aquí hay un ejemplo:

HTML

```
<link href="https://unpkg.com/animate.css@3.5.1/animate.min.css" rel="stylesheet" />

<div id="example-3">
  <button @click="show = !show">
    Toggle render
  </button>
  <transition
    name="custom-classes-transition"
    enter-active-class="animated tada"
    leave-active-class="animated bounceOutRight"
  >
    <p v-if="show">hello</p>
  </transition>
</div>
```

JS

```
new Vue({
  el: '#example-3',
  data: {
    show: true
  }
})
```

Toggle render

hello

## # Utilizando transiciones y animaciones en conjunto

Vue necesita adjuntar *event listeners* para saber cuando una transición ha finalizado. Puede ser tanto `transitionend` como `animationend`, dependiendo del tipo de regla CSS aplicada. Si solo estás utilizando una de ellas, Vue puede detectar automáticamente el tipo correcto.

Sin embargo, en algunos casos puede que quieras tener ambas en el mismo elemento, por ejemplo, una animación disparada por Vue en conjunto con un efecto de transición *hover*. En estos casos, tendrás que declarar explícitamente con el atributo `typ` el tipo del cual quieres que Vue se encargue, con un valor igual a `animation` o `transition`.

## # Hooks de JavaScript

También puedes definir hooks de JavaScript en los atributos:

HTML

```
<transition
  v-on:before-enter="beforeEnter"
  v-on:enter="enter"
  v-on:after-enter="afterEnter"
  v-on:enter-cancelled="enterCancelled"

  v-on:before-leave="beforeLeave"
  v-on:leave="leave"
  v-on:after-leave="afterLeave"
  v-on:leave-cancelled="leaveCancelled"
>
  <!-- ... -->
</transition>
```

JS

```
// ...
methods: {
  // -----
  // Entrando
  // -----

  beforeEnter: function (el) {
    // ...
  },
  // la función "done" es opcional
  // cuando se utiliza en conjunto con CSS
```

```

enter: function (el, done) {
  // ...
  done()
},
afterEnter: function (el) {
  // ...
},
enterCancelled: function (el) {
  // ...
},

// -----
// Saliendo
// -----

beforeLeave: function (el) {
  // ...
},
// la función "done" es opcional
// cuando se utiliza en conjunto con CSS
leave: function (el, done) {
  // ...
  done()
},
afterLeave: function (el) {
  // ...
},
// leaveCancelled solo está disponible con v-show
leaveCancelled: function (el) {
  // ...
}
}

```

Estos *hooks* pueden ser utilizados solos o combinados con transiciones/animaciones CSS.

!

Cuando estés utilizando solo transiciones JavaScript, las **callbacks** **done** son **obligatorias en los hooks enter y leave**. En otros casos, serán ejecutadas síncronicamente y la transición finalizará inmediatamente.

!

También es una buena idea agregar explícitamente **v-bind:css="false"** para transiciones con JavaScript únicamente para que Vue pueda saltar la detección de CSS. Esto además previene que alguna regla CSS interfiera accidentalmente con la transición.

Veamos un ejemplo. Aquí hay una transición JavaScript muy simple utilizando Velocity.js:

```

<!--
Velocity funciona de manera similar a jQuery.animate es
una excelente opción para animaciones JavaScript
-->
<script src="https://cdnjs.cloudflare.com/ajax/libs/velocity/1.2.3/velocity.min.js"

<div id="example-4">
  <button @click="show = !show">
    Toggle
  </button>
  <transition
    v-on:before-enter="beforeEnter"
    v-on:enter="enter"
    v-on:leave="leave"
    v-bind:css="false"
  >
    <p v-if="show">
      Demo
    </p>
  </transition>
</div>

```

```

new Vue({
  el: '#example-4',
  data: {
    show: false
  },

  methods: {
    beforeEnter: function (el) {
      el.style.opacity = 0
    },
    enter: function (el, done) {
      Velocity(el, { opacity: 1, fontSize: '1.4em' }, { duration: 300 })
      Velocity(el, { fontSize: '1em' }, { complete: done })
    },
    leave: function (el, done) {
      Velocity(el, { translateX: '15px', rotateZ: '50deg' }, { duration: 600 })
      Velocity(el, { rotateZ: '100deg' }, { loop: 2 })
      Velocity(el, {
        rotateZ: '45deg',
        translateY: '30px',
        translateX: '30px',
        opacity: 0
      }, { complete: done })
    }
  }
})

```

Toggle

## Transiciones en el renderizado inicial

---

Si también deseas aplicar una transición al renderizado inicial de un nodo, puedes añadir el atributo `appear` :

HTML

```
<transition appear>
  <!-- ... -->
</transition>
```

Por defecto, utilizará las transiciones especificadas para la entrada y salida. Si quisieras, podrías especificar clases CSS personalizadas:

HTML

```
<transition
  appear
  appear-class="custom-appear-class"
  appear-to-class="custom-appear-to-class" (solo >= 2.1.8)
  appear-active-class="custom-appear-active-class"
>
  <!-- ... -->
</transition>
```

y *hooks* de JavaScript personalizados:

HTML

```
<transition
  appear
  v-on:before-appear="customBeforeAppearHook"
  v-on:appear="customAppearHook"
  v-on:after-appear="customAfterAppearHook"
  v-on:appear-cancelled="customAppearCancelledHook"
>
  <!-- ... -->
</transition>
```

## Transiciones entre elementos



---

Discutiremos **las transiciones entre componentes** luego, pero también puedes aplicar transiciones al cambio de elementos puros cuando utilices `v-if` / `v-else` . Una de las transiciones más comunes entre dos elementos es la que sucede entre un contenedor de una lista y un mensaje indicando que esa lista esta vacía:

HTML

```
<transition>
  <table v-if="items.length > 0">
    <!-- ... -->
  </table>
  <p v-else>Sorry, no items found.</p>
</transition>
```

Esto funciona correctamente, pero ten en cuenta la siguiente advertencia:

!

Cuando se intercambian elementos que tienen **la misma etiqueta**, debes indicarle a Vue que son elementos diferentes, dándoles un atributo `key` único. De otra forma, el compilador de Vue solo reemplazará el contenido del elemento para ser más eficiente. Incluso cuando es técnicamente innecesario, **se considera una buena práctica agregar este atributo cuando haya múltiples elementos dentro de un componente**

`<transition>` .

Por ejemplo:

HTML

```
<transition>
  <button v-if="isEditing" key="save">
    Save
  </button>
  <button v-else key="edit">
    Edit
  </button>
</transition>
```

En estos casos, puedes utilizar el atributo `key` para generar transiciones entre diferentes estados del mismo elemento. En lugar de utilizar `v-if` y `v-else` , el ejemplo anterior puede ser reescrito como:

HTML

```
<transition>
  <button v-bind:key="isEditing">
```

```

    {{ isEditing ? 'Save' : 'Edit' }}
  </button>
</transition>

```

Incluso, es posible generar transiciones entre cualquier número de elementos, ya sea utilizando múltiples `v-if` o enlazando un solo elemento a una propiedad dinámica. Por ejemplo:

HTML

```

<transition>
  <button v-if="docState === 'saved'" key="saved">
    Edit
  </button>
  <button v-if="docState === 'edited'" key="edited">
    Save
  </button>
  <button v-if="docState === 'editing'" key="editing">
    Cancel
  </button>
</transition>

```

Lo cual puede reescribirse como:

HTML

```

<transition>
  <button v-bind:key="docState">
    {{ buttonMessage }}
  </button>
</transition>

```

JS

```

// ...
computed: {
  buttonMessage: function () {
    switch (docState) {
      case 'saved': return 'Edit'
      case 'edited': return 'Save'
      case 'editing': return 'Cancel'
    }
  }
}
}

```

## # Modos de transición

Todavía queda un problema. Intenta hacer clic en el botón debajo:

off

Mientras se realiza la transición entre el botón “on” y el botón “off”, ambos son renderizados - uno saliendo y otro entrando. Este es el comportamiento por defecto de `<transition>` - ambos ocurren simultáneamente.

En ocasiones esto funciona genial, como cuando los elementos en transición están posicionados absolutamente, uno sobre el otro:

off

Y luego quizá sean trasladados para que luzcan como una transición deslizante:

off

Sin embargo, las transiciones de entrada y salida simultáneas no siempre son deseables, por lo que Vue ofrece algunos **modos de transición** alternativos:

- **in-out** : El nuevo elemento ingresa y, cuando termina de hacerlo, el elemento actual se remueve.
- **out-in** : El elemento actual se remueve primero y, cuando termina de hacerlo, el nuevo elemento ingresa.

Actualicemos las transiciones para nuestros botones on/off con **out-in** :

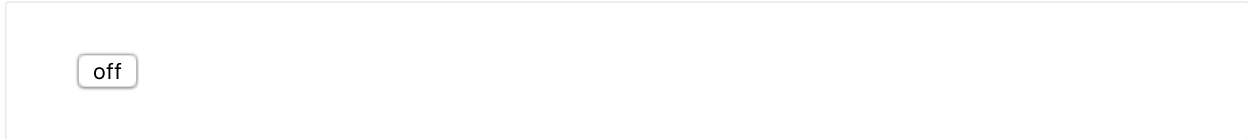
HTML

```
<transition name="fade" mode="out-in">
  <!-- ... the buttons ... -->
</transition>
```

off

Con agregar un simple atributo, hemos arreglado la transición original sin tener que agregar estilos adicionales.

El modo `in-out` no se utiliza muy seguido, pero puede ser útil en algunos casos para obtener un efecto de transición ligeramente diferente. Intentemos combinarlo con la transición deslizante-esfumante en la que estuvimos trabajando anteriormente:



Excelente, ¿no?

## Transiciones entre componentes

---

Las transiciones entre componentes son más simples - no necesitamos el atributo `key`. En su lugar, simplemente utilizamos un **componente dinámico**:

HTML

```
<transition name="component-fade" mode="out-in">
  <component v-bind:is="view"></component>
</transition>
```

JS

```
new Vue({
  el: '#transition-components-demo',
  data: {
    view: 'v-a'
  },
  components: {
    'v-a': {
      template: '<div>Component A</div>'
    },
    'v-b': {
      template: '<div>Component B</div>'
    }
  }
})
```

CSS

```
.component-fade-enter-active, .component-fade-leave-active {
  transition: opacity .3s ease;
```

```

}
.component-fade-enter, .component-fade-leave-to
/* .component-fade-leave-active for <2.1.8 */ {
  opacity: 0;
}

```

○A ○B

Component A

## Transiciones de listas

Hasta ahora, hemos utilizado transiciones para:

- Nodos individuales
- Nodos múltiples donde solo uno es renderizado a la vez

Entonces, ¿qué sucede cuando tenemos una lista completa de elementos que queremos renderizar en simultáneo, por ejemplo, con `v-for` ? En este caso, utilizaremos el componente `<transition-group>` . Antes de ver un ejemplo, hay algunas cosas importantes que saber acerca de este componente:

- A diferencia de `<transition>` , renderiza un elemento específico: por defecto, `<span>` . Puedes cambiar el elemento renderizado con el atributo `tag` .
- **Siempre se requiere** que los elementos dentro del componente tengan un atributo `key` único.

### # Transiciones de listas de entrada/salida

Ahora veamos un ejemplo simple, aplicando transiciones de entrada y salida utilizando las mismas clases CSS que antes:

HTML

```

<div id="list-demo">
  <button v-on:click="add">Add</button>
  <button v-on:click="remove">Remove</button>
  <transition-group name="list" tag="p">
    <span v-for="item in items" v-bind:key="item" class="list-item">
      {{ item }}
    </span>
  </transition-group>
</div>

```

```
</transition-group>
</div>
```

JS

```
new Vue({
  el: '#list-demo',
  data: {
    items: [1,2,3,4,5,6,7,8,9],
    nextNum: 10
  },
  methods: {
    randomIndex: function () {
      return Math.floor(Math.random() * this.items.length)
    },
    add: function () {
      this.items.splice(this.randomIndex(), 0, this.nextNum++)
    },
    remove: function () {
      this.items.splice(this.randomIndex(), 1)
    },
  }
})
```

CSS

```
.list-item {
  display: inline-block;
  margin-right: 10px;
}
.list-enter-active, .list-leave-active {
  transition: all 1s;
}
.list-enter, .list-leave-to /* .list-leave-active for <2.1.8 */ {
  opacity: 0;
  transform: translateY(30px);
}
```

Add Remove

1 2 3 4 5 6 7 8 9

Hay un problema con este ejemplo. Cuando agregues o quites un elemento, los que lo estén pegados a este se posicionaran instantáneamente en su nuevo lugar en vez de desplazarse suavemente. Lo arreglaremos luego.

## # Transiciones en los movimientos de la lista

El componente `<transition-group>` tiene otro truco escondido bajo la manga. Puede no solo animar entrada y salida sino también cambios en la posición dentro de la lista. El único nuevo concepto que debes conocer para utilizar esta característica es el agregado de la clase `v-move`, la cual se añade cuando los elementos están cambiando de posición. Como las otras clases, se le agregará un prefijo que coincide con el valor del atributo `name`, el cual puedes especificar manualmente a través del atributo `move-class`.

Esta clase es especialmente útil para indicar la curva de alivio y tiempos de la transición, como verás debajo:

HTML

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.14.1/lodash.min.js">

<div id="flip-list-demo" class="demo">
  <button v-on:click="shuffle">Shuffle</button>
  <transition-group name="flip-list" tag="ul">
    <li v-for="item in items" v-bind:key="item">
      {{ item }}
    </li>
  </transition-group>
</div>
```

JS

```
new Vue({
  el: '#flip-list-demo',
  data: {
    items: [1,2,3,4,5,6,7,8,9]
  },
  methods: {
    shuffle: function () {
      this.items = _.shuffle(this.items)
    }
  }
})
```

CSS

```
.flip-list-move {
  transition: transform 1s;
}
```

Shuffle

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9

Esto puede parecer magia pero, internamente, Vue está utilizando una técnica de animación muy simple llamada **FLIP** para aplicar transiciones suaves a un elemento desde su posición anterior a su nueva posición utilizando transformaciones.

¡Podemos combinar esta técnica con nuestra implementación anterior para animar todos los posibles cambios de nuestra lista!

HTML

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.14.1/lodash.min.js"></script>

<div id="list-complete-demo" class="demo">
  <button v-on:click="shuffle">Shuffle</button>
  <button v-on:click="add">Add</button>
  <button v-on:click="remove">Remove</button>
  <transition-group name="list-complete" tag="p">
    <span
      v-for="item in items"
      v-bind:key="item"
      class="list-complete-item"
    >
      {{ item }}
    </span>
  </transition-group>
</div>
```

JS

```
new Vue({
  el: '#list-complete-demo',
```



```

data: {
  items: [1,2,3,4,5,6,7,8,9],
  nextNum: 10
},
methods: {
  randomIndex: function () {
    return Math.floor(Math.random() * this.items.length)
  },
  add: function () {
    this.items.splice(this.randomIndex(), 0, this.nextNum++)
  },
  remove: function () {
    this.items.splice(this.randomIndex(), 1)
  },
  shuffle: function () {
    this.items = _.shuffle(this.items)
  }
}
})

```

CSS

```

.list-complete-item {
  transition: all 1s;
  display: inline-block;
  margin-right: 10px;
}
.list-complete-enter, .list-complete-leave-to
/* .list-complete-leave-active for <2.1.8 */ {
  opacity: 0;
  transform: translateY(30px);
}
.list-complete-leave-active {
  position: absolute;
}

```

Shuffle Add Remove

1 2 3 4 5 6 7 8 9

! Una aclaración importante es que estas transiciones *FLIP* no funcionan con elementos estilizados con `display: inline`. Como alternativa, puedes cambiar el valor por `display: inline-block` o posicionarlos en un contexto *flex*.

Estas animaciones FLIP no están limitadas a un solo eje. Se pueden aplicar transiciones a los elementos en un arreglo multidimensional **de la misma manera**:

## Lazy Sudoku

Keep hitting the shuffle button until you win.

Shuffle

1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9

## # Escalonando transiciones de lista

Comunicándonos con las transiciones JavaScript a través de los atributos de datos, también es posible escalar transiciones en una lista:

HTML

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/velocity/1.2.3/velocity.min.js">

<div id="staggered-list-demo">
  <input v-model="query">
  <transition-group
    name="staggered-fade"
    tag="ul"
    v-bind:css="false"
    v-on:before-enter="beforeEnter"
    v-on:enter="enter"
    v-on:leave="leave"
  >
    <li
      v-for="(item, index) in computedList"
      v-bind:key="item.msg"
      v-bind:data-index="index"
    >{{ item.msg }}</li>
  </transition-group>
</div>
```

```

new Vue({
  el: '#staggered-list-demo',
  data: {
    query: '',
    list: [
      { msg: 'Bruce Lee' },
      { msg: 'Jackie Chan' },
      { msg: 'Chuck Norris' },
      { msg: 'Jet Li' },
      { msg: 'Kung Fury' }
    ]
  },
  computed: {
    computedList: function () {
      var vm = this
      return this.list.filter(function (item) {
        return item.msg.toLowerCase().indexOf(vm.query.toLowerCase()) !== -1
      })
    }
  },
  methods: {
    beforeEnter: function (el) {
      el.style.opacity = 0
      el.style.height = 0
    },
    enter: function (el, done) {
      var delay = el.dataset.index * 150
      setTimeout(function () {
        Velocity(
          el,
          { opacity: 1, height: '1.6em' },
          { complete: done }
        )
      }, delay)
    },
    leave: function (el, done) {
      var delay = el.dataset.index * 150
      setTimeout(function () {
        Velocity(
          el,
          { opacity: 0, height: 0 },
          { complete: done }
        )
      }, delay)
    }
  }
})

```

- Bruce Lee
- Jackie Chan
- Chuck Norris
- Jet Li
- Kung Fury

## Transiciones reutilizables

---

Las transiciones pueden ser reutilizadas a lo largo de todo el sistema de componentes de Vue. Para crear una transición reutilizable, todo lo que debes hacer es ubicar un componente `<transition>` o `<transition-group>` en la raíz y luego pasar cualquier hijo dentro del componente de transición:

Aquí hay un ejemplo utilizando un componente creado con plantillas:

JS

```
Vue.component('my-special-transition', {
  template: '\
    <transition\
      name="very-special-transition"\
      mode="out-in"\
      v-on:before-enter="beforeEnter"\
      v-on:after-enter="afterEnter"\
    >\
      <slot></slot>\
    </transition>\
  ',
  methods: {
    beforeEnter: function (el) {
      // ...
    },
    afterEnter: function (el) {
      // ...
    }
  }
})
```

Y los componentes funcionales están preparados para esta tarea:

```
Vue.component('my-special-transition', {
  functional: true,
  render: function (createElement, context) {
    var data = {
      props: {
        name: 'very-special-transition'
        mode: 'out-in'
      },
      on: {
        beforeEnter: function (el) {
          // ...
        },
        afterEnter: function (el) {
          // ...
        }
      }
    }
    return createElement('transition', data, context.children)
  }
})
```

## Transiciones dinámicas

Si, ¡incluso las transiciones en Vue están orientadas a los datos! El ejemplo más básico de una transición dinámica enlaza el atributo `name` a una propiedad dinámica.

HTML

```
<transition v-bind:name="transitionName">
  <!-- ... -->
</transition>
```

Esto puede ser útil cuando has definido transiciones/animaciones CSS utilizando las convenciones de clases para las transiciones de Vue y simplemente quieres intercambiarlas.

Realmente, cualquier atributo puede ser enlazado dinámicamente. Y no solo atributos. Dado que los *hooks* de eventos son métodos, tienen acceso a cualquier porción de los datos dentro del contexto. Eso significa que dependiendo del estado de tu componente, tus transiciones JavaScript pueden comportarse diferente.

HTML

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/velocity/1.2.3/velocity.min.js">

<div id="dynamic-fade-demo">
  Fade In: <input type="range" v-model="fadeInDuration" min="0" v-bind
```

```

Fade Out: <input type="range" v-model="fadeOutDuration" min="0" v-bind:max="maxF:
<transition
  v-bind:css="false"
  v-on:before-enter="beforeEnter"
  v-on:enter="enter"
  v-on:leave="leave"
>
  <p v-if="show">hello</p>
</transition>
<button v-on:click="stop = true">Stop it!</button>
</div>

```

JS

```

new Vue({
  el: '#dynamic-fade-demo',
  data: {
    show: true,
    fadeInDuration: 1000,
    fadeOutDuration: 1000,
    maxFadeDuration: 1500,
    stop: false
  },
  mounted: function () {
    this.show = false
  },
  methods: {
    beforeEnter: function (el) {
      el.style.opacity = 0
    },
    enter: function (el, done) {
      var vm = this
      Velocity(el,
        { opacity: 1 },
        {
          duration: this.fadeInDuration,
          complete: function () {
            done()
            if (!vm.stop) vm.show = false
          }
        }
      )
    },
    leave: function (el, done) {
      var vm = this
      Velocity(el,
        { opacity: 0 },
        {
          duration: this.fadeOutDuration,
          complete: function () {
            done()
            vm.show = true
          }
        }
      )
    }
  }
})

```

```
}  
)  
}  
}  
})
```

Fade In:  Fade Out: 

hello

Stop it!

Por último, la forma esencial de crear transiciones dinámicas es a través de componentes que aceptan propiedades para modificar la naturaleza de las transiciones utilizadas. Puede sonar tonto, pero el único límite es tu imaginación.

# Transiciones en el estado

El sistema de transiciones de Vue ofrece muchas formas simples de animar entrada, salida y listado de elementos pero, ¿qué sucede si quieres animar los datos? Por ejemplo:

- números y cálculos
- colores exhibidos
- las posiciones de nodos SVG
- tamaño y otras propiedades de los elementos

Todos estos datos están almacenados como números o pueden ser convertidos fácilmente. Una vez que lo hagamos, podemos animar estos cambios de estado utilizando bibliotecas de terceros en combinación con los sistemas de reactividad y componentes de Vue.

## Animando estado con observadores

---

Los observadores nos permiten animar cambios de cualquier propiedad numérica en otra propiedad. Esto puede parecer complicado, así que veamos un ejemplo utilizando Tween.js:

HTML

```
<script src="https://unpkg.com/tween.js@16.3.4"></script>

<div id="animated-number-demo">
  <input v-model.number="number" type="number" step="20">
  <p>{{ animatedNumber }}</p>
</div>
```

JS

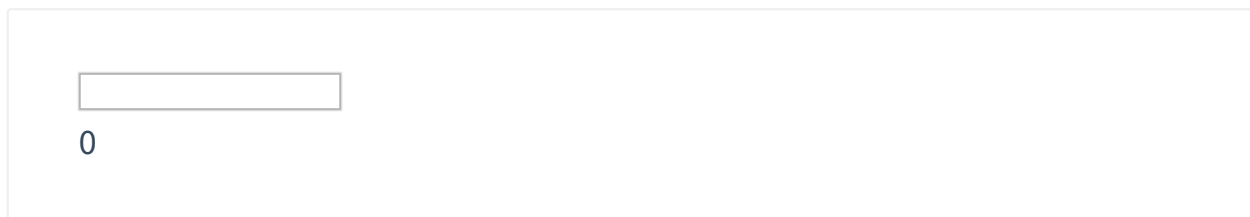
```
new Vue({
  el: '#animated-number-demo',
  data: {
    number: 0,
    animatedNumber: 0
  },
  watch: {
    number: function(newValue, oldValue) {
      var vm = this
      var animationFrame
      function animate (time) {
        TWEEN.update(time)
        animationFrame = requestAnimationFrame(animate)
      }
    }
  }
})
```



```

    }
    new TWEEN.Tween({ tweeningNumber: oldValue })
      .easing(TWEEN.Easing.Quadratic.Out)
      .to({ tweeningNumber: newValue }, 500)
      .onUpdate(function () {
        vm.animatedNumber = this.tweeningNumber.toFixed(0)
      })
      .onComplete(function () {
        cancelAnimationFrame(animationFrame)
      })
      .start()
    animationFrame = requestAnimationFrame(animate)
  }
}
})

```



Cuando actualices el número, el cambio se anima debajo del campo de texto. De esta manera logramos una hermosa demostración pero, ¿qué sucede con algo que no está directamente almacenado como un número, como un color CSS válido por ejemplo? Aquí puedes ver como lo logramos añadiendo Color.js:

HTML

```

<script src="https://unpkg.com/tween.js@16.3.4"></script>
<script src="https://unpkg.com/color.js@1.0.3/color.js"></script>

<div id="example-7">
  <input
    v-model="colorQuery"
    v-on:keyup.enter="updateColor"
    placeholder="Enter a color"
  >
  <button v-on:click="updateColor">Update</button>
  <p>Preview:</p>
  <span
    v-bind:style="{ backgroundColor: tweenedCSSColor }"
    class="example-7-color-preview"
  ></span>
  <p>{{ tweenedCSSColor }}</p>
</div>

```

```

var Color = net.brehaut.Color

new Vue({
  el: '#example-7',
  data: {
    colorQuery: '',
    color: {
      red: 0,
      green: 0,
      blue: 0,
      alpha: 1
    },
    tweenedColor: {}
  },
  created: function () {
    this.tweenedColor = Object.assign({}, this.color)
  },
  watch: {
    color: function () {
      var animationFrame
      function animate (time) {
        TWEEN.update(time)
        animationFrame = requestAnimationFrame(animate)
      }
      new TWEEN.Tween(this.tweenedColor)
        .to(this.color, 750)
        .onComplete(function () {
          cancelAnimationFrame(animationFrame)
        })
        .start()
      animationFrame = requestAnimationFrame(animate)
    }
  },
  computed: {
    tweenedCSSColor: function () {
      return new Color({
        red: this.tweenedColor.red,
        green: this.tweenedColor.green,
        blue: this.tweenedColor.blue,
        alpha: this.tweenedColor.alpha
      }).toCSS()
    }
  },
  methods: {
    updateColor: function () {
      this.color = new Color(this.colorQuery).toRGB()
      this.colorQuery = ''
    }
  }
})

```

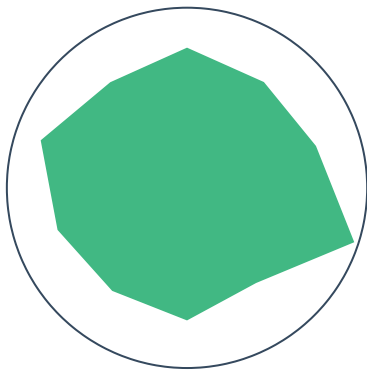
```
.example-7-color-preview {  
  display: inline-block;  
  width: 50px;  
  height: 50px;  
}
```

Preview:

#000000

## Transiciones de estado dinámicas

Así como Vue con los componentes de transición, los datos respaldando las transiciones de estado pueden ser actualizados en tiempo real, ¡lo cual es particularmente útil para el prototipado! Incluso utilizando un polígono SVG simple, puedes lograr muchos efectos que serían difíciles de concebir hasta que hayas jugado un poco con las variables.



Sides: 10



Minimum Radius: 50%



Update Interval: 500 milliseconds



**Fiddle** con el código completo del ejemplo.

## Organizando las transiciones como componentes

Manejar transiciones de estado puede incrementar rápidamente la complejidad de nuestra instancia de Vue o componente. Afortunadamente, muchas animaciones pueden ser extraídas creando componentes dedicados. Hagámoslo con el entero animado de nuestro ejemplo anterior:

HTML

```
<script src="https://unpkg.com/tween.js@16.3.4"></script>

<div id="example-8">
  <input v-model.number="firstNumber" type="number" step="20"> +
  <input v-model.number="secondNumber" type="number" step="20"> =
  {{ result }}
  <p>
    <animated-integer v-bind:value="firstNumber"></animated-integer> +
    <animated-integer v-bind:value="secondNumber"></animated-integer> =
    <animated-integer v-bind:value="result"></animated-integer>
  </p>
</div>
```

JS

```
// Esta lógica compleja de intercambio puede ser reutilizada entre
// todos los enteros que deseemos animar en nuestra aplicación.
// Los componentes también ofrecen una interface limpia para configurar
// más transiciones dinámicas y transiciones complejas
// strategies.
Vue.component('animated-integer', {
  template: '<span>{{ tweeningValue }}</span>',
  props: {
    value: {
      type: Number,
      required: true
    }
  },
  data: function () {
    return {
      tweeningValue: 0
    }
  },
  watch: {
    value: function (newValue, oldValue) {
      this.tween(oldValue, newValue)
    }
  },
  mounted: function () {
```

```

    this.tween(0, this.value)
  },
  methods: {
    tween: function (startValue, endValue) {
      var vm = this
      var animationFrame
      function animate (time) {
        TWEEN.update(time)
        animationFrame = requestAnimationFrame(animate)
      }
      new TWEEN.Tween({ tweeningValue: startValue })
        .to({ tweeningValue: endValue }, 500)
        .onUpdate(function () {
          vm.tweeningValue = this.tweeningValue.toFixed(0)
        })
        .onComplete(function () {
          cancelAnimationFrame(animationFrame)
        })
        .start()
      animationFrame = requestAnimationFrame(animate)
    }
  }
})

// ¡Toda la complejidad fue removida de la instancia principal de Vue!

new Vue({
  el: '#example-8',
  data: {
    firstNumber: 20,
    secondNumber: 40
  },
  computed: {
    result: function () {
      return this.firstNumber + this.secondNumber
    }
  }
})

```

$$\begin{array}{c}
 \boxed{\phantom{00}} + \boxed{\phantom{00}} = 60 \\
 20 + 40 = 60
 \end{array}$$

Dentro de componentes hijo, podemos utilizar cualquier combinación de estrategias de transición que hemos cubierto en esta página, junto con las ofrecidas por el **sistema de transición incorporado** de Vue. Juntas, hay pocos límites para lo que queramos lograr.



# Funciones de renderizado

## Fundamentos

---

Vue recomienda utilizar plantillas para generar tu HTML in la gran mayoría de los casos. Sin embargo, hay situaciones donde puedes necesitar todo el poder programático de JavaScript. Ahí es donde puedes utilizar **la función render**, una alternativa a las plantillas más parecida a un compilador.

Veamos un ejemplo sencillo donde la función `render` sería práctica. Digamos que quieres generar encabezados con anclas:

HTML

```
<h1>
  <a name="hello-world" href="#hello-world">
    Hello world!
  </a>
</h1>
```

Para el HTML anterior, decides que deseas la siguiente interface:

HTML

```
<anchored-heading :level="1">Hello world!</anchored-heading>
```

Cuando comienzas con un componente que simplemente genera un encabezado basado en la propiedad `level`, rápidamente llegas a lo siguiente:

HTML

```
<script type="text/x-template" id="anchored-heading-template">
  <div>
    <h1 v-if="level === 1">
      <slot></slot>
    </h1>
    <h2 v-if="level === 2">
      <slot></slot>
    </h2>
    <h3 v-if="level === 3">
      <slot></slot>
    </h3>
    <h4 v-if="level === 4">
      <slot></slot>
    </h4>
  </div>
</script>
```

```

</h4>
<h5 v-if="level === 5">
  <slot></slot>
</h5>
<h6 v-if="level === 6">
  <slot></slot>
</h6>
</div>
</script>

```

JS

```

Vue.component('anchored-heading', {
  template: '#anchored-heading-template',
  props: {
    level: {
      type: Number,
      required: true
    }
  }
})

```

Esa plantilla no parece estar bien. No solo es repetitiva, sino que estamos duplicando `<slot></slot>` en cada nivel de encabezado y tendremos que hacer lo mismo cuando agreguemos el elemento ancla. Además, hemos envuelto todo en un `div` inútil porque los componentes deben contener solo un nodo raíz.

A pesar que las plantillas funcionan muy bien para la mayoría de los componentes, es claro que este no es el caso. Intentemos reescribirlo con una función `render` :

JS

```

Vue.component('anchored-heading', {
  render: function (createElement) {
    return createElement(
      'h' + this.level, // nombre de la etiqueta
      this.$slots.default // arreglo de hijos
    )
  },
  props: {
    level: {
      type: Number,
      required: true
    }
  }
})

```

¡Mucho más simple! Casi. El código es más simple, pero requiere estar familiarizado con las propiedades de instancia de Vue. En este caso, debes saber que cuando pasas hijos



atributo `slot` a un componente, tales como el `Hello world!` dentro de `anchored-heading`, esos hijos son almacenados en la instancia del componente como `$slots.default`. Si todavía no lo hiciste, es recomendable que leas la [API de las propiedades de instancia](#) antes que utilices las funciones de renderizado.

## Parámetros de `createElement`

---

Lo segundo con lo que tendrás que familiarizarte es con como usar las características de las plantillas en la función `createElement`. Aquí están los parámetros que acepta `createElement`:

JS

```
// @returns {VNode}
createElement(
  // {String | Object | Function}
  // Un nombre de etiqueta HTML, opciones de componente, o función
  // que devuelva uno de los dos anteriores. Requerido.
  'div',

  // {Object}
  // Un objeto de datos correspondiente a los atributos
  // que usarías en una plantilla. Opcional.
  {
    // (los detalles en la siguiente sección)
  },

  // {String | Array}
  // VNodes hijos. Opcional.
  [
    createElement('h1', 'hello world'),
    createElement(MyComponent, {
      props: {
        someProp: 'foo'
      }
    }),
    'bar'
  ]
)
```

### # El objeto de datos en profundidad

Una cosa a tener en cuenta: similar a la manera en que `v-bind:class` y `v-bind:style` tienen un tratamiento especial en las plantillas, tienen sus propios campos de nivel superior en los objetos de datos `VNode`.

```

{
  // Misma API que `v-bind:class`
  'class': {
    foo: true,
    bar: false
  },
  // Misma API que `v-bind:style`
  style: {
    color: 'red',
    fontSize: '14px'
  },
  // Atributos HTML normales
  attrs: {
    id: 'foo'
  },
  // Propiedades del componente
  props: {
    myProp: 'bar'
  },
  // Propiedades del DOM
  domProps: {
    innerHTML: 'baz'
  },
  // Los controladores de eventos están anidados bajo "on", sin
  // embargo los modificadores como in v-on:keyup.enter no
  // están soportados. Tendrás que verificar manualmente
  // el código de la tecla en la función.
  on: {
    click: this.clickHandler
  },
  // Solo para componentes. Permite escuchar
  // eventos nativos en lugar de eventos emitidos desde
  // el componente utilizando vm.$emit.
  nativeOn: {
    click: this.nativeClickHandler
  },
  // Custom directives. Note that the binding's
  // oldValue cannot be set, as Vue keeps track
  // of it for you.
  directives: [
    {
      name: 'my-custom-directive',
      value: '2'
      expression: '1 + 1',
      arg: 'foo',
      modifiers: {
        bar: true
      }
    }
  ],
  // _Slots_ dentro del ámbito de la forma
  // { name: props => VNode | Array<VNode> }

```

```

scopedSlots: {
  default: props => createElement('span', props.text)
},
// El nombre del _slot_, si este componente
// es hijo de otro componente.
slot: 'name-of-slot'
// Otras propiedades especiales de alto nivel
key: 'myKey',
ref: 'myRef'
}

```

## # Ejemplo completo

Con este conocimiento, podemos ahora terminar el componente que iniciamos:

JS

```

var getChildrenTextContent = function (children) {
  return children.map(function (node) {
    return node.children
      ? getChildrenTextContent(node.children)
      : node.text
  }).join('')
}

Vue.component('anchored-heading', {
  render: function (createElement) {
    // create kebabCase id
    var headingId = getChildrenTextContent(this.$slots.default)
      .toLowerCase()
      .replace(/\W+/g, '-')
      .replace(/(^|-|\-$)/g, '')

    return createElement(
      'h' + this.level,
      [
        createElement('a', {
          attrs: {
            name: headingId,
            href: '#' + headingId
          }
        }, this.$slots.default)
      ]
    )
  },
  props: {
    level: {
      type: Number,
      required: true
    }
  }
}

```

```
})
```

## # Restricciones

### Los VNodes deben ser únicos

Todos los VNodes en un árbol de componente deben ser únicos. Eso significa que la siguiente función de renderizado es inválida:

JS

```
render: function (createElement) {  
  var myParagraphVNode = createElement('p', 'hi')  
  return createElement('div', [  
    // Aaagh - ¡VNodes duplicados!  
    myParagraphVNode, myParagraphVNode  
  ])  
}
```

Si realmente deseas duplicar el mismo elemento/componente varias veces, puedes hacerlo con una función factoría. Por ejemplo, la siguiente función de renderizado es una forma perfectamente válida de renderizar 20 párrafos idénticos:

JS

```
render: function (createElement) {  
  return createElement('div',  
    Array.apply(null, { length: 20 }).map(function () {  
      return createElement('p', 'hi')  
    })  
  )  
}
```

## Reemplazando características de plantillas con JavaScript puro

---

### # `v-if` y `v-for`

Donde algo pueda ser solucionado con JavaScript puro, las funciones de renderizado de Vue no proveen una alternativa propietaria. Por ejemplo, en una plantilla utilizando `v-if` y `v-for` :

```

<ul v-if="items.length">
  <li v-for="item in items">{{ item.name }}</li>
</ul>
<p v-else>No items found.</p>

```

Esto puede ser reescrito con `if / else` y `map` de JavaScript en una función de renderizado:

JS

```

render: function (createElement) {
  if (this.items.length) {
    return createElement('ul', this.items.map(function (item) {
      return createElement('li', item.name)
    }))
  } else {
    return createElement('p', 'No items found.')
  }
}

```

## # v-model

No existe una contraparte directa de `v-model` en las funciones de renderizado - tendrás que implementar la lógica tú mismo:

JS

```

render: function (createElement) {
  var self = this
  return createElement('input', {
    domProps: {
      value: self.value
    },
    on: {
      input: function (event) {
        self.value = event.target.value
      }
    }
  })
}

```

Este es el costo de trabajar a bajo nivel, pero también obtienes un mayor control sobre la interacción a comparación de `v-model`.

## # Modificadores de eventos y teclas

Para los modificadores de eventos `.capture` y `.once`, Vue ofrece prefijos que pueden ser utilizados con `on`:

Modificador(es)	Prefijo
<code>.capture</code>	<code>!</code>
<code>.once</code>	<code>~</code>
<code>.capture.once</code> o <code>.once.capture</code>	<code>~!</code>

Por ejemplo:

```
on: {
  '!click': this.doThisInCapturingMode,
  '~keyup': this.doThisOnce,
  '~!mouseover': this.doThisOnceInCapturingMode
}
```

Para todos los otros modificadores de eventos y teclas, no es necesario un prefijo propietario porque puedes simplemente utilizar los métodos de eventos en la función controladora:

Modificador(es)	Equivalencia
<code>.stop</code>	<code>event.stopPropagation()</code>
<code>.prevent</code>	<code>event.preventDefault()</code>
<code>.self</code>	<code>if (event.target !== event.currentTarget) return</code>
Teclas: <code>.enter</code> , <code>.13</code>	<code>if (event.keyCode !== 13) return</code> (cambia <code>13</code> por <b>otro código de tecla</b> para otros modificadores)
Teclas modificadoras: <code>.ctrl</code> , <code>.alt</code> , <code>.shift</code> , <code>.meta</code>	<code>if (!event.ctrlKey) return</code> (cambia <code>ctrlKey</code> por <code>altKey</code> , <code>shiftKey</code> , o <code>metaKey</code> , respectivamente)

Aquí hay un ejemplo con todos estos modificadores utilizados en conjunto:

```
on: {
  keyup: function (event) {
    // Retorna si el elemento emitiendo el evento no es
    // el elemento al cual está enlazado
    if (event.target !== event.currentTarget) return
    // Retorna si la tecla presionada no es _enter_
    // (13) y la tecla _shift_ no fue presionada
    // al mismo tiempo
    if (!event.shiftKey || event.keyCode !== 13) return
    // Detiene la propagación del evento
    event.stopPropagation()
    // Previene la ejecución de la controladora por defecto
    // para este evento
    event.preventDefault()
    // ...
  }
}
```

## # Slots

Puedes acceder al contenido de los *slots* estáticos como un arreglo de VNodes en **this.\$slots** :

JS

```
render: function (createElement) {
  // <div><slot></slot></div>
  return createElement('div', this.$slots.default)
}
```

Y acceder a slots del ámbito como funciones que devuelven VNodes en **this.\$scopedSlots** :

JS

```
render: function (createElement) {
  // <div><slot :text="msg"></slot></div>
  return createElement('div', [
    this.$scopedSlots.default({
      text: this.msg
    })
  ])
}
```

Para pasar slots del ámbito a un componente hijo utilizando funciones de renderizado, agrega el campo **scopedSlots** en los datos del VNode:

```
render (createElement) {
  return createElement('div', [
    createElement('child', {
      // pasa scopedSlots en el objeto de datos
      // con la forma { name: props => VNode | Array<VNode> }
      scopedSlots: {
        default: function (props) {
          return createElement('span', props.text)
        }
      }
    })
  ])
}
```

## JSX

---

Si estás escribiendo muchas funciones `render`, puede parecer desagradable escribir algo como:

```
createElement(
  'anchored-heading', {
    props: {
      level: 1
    }
  }, [
    createElement('span', 'Hello'),
    ' world!'
  ]
)
```

Especialmente cuando la versión de plantillas es mucho más simple en comparación:

```
<anchored-heading :level="1">
  <span>Hello</span> world!
</anchored-heading>
```

Por eso es que existe un **complemento de Babel** para utilizar JSX con Vue, devolviéndonos a una sintaxis más parecida a las plantillas:

```
import AnchoredHeading from './AnchoredHeading.vue'
```



```

new Vue({
  el: '#demo',
  render (h) {
    return (
      <AnchoredHeading level={1}>
        <span>Hello</span> world!
      </AnchoredHeading>
    )
  }
})

```

! Darle el alias `h` a `createElement` es una convención común que verás en el ecosistema de Vue y que es obligatoria para JSX. Si `h` no está disponible en el ámbito, tu aplicación lanzará un error.

Para más información acerca de como JSX se mapea a JavaScript, revisa [su documentación](#).

## Componentes funcionales

El componente de encabezado con anclas que creamos anteriormente es relativamente sencillo. No maneja estado, no observa ninguna porción de estado que haya recibido y no tiene métodos de ciclo de vida. Es simplemente una función con algunas propiedades.

En casos como este, podemos marcar el componente como `funcional`, lo cual significa que carecen de estado (no tienen `data`) y de instancia (no tienen un contexto `this`). Un **componente funcional** luce así:

JS

```

Vue.component('my-component', {
  functional: true,
  // Para compensar la falta de instancia,
  // se nos provee un segundo argumento de contexto
  render: function (createElement, context) {
    // ...
  },
  // Las propiedades son opcionales
  props: {
    // ...
  }
})

```

Todo lo que necesita el componente se pasa a través de `context` , el cual es un objeto que contiene:

- `props` : Un objeto con las propiedades provistas
- `children` : Un arreglo de VNode hijos
- `slots` : Una función que devuelve objetos *slot*
- `data` : El objeto *data* entero pasado al componente
- `parent` : Una referencia al componente padre

Luego de agregar `functional: true` , actualizar la función de renderizado de nuestro componente de encabezado con anclas requeriría añadir el parámetro `context` , actualizar `this.$slots.default` a `context.children` , y luego `this.level` a `context.props.level` .

Dado que los componentes funcionales son funciones, son mucho menos costosas de renderizar. También son muy útiles como componentes envolventes. Por ejemplo, cuando necesitas:

- Escoger programáticamente uno de varios componentes para delegar
- Manipular hijos, propiedades o datos antes de pasarlos a un componente hijo

Aquí hay un ejemplo de un componente `smart-list` que delega a componentes más específicos, dependiendo de las propiedades que reciba:

JS

```
var EmptyList = { /* ... */ }
var TableList = { /* ... */ }
var OrderedList = { /* ... */ }
var UnorderedList = { /* ... */ }

Vue.component('smart-list', {
  functional: true,
  render: function (createElement, context) {
    function appropriateListComponent () {
      var items = context.props.items

      if (items.length === 0)           return EmptyList
      if (typeof items[0] === 'object') return TableList
      if (context.props.isOrdered)     return OrderedList

      return UnorderedList
    }

    return createElement(
      appropriateListComponent(),
      context.data,
      context.children
    )
  },
  props: {
```

```

    items: {
      type: Array,
      required: true
    },
    isOrdered: Boolean
  }
})

```

## # slots() vs children

Te debes estar preguntando por qué necesitamos tanto `slots()` como `children` . ¿No sería `slots().default` lo mismo que `children` ? En algunos casos, si. Pero, ¿qué sucede si tienes un componente funcional con los siguientes hijos?

HTML

```

<my-functional-component>
  <p slot="foo">
    first
  </p>
  <p>second</p>
</my-functional-component>

```

Para este componente, `children` te dará ambos párrafos, `slots().default` solo te dará el segundo, y `slots().foo` solo el primero. Tener tanto `children` como `slots()` te permite elegir si este componente conoce el sistema de *slot* o tal vez delegar esa responsabilidad en otro componente simplemente pasando `children` .

## Compilación de plantillas

Puede que estés interesado en saber que las plantillas de Vue son compiladas a funciones de renderizado. Este es un detalle de implementación que normalmente no necesitas conocer, pero si desearas ver como las características específicas de las plantillas son compiladas, puede resultarte interesante. Debajo hay una pequeña demostración utilizando `Vue.compile` para compilar en vivo una plantilla en forma de cadena de texto:



render:

```
function anonymous(  
  ) {  
    with(this){return _c('div',[_c('h1',[_v("I'm a template!")]),(message)?_c('p'  
  }  
}
```

staticRenderFns:

# Directivas personalizadas

## Introducción

---

Además del conjunto por defecto de directivas en su núcleo ( `v-model` y `v-show` ), Vue también te permite registrar tus propias directivas personalizadas. Nota que en Vue 2.0, la forma básica de abstracción y reutilización de código es a través de componentes. Sin embargo, puede haber casos donde necesites acceso de bajo nivel al DOM en elementos puros, y aquí es donde las directivas personalizadas todavía serían útiles. Un ejemplo sería hacer foco en un elemento `input` como el siguiente:



Cuando la página carga, ese elemento recibe el foco (nota: el auto foco no funciona en Safari). De hecho, si no has hecho clic en ningún lado desde que visitaste esta página, el campo de texto anterior debería tener el foco ahora mismo. Codifiquemos una directiva que lo realice:

JS

```
// Registra una directiva personalizada global llamada v-focus
Vue.directive('focus', {
  // Cuando el elemento enlazado es insertado en el DOM
  inserted: function (el) {
    // Pon el foco en el elemento
    el.focus()
  }
})
```

Si en cambio quieres registrar una directiva localmente, los componentes también aceptan la opción `directives` :

JS

```
directives: {
  focus: {
    // definición de la directiva
  }
}
```

Luego dentro de una plantilla, puedes utilizar el nuevo atributo `v-focus` en cualquier elemento, por ejemplo:

```
<input v-focus>
```

HTML

## Funciones *hook*

---

Un objeto de definición de directivas puede proveer distintas funciones *hook* (todas opcionales):

- `bind` : ejecutada solo una vez, cuando la directiva es enlazada en primer turno al elemento. Aquí es donde puedes realizar las configuraciones necesarias una vez.
- `inserted` : ejecutada cuando el elemento enlazado ha sido insertado en su nodo padre (esto solo garantiza que el nodo padre exista, no que este necesariamente dentro del documento).
- `update` : ejecutada luego que el componente contenedor ha sido actualizado, **pero posiblemente antes que su hijo hayan sido actualizados**. El valor de la directiva puede haber cambiado o no, pero puedes saltarte actualizaciones innecesarias comparando el valor actual del enlace con el anterior (más información debajo en parámetros *hook*).
- `componentUpdated` : ejecutada luego de que el componente contenedor **y sus hijos** han sido actualizados.
- `unbind` : ejecutada solo una vez, cuando la directiva es desenlazada del elemento.

Exploraremos los parámetros pasados a estos *hooks* (por ejemplo: `el` , `binding` , `vnode` , y `oldVnode` ) en la siguiente sección.

## Parámetros de los *hooks* de directivas

---

A los *hooks* de directivas se les pasan estos parámetros:

- **el**: El elemento al cual está enlazada la directiva. Puede ser usado para manipular directamente el DOM.
- **binding**: Un objeto que contiene las siguientes propiedades.
  - **name**: El nombre de la directiva, sin el prefijo `v-` .

- **value:** El valor pasado a la directiva. Por ejemplo en `v-my-directive="1 + 1"` , el valor sería `2` .
- **oldValue:** El valor anterior, solo disponible en `update` y `componentUpdated` . Existe haya o no cambiado.
- **expression:** La expresión del enlace como una cadena de texto. Por ejemplo in `v-my-directive="1 + 1"` , la expresión sería `"1 + 1"` .
- **arg:** Los parámetros pasados a la directivas, si existen. Por ejemplo en `v-my-directive:foo` , el parámetro sería `"foo"` .
- **modifiers:** Un objeto que contiene modificadores, si existen. Por ejemplo en `v-my-directive.foo.bar` , el objeto de modificadores sería `{ foo: true, bar: true }` .
- **vnode:** El nodo virtual generado por el compilador de Vue. Ingresa a la **API de VNode** para más detalles.
- **oldVnode:** El nodo virtual anterior, solo disponible en los *hooks* `update` y `componentUpdated` .

Dejando de lado `el` , deberías tratar estos argumentos como de solo lectura y nunca modificarlos. Si necesitas compartir información entre *hooks*, es recomendable hacerlo a través del **dataset** del elemento.

Un ejemplo de una directiva personalizada utilizando algunas de estas propiedades:

HTML

```
<div id="hook-arguments-example" v-demo:hello.a.b="message"></div>
```

JS

```
Vue.directive('demo', {
  bind: function (el, binding, vnode) {
    var s = JSON.stringify
    el.innerHTML =
      'name: ' + s(binding.name) + '<br>' +
      'value: ' + s(binding.value) + '<br>' +
      'expression: ' + s(binding.expression) + '<br>' +
      'argument: ' + s(binding.arg) + '<br>' +
      'modifiers: ' + s(binding.modifiers) + '<br>' +
      'vnode keys: ' + Object.keys(vnode).join(', ')
  }
})

new Vue({
  el: '#hook-arguments-example',
  data: {
    message: 'hello!'
  }
})
```

```
})
```

```
name: "demo"  
value: "hello!"  
expression: "message"  
argument: "hello"  
modifiers: {"a":true,"b":true}  
vnode keys: tag, data, children, text, elm, ns, context, functionalContext,  
key, componentOptions, componentInstance, parent, raw, isStatic,  
isRootInsert, isComment, isCloned, isOnce
```

## Abreviación de funciones

---

En muchos casos, puedes querer el mismo comportamiento para `bind` y `update`, pero no te interesan los otros *hooks*. Por ejemplo:

JS

```
Vue.directive('color-swatch', function (el, binding) {  
  el.style.backgroundColor = binding.value  
})
```

## Objetos literales

---

Si tus directivas necesitan múltiples valores, puedes pasar un objeto literal de JavaScript. Recuerda, las directivas pueden recibir cualquier expresión JavaScript válida.

HTML

```
<div v-demo="{ color: 'white', text: 'hello!' }"></div>
```

JS

```
Vue.directive('demo', function (el, binding) {  
  console.log(binding.value.color) // => "white"  
  console.log(binding.value.text)  // => "hello!"  
})
```





# Mixins

## Lo básico

---

Los *mixins* son una manera flexible de distribuir funcionalidades reusables para componentes de Vue. Un objeto *mixin* puede contener cualquier opción de componente. Cuando un componente utiliza un *mixin*, todas las opciones en él serán fusionadas con las opciones propias del componente.

Oor ejemplo:

JS

```
// Define un objeto _mixin_
var myMixin = {
  created: function () {
    this.hello()
  },
  methods: {
    hello: function () {
      console.log('hello from mixin!')
    }
  }
}

// Define un componente que use ese _mixin_
var Component = Vue.extend({
  mixins: [myMixin]
})

var component = new Component() // -> "hello from mixin!"
```

## Opciones de fusionado

---

Cuando un *mixin* y el propio componente contienen opciones solapadas, serán “fusionadas” utilizando estrategias apropiadas. Por ejemplo, las funciones *hook* serán añadidas a un arreglo para que todas ellas sean ejecutadas. Además, los *hooks* del *mixin* serán ejecutados antes que los propios del componente:

JS

```
var mixin = {
```

```

    created: function () {
      console.log('mixin hook called')
    }
  }

  new Vue({
    mixins: [mixin],
    created: function () {
      console.log('component hook called')
    }
  })

// -> "mixin hook called"
// -> "component hook called"

```

Las opciones que esperarán valores de tipo objeto, por ejemplo `methods` , `components` y `directives` , serán fusionadas en un mismo objeto. Las opciones del componente tendrán mayor precedencia cuando haya llaves en conflicto en estos objetos:

JS

```

var mixin = {
  methods: {
    foo: function () {
      console.log('foo')
    },
    conflicting: function () {
      console.log('from mixin')
    }
  }
}

var vm = new Vue({
  mixins: [mixin],
  methods: {
    bar: function () {
      console.log('bar')
    },
    conflicting: function () {
      console.log('from self')
    }
  }
})

vm.foo() // -> "foo"
vm.bar() // -> "bar"
vm.conflicting() // -> "from self"

```

Nota que las mismas estrategias de fusión son utilizadas en `Vue.extend()` .

## Mixin globales

---

Puedes aplicar globalmente un *mixin*. ¡Ten cuidado! Una vez que aplicas globalmente un *mixin*, afectará a **cada** instancia de Vue que crees luego. Cuando se utiliza apropiadamente, puede ser utilizado para inyectar lógica de procesamiento para opciones personalizadas:

JS

```
// inyecta una función controladora para la opción personalizada `myOption`
Vue.mixin({
  created: function () {
    var myOption = this.$options.myOption
    if (myOption) {
      console.log(myOption)
    }
  }
})

new Vue({
  myOption: 'hello!'
})
// -> "hello!"
```

!

Utiliza *mixins* globales con cuidado y no muy seguido, porque afecta a cada una de las instancias de Vue creadas, inclusive los componentes de terceros. La mayor parte del tiempo, deberías usarlos para el manejo de opciones personalizadas como la que mostramos en el ejemplo anterior. También es una buena idea empaquetarlos como **complementos** para evitar ejecutarlos más de una vez.

## Estrategias de fusión de opciones personalizadas

---

Cuando fusionas opciones personalizadas, utilizan una estrategia por defecto, la cual simplemente sobrescribe el valor actual. Si deseas que una opción personalizada sea fusionada utilizando lógica propia, necesitas agregar una función a `Vue.config.optionMergeStrategies` :

JS

```
Vue.config.optionMergeStrategies.myOption = function (toVal, fromVal) {
  // devuelve mergedVal
}
```

Para la mayoría de las opciones basadas en objetos, puedes utilizar la misma estrategia que en `methods` :

JS

```
var strategies = Vue.config.optionMergeStrategies
strategies.myOption = strategies.methods
```

Puedes encontrar un ejemplo más avanzado en la estrategia de fusión de **Vuex 1.x**:

JS

```
const merge = Vue.config.optionMergeStrategies.computed
Vue.config.optionMergeStrategies.vuex = function (toVal, fromVal) {
  if (!toVal) return fromVal
  if (!fromVal) return toVal
  return {
    getters: merge(toVal.getters, fromVal.getters),
    state: merge(toVal.state, fromVal.state),
    actions: merge(toVal.actions, fromVal.actions)
  }
}
```

# Complementos

## Desarrollando un complemento

---

Los complementos normalmente añaden funcionalidad a nivel global a Vue. No hay un ámbito estrictamente definido para un complemento - hay diferentes tipos típicos de complemento que puedes desarrollar:

1. Añadir algunos métodos o propiedades globales, por ejemplo **vue-element**
2. Añadir uno o más recursos globales: directivas/filtros/transiciones, por ejemplo **vue-touch**
3. Añadir opciones de componentes a través de *mixin* globales, por ejemplo **vuex**
4. Añadir métodos de instancia de Vue agregándolos a `Vue.prototype`.
5. Una biblioteca que provea una API propia, a la vez que inyecta una combinación de las anteriores, por ejemplo **vue-router**

Un complemento de Vue.js debe exponer un método `install`. Este será ejecutado con el constructor de `Vue` como primer parámetro, junto con las opciones posibles:

JS

```
MyPlugin.install = function (Vue, options) {  
  // 1. agrega un método o propiedad global  
  Vue.myGlobalMethod = function () {  
    // algo de lógica  
  }  
  
  // 2. agrega un recurso global  
  Vue.directive('my-directive', {  
    bind (el, binding, vnode, oldVnode) {  
      // algo de lógica  
    }  
    ...  
  })  
  
  // 3. inyecta algunas opciones de componente  
  Vue.mixin({  
    created: function () {  
      // algo de lógica  
    }  
    ...  
  })  
}
```

```
// 4. agrega un método de instancia
Vue.prototype.$myMethod = function (options) {
  // algo de lógica
}
}
```

## Utilizando un complemento

---

Utiliza un plugin ejecutando el método global `Vue.use()` :

JS

```
// ejecuta `MyPlugin.install(Vue)`
Vue.use(MyPlugin)
```

Puedes pasar algunas opciones si lo necesitas:

JS

```
Vue.use(MyPlugin, { someOption: true })
```

`Vue.use` automáticamente previene que utilices un complemento más de una vez, por lo que ejecutarlo repetidamente con el mismo complemento como parámetro solo lo instalará una vez.

Algunos complementos oficiales provistos por Vue.js como `vue-router` ejecutan automáticamente `Vue.use()` si `Vue` está disponible como una variable global. Sin embargo, en un ambiente modular como CommonJS, necesitas explícitamente ejecutar `Vue.use()` :

JS

```
// Utilizando CommonJS a través de Browserify o Webpack
var Vue = require('vue')
var VueRouter = require('vue-router')

// ¡No te olvides de ejecutar esto!
Vue.use(VueRouter)
```

Échale un vistazo a [awesome-vue](#) para obtener un listado enorme de bibliotecas y complementos desarrollados por la comunidad.

# Componentes de un solo archivo

## Introducción

---

En muchos proyectos Vue, los componentes globales serán definidos utilizando `Vue.component`, seguido de `new Vue({ el: '#container' })` para especificar un elemento contenedor en el cuerpo de cada página.

Esto puede funcionar muy bien en proyectos pequeños a medianos, donde JavaScript se utiliza solo para mejorar ciertas vistas. Sin embargo, en proyectos más complejos o donde el lado cliente sea manejado totalmente con JavaScript, las siguientes desventajas se vuelven obvias:

- **Definiciones globales** fuerza a definir nombres únicos para cada componente
- **Plantillas como cadenas de texto** no se obtiene sintaxis resaltada y requiere utilizar horribles barras para HTML multilínea
- **Sin soporte CSS** significa que mientras el HTML y JavaScript son modularizados en componentes, el CSS es claramente dejado de lado
- **Sin proceso de construcción** nos restringe a utilizar HTML y ES5 JavaScript, en lugar de preprocesadores como Pug (anteriormente Jade) y Babel

Todo lo anterior es resuelto mediante **componentes de un solo archivo** con extensión `.vue`, gracias a herramientas de construcción como Webpack o Browserify.

Aquí hay un ejemplo sencillo de un archivo que llamaremos `Hello.vue`:



Ahora obtenemos:

- **Sintaxis resaltada**
- **Módulos CommonJS**
- **CSS en el ámbito del componente**

Como prometimos, también podemos usar preprocesadores como Pug, Babel (con módulos ES2015) y Stylus para obtener componentes más limpios y con mejores características.



Estos lenguajes específicos son solo ejemplos. Podrías utilizar igual de fácil Bublé, TypeScript, SCSS, PostCSS - o cualquier otro preprocesador que te ayude a ser productivo. Si utilizas Webpack



con `vue-loader` , también tienes soporte *first-class* para módulos CSS.

## # ¿Qué sucede con la separación de intereses?

Una cosa importante a tener en cuenta es que **separación de intereses no es equivalente a separación de tipos de archivo**. En el desarrollo UI moderno, hemos descubierto que en lugar de dividir el código base en tres capas enormes que se entrelazan entre ellas, tiene mucho más sentido dividir las en componentes poco acoplados y componerlas. Dentro de un componente, su plantilla, lógica y estilos están inherentemente acoplados, y agruparlos hace que el componente sea más cohesivo y fácil de mantener.

Incluso si no te gusta la idea de los componentes de un solo archivo, puedes tomar ventaja de sus características de recarga en caliente y precompilación separando tu JavaScript y CSS en distintos archivos:

```
<!-- my-component.vue -->
<template>
  <div>This will be pre-compiled</div>
</template>
<script src="./my-component.js"></script>
<style src="./my-component.css"></style>
```

HTML

## Primeros pasos

---

### Para usuarios sin experiencia con sistemas de empaquetamiento de módulos en # JavaScript

Con los componentes `.vue` , entramos en el ámbito de las aplicaciones avanzadas JavaScript. Esto significa aprender a usar algunas herramientas adicionales si todavía no lo has hecho:

- **Gestor de paquetes de Node (NPM por sus siglas en inglés):** Lee la [guía de primeros pasos](#) hasta la sección *10: Desinstalando paquetes globales*.
- **JavaScript moderno con ES2015/16:** Lee la guía de Babel [para aprender ES2015](#). No tienes que memorizar cada nueva característica ahora mismo, pero mantén esa página como una referencia a la que puedes consultar.

Luego de que te hayas tomado un día para profundizar esos recursos, te recomendamos dar un vistazo a la plantilla de proyecto [webpack-simple](#). Sigue las instrucciones y ¡deberías tener un proyecto Vue con componentes `.vue`, ES2015 y recarga en caliente casi inmediatamente!

Las plantillas de proyecto utilizan **Webpack**, un sistema de empaquetamiento de módulos que recibe una cantidad de módulos X y los empaqueta en tu aplicación final. Para aprender más acerca de Webpack, [este video](#) ofrece una buena introducción. Una vez que hayas entendido los conceptos básicos, puede que quieras mirar [este curso avanzado de Webpack en Egghead.io](#).

En Webpack, cada módulo puede ser transformado por un “cargador” antes de ser incluido en el paquete final, y Vue ofrece el complemento **vue-loader** que se encarga de traducir los componentes de un solo archivo `.vue`. La plantilla de proyecto [webpack-simple](#) ya tiene todo esto configurado para ti, pero si quieres aprender más acerca de como funcionan los componentes `.vue` en conjunto con Webpack, puedes leer [la documentación de vue-loader](#).

## # Para usuarios avanzados

Ya sea que prefieras Webpack o Browserify, hemos documentado las plantillas tanto para proyectos simples como complejos. Te recomendamos visitar [github.com/vuejs-templates](https://github.com/vuejs-templates), elegir la plantilla de proyecto que se ajuste a tu trabajo, y luego seguir las instrucciones en el README para generar un nuevo proyecto con **vue-cli**.

# Consejos para el despliegue a producción

## Habilita el modo producción

---

Durante el desarrollo, Vue provee gran cantidad de advertencias para ayudarte con errores comunes y dificultades. Sin embargo, estas advertencias se vuelven inútiles en producción e inflan el tamaño de tu aplicación final. Además, algunas de estas advertencias tienen un pequeño costo en terminos de rendimiento que pueden ser evitados en el modo producción.

### # Sin herramientas de empaquetado

Si estás utilizando la versión independiente, por ejemplo, incluyendo Vue directamente a través de una etiqueta `script` sin herramientas de empaquetado, asegúrate de utilizar la versión minificada ( `vue.min.js` ) en producción.

### # Con herramientas de empaquetado

Cuando utilices herramientas de empaquetado como Webpack o Browserify, el modo producción será determinado por `process.env.NODE_ENV` dentro del código fuente de Vue, y estará en modo desarrollo por defecto. Ambas herramientas proveen formas de sobrescribir esta variable para habilitar el modo producción de Vue, y las advertencias serán eliminadas por los minificadores durante el empaquetado. Todas las plantillas de proyectos de `vue-cli` tienen estas características ya configuradas, pero sería bueno que sepas como se hace:

#### Webpack

Usa **DefinePlugin** de Webpack para indicar un entorno de producción, para que los bloques de advertencias puedan ser automáticamente descartados por UglifyJS durante la minificación. Configuración de ejemplo:

```
var webpack = require('webpack')

module.exports = {
  // ...
```

JS

```

plugins: [
  // ...
  new webpack.DefinePlugin({
    'process.env': {
      NODE_ENV: '"production"'
    }
  })
]
}

```

## Browserify

- Ejecuta tu comando de empaquetamiento con la variable de entorno `NODE_ENV` con el valor `"production"`. Esto indica a `vueify` que evite incluir la recarga en caliente y el código relativo al desarrollo.
- Aplica una transformación global `envify` a tu paquete. Esto permite al minificador descartar todas las advertencias envueltas en bloques condicionales de la variable `env` dentro del código fuente de Vue. Por ejemplo:

```

Shell
NODE_ENV=production browserify -g envify -e main.js | uglifyjs -c -m > build.js

```

## Rollup

Utiliza `rollup-plugin-replace`:

```

JS
const replace = require('rollup-plugin-replace')

rollup({
  // ...
  plugins: [
    replace({
      'process.env.NODE_ENV': JSON.stringify( 'production' )
    })
  ]
}).then(...)

```

## Plantillas precompiladas

---

Cuando utilices plantillas dentro del DOM o plantillas como cadenas de texto en JavaScript, la compilación de *funciones de renderizado a plantillas* es ejecutada sobre la marcha. Esto normalmente es lo suficientemente rapido en la mayoría de los casos, pero es mejor evitarlo si tu aplicación es sensible al rendimiento.

La manera mas fácil de precompilar plantillas es utilizando **componentes de un solo archivo** - los procesos de empaquetamiento asociados automáticamente realizan la precompilacion por ti, por lo que el código resultante ya contiene las funciones de renderizado en lugar de las plantillas como cadenas de texto.

Si estas utilizando Webpack, y prefieres separar JavaScript y archivos de plantillas, puedes utilizar **vue-template-loader**, el cual también transforma los archivos de plantilla en funciones JavaScript de renderizado durante el proceso de empaquetado.

## Extrayendo CSS de los componentes

---

Cuando utilices componentes de un solo archivo, el CSS dentro de los mismos es inyectado dinámicamente como etiquetas `<style>` a través de JavaScript. Esto tiene un pequeño costo en términos de rendimiento, y si estás utilizando renderizado del lado servidor causará un “flash de contenido sin estilizar”. Extrae el CSS de todos los componentes en un mismo archivo y evita estos problemas, además resulta en una mejor minificación y cacheo del CSS.

Verifica la documentación de las herramientas correspondientes para ver como se realiza:

- **Webpack + vue-loader** (la plantilla de proyecto webpack de `vue-cli` lo incluye preconfigurado)
- **Browserify + vueify**
- **Rollup + rollup-plugin-vue**

## Registrando errores en tiempo de ejecución

---

Si un error en tiempo de ejecución ocurre durante el renderizado de un componente, será pasado a la función de configuración global `Vue.config.errorHandler` si ha sido agregada. Puede ser una buena idea apoyarse en esta herramienta en conjunto con algún servicio de registro de errores como **Sentry**, el cual provee **integración oficial** para Vue.



# Enrutamiento

## Enrutador oficial

---

Para la mayoría de las aplicaciones de una sola página, es recomendable utilizar la biblioteca con soporte oficial **vue-router library**. Para más detalles, lee la **documentación** de vue-router.

## Ruteo sencillo desde cero

---

Si solo necesitas enrutamiento simple y no deseas utilizar una biblioteca de enrutamiento completa, puedes hacerlo renderizando dinámicamente un componente de página de la siguiente manera:

JS

```
const NotFound = { template: '<p>Page not found</p>' }
const Home = { template: '<p>home page</p>' }
const About = { template: '<p>about page</p>' }

const routes = {
  '/': Home,
  '/about': About
}

new Vue({
  el: '#app',
  data: {
    currentRoute: window.location.pathname
  },
  computed: {
    ViewComponent () {
      return routes[this.currentRoute] || NotFound
    }
  },
  render (h) { return h(this.ViewComponent) }
})
```

En combinación con la API de historial de HTML5, puedes construir un enrutador de lado cliente muy básico completamente funcional. Para ver esto en práctica, revisa esta **aplicación de ejemplo**.

## Integrando enrutadores de terceros

---

Si hay algún enrutador de terceros que prefieras utilizar, como **Page.js** o **Director**, la integración es **igual de sencilla**. Aquí hay un **ejemplo completo** utilizando Page.js.



# Manejo de estado

## Implementación oficial estilo Flux

---

Las grandes aplicaciones pueden tornarse complejas, debido a múltiples porciones de estado dispersadas a través de muchos componentes y las interacciones entre ellos. Para resolver este problema, Vue ofrece **vuex**: nuestra biblioteca de manejo de estado propia inspirada en Elm. Incluso se integra con **vue-devtools**, dando un acceso con configuración cero a los viajes en el tiempo.

### # Información para desarrolladores de React

Si vienes de React, puede estar preguntandote como vuex se compara con **redux**, la implementación Flux más popular en ese ecosistema. En realidad, Redux es independiente de la capa “Vista”, por lo que puede ser utilizada con Vue a través de **algunos enlaces sencillos**. Vuex es diferente en el sentido que *sabe* que se encuentra en una aplicación Vue. Esto permite una mejor integración, ofreciendo una API más intuitiva y una experiencia de desarrollo mejorada.

## Manejo de estado sencillo desde cero

---

Es normal pasar por alto que la fuente de verdad en las aplicaciones Vue es el objeto **data** - una instancia de Vue simplemente delega el acceso a él. Por lo tanto, si tienes una porción de estado que debería ser compartido por varias instancias, puedes compartirla por identidad:

```
const sourceOfTruth = {}

const vmA = new Vue({
  data: sourceOfTruth
})

const vmB = new Vue({
  data: sourceOfTruth
})
```

JS

Ahora, cuando `sourceOfTruth` sea modificada, tanto `vmA` como `vmB` actualizarán sus vistas automáticamente. Los subcomponentes dentro de cada una de estas instancias también tendrán acceso a través de `this.$root.$data`. Ahora tenemos una sola fuente de verdad, pero depurar sería una pesadilla. Cualquier porción de los datos puede ser cambiada en cualquier lugar de nuestra aplicación en cualquier momento, sin dejar rastro.

Para ayudar a resolver este problema, podemos adoptar un sencillo **patrón store**:

JS

```
var store = {
  debug: true,
  state: {
    message: 'Hello!'
  },
  setMessageAction (newValue) {
    this.debug && console.log('setMessageAction triggered with', newValue)
    this.state.message = newValue
  },
  clearMessageAction () {
    this.debug && console.log('clearMessageAction triggered')
    this.state.message = ''
  }
}
```

Nota que todas las acciones que modifican el estado del *store* se encuentran dentro del mismo. Este tipo de manejo de estado centralizado hace más simple entender que tipo de mutaciones pueden ocurrir y como son disparadas. Ahora, cuando algo salga mal, también tendremos un registro de lo sucedido indicándonos donde está el error.

Además, cada instancia/componente puede manejar todavía su propio estado privado:

JS

```
var vmA = new Vue({
  data: {
    privateState: {},
    sharedState: store.state
  }
})

var vmB = new Vue({
  data: {
    privateState: {},
    sharedState: store.state
  }
})
```



Es importante notar que nunca deberías reemplazar el objeto de estado original in tus acciones - los componentes y el *store* necesitan compartir referencias al mismo objeto para que las mutaciones sean observadas.

A medid que continuamos desarrollando la convención donde los componentes no tienen permitido modificar directamente el estado que pertenece a un *store*, sino que deberían emitir eventos que notifiquen al *store* para que realice acciones, eventualmente llegamos a la arquitectura **Flux**. El beneficio de esta convención es que podemos registrar todas las mutaciones de estado que ocurren en el *store* e implementar características de depuración avanzada como registros de modificaciones, instantáneas, y recorrer hacia atrás/adelante el historial.

Esto nos pone nuevamente frente a **vuex**, por lo que si ya has leído hasta aquí, ¡probablemente es tiempo de probarlo!

# Pruebas unitarias

## Configuración y herramientas

---

Cualquier cosa compatible con un sistema de empaquetamiento basado en módulos funcionará, pero si buscas recomendaciones específicas, prueba el ejecutor de pruebas **Karma**. Tiene muchos complementos desarrollados por la comunidad, incluyendo soporte para **Webpack** y **Browserify**. Para una configuración detallada, por favor lee la documentación respectiva de cada proyecto, aunque estas configuraciones de ejemplo de Karma para **Webpack** y **Browserify** deberían ayudarte a empezar.

## Verificaciones simples

---

En términos de estructura de código para las pruebas, no tienes que hacer nada especial en tus componentes para poder probarlos. Simplemente exporta las opciones:

HTML

```
<template>
  <span>{{ message }}</span>
</template>

<script>
  export default {
    data () {
      return {
        message: 'hello!'
      }
    },
    created () {
      this.message = 'bye!'
    }
  }
</script>
```

Cuando pruebes ese componente, todo lo que tendrás que hacer es importar el objeto junto con Vue para intentar algunas verificaciones comunes:

JS

```
// Importa Vue y el componente bajo pruebas
```

```

import Vue from 'vue'
import MyComponent from 'path/to/MyComponent.vue'

// Aquí hay algunas pruebas para Jasmine 2.0, aunque puedes
// utilizar cualquier biblioteca que prefieras
describe('MyComponent', () => {
  // Inspecciona las opciones base del componente
  it('has a created hook', () => {
    expect(typeof MyComponent.created).toBe('function')
  })

  // Evalúa los resultados de las funciones
  // en las opciones base del componente
  it('sets the correct default data', () => {
    expect(typeof MyComponent.data).toBe('function')
    const defaultData = MyComponent.data()
    expect(defaultData.message).toBe('hello!')
  })

  // Inspecciona la instancia del componente al montarla
  it('correctly sets the message when created', () => {
    const vm = new Vue(MyComponent).$mount()
    expect(vm.message).toBe('bye!')
  })

  // Monta la instancia e inspecciona el resultado del renderizado
  it('renders the correct message', () => {
    const Ctor = Vue.extend(MyComponent)
    const vm = new Ctor().$mount()
    expect(vm.$el.textContent).toBe('bye!')
  })
})

```

## Escribiendo componentes testeables

---

Muchas de las salidas del renderizado de componentes están determinadas por las propiedades que recibe. De hecho, si el resultado del renderizado de un componente depende únicamente de sus propiedades, es bastante sencillo testearlo, parecido a verificar el valor de retorno de una función pura con diferentes parámetros. Un ejemplo inventado:

HTML

```

<template>
  <p>{{ msg }}</p>
</template>

<script>
  export default {
    props: ['msg']
  }

```

```
}  
</script>
```

Puedes verificar el resultado del renderizado con diferentes propiedades utilizando la opción

`propsData` :

JS

```
import Vue from 'vue'  
import MyComponent from './MyComponent.vue'  
  
// función auxiliar que monta y retorna el texto renderizado  
function getRenderedText (Component, propsData) {  
  const Ctor = Vue.extend(Component)  
  const vm = new Ctor({ propsData }).$mount()  
  return vm.$el.textContent  
}  
  
describe('MyComponent', () => {  
  it('renders correctly with different props', () => {  
    expect(getRenderedText(MyComponent, {  
      msg: 'Hello'  
    })).toBe('Hello')  
  
    expect(getRenderedText(MyComponent, {  
      msg: 'Bye'  
    })).toBe('Bye')  
  })  
})
```

## Verificando actualizaciones asíncronas

---

Dado que Vue realiza **actualizaciones del DOM asíncronicamente**, las verificaciones sobre actualizaciones del DOM resultado de un cambio de estado tendrán que ser hechas en la *callback*

`Vue.nextTick` :

JS

```
// Inspecciona el HTML generado luego de una actualización de estado  
it('updates the rendered message when vm.message updates', done => {  
  const vm = new Vue(MyComponent).$mount()  
  vm.message = 'foo'  
  
  // espera un "tick" luego del cambio del estado y antes de verificar actualizaci  
  Vue.nextTick(() => {  
    expect(vm.$el.textContent).toBe('foo')  
    done()  
  })  
})
```

})

Tenemos en vista trabajar en una colección de funciones auxiliares de pruebas comunes que hacen más simple renderizar componentes con diferentes restricciones (por ejemplo, renderizado superficial ignorando los componentes hijo) y verificar sus resultados.

# Renderizado del lado servidor

## ¿Necesitas SSR?

---

Antes de introducirnos en el SSR, veamos que es lo que realmente hace por ti y cuando puede ser que lo necesites.

### # SEO

Google y Bing pueden indexar correctamente aplicaciones JavaScript síncronas. *Síncronas* siendo la palabra clave. Si tu aplicación comienza con un indicador de carga y luego obtiene información utilizando AJAX, el *crawler* no esperará a que termines.

Esto significa que si tienes contenido recuperado asincrónicamente en páginas donde el SEO es importante, SSR puede ser necesario.

### # Clientes con una conexión lenta a Internet

Puede que los usuarios naveguen a tu sitio desde un área remota con conexiones lentas a Internet - o simplemente con una mala conexión desde el celular. En estos casos, querrás minimizar el número y tamaño de las peticiones necesarias para que los usuarios puedan ver el contenido básico.

Puedes utilizar **el divisor de código de Webpack** para evitar forzar a los usuarios a descargar tu aplicación entera para ver una única página, pero aún no será igual de rápido que bajar un solo archivo HTML pre-renderizado.

### # Clientes con un motor JavaScript antiguo (o sin soporte JavaScript)

En algunos lugares del mundo, utilizar una computadora del año 1998 para acceder a Internet puede ser la única opción. Mientras que Vue solo funciona con IE9 o superiores, tal vez quieras



devolver algo de contenido básico a estos usuarios con navegadores antiguos - o a los hackers *hipsters* que utilizan **Lynx** en una consola.

## # SSR vs Pre-renderizado

Si solo estás invetigando SSR para mejorar el SEO de un puñado de páginas de marketing/presentación (por ejemplo. `/` , `/about` , `/contact` , etc), entonces lo que probablemente desees sea **pre-renderizado**. En lugar de utilizar un servidor web para compilar HTML en tiempo real, el pre-renderizado simplemente genera archivos HTML estáticos para rutas específicas en el momento de compilación. La ventaja es que configurar el pre-renderizado es mucho más sencillo y te permite mantener tu *frontend* como un sitio completamente estático.

Si estás utilizando Webpack, puedes añadir pre-renderizado muy fácilmente con **prerender-spa-plugin**. Ha sido probado extensamente con aplicaciones Vue y, de hecho, el creador es un miembro del equipo de trabajo de Vue.

## Hola mundo

---

Si llegaste hasta aquí, estás listo para ver SSR en acción. Suena complejo, pero un *script* node con esta característica requiere solo de 3 pasos:

JS

```
// Paso 1: Crear una instancia de Vue
var Vue = require('vue')
var app = new Vue({
  render: function (h) {
    return h('p', 'hello world')
  }
})

// Paso 2: Crear un renderizador
var renderer = require('vue-server-renderer').createRenderer()

// Paso 3: Renderizar la instancia de Vue en el HTML
renderer.renderToString(app, function (error, html) {
  if (error) throw error
  console.log(html)
  // => <p server-rendered="true">hello world</p>
})
```

No es tan complicado, ¿verdad?. Por supuesto, este ejemplo es mucho más simple que la mayoría de las aplicaciones. No tenemos que preocuparnos todavía por:

- Un servidor web
- Respuestas como Streaming
- Cacheo de componentes
- Un proceso de compilación/empaquetamiento
- Enrutamiento
- Regeneración del estado de Vuex

En el resto de esta guía, repasaremos como trabajar con alguna de estas características. Una vez que entiendas los conceptos básicos, te indicaremos donde leer más documentación y ejemplos avanzados para ayudarte a manejar casos extremos.

## SSR sencillo con el servidor web Express

---

Es un poco difícil llamarlo “renderizado del lado servidor” cuando en realidad no tenemos un servidor web, así que arreglemos eso. Vamos a construir una aplicación SSR muy sencilla utilizando solo ES5 sin ningún proceso de compilación ni complementos de Vue.

Empezaremos con una aplicación que le indica al usuario cuantos segundos ha estado en la página:

JS

```
new Vue({
  template: '<div>You have been here for {{ counter }} seconds.</div>',
  data: {
    counter: 0
  },
  created: function () {
    var vm = this
    setInterval(function () {
      vm.counter += 1
    }, 1000)
  }
})
```

Para adaptar esto para SSR, hay algunas pocas modificaciones que tendremos que hacer, para que funcione tanto en los navegadores como con node:

- Cuando estemos en un navegador, añadir una instancia de nuestra aplicación al contexto global (es decir, `window`), para que podamos montarla.

- Cuando estemos en node, exportar una función factoría para que podamos crear una instancia nueva de la aplicación para cada petición.

Lograr esto requiere un poco más de código:

JS

```
// assets/app.js
(function () { 'use strict'
  var createApp = function () {
    // -----
    // COMIENZA EL CÓDIGO DE LA APP
    // -----

    // La instancia principal debe ser devuelta y tener un
    // nodo raíz con el id "app", para que la versión del lado
    // cliente pueda tomar el control una vez que cargue.
    return new Vue({
      template: '<div id="app">You have been here for {{ counter }} seconds.</div>'
      data: {
        counter: 0
      },
      created: function () {
        var vm = this
        setInterval(function () {
          vm.counter += 1
        }, 1000)
      }
    })

    // -----
    // TERMINA EL CÓDIGO DE LA APP
    // -----
  }
  if (typeof module !== 'undefined' && module.exports) {
    module.exports = createApp
  } else {
    this.app = createApp()
  }
}).call(this)
```

Ahora que tenemos el código de nuestra aplicación, creemos un archivo `index.html` :

HTML

```
<!-- index.html -->
<!DOCTYPE html>
<html>
<head>
  <title>My Vue App</title>
  <script src="/assets/vue.js"></script>
</head>
<body>
```

```

<div id="app"></div>
<script src="/assets/app.js"></script>
<script>app.$mount('#app')</script>
</body>
</html>

```

Mientras que el directorio `assets` referenciado contenga el archivo `app.js` que creamos anteriormente junto con el archivo `vue.js` con Vue, ¡deberíamos tener una aplicación de una sola página funcionando!

Luego, para hacerla funcionar con el renderizado del lado servidor, hay un solo paso extra - el servidor web:

JS

```

// server.js
'use strict'

var fs = require('fs')
var path = require('path')

// Define global.Vue para app.js funcionando del lado servidor
global.Vue = require('vue')

// Obtén la estructura HTML
var layout = fs.readFileSync('./index.html', 'utf8')

// Crea un renderizador
var renderer = require('vue-server-renderer').createRenderer()

// Crea un servidor express
var express = require('express')
var server = express()

// Sirve los archivos desde el directorio de recursos
server.use('/assets', express.static(
  path.resolve(__dirname, 'assets')
))

// Maneja las peticiones GET
server.get('*', function (request, response) {
  // Renderiza nuestra app Vue a una cadena de texto
  renderer.renderToString(
    // Crea una instancia de app
    require('./assets/app')(),
    // Maneja el resultado del renderizado
    function (error, html) {
      // Si ocurre un error durante el renderizado
      if (error) {
        // Registra el error en la consola
        console.error(error)
        // Avisa al cliente que algo sucedio

```

```

        return response
            .status(500)
            .send('Server Error')
    }
    // Envía la estructura con el HTML renderizado de la aplicación
    response.send(layout.replace('<div id="app"></div>', html))
}
)
})

// Escucha en el puerto 5000
server.listen(5000, function (error) {
    if (error) throw error
    console.log('Server is running at localhost:5000')
})

```

¡Y eso es todo! Aquí está **la aplicación completa**, en caso que quieras clonarla y experimentar un poco más. Una vez que la ejecutes localmente, puedes confirmar que el renderizado del lado servidor está realmente funcionando haciendo clic derecho en la página y seleccionando

[View Page Source](#) (o similar). Deberías ver esto en el **body** :

HTML

```
<div id="app" server-rendered="true">You have been here for 0 seconds&period;</div>
```

en lugar de:

HTML

```
<div id="app"></div>
```

## Respuesta como Streaming

---

Vue también soporta renderizar a un **stream**, lo cual se prefiere para servidores web que soporten *streaming*. Esto permite que el HTML sea escrito a la respuesta *a medida que es generado*, en lugar de escribirlo todo junto al final. El resultado es que las respuestas son devueltas más rápido, ¡y no hay contras!

Para adaptar tu aplicación de la sección anterior y utilizar *streaming*, podemos simplemente reemplazar los bloques `server.get('*', ...)` con lo siguiente:

JS

```

// Divide la estructura en dos secciones de HTML
var layoutSections = layout.split('<div id="app"></div>')
var preAppHTML = layoutSections[0]

```

```

var postAppHTML = layoutSections[1]

// Maneja todas las peticiones GET
server.get('*', function (request, response) {
  // Renderiza nuestra aplicación Vue a un _stream_
  var stream = renderer.renderToStream(require('./assets/app'))()

  // Escribe el HTML preAppHtml en la respuesta
  response.write(preAppHTML)

  // Cada vez que nuevas porciones son renderizadas...
  stream.on('data', function (chunk) {
    // Escribe la porción en la respuesta
    response.write(chunk)
  })

  // Cuando todas las porciones son renderizadas...
  stream.on('end', function () {
    // Escribe el HTML postAppHTML en la respuesta
    response.end(postAppHTML)
  })

  // Si ocurre un error durante el renderizado...
  stream.on('error', function (error) {
    // Registra el error en la consola
    console.error(error)
    // Informa al cliente que algo sucedió
    return response
      .status(500)
      .send('Server Error')
  })
})

```

Como puedes ver, no es mucho más complicada que la versión anterior, incluso si los streams son conceptualmente nuevos para ti. Simplemente:

1. Inicializamos el stream
2. Escribimos en la respuesta el HTML necesario previo a la aplicación
3. Escribimos en la respuesta el HTML de la aplicación a medida que está disponible
4. Escribimos en la respuesta el HTML necesario posterior a la aplicación y luego la finalizamos
5. Manejamos cualquier error que haya surgido

## Cacheo de componentes

---

El SSR de Vue es muy rápido por defecto, pero puedes mejorar aún más el rendimiento cacheando componentes renderizados. Sin embargo, esto debe considerarse una característica

dado que cachear los componentes equivocados (o los componentes correctos con la llave equivocada) puede conducir a errores en el renderizado de tu aplicación. Específicamente:

! No debes cachear un componente que contiene componentes hijo que dependen de estado global (por ejemplo, desde un *store* de vuex). Si lo haces, esos componentes hijo (y, de hecho, todo el sub-árbol) será también cacheado. Se especialmente cuidadoso con componentes que aceptan slots/hijos.

## # Configuración

Con esa advertencia fuera del camino, así es como cacheas tus componentes.

Primero, necesitarás asignar a tu renderizador un **objeto cache**. Aquí hay un ejemplo sencillo utilizando **lru-cache**:

JS

```
var createRenderer = require('vue-server-renderer').createRenderer
var lru = require('lru-cache')

var renderer = createRenderer({
  cache: lru(1000)
})
```

Eso cacheará hasta 1000 renderizados únicos. Para otras configuraciones que se ajustan mejor al uso de memoria, verifica las **opciones de lru-cache**.

Luego, para componentes que desees cachear, debes asignarles:

- un **name** único
- una función **serverCacheKey**, devolviendo una llave única en el ámbito del componente

Por ejemplo:

JS

```
Vue.component({
  name: 'list-item',
  template: '<li>{{ item.name }}</li>',
  props: ['item'],
  serverCacheKey: function (props) {
    return props.item.type + '::' + props.item.id
  }
})
```

## # Componentes ideales para cacheo

Cualquier componente “puro” puede ser cacheado sin problemas - esto es, cualquier componente que puedas garantizar que renderizará el mismo HTML dadas las mismas propiedades. Ejemplos comunes de esto son:

- Componentes estáticos (i.e. siempre generan el mismo HTML, por lo que la función `serverCacheKey` puede directamente devolver `true` )
- Componentes elemento de una lista (cuando se trabaja con listas grandes, cachear estos puede mejorar el rendimiento significativamente)
- Componentes de UI genéricos (e.g. botones, alertas, etc - al menos aquellos que aceptan contenido a través de propiedades en lugar de slots/hijos)

## Proceso de compilación, ruteo y regeneración de estado de Vuex

---

En este punto, deberías entender los conceptos fundamentales detrás del renderizado del lado servidor. Sin embargo, a medida que agregas un proceso de compilación/empaquetamiento, ruteo y vuex, cada uno introduce sus propias consideraciones.

Para dominar realmente el renderizado del lado servidor en aplicaciones complejas, te recomendamos leer detenidamente los siguientes enlaces:

- **Documentación de vue-server-renderer**: mayor detalle en los temas tratados aquí, así como documentación de temas más avanzados, como **prevenir contaminación de peticiones cruzadas** y **añadir un build de servidor separado**
- **vue-hackernews-2.0**: el ejemplo definitivo de integración de todas las bibliotecas principales y conceptos de Vue en una sola aplicación.

## Nuxt.js

---

Configurar apropiadamente todos los aspectos discutidos de una aplicación renderizada del lado servidor preparada para producción puede ser una tarea intimidante. Por suerte, existe un excelente proyecto de la comunidad que apunta a hacer todo esto mucho más sencillo: **Nuxt.js**. Nuxt.js es un framework de nivel superior construido sobre el ecosistema de Vue, el cual provee una experiencia de desarrollo extremadamente eficiente para escribir aplicaciones \



universales. Aún mejor, puedes usarlo como un generador de sitios estático (con las páginas creadas como componentes de un solo archivo de Vue). Recomendamos fervientemente que lo pruebes.

# Soporte para TypeScript

## Archivos oficiales de declaración

---

Un sistema de tipos estáticos puede ayudar a prevenir varios errores potenciales en tiempo de ejecución, especialmente cuando la aplicación crece. Por esto es que Vue incluye **declaraciones oficiales de tipo** para **TypeScript** - no solo para Vue, sino también **para Vue Router** y **para Vuex**.

Dado que son publicadas en **NPM**, ni siquiera necesitas herramientas externas como **Typings**, ya que las declaraciones son importadas automáticamente con Vue. Esto significa que todo lo que necesitas es simplemente:

```
import Vue = require('vue')
```

Entonces se verificarán los tipos de todos los métodos, propiedades y parámetros. Por ejemplo, si por un error de tipeo escribes `template` como `tempate` (sin la `l`), el compilador de TypeScript imprimirá un mensaje de error en tiempo de compilación. Si estás utilizando un editor que pueda *lintear* TypeScript, como **Visual Studio Code**, será posible encontrar estos errores incluso antes de compilar:

 TypeScript Type Error in Visual Studio Code

## # Opciones de compilación

Los archivos de declaración de Vue requieren **la opción de compilador**

`--lib DOM,ES5,ES2015.Promise`. Puedes pasar esta opción al comando `tsc` o agregar el equivalente en un archivo `tsconfig.json`.

## # Accediendo a las declaraciones de tipo de Vue

Si quieres *anotar* a tu propio código con los tipos de Vue, puedes acceder a ellos en el objeto exportado de Vue. Por ejemplo, para *anotar* un objeto de opciones de componente exportado (en un archivo `.vue`):

```
import Vue = require('vue')

export default {
  props: ['message'],
  template: '<span>{{ message }}</span>'
} as Vue.ComponentOptions<Vue>
```

## Componentes de Vue como clases JavaScript

---

Las opciones de componentes de Vue pueden ser *anotadas* fácilmente con tipos:

```
import Vue = require('vue')

// Declara el tipo del componente
interface MyComponent extends Vue {
  message: string
  onClick (): void
}

export default {
  template: '<button @click="onClick">Click!</button>',
  data: function () {
    return {
      message: 'Hello!'
    }
  },
  methods: {
    onClick: function () {
      // TypeScript sabe que `this` es de tipo MyComponent
      // y que `this.message` será una cadena de texto
      window.alert(this.message)
    }
  }
}

// Necesitamos _anotar_ explícitamente el objeto de
// opciones exportado con el tipo MyComponent
} as Vue.ComponentOptions<MyComponent>
```

Desafortunadamente, hay algunas limitaciones aquí:

- **TypeScript no puede inferir todos los tipos de la API de Vue.** Por ejemplo, no sabe que la propiedad `message` devuelta en nuestra función `data` será agregada a la instancia de `MyComponent`. Eso significa que si asignamos un valor numérico o booleano a `message`, los

*linters* y compiladores no serían capaces de lanzar un error, indicando que debería ser una cadena de texto.

- Debido a la limitación anterior, **anotando tipos de esta manera puede tornarse tedioso y largo**. La única razón por la que tenemos que declarar manualmente `message` como una cadena de texto es porque TypeScript no puede inferir el tipo en este caso.

Afortunadamente, **vue-class-component** puede resolver ambos problemas. Es una biblioteca oficial que te permite declarar componentes como clases nativas de JavaScript, con el decorador `@Component`. Como ejemplo, reescribamos el componente anterior:

```
import Vue = require('vue')
import Component from 'vue-class-component'

// El decorador @Component indica que la clase es un componente de Vue
@Component({
  // Todas las opciones del componente están permitidas aquí
  template: '<button @click="onClick">Click!</button>'
})
export default class MyComponent extends Vue {
  // Los datos iniciales pueden ser declarados como propiedades de instancia
  message: string = 'Hello!'

  // Los métodos del componente pueden ser declarados por métodos de la instancia
  onClick (): void {
    window.alert(this.message)
  }
}
```

Con esta sintaxis alternativa, nuestra definición de componente no solo es más corta, sino que TypeScript puede también inferir los tipos de `message` y `onClick` sin declaraciones de interface explícitas. Esta estrategia incluso te permite manejar los tipos de las propiedades computadas, *hooks* de ciclo de vida y funciones de renderizado. Para los detalles completos de uso, lee la **documentación de vue-class-component**.

# API

---

# Configuración global

---

`Vue.config` es un objeto que contiene las configuraciones globales de Vue. Puedes modificar las propiedades listadas debajo antes de iniciar tu aplicación:

## # silent

- Tipo: `boolean`
- Valor por defecto: `false`
- Uso:

JS

```
Vue.config.silent = true
```

Elimina todos los registros y advertencias de Vue.

- [Source](#)

## # optionMergeStrategies

- Tipo: `{ [key: string]: Function }`
- Valor por defecto: `{}`
- Uso:

JS

```
Vue.config.optionMergeStrategies._my_option = function (parent, child, vm) {  
  return child + 1  
}
```

```
const Profile = Vue.extend({
  _my_option: 1
})

// Profile.options._my_option = 2
```

Define estrategias de fusionado personalizadas para las opciones.

La estrategia de fusión recibe el valor de la opción definida en las instancias padre e hijo como primer y segundo argumento respectivamente. La instancia Vue de contexto es pasada como tercer argumento.

- Lee también: **Estrategias personalizadas de fusionado de opciones**
- [Source](#)

## # devtools

- Tipo: `boolean`
- Valor por defecto: `true` ( `false` en versiones de producción)
- Uso:

```
JS
// asegúrate de establecer sincronicamente lo siguiente inmediatamente después d
Vue.config.devtools = true
```

Permite o no la inspección de **vue-devtools**. EL valor por defecto de esta opción es `true` en las versiones de desarrollo y `false` en versiones de producción. Puedes establecer este valor en `true` para habilitar la inspección en versiones de producción.

- [Source](#)

## # errorHandler

- **Tipo:** `Function`
- **Valor por defecto:** El error es lanzado en el lugar
- **Uso:**

JS

```
Vue.config.errorHandler = function (err, vm) {
  // maneja el error
}
```

Asigna una función controladora para los errores no capturados durante renderización de componentes y observadores. La función controladora es llamada con el error y la instancia de Vue.

**Sentry**, un servicio de seguimiento de errores, provee **integración oficial** utilizando esta opción.

- **Source**

## # ignoredElements

- **Tipo:** `Array<string>`
- **Valor por defecto:** `[]`
- **Uso:**

JS

```
Vue.config.ignoredElements = [
  'my-custom-web-component', 'another-web-component'
]
```

Hace que Vue ignore los elementos personalizados definidos fuera de si mismo (por ejemplo, utilizando la API de componentes Web). De otra forma, lanzará una advertencia



Unknown custom element , asumiendo que has olvidado registrar un componente global o escrito mal un nombre de componente.

- [Source](#)

## # keyCode

- Tipo: `{ [key: string]: number | Array<number> }`
- Valor por defecto: `{}`
- Uso:

JS

```
Vue.config.keyCodes = {  
  v: 86,  
  f1: 112,  
  mediaPlayPause: 179,  
  up: [38, 87]  
}
```

Define alias personalizados de teclas para v-on.

- [Source](#)

## API Global

---

### # Vue.extend( options )

- Argumentos:
  - `{Object} options`
- Uso:

Crea una “subclase” del constructor base de Vue. El argumento debe ser un objeto que contenga las opciones de componente.

El caso especial a tener en cuenta aquí es la opción `data` - debe ser una función cuando se utiliza con `Vue.extend()` .

HTML

```
<div id="mount-point"></div>
```

JS

```
// crea el constructor
var Profile = Vue.extend({
  template: '<p>{{firstName}} {{lastName}} aka {{alias}}</p>',
  data: function () {
    return {
      firstName: 'Walter',
      lastName: 'White',
      alias: 'Heisenberg'
    }
  }
})
// crea una instancia de 'Profile' y la monta en un elemento
new Profile().$mount('#mount-point')
```

Se obtiene como resultado:

HTML

```
<p>Walter White aka Heisenberg</p>
```

- Lee también: [Componentes](#)
- [Source](#)

## # Vue.nextTick( [callback, context] )

- Argumentos:
  - `{Function}` [callback]
  - `{Object}` [context]

- **Uso:**

Posterga la llamada a la función *callback* hasta el próximo ciclo de actualización del DOM. Utilízala inmediatamente luego de que hayas cambiado datos para esperar a la siguiente actualización del DOM.

JS

```
// modifica los datos
vm.msg = 'Hello'
// el DOM no fue actualizado todavía
Vue.nextTick(function () {
  // DOM actualizado
})
```

**Nuevo en 2.1.0:** devuelve una *Promise* si no se provee una función *callback* y el ambiente de ejecución las soporta.

- Lee también: **Coloa de actualización asíncrona**
- **Source**

## # Vue.set( object, key, value )

- **Argumentos:**

- {Object} object
- {string} key
- {any} value

- **Devuelve:** el valor establecido.

- **Uso:**

Establece una propiedad en un objeto. Si el objeto es reactivo, asegura que la propiedad es creada como una propiedad reactiva y dispara actualizaciones de la vista. Es utilizado principalmente para solucionar la limitación de Vue de no poder detectar el agregado de propiedades.

Nota que el objeto no puede ser una instancia de Vue, ni el objeto de datos raíz de una instancia de Vue.

- Lee también: [Reactividad en profundidad](#)
- [Source](#)

## # Vue.delete( object, key )

- Argumentos:
  - `{Object} object`
  - `{string} key`

- Uso:

Borra una propiedad de un objeto. Si el objeto es reactivo, asegura el borrado de los disparadores de actualizaciones de vistas. Es utilizado principalmente para solucionar la limitación de Vue de no poder detectar el quitado de propiedades, pero no debería ser normal que necesites utilizarlo.

Nota que el objeto no puede ser una instancia de Vue, ni el objeto de datos raíz de una instancia de Vue.

- Lee también: [Reactividad en profundidad](#)
- [Source](#)

## # Vue.directive( id, [definition] )

- Argumentos:
  - `{string} id`
  - `{Function | Object} [definition]`

- **Uso:**

Registra o recupera una directiva global.

JS

```
// registra
Vue.directive('my-directive', {
  bind: function () {},
  inserted: function () {},
  update: function () {},
  componentUpdated: function () {},
  unbind: function () {}
})

// registra (función directiva sencilla)
Vue.directive('my-directive', function () {
  // esto será ejecutado como `bind` y `update`
})

// 'getter', devuelve la definición de la directiva si ha sido registrada
var myDirective = Vue.directive('my-directive')
```

- **Lee también: Directivas personalizadas**
- **Source**

## # Vue.filter( id, [definition] )

- **Argumentos:**
  - {string} id
  - {Function} [definition]
- **Uso:**

Registra o recupera un filtro global.

JS

```
// registra
Vue.filter('my-filter', function (value) {
  // devuelve el valor procesado
})
```

```
// 'getter', devuelve el filtro si ha sido registrado  
var myFilter = Vue.filter('my-filter')
```

- [Source](#)

## # Vue.component( id, [definition] )

- **Argumentos:**

- `{string} id`
- `{Function | Object} [definition]`

- **Uso:**

Registra o recupera un componente global. El registro establece automáticamente la propiedad `name` del componente como el `id` dado.

JS

```
// registra un constructor extendido  
Vue.component('my-component', Vue.extend({ /* ... */ }))  
  
// registra un objeto de opciones (automáticamente llama a Vue.extend)  
Vue.component('my-component', { /* ... */ })  
  
// recupera un componente registrado (siempre devuelve un constructor)  
var MyComponent = Vue.component('my-component')
```

- **Lee también:** [Componentes](#)

- [Source](#)

## # Vue.use( plugin )

- **Argumentos:**

- `{Object | Function} plugin`

- **Uso:**

Instala un complemento de Vue.js. Si el complemento es un objeto, debe exponer un método `install` . Si es una función, será tratada como el método de instalación. Este método será llamado con Vue como argumento.

Cuando este método es ejecutado con el mismo complemento varias veces, el complemento será instalado solo una vez.

- **Lee también: Complementos**
- **Source**

## # Vue.mixin( mixin )

- **Argumentos:**

- `{Object} mixin`

- **Uso:**

Aplica un *mixin* globalmente, el cual afecta a cada instancia de Vue creada luego de la llamada. Puede ser utilizado por creadores de complementos para inyectar comportamiento personalizado dentro de los componentes.

**No se recomienda en el código de la aplicación.**

- **Lee también: Mixins globales**
- **Source**

## # Vue.compile( template )

- **Argumentos:**

- `{string} template`

- **Uso:**

Compila una plantilla de texto en una función de renderizado. **Solo disponible en la versión independiente.**

JS

```
var res = Vue.compile('<div><span>{{ msg }}</span></div>')

new Vue({
  data: {
    msg: 'hello'
  },
  render: res.render,
  staticRenderFns: res.staticRenderFns
})
```

- **Lee también:** [Funciones de renderizado](#)
- [Source](#)

## Opciones / Datos

---

### # data

- **Tipo:** `Object` | `Function`
- **Restricciones:** Solo acepta `Function` cuando se utiliza en la definición de componentes.
- **Detalles:**

El objeto de datos para la instancia de Vue. Vue convertirá recursivamente sus propiedades en *getters/setters* para hacerlos “reactivos”. **Debe ser un objeto plano:** los objetos nativos tales como los objetos de API de los navegadores y propiedades de prototipo son ignorados. Una regla sencilla es que los datos deben ser solo datos - no es recomendable observar objetos con comportamiento propio.



Una vez observado, no puedes añadir más propiedades reactivas al objeto de datos raíz. Por lo tanto, es recomendable declarar todas las propiedades reactivas en el nivel raíz por adelantado, antes de crear la instancia.

Luego que la instancia ha sido creada, el objeto original de dato puede ser accedido a través de `vm.$data` . La instancia de Vue también espeja todas las propiedades encontradas en el objeto de datos, por lo que `vm.a` será equivalente a `vm.$data.a` .

Las propiedades que comienzan con `_` o `$` **no** serán espejadas en la instancia de Vue porque pueden entrar en conflicto con las propiedades internas y métodos de la API de Vue. Tendrás que acceder a ellos como `vm.$data._property` .

Cuando se define un **componente**, `data` debe ser declarado como una función que devuelve el objeto de datos inicial, porque habrá varias instancias creadas utilizando la misma definición. Si utilizamos un objeto plano para `data` , ese mismo objeto será **compartido por referencia** en todas las instancias creadas! Creando una función `data` , cada vez que una nueva instancia es creada, podemos simplemente ejecutarla para devolver una nueva copia de los datos originales.

Si se requiere, se puede obtener una copia profunda del objeto original pasando `vm.$data` a `JSON.parse(JSON.stringify(...))` .

- **Ejemplo:**

JS

```
var data = { a: 1 }

// creación directa de instancia
var vm = new Vue({
  data: data
})
vm.a // -> 1
vm.$data === data // -> true

// debe utilizarse una función cuando se trabaja con Vue.extend()
var Component = Vue.extend({
  data: function () {
    return { a: 1 }
  }
})
```

! Nota que **no debes utilizar una función flecha con la propiedad `data`** (por ejemplo: `data: () => { return { a: this.myProp }}` ). La razón es que las funciones flecha enlazan el contexto padre, por lo que `this` no será la instancia de Vue como esperarías y `this.myProp` tendrá un valor `undefined` .

- Lee también: **Reactividad en profundidad**
- **Source**

## # props

- **Tipo:** `Array<string> | Object`
- **Detalles:**

Una lista/hash de atributos expuestos para aceptar datos desde el componente padre. Tiene una sintaxis sencilla basada en vectores y otra alternativa basada en objetos que permite configuraciones avanzadas como verificación de tipos, validación personalizada y valores por defecto.

- **Ejemplo:**

JS

```
// sintaxis simple
Vue.component('props-demo-simple', {
  props: ['size', 'myMessage']
})

// sintaxis de objeto con validación
Vue.component('props-demo-advanced', {
  props: {
    // solo verificación de tipo
    height: Number,
    // verificación de tipo junto con otras validaciones
    age: {
      type: Number,
      default: 0,
      required: true,
      validator: function (value) {
        return value >= 0
      }
    }
  }
})
```

```
}  
})
```

- Lee también: [Propiedades](#)
- [Source](#)

## # propsData

- **Tipo:** `{ [key: string]: any }`
- **Restricciones:** solo respetadaa en la creación de una instancia utilizando `new` .
- **Detalles:**

Pasa propiedades a una instancia durante su creación. Está pensado principalmente para facilitar las pruebas unitarias.

- **Ejemplo:**

```
var Comp = Vue.extend({  
  props: ['msg'],  
  template: '<div>{{ msg }}</div>'  
})  
  
var vm = new Comp({  
  propsData: {  
    msg: 'hello'  
  }  
})
```

JS

- [Source](#)

## # computed

- **Tipo:** `{ [key: string]: Function | { get: Function, set: Function } }`

- **Detalles:**

Las propiedades computadas a ser fusionadas en la instancia de Vue. Todos los *getters* y *setters* tienen su contexto `this` automáticamente enlazado con la instancia de Vue.

Nota que **no deberías utilizar una función flecha para definir una propiedad computada** (por ejemplo: `aDouble: () => this.a * 2`). La razón es que las funciones flecha enlazan el contexto padre, por lo que `this` no será la instancia de Vue como esperarías y `this.a` tendrá un valor `undefined`.

Las propiedades computadas son guardadas en memoria caché, y solo son re-evaluadas cuando las dependencias cambian. Nota que si cierta dependencia está fuera del ámbito de la instancia (i.e. no es reactiva), la propiedad computada **no** será actualizada.

- **Ejemplo:**

JS

```
var vm = new Vue({
  data: { a: 1 },
  computed: {
    // si solo necesitas obtener datos, se pasa una única función
    aDouble: function () {
      return this.a * 2
    },
    // tanto obtener como establecer datos
    aPlus: {
      get: function () {
        return this.a + 1
      },
      set: function (v) {
        this.a = v - 1
      }
    }
  }
})
vm.aPlus    // -> 2
vm.aPlus = 3
vm.a        // -> 2
vm.aDouble  // -> 4
```

- **Lee también:**

- **Propiedades computadas**

- [Source](#)

## # methods

- **Tipo:** `{ [key: string]: Function }`
- **Detalles:**

Los métodos a ser fusionados en la instancia de Vue. Puedes acceder a estos métodos directamente en la instancia de la VM, o usarlos en expresiones de directivas. Todos los métodos tendrán su contexto `this` enlazado con la instancia de Vue.

! Nota que **no deberías utilizar una función flecha para definir un método** (por ejemplo: `plus: () => this.a++`). La razón es que las funciones flecha enlazan el contexto padre, por lo que `this` no será la instancia de Vue como esperarías y `this.a` tendrá un valor `undefined`.

- **Ejemplo:**

```
var vm = new Vue({
  data: { a: 1 },
  methods: {
    plus: function () {
      this.a++
    }
  }
})
vm.plus()
vm.a // 2
```

JS

- [Lee también: Métodos y Manejo de eventos](#)
- [Source](#)

## # watch

- **Tipo:** `{ [key: string]: string | Function | Object }`

- **Detalles:**

Un objeto donde las llaves son expresiones a observar y los valores son sus correspondientes funciones *callback*. El valor puede ser también una cadena de texto equivalente al nombre de un método, o un objeto que contenga opciones adicionales. La instancia de Vue ejecutará `$watch()` por cada entrada del objeto al momento de la creación de la instancia.

- **Ejemplo:**

JS

```
var vm = new Vue({
  data: {
    a: 1,
    b: 2,
    c: 3
  },
  watch: {
    a: function (val, oldVal) {
      console.log('new: %s, old: %s', val, oldVal)
    },
    // cadena de texto con el nombre del método
    b: 'someMethod',
    // deep watcher
    c: {
      handler: function (val, oldVal) { /* ... */ },
      deep: true
    }
  }
})
vm.a = 2 // -> nuevo: 2, viejo: 1
```

!

Nota que **no deberías utilizar una función flecha para definir un observador** (por ejemplo: `searchQuery: newValue => this.updateAutocomplete(newValue)`). La razón es que las funciones flecha enlazan el contexto padre, por lo que `this` no será la instancia de Vue como esperarías y `this.updateAutocomplete` tendrá un valor `undefined`.

- **Source**

- **Lee también: Métodos de instancia - `vm.$watch`**

# Opciones / DOM

---

## # el

- **Tipo:** `string` | `HTMLElement`
- **Restricciones:** solo respetado en la creación de instancias a través de `new` .
- **Detalles:**

Provee a la instancia de Vue un elemento del DOM existente sobre el que montarse. Puede ser un selector CSS o un elemento `HTMLElement`.

Luego que la instancia haya sido montada, el elemento resuelto será accesible como `vm.$el` .

Si esta opción está disponible al momento de la instanciación, la instancia entrará inmediatamente en la compilación; en otro caso, el usuario tendrá que llamar explícitamente a `vm.$mount()` para iniciar manualmente la compilación.



El elemento proviste sirve únicamente como punto de montaje. A diferencia de Vue 1.x, el elemento montado será reemplazado con DOM generado por Vue en todos los casos. Por lo tanto no es recomendable montar la instancia principal en `<html>` o `<body>` .

- **Lee también:** [Diagrama del ciclo de vida](#)
- **Source**

## # template

- **Tipo:** `string`
- **Detalles:**

Plantilla de texto a ser usada como estructura para la instancia de Vue. La plantilla **reemplazará** al elemento montado. Cualquier estructura existente dentro del elemento montado será ignorada, a menos que se encuentren *slots* de distribución de contenido en la plantilla.

Si la cadena de texto comienza con `#` será utilizada como *querySelector* y se usará el contenido interno HTML del elemento seleccionado como plantilla de texto. Esto permite la utilización del conocido truco `<script type="x-template">` para incluir plantillas.



Desde una perspectiva de seguridad, solo deberías utilizar plantillas de Vue en las que confíes. Nunca utilices contenido generado por el usuario como tu plantilla.

- **Lee también:**
  - **Diagrama de ciclo de vida**
  - **Distribución de contenido**
- **Source**

## # render

- **Tipo:** `Function`
- **Detalles:**

Una alternativa a las plantillas de texto que te permite apoyarte en el poder de JavaScript. La función de renderizado recibe un método `createElement` como su primer argumento el cual se utiliza para crear un `VNode` .

Si el componente es uno funcional, la función de renderizado también recibe el argumento extra `context` , el cual provee acceso a los datos contextuales dado que los componentes funcionales no son instanciables.

- **Lee también:**
  - **Funciones de renderizado**
- **Source**



# Opciones / Hooks de ciclo de vida

---

Todos los *hooks* de ciclo de vida tienen su contexto `this` enlazado a la instancia, para que puedas acceder a los datos, propiedades computadas y métodos. Esto significa que **no deberías utilizar una función flecha para definir un método del ciclo de vida** (por ejemplo: `created: () => this.fetchTodos()`). La razón es que las funciones flecha enlazan el contexto padre, por lo que `this` no será la instancia de Vue como esperarías y `this.fetchTodos` tendrá un valor `undefined`.

## # beforeCreate

- Tipo: `Function`

- Detalles:

Llamada asíncronicamente apenas la instancia ha sido inicializada, antes que la observación de datos y la configuración de eventos/observadores.

- Lee también: [Diagrama del ciclo de vida](#)
- [Source](#)

## # created

- Tipo: `Function`

- Detalles:

Llamada sincrónicamente luego de que la instancia ha sido creada. En este momento, la instancia ha terminado de procesar las opciones lo cual significa que ya han sido configuradas: la observación de datos, propiedades computadas, métodos, funciones *callback* para eventos/observadores. Sin embargo, la fase de montaje no ha sido iniciada, y la propiedad `$el` no estará disponible todavía.

- Lee también: [Diagrama del ciclo de vida](#)
- [Source](#)

## # beforeMount

- Tipo: `Function`
- Detalles:

Llamada apenas comienza el montaje: la función `render` está por ser llamada por primera vez.

Este *hook* no se llama durante el renderizado del lado servidor.

- Lee también: [Diagrama del ciclo de vida](#)
- [Source](#)

## # mounted

- Tipo: `Function`
- Detalles:

Llamada apenas ha sido montada la instancia en el lugar de `el` reemplazado con el recientemente creado `vm.$el`. Si la instancia principal es montada en un elemento dentro del documento, `vm.$el` estará también en el documento cuando `mounted` es llamada.

Este *hook* no se llama durante el renderizado del lado servidor.

- Lee también: [Diagrama del ciclo de vida](#)
- [Source](#)

## # beforeUpdate

- Tipo: **Function**

- Detalles:

Llamada cuando los datos cambian, antes que el DOM virtual sea re-renderizado y actualizado.

Puedes realizar otros cambios de estado en este *hook* y no dispararán re-renderizaciones adicionales.

Este *hook* no se llama durante el renderizado del lado servidor.

- Lee también: **Diagrama del ciclo de vida**
- **Source**

## # updated

- Tipo: **Function**

- Detalles:

Llamada luego que un cambio en los datos hayan generado que el DOM virtual haya sido re-renderizado y actualizado.

El DOM del componente ya habrá sido actualizado cuando este *hook* es llamado, por lo que puedes realizar operaciones dependientes del DOM aquí. Sin embargo, en la mayoría de los casos deberías evitar cambiar el estado dentro de este *hook*. Para reaccionar ante cambios de estado, normalmente es mejor utilizar **propiedades computadas** u **observadores** en su lugar.

Este *hook* no se llama durante el renderizado del lado servidor.

- Lee también: **Diagrama del ciclo de vida**
- **Source**

## # activated

- Tipo: `Function`

- Detalles:

Llamada cuando un componente `keep-alive` es activado.

Este *hook* no se llama durante el renderizado del lado servidor.

- Lee también:
  - Componentes incorporados - `keep-alive`
  - Componentes dinámicos - `keep-alive`
- Source

## # deactivated

- Tipo: `Function`

- Detalles:

Llamada cuando un componente `keep-alive` es desactivado.

Este *hook* no se llama durante el renderizado del lado servidor.

- Lee también:
  - Componentes incorporados - `keep-alive`
  - Componentes dinámicos - `keep-alive`
- Source

## # beforeDestroy

- Tipo: **Function**

- Detalles:

Llamada apenas antes que una instancia Vue es destruida. En este momento la instancia todavía es completamente funcional.

Este *hook* no se llama durante el renderizado del lado servidor.

- Lee también: **Diagrama del ciclo de vida**
- **Source**

## # destroyed

- Tipo: **Function**

- Detalles:

Llamada luego que una instancia de Vue ha sido destruida. Cuando este *hook* es llamado, todas la directivas de la instancia de Vue han sido desenlazadas, todos los *listeners* de eventos han sido quitados y todas las intancias Vue hijas también han sido destruidas.

Este *hook* no se llama durante el renderizado del lado servidor.

- Lee también: **Diagrama del ciclo de vida**
- **Source**

## Opciones / Recursos

---

## # directives

- Tipo: **Object**

- Detalles:

Un hash de directivas para hacer disponibles a la instancia de Vue.

- Lee también:
  - [Directivas personalizadas](#)
  - [Convención de nombres de recursos](#)
- [Source](#)

## # filters

- Tipo: [Object](#)

- Detalles:

Un hash de filtros para hacer disponibles a la instancia de Vue.

- Lee también:
  - [Vue.filter](#)
- [Source](#)

## # components

- Tipo: [Object](#)

- Detalles:

A hash of componentes to be made available to the Vue instance.

- Lee también:
  - [Components](#)

- [Source](#)

## Opciones / Misc

---

### # parent

- **Tipo:** `Vue instance`
- **Detalles:**

Especifica la instancia padre de la instancia a ser creada. Establece una relación padre-hijo entre las dos. El padre será accesible a través de `this.$parent` para el hijo, y el hijo será agregado al vector `$children` del padre.



Utiliza `$parent` y `$children` con mesura - sirven principalmente como una vía de escape. Es preferible utilizar propiedades y eventos para la comunicación padre-hijo.

- [Source](#)

### # mixins

- **Tipo:** `Array<Object>`
- **Detalles:**

La opción `mixins` acepta un vector de objetos *mixin*. Estos objetos pueden contener opciones de instancia como cualquier objeto de instancia normal, y pueden ser fusionados con las opciones finales utilizando la misma lógica encontrada en `Vue.extend()`. Si tu *mixin* contiene un *hook created* y el componente también, ambas funciones serán ejecutadas.

Los *hooks de mixin* son llamados en el orden que son provistos y luego de los propios del componente.

- Ejemplo:

JS

```
var mixin = {
  created: function () { console.log(1) }
}
var vm = new Vue({
  created: function () { console.log(2) },
  mixins: [mixin]
})
// -> 1
// -> 2
```

- Lee también: [Mixins](#)
- [Source](#)

## # name

- Tipo: `string`
- Restricciones: solo respetado cuando se utiliza como opción de componente.
- Detalles:

Permite al componente invocarse recursivamente en su plantilla. Nota que cuando un componente es registrado globalmente con `Vue.component()`, el ID global es automáticamente establecido como su nombre.

Otro beneficio de especificar la opción `name` es la depuración. Los componentes con nombre permiten entregar mensajes de advertencia de más ayuda. También, cuando se inspecciona una aplicación con **vue-devtools**, los componentes sin nombre aparecerán como `<AnonymousComponent>`, lo cual no es muy informativo. Al proveer la opción `name`, obtendrás un árbol de componentes mucho más descriptivo.

- [Source](#)



## # extends

- **Tipo:** `Object` | `Function`

- **Detalles:**

Permite extender declarativamente otro componente (puede ser tanto un objeto de componentes como un constructor) sin tener que utilizar `Vue.extend`. Esto está pensado principalmente para hacer más sencillo el extender componentes de un solo archivo.

Es similar a `mixins`, siendo la diferencia que las opciones propias de los componentes tienen mayor prioridad que el componente base que está siendo extendido.

- **Ejemplo:**

```
var CompA = { ... }

// extiende CompA sin tener que llamar a Vue.extend
var CompB = {
  extends: CompA,
  ...
}
```

JS

- **Source**

## # delimiters

- **Tipo:** `Array<string>`
- **Valor por defecto:** `["{{", "}}"]`

- **Detalles:**

Change the plain text interpolation delimiters. **This option is only available in the standalone build.**

- **Ejemplo:**

```
new Vue({  
  delimiters: ['${', '}']  
})  
  
// Delimiters changed to ES6 template string style
```

- [Source](#)

## # functional

- **Tipo:** `boolean`
- **Detalles:**

Causes a component to be stateless (no `data` ) and instanceless (no `this` context). They are simply a `render` function that returns virtual nodes making them much cheaper to render.

- **Lee también:** [Functional Components](#)
- [Source](#)

## Instance Properties

---

### # vm.\$data

- **Tipo:** `Object`
- **Detalles:**

The data object that the Vue instance is observing. The Vue instance proxies access to the properties on its data object.

- **Lee también:** [Options - data](#)

- [Source](#)

## # vm.\$el

- Tipo: `HTMLElement`
- Read only
- Detalles:

The root DOM element that the Vue instance is managing.

- [Source](#)

## # vm.\$options

- Tipo: `Object`
- Read only
- Detalles:

The instantiation options used for the current Vue instance. This is useful when you want to include custom properties in the options:

JS

```
new Vue({
  customOption: 'foo',
  created: function () {
    console.log(this.$options.customOption) // -> 'foo'
  }
})
```

- [Source](#)

## # vm.\$parent

- Tipo: `Vue instance`
- Read only
- Detalles:

The parent instance, if the current instance has one.

- Source

## # vm.\$root

- Tipo: `Vue instance`
- Read only
- Detalles:

The root Vue instance of the current component tree. If the current instance has no parents this value will be itself.

- Source

## # vm.\$children

- Tipo: `Array<Vue instance>`
- Read only
- Detalles:

The direct child components of the current instance. **Note there's no order guarantee for `$children` , and it is not reactive.** If you find yourself trying to use `$children` for data

binding, consider using an Array and `v-for` to generate child components, and use the Array as the source of truth.

- **Source**

## # vm.\$slots

- **Tipo:** `{ [name: string]: ?Array<VNode> }`
- **Read only**
- **Detalles:**

Used to programmatically access content **distributed by slots**. Each **named slot** has its own corresponding property (e.g. the contents of `slot="foo"` will be found at `vm.$slots.foo`). The `default` property contains any nodes not included in a named slot.

Accessing `vm.$slots` is most useful when writing a component with a **render function**.

- **Ejemplo:**

HTML

```
<blog-post>
  <h1 slot="header">
    About Me
  </h1>

  <p>Here's some page content, which will be included in vm.$slots.default, beca

  <p slot="footer">
    Copyright 2016 Evan You
  </p>

  <p>If I have some content down here, it will also be included in vm.$slots.def
</blog-post>
```

JS

```
Vue.component('blog-post', {
  render: function (createElement) {
    var header = this.$slots.header
    var body   = this.$slots.default
```

```

    var footer = this.$slots.footer
    return createElement('div', [
      createElement('header', header),
      createElement('main', body),
      createElement('footer', footer)
    ])
  }
})

```

- Lee también:
  - `<slot>` Component
  - Content Distribution with Slots
  - Render Functions: Slots
- Source

## # vm.\$scopedSlots

New in 2.1.0

- Tipo: `{ [name: string]: props => VNode | Array<VNode> }`

- Read only

- Detalles:

Used to programmatically access **scoped slots**. For each slot, including the `default` one, the object contains a corresponding function that returns VNodes.

Accessing `vm.$scopedSlots` is most useful when writing a component with a **render function**.

- Lee también:
  - `<slot>` Component
  - Scoped Slots
  - Render Functions: Slots

## # vm.\$refs

- Tipo: `Object`
- Read only
- Detalles:

An object that holds child components that have `ref` registered.

- Lee también:
  - `Child Component Refs`
  - `ref`
- `Source`

## # vm.\$isServer

- Tipo: `boolean`
- Read only
- Detalles:

Whether the current Vue instance is running on the server.

- Lee también: `Server-Side Rendering`
- `Source`

## Instance Methods / Data

---

## # vm.\$watch( expOrFn, callback, [options] )

- Argumentos:

- {string | Function} expOrFn
- {Function} callback
- {Object} [options]
  - {boolean} deep
  - {boolean} immediate

- Devuelve: {Function} unwatch

- Uso:

Watch an expression or a computed function on the Vue instance for changes. The callback gets called with the new value and the old value. The expression only accepts simple dot-delimited paths. For more complex expression, use a function instead.

- Source

! Note: when mutating (rather than replacing) an Object or an Array, the old value will be the same as new value because they reference the same Object/Array. Vue doesn't keep a copy of the pre-mutate value.

- Ejemplo:

```
// keypath
vm.$watch('a.b.c', function (newVal, oldVal) {
  // do something
})

// function
vm.$watch(
  function () {
    return this.a + this.b
  },
  function (newVal, oldVal) {
    // do something
  }
)
```

JS



)

`vm.$watch` returns an `unwatch` function that stops firing the callback:

JS

```
var unwatch = vm.$watch('a', cb)
// later, teardown the watcher
unwatch()
```

- **Option: deep**

To also detect nested value changes inside Objects, you need to pass in `deep: true` in the options argument. Note that you don't need to do so to listen for Array mutations.

JS

```
vm.$watch('someObject', callback, {
  deep: true
})
vm.someObject.nestedValue = 123
// callback is fired
```

- **Option: immediate**

Passing in `immediate: true` in the option will trigger the callback immediately with the current value of the expression:

JS

```
vm.$watch('a', callback, {
  immediate: true
})
// callback is fired immediately with current value of `a`
```

## # `vm.$set( object, key, value )`

- **Argumentos:**

- `{Object}` object
  - `{string}` key
  - `{any}` value
- **Devuelve:** the set value.
  - **Uso:**

This is the **alias** of the global `Vue.set` .

- **Lee también:** `Vue.set`
- **Source**

## # `vm.$delete( object, key )`

- **Argumentos:**
  - `{Object}` object
  - `{string}` key

- **Uso:**

This is the **alias** of the global `Vue.delete` .

- **Lee también:** `Vue.delete`
- **Source**

## Instance Methods / Events

---

## # `vm.$on( event, callback )`

- **Argumentos:**

- `{string} event`
- `{Function} callback`

- **Uso:**

Listen for a custom event on the current vm. Events can be triggered by `vm.$emit`. The callback will receive all the additional arguments passed into these event-triggering methods.

- **Ejemplo:**

```
vm.$on('test', function (msg) {
  console.log(msg)
})
vm.$emit('test', 'hi')
// -> "hi"
```

JS

- **Source**

## # `vm.$once( event, callback )`

- **Argumentos:**

- `{string} event`
- `{Function} callback`

- **Uso:**

Listen for a custom event, but only once. The listener will be removed once it triggers for the first time.

- **Source**

## # `vm.$off( [event, callback] )`

- **Argumentos:**

- `{string} [event]`
- `{Function} [callback]`

- **Uso:**

Remove event listener(s).

- If no arguments are provided, remove all event listeners;
- If only the event is provided, remove all listeners for that event;
- If both event and callback are given, remove the listener for that specific callback only.

- **Source**

## # `vm.$emit( event, [...args] )`

- **Argumentos:**

- `{string} event`
- `[...args]`

Trigger an event on the current instance. Any additional arguments will be passed into the listener's callback function.

- **Source**

## Instance Methods / Lifecycle

---

## # `vm.$mount( [elementOrSelector] )`

- **Argumentos:**

- `{Element | string} [elementOrSelector]`

- `{boolean} [hydrating]`
- **Devuelve:** `vm` - the instance itself
- **Uso:**

If a Vue instance didn't receive the `el` option at instantiation, it will be in “unmounted” state, without an associated DOM element. `vm.$mount()` can be used to manually start the mounting of an unmounted Vue instance.

If `elementOrSelector` argument is not provided, the template will be rendered as an off-document element, and you will have to use native DOM API to insert it into the document yourself.

The method returns the instance itself so you can chain other instance methods after it.

- **Ejemplo:**

JS

```
var MyComponent = Vue.extend({
  template: '<div>Hello!</div>'
})

// create and mount to #app (will replace #app)
new MyComponent().$mount('#app')

// the above is the same as:
new MyComponent({ el: '#app' })

// or, render off-document and append afterwards:
var component = new MyComponent().$mount()
document.getElementById('app').appendChild(component.$el)
```

- **Lee también:**
  - **Diagrama del ciclo de vida**
  - **Server-Side Rendering**
- **Source**

## # vm.\$forceUpdate()

- **Uso:**

Force the Vue instance to re-render. Note it does not affect all child components, only the instance itself and child components with inserted slot content.

- **Source**

## # vm.\$nextTick( [callback] )

- **Argumentos:**

- `{Function} [callback]`

- **Uso:**

Defer the callback to be executed after the next DOM update cycle. Use it immediately after you've changed some data to wait for the DOM update. This is the same as the global `Vue.nextTick`, except that the callback's `this` context is automatically bound to the instance calling this method.

**New in 2.1.0:** returns a Promise if no callback is provided and Promise is supported in the execution environment.

- **Ejemplo:**

```
new Vue({
  // ...
  methods: {
    // ...
    example: function () {
      // modify data
      this.message = 'changed'
      // DOM is not updated yet
      this.$nextTick(function () {
        // DOM is now updated
        // `this` is bound to the current instance
      })
    }
  }
})
```

```
        this.doSomethingElse()
      })
    }
  }
})
```

- Lee también:
  - [Vue.nextTick](#)
  - [Async Update Queue](#)
- [Source](#)

## # vm.\$destroy()

- **Uso:**

Completely destroy a vm. Clean up its connections with other existing vms, unbind all its directives, turn off all event listeners.

Triggers the `beforeDestroy` and `destroyed` hooks.



In normal use cases you shouldn't have to call this method yourself. Prefer controlling the lifecycle of child components in a data-driven fashion using `v-if` and `v-for`.

- Lee también: [Diagrama del ciclo de vida](#)
- [Source](#)

## Directives

---

### # v-text

- Expects: `string`

- Detalles:

Updates the element's `textContent` . If you need to update the part of `textContent` , you should use `{{ Mustache }}` interpolations.

- Ejemplo:

HTML

```
<span v-text="msg"></span>
<!-- same as -->
<span>{{msg}}</span>
```

- Lee también: [Data Binding Syntax - interpolations](#)

- [Source](#)

## # v-html

- Expects: `string`

- Detalles:

Updates the element's `innerHTML` . **Note that the contents are inserted as plain HTML - they will not be compiled as Vue templates.** If you find yourself trying to compose templates using `v-html` , try to rethink the solution by using components instead.

!

Dynamically rendering arbitrary HTML on your website can be very dangerous because it can easily lead to **XSS attacks**. Only use `v-html` on trusted content and **never** on user-provided content.

- Ejemplo:

HTML

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

- Lee también: [Data Binding Syntax - interpolations](#)



- **Source**

## # v-show

- **Expects:** `any`
- **Uso:**

Toggle's the element's `display` CSS property based on the truthy-ness of the expression value.

This directive triggers transitions when its condition changes.

- **Lee también: Conditional Rendering - v-show**
- **Source**

## # v-if

- **Expects:** `any`
- **Uso:**

Conditionally render the element based on the truthy-ness of the expression value. The element and its contained directives / components are destroyed and re-constructed during toggles. If the element is a `<template>` element, its content will be extracted as the conditional block.

This directive triggers transitions when its condition changes.



! When used together with v-if, v-for has a higher priority than v-if. See the **list rendering guide** for details.

- **Lee también: Conditional Rendering - v-if**

- [Source](#)

## # v-else

- Does not expect expression
- **Restricciones:** previous sibling element must have `v-if` or `v-else-if` .
- **Uso:**

Denote the “else block” for `v-if` or a `v-if / v-else-if` chain.

HTML

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

- **Lee también:**
  - [Conditional Rendering - v-else](#)
- [Source](#)

## # v-else-if

New in 2.1.0

- **Expects:** `any`
- **Restricciones:** previous sibling element must have `v-if` or `v-else-if` .
- **Uso:**

Denote the “else if block” for `v-if` . Can be chained.

HTML

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

- Lee también: **Conditional Rendering - v-else-if**

## # v-for

- **Expects:** `Array` | `Object` | `number` | `string`
- **Uso:**

Render the element or template block multiple times based on the source data. The directive's value must use the special syntax `alias in expression` to provide an alias for the current element being iterated on:

HTML

```
<div v-for="item in items">
  {{ item.text }}
</div>
```

Alternatively, you can also specify an alias for the index (or the key if used on an Object):

HTML

```
<div v-for="(item, index) in items"></div>
<div v-for="(val, key) in object"></div>
<div v-for="(val, key, index) in object"></div>
```

The default behavior of `v-for` will try to patch the elements in-place without moving them. To force it to reorder elements, you need to provide an ordering hint with the `key` special attribute:

HTML

```
<div v-for="item in items" :key="item.id">
  {{ item.text }}
</div>
```

! When used together with `v-if`, `v-for` has a higher priority than `v-if`. See the [list rendering guide](#) for details.

The detailed usage for `v-for` is explained in the guide section linked below.

- **Lee también:**

- [List Rendering](#)
- [key](#)

- [Source](#)

## # v-on

- **Shorthand:** `@`
- **Expects:** `Function | Inline Statement`
- **Argument:** `event (required)`
- **Modifiers:**
  - `.stop` - call `event.stopPropagation()` .
  - `.prevent` - call `event.preventDefault()` .
  - `.capture` - add event listener in capture mode.
  - `.self` - only trigger handler if event was dispatched from this element.
  - `.{keyCode | keyAlias}` - only trigger handler on certain keys.

- `.native` - listen for a native event on the root element of component.
  - `.once` - trigger handler at most once.
- **Uso:**

Attaches an event listener to the element. The event type is denoted by the argument. The expression can either be a method name or an inline statement, or simply omitted when there are modifiers present.

When used on a normal element, it listens to **native DOM events** only. When used on a custom element component, it also listens to **custom events** emitted on that child component.

When listening to native DOM events, the method receives the native event as the only argument. If using inline statement, the statement has access to the special `$event` property:

```
v-on:click="handle('ok', $event)" .
```

- **Ejemplo:**

HTML

```
<!-- method handler -->
<button v-on:click="doThis"></button>

<!-- inline statement -->
<button v-on:click="doThat('hello', $event)"></button>

<!-- shorthand -->
<button @click="doThis"></button>

<!-- stop propagation -->
<button @click.stop="doThis"></button>

<!-- prevent default -->
<button @click.prevent="doThis"></button>

<!-- prevent default without expression -->
<form @submit.prevent></form>

<!-- chain modifiers -->
<button @click.stop.prevent="doThis"></button>

<!-- key modifier using keyAlias -->
<input @keyup.enter="onEnter">

<!-- key modifier using keyCode -->
<input @keyup.13="onEnter">
```

```
<!-- the click event will be triggered at most once -->
<button v-on:click.once="doThis"></button>
```

Listening to custom events on a child component (the handler is called when “my-event” is emitted on the child):

HTML

```
<my-component @my-event="handleThis"></my-component>

<!-- inline statement -->
<my-component @my-event="handleThis(123, $event)"></my-component>

<!-- native event on component -->
<my-component @click.native="onClick"></my-component>
```

- Lee también:
  - **Methods and Event Handling**
  - **Components - Custom Events**
- **Source**

## # v-bind

- **Shorthand:** `:`
- **Expects:** `any (with argument) | Object (without argument)`
- **Argument:** `attrOrProp (optional)`
- **Modifiers:**
  - `.prop` - Bind as a DOM property instead of an attribute. (**what's the difference?**)
  - `.camel` - transform the kebab-case attribute name into camelCase. (supported since 2.1.0)
- **Uso:**

Dynamically bind one or more attributes, or a component prop to an expression.

When used to bind the `class` or `style` attribute, it supports additional value types such as Array or Objects. See linked guide section below for more details.

When used for prop binding, the prop must be properly declared in the child component.

When used without an argument, can be used to bind an object containing attribute name-value pairs. Note in this mode `class` and `style` does not support Array or Objects.

- **Ejemplo:**

HTML

```
<!-- bind an attribute -->


<!-- shorthand -->


<!-- with inline string concatenation -->


<!-- class binding -->
<div :class="{ red: isRed }"></div>
<div :class="[classA, classB]"></div>
<div :class="[classA, { classB: isB, classC: isC }]">

<!-- style binding -->
<div :style="{ fontSize: size + 'px' }"></div>
<div :style="[styleObjectA, styleObjectB]"></div>

<!-- binding an object of attributes -->
<div v-bind="{ id: someProp, 'other-attr': otherProp }"></div>

<!-- DOM attribute binding with prop modifier -->
<div v-bind:text-content.prop="text"></div>

<!-- prop binding. "prop" must be declared in my-component. -->
<my-component :prop="something"></my-component>

<!-- XLink -->
<svg><a :xlink:special="foo"></a></svg>
```

The `.camel` modifier allows camelizing a `v-bind` attribute name when using in-DOM templates, e.g. the SVG `viewBox` attribute:

HTML

```
<svg :view-box.camel="viewBox"></svg>
```

`.camel` is not needed if you are using string templates, or compiling with `vue-loader` / `vueify`.

- **Lee también:**
  - **Class and Style Bindings**
  - **Components - Component Props**
- **Source**

## # v-model

- **Expects:** varies based on value of form inputs element or output of components
- **Limited to:**
  - `<input>`
  - `<select>`
  - `<textarea>`
  - components
- **Modifiers:**
  - `.lazy` - listen to `change` events instead of `input`
  - `.number` - cast input string to numbers
  - `.trim` - trim input

- **Uso:**

Create a two-way binding on a form input element or a component. For detailed usage and other notes, see the Guide section linked below.

- **Lee también:**
  - **Form Input Bindings**
  - **Components - Form Input Components using Custom Events**



- [Source](#)

## # v-pre

- Does not expect expression
- Uso:

Skip compilation for this element and all its children. You can use this for displaying raw mustache tags. Skipping large numbers of nodes with no directives on them can also speed up compilation.

- Ejemplo:

HTML

```
<span v-pre>{{ this will not be compiled }}</span>
```

- [Source](#)

## # v-cloak

- Does not expect expression
- Uso:

This directive will remain on the element until the associated Vue instance finishes compilation. Combined with CSS rules such as `[v-cloak] { display: none; }`, this directive can be used to hide un-compiled mustache bindings until the Vue instance is ready.

- Ejemplo:

CSS

```
[v-cloak] {  
  display: none;  
}
```

```
<div v-cloak>
  {{ message }}
</div>
```

The `<div>` will not be visible until the compilation is done.

- **Source**

## # v-once

- Does not expect expression
- Detalles:

Render the element and component **once** only. On subsequent re-renders, the element/component and all its children will be treated as static content and skipped. This can be used to optimize update performance.

```
<!-- single element -->
<span v-once>This will never change: {{msg}}</span>
<!-- the element have children -->
<div v-once>
  <h1>comment</h1>
  <p>{{msg}}</p>
</div>
<!-- component -->
<my-component v-once :comment="msg"></my-component>
<!-- v-for directive -->
<ul>
  <li v-for="i in list" v-once>{{i}}</li>
</ul>
```

- **Lee también:**
  - **Data Binding Syntax - interpolations**
  - **Components - Cheap Static Components with v-once**
- **Source**

# Special Attributes

---

## # key

- Expects: `string`

The `key` special attribute is primarily used as a hint for Vue's virtual DOM algorithm to identify VNodes when diffing the new list of nodes against the old list. Without keys, Vue uses an algorithm that minimizes element movement and tries to patch/reuse elements of the same type in-place as much as possible. With keys, it will reorder elements based on the order change of keys, and elements with keys that are no longer present will always be removed/destroyed.

Children of the same common parent must have **unique keys**. Duplicate keys will cause render errors.

The most common use case is combined with `v-for` :

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

HTML

It can also be used to force replacement of an element/component instead of reusing it. This can be useful when you want to:

- Properly trigger lifecycle hooks of a component
- Trigger transitions

For example:

```
<transition>
  <span :key="text">{{ text }}</span>
</transition>
```

HTML

When `text` changes, the `<span>` will always be replaced instead of patched, so a transition will be triggered.

- **Source**

## # ref

- **Expects:** `string`

`ref` is used to register a reference to an element or a child component. The reference will be registered under the parent component's `$refs` object. If used on a plain DOM element, the reference will be that element; if used on a child component, the reference will be component instance:

HTML

```
<!-- vm.$refs.p will be the DOM node -->
<p ref="p">hello</p>

<!-- vm.$refs.child will be the child comp instance -->
<child-comp ref="child"></child-comp>
```

When used on elements/components with `v-for`, the registered reference will be an Array containing DOM nodes or component instances.

An important note about the ref registration timing: because the refs themselves are created as a result of the render function, you cannot access them on the initial render - they don't exist yet! `$refs` is also non-reactive, therefore you should not attempt to use it in templates for data-binding.

- **Lee también: Child Component Refs**
- **Source**

## # slot

- **Expects:** `string`

Used on content inserted into child components to indicate which named slot the content belongs to.

For detailed usage, see the guide section linked below.

- Lee también: [Named Slots](#)
- [Source](#)

## Built-In Components

---

### # component

- **Props:**
  - `is` - string | ComponentDefinition | ComponentConstructor
  - `inline-template` - boolean
- **Uso:**

A “meta component” for rendering dynamic components. The actual component to render is determined by the `is` prop:

```
<!-- a dynamic component controlled by -->
<!-- the `componentId` property on the vm -->
<component :is="componentId"></component>

<!-- can also render registered component or component passed as prop -->
<component :is="$options.components.child"></component>
```

HTML

- Lee también: [Dynamic Components](#)
- [Source](#)

## # transition

- Props:

- `name` - string, Used to automatically generate transition CSS class names. e.g. `name: 'fade'` will auto expand to `.fade-enter` , `.fade-enter-active` , etc. Defaults to `"v"` .
- `appear` - boolean, Whether to apply transition on initial render. Defaults to `false` .
- `css` - boolean, Whether to apply CSS transition classes. Defaults to `true` . If set to `false` , will only trigger JavaScript hooks registered via component events.
- `type` - string, Specify the type of transition events to wait for to determine transition end timing. Available values are `"transition"` and `"animation"` . By default, it will automatically detect the type that has a longer duration.
- `mode` - string, Controls the timing sequence of leaving/entering transitions. Available modes are `"out-in"` and `"in-out"` ; defaults to simultaneous.
- `enter-class` - string
- `leave-class` - string
- `appear-class` - string
- `enter-to-class` - string
- `leave-to-class` - string
- `appear-to-class` - string
- `enter-active-class` - string
- `leave-active-class` - string
- `appear-active-class` - string

- Events:

- `before-enter`
- `before-leave`
- `before-appear`
- `enter`
- `leave`
- `appear`
- `after-enter`
- `after-leave`
- `after-appear`
- `enter-cancelled`
- `leave-cancelled` ( `v-show` only)
- `appear-cancelled`

- Uso:

`<transition>` serve as transition effects for **single** element/component. The `<transition>` does not render an extra DOM element, nor does it show up in the inspected component hierarchy. It simply applies the transition behavior to the wrapped content inside.

HTML

```
<!-- simple element -->
<transition>
  <div v-if="ok">toggled content</div>
</transition>

<!-- dynamic component -->
<transition name="fade" mode="out-in" appear>
  <component :is="view"></component>
</transition>

<!-- event hooking -->
<div id="transition-demo">
  <transition @after-enter="transitionComplete">
    <div v-show="ok">toggled content</div>
  </transition>
</div>
```

JS

```
new Vue({
  ...
  methods: {
    transitionComplete: function (el) {
      // for passed 'el' that DOM element as the argument, something ...
    }
  }
  ...
}).$mount('#transition-demo')
```

- **Lee también: Transitions: Entering, Leaving, and Lists**
- **Source**

## # transition-group

- **Props:**
  - `tag` - string, defaults to `span` .
  - `move-class` - overwrite CSS class applied during moving transition.

- exposes the same props as `<transition>` except `mode` .

- **Events:**

- exposes the same events as `<transition>` .

- **Uso:**

`<transition-group>` serve as transition effects for **multiple** elements/components. The `<transition-group>` renders a real DOM element. By default it renders a `<span>` , and you can configure what element it should render via the `tag` attribute.

Note every child in a `<transition-group>` must be **uniquely keyed** for the animations to work properly.

`<transition-group>` supports moving transitions via CSS transform. When a child's position on screen has changed after an update, it will get applied a moving CSS class (auto generated from the `name` attribute or configured with the `move-class` attribute). If the CSS `transform` property is "transition-able" when the moving class is applied, the element will be smoothly animated to its destination using the **FLIP technique**.

HTML

```
<transition-group tag="ul" name="slide">
  <li v-for="item in items" :key="item.id">
    {{ item.text }}
  </li>
</transition-group>
```

- **Lee también: Transitions: Entering, Leaving, and Lists**

- **Source**

## # keep-alive

- **Props:**

- `include` - string or RegExp. Only components matched by this will be cached.
- `exclude` - string or RegExp. Any component matched by this will not be cache ' .



- **Uso:**

When wrapped around a dynamic component, `<keep-alive>` caches the inactive component instances without destroying them. Similar to `<transition>`, `<keep-alive>` is an abstract component: it doesn't render a DOM element itself, and doesn't show up in the component parent chain.

When a component is toggled inside `<keep-alive>`, its `activated` and `deactivated` lifecycle hooks will be invoked accordingly.

Primarily used with preserve component state or avoid re-rendering.

HTML

```
<!-- basic -->
<keep-alive>
  <component :is="view"></component>
</keep-alive>

<!-- multiple conditional children -->
<keep-alive>
  <comp-a v-if="a > 1"></comp-a>
  <comp-b v-else></comp-b>
</keep-alive>

<!-- used together with <transition> -->
<transition>
  <keep-alive>
    <component :is="view"></component>
  </keep-alive>
</transition>
```

- **include and exclude**

### New in 2.1.0

The `include` and `exclude` props allow components to be conditionally cached. Both props can either be a comma-delimited string or a RegExp:

HTML

```
<!-- comma-delimited string -->
```

```

<keep-alive include="a,b">
  <component :is="view"></component>
</keep-alive>

<!-- regex (use v-bind) -->
<keep-alive :include="/a|b/">
  <component :is="view"></component>
</keep-alive>

```

The match is first checked on the component's own `name` option, then its local registration name (the key in the parent's `components` option) if the `name` option is not available. Anonymous components cannot be matched against.

! `<keep-alive>` does not work with functional components because they do not have instances to be cached.

- Lee también: [Dynamic Components - keep-alive](#)
- [Source](#)

## # slot

- Props:
  - `name` - string, Used for named slot.

- Uso:

`<slot>` serve as content distribution outlets in component templates. `<slot>` itself will be replaced.

For detailed usage, see the guide section linked below.

- Lee también: [Content Distribution with Slots](#)
- [Source](#)

## VNode Interface

---

- Please refer to the [VNode class declaration](#).

## Server-Side Rendering

---

- Please refer to the [vue-server-renderer package documentation](#).