Vue.js

1. Vue.js Concepto y componentes
2. Representación declarativa
3. Utilizar Vue en HTML
4. Primeros Pasos
5. Condicionales
6. Bucles
7. Entrada de datos
8. Componentes
9. Instancias
10. Hooks de instancias
11. Sintaxis de plantillas
12. Directivas
13. Condicionales
14. Representación de Listas
15. Vinculación de Clase y Estilo
16. Consumo de Servicios con Vue / Axios

¿Qué es Vue.js?

Vue es un **marco progresivo** para construir interfaces de usuario. A diferencia de otros marcos monolíticos, Vue está diseñado desde cero para ser adoptable gradualmente. La biblioteca principal se centra solo en la capa de vista y es fácil de integrar con otras bibliotecas o proyectos existentes. Por otro lado, Vue también es perfectamente capaz de impulsar aplicaciones sofisticadas de una sola página cuando se usa en combinación con **herramientas modernas** y **bibliotecas de soporte** .

Empezando

Instalación

La forma más fácil de probar Vue.js es usando el **ejemplo de Hello World** . Sentite libre de abrirlo en un navegador web y seguirlo mientras revisamos algunos ejemplos básicos. O ppodés **crear un index.html** e incluir Vue con:

```
<!-- development version, includes helpful console warnings -->
<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
o:
<!-- production version, optimized for size and speed →
<script src="https://cdn.jsdelivr.net/npm/vue"></script>
```

Ejemplo:

```
<!DOCTYPE html>
<html>
<head>
  <title>Mi primer Vue app</title>
  <script src="https://unpkg.com/vue"></script>   <!-- otra forma de incluir Vue -->
</head>
<body>
  <div id="app">
    {{ message }}
  </div>
  <script>
    var app = new Vue({
      el: '#app',
      data: {
        message: 'Hola Vue!'
      }
    })
  </script>
</body>
</html>
```

Representación declarativa

En el núcleo de Vue.js hay un sistema que nos permite renderizar datos de forma declarativa al DOM utilizando una sintaxis de plantilla sencilla:

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

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

Veremos como resultado:

¡Hola Vue!
Esto se parece bastante a renderizar una plantilla de cadena, pero Vue ha hecho mucho trabajo bajo la capa de instancias. Los datos y el DOM ahora están vinculados y todo es **reactivo** . ¿Como sabemos? Ejecutá el ejemplo anterior y abrí la consola JavaScript de tu navegador y establecé a `app.message` un valor diferente. Deberías ver el ejemplo renderizado anterior actualizado en consecuencia.

Tené en cuenta que ya no tenemos que interactuar con el HTML directamente. Una aplicación Vue se adjunta a un solo elemento DOM (`#app` en nuestro caso) y luego lo controla por completo. El HTML es nuestro punto de entrada, pero todo lo demás sucede dentro de la instancia de Vue recién creada.

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

```
<!DOCTYPE html>
<html>
<head>
  <title>Mi primer Vue app</title>
  <script src="https://unpkg.com/vue"></script>   <!-- otra forma de incluir Vue -->
</head>
<body>
<div id="app-2">
  <span v-bind:title="message">
    Mové tu mouse acá arriba y verás la hora de carga de esta página
  </span>
</div>
  <script>
   var app2 = new Vue({
  el: '#app-2',
  data: {
```

```
    message: 'Iniciada en ' + new Date().toLocaleString()
  }
})
  </script>
</body>
</html>
```

Nos encontramos con algo nuevo. El atributo `v-bind` que estás viendo se llama **directiva** . Las directivas tienen el prefijo `v-` para indicar que son atributos especiales proporcionados por Vue y, como habrás adivinado, aplican un comportamiento reactivo especial al DOM renderizado. Aquí, básicamente está diciendo "mantenga el atributo `title` de este elemento actualizado con la propiedad `message` en la instancia de Vue".

Si volvés a abrir la consola de JavaScript e ingresa `app2.message = 'algún mensaje nuevo'`, verás nuevamente que el HTML enlazado, en este caso el atributo `title`, se ha actualizado.

Condicionales y bucles

También es fácil alternar la presencia de un elemento:

```
<!DOCTYPE html>
<html>
<head>
  <title>Mi primer Vue app</title>
  <script src="https://unpkg.com/vue"></script>   <!-- otra forma de incluir Vue -->
</head>
<body>
<div id="app-3">
  <span v-if="seen">Ahora me ves</span>
</div>
  <script>
var app3 = new Vue({
  el: '#app-3',
  data: {
    seen: false
  }
})
  </script>
</body>
</html>
```

Entrá en la consola y cambiá `app3.seen = false`. Deberías ver desaparecer el mensaje.

Este ejemplo demuestra que podemos vincular datos no solo a texto y atributos, sino también a la **estructura** del DOM. Además, Vue también proporciona un potente sistema de efectos de transición que puede aplicar automáticamente cuando Vue inserta / actualiza / elimina elementos.

Hay bastantes otras directivas, cada una con su propia funcionalidad especial. Por ejemplo, la directiva `v-for` se puede utilizar para mostrar una lista de elementos utilizando los datos de una matriz:

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

var app4 = new Vue({
  el: '#app-4',
  data: {
    todos: [
      { text: 'Estudio JavaScript' },
```

```
    { text: 'Estudio Vue' },
    { text: 'Programo FrontEnd' }
  ]
 }
})
```

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

Manejo de la entrada del usuario

Para permitir que los usuarios interactúen con su aplicación, podemos usar la directiva v-on para adjuntar detectores de eventos que invocan métodos en nuestras instancias de Vue:

```
<!DOCTYPE html>
<html>
<head>
  <title>Mi primer Vue app</title>
  <script src="https://unpkg.com/vue"></script>   <!-- otra forma de incluir Vue -->
</head>
<body>
<div id="app-5">
  <p>{{ message }}</p>
  <button v-on:click="Mensajereverso">Mensaje al reverso</button>
</div>
<script>
var app5 = new Vue({
  el: '#app-5',
  data: {
    message: 'Hola Vue.js!'
  },
  methods: {
    Mensajereverso: function () {
      this.message = this.message.split('').reverse().join('')
    }
  }
})
</script>
</body>
</html>
```

Tené en cuenta que en este método actualizamos el estado de nuestra aplicación sin tocar el DOM: todas las manipulaciones del DOM son manejadas por Vue y el código que escribe se centra en la lógica subyacente.

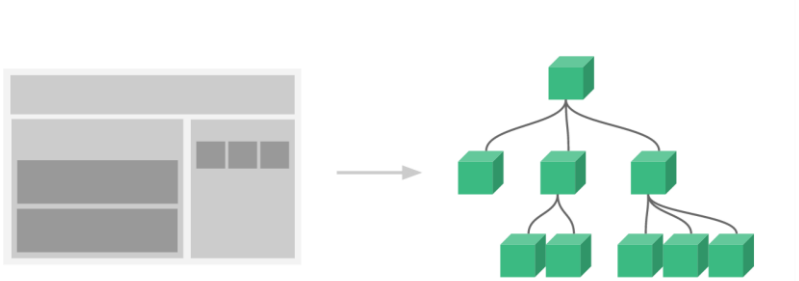
Vue también proporciona la directiva v-model que hace que el enlace bidireccional entre la entrada del formulario y el estado de la aplicación sea muy sencillo:

```
<!DOCTYPE html>
<html>
<head>
  <title>Mi primer Vue app</title>
  <script src="https://unpkg.com/vue"></script>   <!-- otra forma de incluir Vue -->
</head>
<body>
<div id="app-6">
  <p>{{ message }}</p>
  <input v-model="message">
</div>
<script>
var app6 = new Vue({
  el: '#app-6',
  data: {
```

```
    message: 'Hello Vue!'
  }
})
</script>
</body>
</html>
```

Componer con componentes

El sistema de componentes es otro concepto importante en Vue, porque es una abstracción que nos permite crear aplicaciones a gran escala compuestas por componentes pequeños, autónomos y, a menudo, reutilizables. Si lo pensamos bien, casi cualquier tipo de interfaz de aplicación se puede abstraer en un árbol de componentes:



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

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

var app = new Vue(...)
```

Ahora puede componerlo en la plantilla de otro componente:

```
<ol>
  <!-- Creo una instancia de todo-item -->
  <todo-item></todo-item>
</ol>
```

Pero esto generaría el mismo texto para cada tarea, lo cual no es muy interesante. Deberíamos poder pasar datos del ámbito principal a los componentes secundarios. Modifiquemos la definición del componente para que acepte un **accesorio** :

```
Vue.component('todo-item', {
   // El componente de todo-item ahora acepta un
   // "prop", que es como un atributo personalizado.
   // Este accesorio se llama todo.
  props: ['todo'],
  template: '<li>{{ todo.text }}</li>'
})
```

Ahora podemos pasar la tarea a cada componente repetido usando `v-bind`:

```
<div id="app-7">
  <ol>
    <!--
          Ahora proporcionamos a cada elemento de tarea pendiente el objeto de tarea       pendiente
      está representando, para que su contenido pueda ser dinámico.
      También debemos proporcionar a cada componente una "clave",
```

```
      que se explicará más adelante.
    -->
    <todo-item
      v-for="item in groceryList"
      v-bind:todo="item"
      v-bind:key="item.id"
    ></todo-item>
  </ol>
</div>

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

var app7 = new Vue({
  el: '#app-7',
  data: {
    groceryList: [
      { id: 0, text: 'Vegetales' },
      { id: 1, text: 'Queso' },
      { id: 2, text: 'Cualquier otra comida' }
    ]
  }
})
```

Este es un ejemplo artificial, pero logramos separar nuestra aplicación en dos unidades más pequeñas, y el children está razonablemente bien desacoplado del padre a través de la interfaz de accesorios. Ahora podemos mejorar aún más nuestro `<todo-item>`componente con una plantilla y lógica más complejas sin afectar la aplicación principal.

En una aplicación grande, es necesario dividir toda la aplicación en componentes para que el desarrollo sea manejable. Hay un ejemplo de cómo se vería la plantilla de una aplicación con componentes:

```
<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

Es posible que hayas notado que los componentes de Vue son muy similares a los **elementos personalizados** , que forman parte de la **especificación de componentes web** . Esto se debe a que la sintaxis del componente de Vue se modela libremente según la especificación. Por ejemplo, los

componentes de Vue implementan la **API Slot** y el atributo especial is. Sin embargo, existen algunas diferencias clave:

1. La especificación de componentes web se ha finalizado, pero no se implementa de forma nativa en todos los navegadores. Safari 10.1+, Chrome 54+ y Firefox 63+ admiten componentes web de forma nativa. En comparación, los componentes de Vue no requieren polyfills y funcionan de manera consistente en todos los navegadores compatibles (IE9 y superior). Cuando sea necesario, los componentes de Vue también se pueden envolver dentro de un elemento personalizado nativo.

2. Los componentes de Vue brindan características importantes que no están disponibles en elementos personalizados simples, en particular el flujo de datos entre componentes, la comunicación de eventos personalizados e integraciones de herramientas de construcción.

Aunque Vue no usa elementos personalizados internamente, tiene una **gran interoperabilidad** cuando se trata de consumir o distribuir como elementos personalizados. Vue CLI también admite la creación de componentes de Vue que se registran como elementos personalizados nativos.

Instancias de Vue

Creando una instancia de Vue

Cada aplicación de Vue comienza creando una nueva **instancia de Vue** con la función `Vue`:

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

Aunque no está estrictamente asociado con el **patrón MVVM** , el diseño de Vue se inspiró en parte en él. Como convención, a menudo usamos la variable `vm`(abreviatura de ViewModel) para referirnos a nuestra instancia de Vue.

Cuando se crea una instancia de Vue, pasa un **objeto de opciones** . Describiré cómo podés utilizar estas opciones para crear un comportamiento deseado.

Una aplicación de Vue consiste en una **instancia raíz de Vue** creada con `new Vue`, opcionalmente organizada en un árbol de componentes reutilizables anidados. Por ejemplo, el árbol de componentes de una aplicación de tareas pendientes podría verse así:

```
Root Instance
└─ TodoList
   ├─ TodoItem
   │  ├─ TodoButtonDelete
   │  └─ TodoButtonEdit
   └─ TodoListFooter
      ├─ TodosButtonClear
      └─ TodoListStatistics
```

Todos los componentes de Vue también son instancias de Vue y, por lo tanto, aceptan el mismo objeto de opciones (excepto por algunas opciones específicas de raíz).

Datos y métodos

Cuando se crea una instancia de Vue, agrega todas las propiedades que se encuentran en su objeto `data` al **sistema de reactividad** de Vue . Cuando los valores de esas propiedades cambian, la vista "reaccionará", actualizándose para coincidir con los nuevos valores.

```
// Objeto data
var data = { a: 1 }

// el objeto es agregado a la instancia de Vue
var vm = new Vue({
  data: data
})
```

```
// Obteniendo la propiedad en la instancia
// devuelve el de los datos originales
vm.a == data.a // => true

// Estableciendo la propiedad en la instancia
// también afecta a los datos originales
vm.a = 2
data.a // => 2

// ... y vice-versa
data.a = 3
vm.a // => 3
```

Cuando estos datos cambien, la vista volverá a renderizarse. Cabe señalar que las propiedades en `data` solo son **reactivas** si existían cuando se creó la instancia. Eso significa que si agrega una nueva propiedad, como:

```
vm.b = 'hola'
```

Entonces los cambios a `b` no activarán ninguna actualización de vista. Si sabe que necesitará una propiedad más adelante, pero comienza vacía o no existe, deberá establecer un valor inicial. Por ejemplo:

```
data: {
  newTodoText: '',
  visitCount: 0,
  hideCompletedTodos: false,
  todos: [],
  error: null
}
```

La única excepción a esto es el uso de `Object.freeze()`, que evita que se modifiquen las propiedades existentes, lo que también significa que el sistema de reactividad no puede *rastrear los* cambios.

```
var obj = {
  foo: 'bar'
}

Object.freeze(obj)

new Vue({
  el: '#app',
  data: obj
})

<div id="app">
  <p>{{ foo }}</p>
  <!-- esto no se actualizará `foo`! -->
  <button v-on:click="foo = 'baz'">Change it</button>
</div>
```

Además de las propiedades de los datos, las instancias de Vue exponen una serie de propiedades y métodos de instancia útiles. Estos tienen el prefijo `$` para diferenciarlos de las propiedades definidas por el usuario. Por ejemplo:

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

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

// $ watch es un método de instancia
vm.$watch('a', function (newValue, oldValue) {
  // Esta devolución de llamada se llamará cuando cambie `vm.a`
})
```

Hooks del ciclo de vida de la instancia

Cada instancia de Vue pasa por una serie de pasos de inicialización cuando se crea; por ejemplo, necesita configurar la observación de datos, compilar la plantilla, montar la instancia en el DOM y actualizar el DOM cuando los datos cambian. En el camino, también ejecuta funciones llamadas **enlaces de ciclo de vida** , lo que brinda a los usuarios la oportunidad de agregar su propio código en etapas específicas.

Por ejemplo, el llamado a `created` se puede usar para ejecutar código después de que se crea una instancia:

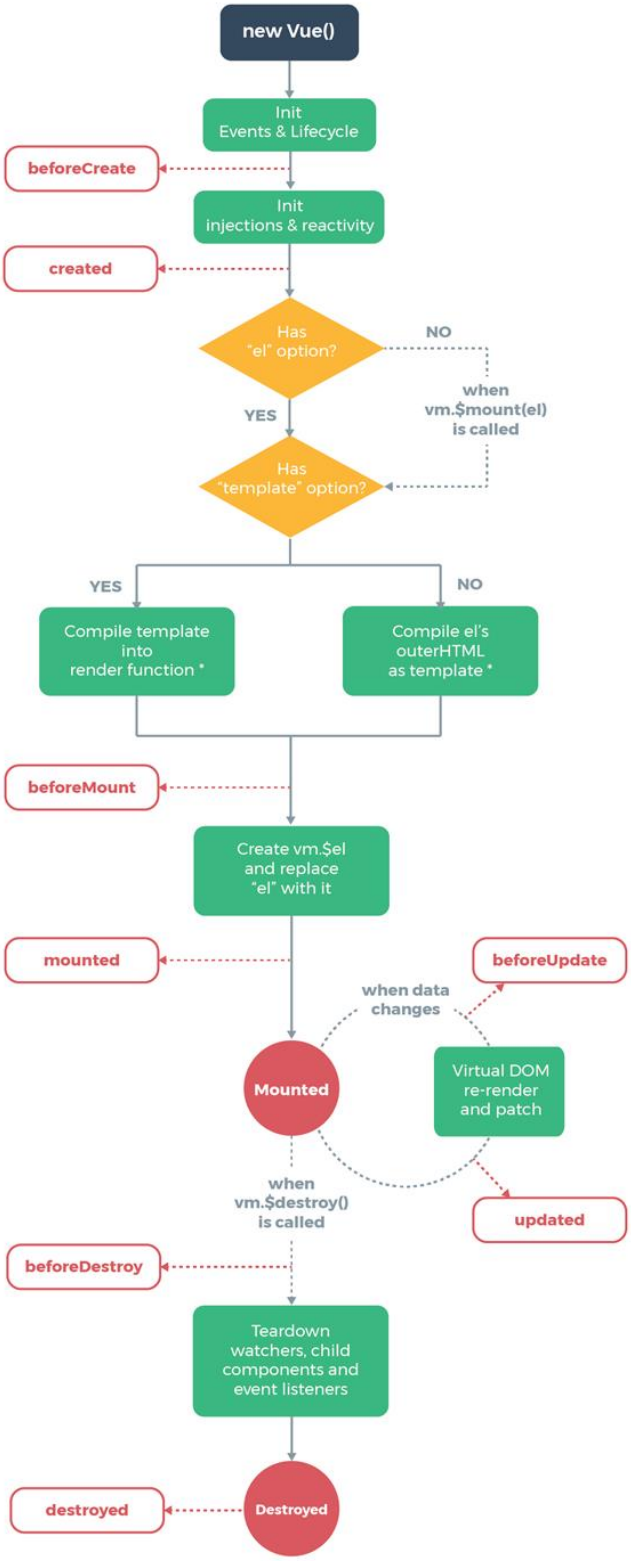
```
new Vue({
  data: {
    a: 1
  },
  created: function () {
    // `this` points to the vm instance
    console.log('a is: ' + this.a)
  }
})
// => "a is: 1"
```

También hay otros llamados que serán ejecutados en diferentes etapas del ciclo de vida de la instancia, como `mounted`, `updated`, y `destroyed`. Todos los hooks del ciclo de vida se llaman con su contexto `this` apuntando a la instancia de Vue que lo invoca.

> No usés **funciones de flecha** en una propiedad de opciones o devolución de llamada, como `created: () => console.log(this.a)` o `vm.$watch('a', newValue => this.myMethod())`. Dado que una función de flecha no tiene un `this`, `this` se tratará como cualquier otra variable y se buscará léxicamente a través de los ámbitos principales hasta que se encuentre, lo que a menudo genera errores como `Uncaught TypeError: Cannot read property of undefinedo Uncaught TypeError: this.myMethod is not a function`.

Diagrama de ciclo de vida

A continuación, se muestra un diagrama del ciclo de vida de la instancia. No necesitás comprender completamente todo lo que está sucediendo en este momento, pero a medida que aprendas y construyas más, será una referencia útil.



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

Sintaxis de la plantilla

Vue.js utiliza una sintaxis de plantilla basada en HTML que le permite vincular declarativamente el DOM renderizado a los datos de la instancia de Vue subyacente. Todas las plantillas de Vue.js son HTML válidas que pueden ser analizadas por navegadores compatibles con las especificaciones y analizadores de HTML.

Bajo las capas, Vue compila las plantillas en funciones de renderizado de DOM virtual. Combinado con el sistema de reactividad, Vue es capaz de determinar de manera inteligente la cantidad mínima de componentes para volver a renderizar y aplicar la cantidad mínima de manipulaciones DOM cuando cambia el estado de la aplicación.

Si está familiarizado con los conceptos de Virtual DOM y prefiere la potencia bruta de JavaScript, también puede **escribir directamente funciones de renderizado en** lugar de plantillas, con soporte JSX opcional.

Interpolaciones

#Texto

La forma más básica de vinculación de datos es la interpolación de texto utilizando la sintaxis "Moustache" (llaves dobles):

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

La etiqueta del bigote se reemplazará con el valor de lapropiedad `msg` en el objeto de datos correspondiente. También se actualizará siempre que cambie la propiedad del objeto de datos `msg`.

También puede realizar interpolaciones de una sola vez que no se actualizan con el cambio de datos mediante el uso de la **directiva v-once** , pero tenga en cuenta que esto también afectará a cualquier otro enlace en el mismo nodo:

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

#HTML sin procesar

El doble bigote interpreta los datos como texto sin formato, no HTML. Para generar HTML real, deberás utilizar la **directiva v-html** :

```
<p>Using mustaches: {{ rawHtml }}</p>
<p>Using v-html directive: <span v-html="rawHtml">

</span></p>Usar bigotes: <span style = "color: red"> Debe ser rojo. </span>
```

Usando la directiva v-html: Esto debería ser rojo.
El contenido de `span` se reemplazará con el valor de la propiedad `rawHtml`, interpretado como HTML simple; los enlaces de datos se ignoran. Tené en cuenta que no podés usar `v-html` para componer parciales de plantilla, porque Vue no es un motor de plantillas basado en cadenas. En cambio, prefiere los componentes como unidad fundamental para la reutilización y composición de la interfaz de usuario.

> Representar dinámicamente HTML arbitrario en su sitio web puede ser muy peligroso porque puede conducir fácilmente a **vulnerabilidades XSS** . Utilizá únicamente la interpolación HTML en contenido confiable y **nunca** en contenido proporcionado por el usuario.

#Atributos

Los bigotes {{ }} no se pueden usar dentro de los atributos HTML. En su lugar, usá una **directiva v-bind** :

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

En el caso de los atributos booleanos, donde su mera existencia implica `true`, `v-bind`funciona de forma un poco diferente. En este ejemplo:

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

Si `isButtonDisabled` tiene el valor de `null`, `undefined` o `false`, el atributo `disabled` ni siquiera se incluirá en el `<button>`elemento renderizado .

#Usar expresiones de JavaScript

Hasta ahora solo hemos vinculado claves de propiedad simples en nuestras plantillas. Pero Vue.js en realidad admite todo el poder de las expresiones de JavaScript dentro de todos los enlaces de datos:

```
{{ number + 1 }}

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

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

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

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

```
<!--esto es una declaración, no una expresión: -->
```

```
{{ var a = 1 }}

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

> Las expresiones de plantilla están en un espacio aislado y solo tienen acceso a una **lista blanca de globales** como `Math`y `Date`. No debes intentar acceder a globales definidos por el usuario en expresiones de plantilla.

Directivas

Las directivas son atributos especiales con el prefijo `v-`. Se espera que los valores de los atributos de la directiva sean **una sola expresión de JavaScript** (con la excepción de `v-for`). El trabajo de una directiva es aplicar de forma reactiva efectos secundarios al DOM cuando cambia el valor de su expresión. Repasemos el ejemplo que vimos en la primer parte:

```
<p v-if="seen">Ahora podés verme</p>
```

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

Argumentos

Algunas directivas pueden tomar un "argumento", indicado por dos puntos después del nombre de la directiva. Por ejemplo, la directiva `v-bind` se utiliza para actualizar de forma reactiva un atributo HTML:

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

Vemos como el argumento `href`, que le dice a la directiva `v-bind` que vincule el atributo `href` del elemento al valor de la expresión `url`.

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

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

Aquí el argumento es el nombre del evento a escuchar.

Argumentos dinámicos

A partir de la versión 2.6.0, también es posible utilizar una expresión de JavaScript en un argumento de directiva envolviéndola entre corchetes:

```
<!--
Tené en cuenta que existen algunas restricciones para la expresión del argumento.
-->
<a v-bind:[attributeName]="url"> ... </a>
```

Aquí `attributeName` se evaluará dinámicamente como una expresión de JavaScript y su valor evaluado se utilizará como el valor final del argumento. Por ejemplo, si su instancia de Vue tiene una propiedad de datos `attributeName`, cuyo valor es `"href"`, entonces este enlace será equivalente a `v-bind:href`.

De manera similar, puede usar argumentos dinámicos para vincular un controlador a un nombre de evento dinámico:

```
<a v-on:[eventName]="doSomething"> ... </a>
```

En este ejemplo, cuando eventName tenga el valor de "focus", v-on:[eventName] será equivalente a v-on:focus.

Restricciones de valor de argumento dinámico

Se espera que los argumentos dinámicos se evalúen en una cadena, con la excepción de null. El valor especial null se puede utilizar para eliminar explícitamente el enlace. Cualquier otro valor que no sea una cadena activará una advertencia.

Restricciones de expresión de argumentos dinámicos

Las expresiones de argumentos dinámicos tienen algunas restricciones de sintaxis porque ciertos caracteres, como espacios y comillas, no son válidos dentro de los nombres de atributos HTML. Por ejemplo, lo siguiente no es válido:

```
<!-- Esto activará una advertencia del compilador. -->
<a v-bind:['foo' + bar]="value"> ... </a>
```

La solución es utilizar expresiones sin espacios ni comillas, o reemplazar la expresión compleja con una propiedad calculada.

Cuando utilices plantillas en DOM (es decir, plantillas escritas directamente en un archivo HTML), también debes evitar nombrar claves con caracteres en mayúscula, ya que los navegadores convertirán los nombres de los atributos en minúsculas:

```
<!--
Esto se convertirá a v-bind: [someattr] en plantillas en DOM.
A menos que tenga una propiedad "someattr" en su instancia, su código no funcionará.
-->
<a v-bind:[someAttr]="value"> ... </a>
```

Modificadores

Los modificadores son sufijos especiales indicados por un punto, que indican que una directiva debe estar vinculada de alguna manera especial. Por ejemplo, el modificador .prevent le dice a la directiva v-on que llame event.preventDefault() al evento activado:

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

Shorthands

El prefijo v-sirve como una pista visual para identificar atributos específicos de Vue en sus plantillas. Esto es útil cuando usa Vue.js para aplicar un comportamiento dinámico a algún marcado existente, pero puede parecer detallado para algunas directivas de uso frecuente. Al mismo tiempo,

la necesidad del prefijo v-se vuelve menos importante cuando está construyendo un **SPA** , donde Vue administra cada plantilla. Por lo tanto, Vue proporciona abreviaturas especiales para dos de las directivas más utilizadas v-bindy v-on:

v-bind Taquigrafía

```
<!-- full syntax -->
<a v-bind:href="url"> ... </a>

<!-- camino abreviado -->
<a :href="url"> ... </a>

<!-- camino abreviado  con argumento dinámico (2.6.0+) -->
<a :[key]="url"> ... </a>
```

v-on Taquigrafía

```
<!-- full syntax -->
<a v-on:click="doSomething"> ... </a>

<!-- camino abreviado  -->
<a @click="doSomething"> ... </a>

<!-- camino abreviado  con argumento dinámico (2.6.0+) -->
<a @[event]="doSomething"> ... </a>
```

Pueden parecer un poco diferente de HTML normal, pero :y @son caracteres válidos para nombres de atributos y todos los navegadores compatibles-Vue puede analizar correctamente. Además, no aparecen en el marcado renderizado final. La sintaxis abreviada es totalmente opcional, pero probablemente la apreciará cuando aprenda más sobre su uso más adelante.

Representación condicional

v-if

La directiva v-ifse usa para renderizar condicionalmente un bloque. El bloque solo se representará si la expresión de la directiva devuelve un valor verdadero.

```
<h1 v-if="awesome">Vue is awesome!</h1>
```

También es posible agregar un "bloque else" con v-else:

```
<h1 v-if="awesome">Vue is awesome!</h1>
<h1 v-else>Oh no !</h1>
```

Grupos condicionales con v-if activado<template>

Debido a que v-if es una directiva, debe estar unida a un solo elemento. Pero, ¿y si queremos alternar más de un elemento? En este caso podemos usar v-if en un elemento <template>, que sirve como envoltorio invisible. El resultado final renderizado no incluirá el elemento <template>.

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

v-else

Podés usar la directiva v-else para indicar un "bloque else" para v-if:

```
<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 seguir inmediatamente a v-if un v-else-if elemento o un elemento; de lo contrario, no se reconocerá.

v-else-if

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

```
<div v-if="type === 'A'">
  A
</div>
```

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

Similar a `v-else`, un elemento `v-else-if` debe seguir inmediatamente a elemento `v-if`un o `v-else-if` .

Controlar elementos reutilizables con `key`

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

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

Luego, cambiar el código `loginType` anterior no borrará lo que el usuario ya haya ingresado. Dado que ambas plantillas usan los mismos elementos, `<input>`no se reemplaza, solo su `placeholder`.

Sin embargo, esto no siempre es deseable, por lo que Vue ofrece una forma de decir: "Estos dos elementos están completamente separados, no los reutilices". Agregue un atributo `key` con valores únicos:

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

Ahora esas entradas se procesarán desde cero cada vez que cambie.

Tené en cuenta que los elementos `<label>`aún se reutilizan de manera eficiente, porque no tienen atributos `key`.

v-show

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

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

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

> Tenga en cuenta que v-show no es compatible con el elemento <template> ni funciona con v-else.

v-if vs v-show

v-if es una representación condicional "real" porque garantiza que los detectores de eventos y los componentes secundarios dentro del bloque condicional se destruyan y se vuelvan a crear correctamente durante los conmutadores.

V-if con una condición es falsa en el renderizado inicial, no hará nada; el bloque condicional no se renderizará hasta que la condición sea verdadera por primera vez.

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

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

v-if con v-for

> **No se recomienda** usar v-if y v-for juntos .

Cuando se usa junto con v-if, v-for tiene una prioridad más alta que v-if.

Representación de listas

Asignación de una matriz a elementos con v-for

Podemos usar la directiva v-for para representar una lista de elementos basada en una matriz. La directiva v-for requiere una sintaxis especial en la forma de item in items, donde items es la matriz de datos de origen y item es un **alias** para el elemento de matriz en el que se itera:

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

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

Resultado:

-Foo
-Bar

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

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

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

Resultado:

```
-Padre - 0 – Foo
-Padre - 1 - Barra
```

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

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

v-for con un objeto

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

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

new Vue({
  el: '#v-for-object',
  data: {
    object: {
      title: 'Cómo hacer listas en Vue',
      author: 'fulano',
      publishedAt: '2016-04-10'
    }
  }
})
```

Resultado:

```
-Cómo hacer listas en Vue
-fulano
-2016-04-10
```

También se puede proporcionar un segundo argumento para el nombre de la propiedad (también conocido como clave):

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

```
</div>
```

-Cómo hacer listas en Vue
-fulano
-2016-04-10

Y otro para el índice:

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

-Cómo hacer listas en Vue
-fulano
-2016-04-10

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

Estado de mantenimiento

Cuando Vue actualiza una lista de elementos renderizados con v-for, de forma predeterminada utiliza una estrategia de "parche en el lugar". Si el orden de los elementos de datos ha cambiado, en lugar de mover los elementos DOM para que coincidan con el orden de los elementos, Vue parcheará cada elemento en su lugar y se asegurará de que refleje lo que se debe representar en ese índice en particular. Esto es similar al comportamiento de track-by="$index"en Vue 1.x.

Este modo predeterminado es eficiente, pero **solo es adecuado cuando la salida de renderizado de su lista no depende del estado del componente hijo o del estado temporal del DOM (por ejemplo, valores de entrada de formulario)** .

Para darle a Vue una pista para que pueda rastrear la identidad de cada nodo y, por lo tanto, reutilizar y reordenar los elementos existentes, debe proporcionar un atributo key único para cada elemento:

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

Se recomienda proporcionar un atributo `key` `v-for` siempre que sea posible, a menos que el contenido del DOM iterado sea simple o que confíe intencionalmente en el comportamiento predeterminado para mejorar el rendimiento.

Dado que es un mecanismo genérico para que Vue identifique nodos, `key` también tiene otros usos que no están específicamente vinculados `v-for`.

No utilicse valores no primitivos como objetos y matrices como claves `v-for`. En su lugar, utilizá valores numéricos o de cadena.

Detección de cambio de matriz
Métodos de mutación

Vue envuelve los métodos de mutación de una matriz observada para que también activen actualizaciones de vista. Los métodos envueltos son:

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

Podés abrir la consola y jugar con la matriz `items` de los ejemplos anteriores llamando a sus métodos de mutación. Por ejemplo: `example1.items.push({ message: 'Juan' })`.

Reemplazo de una matriz

Los métodos de mutación, como su nombre indica, mutan la matriz original a la que se llaman. En comparación, también existen métodos no mutantes, por ejemplo `filter()`, `concat()`y `slice()`, que no mutan la matriz original pero **siempre devuelven una nueva matriz** . Cuando trabaje con métodos que no sean mutantes, podés reemplazar la matriz anterior por la nueva:

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

Es posible pensar que esto hará que Vue descarte el DOM existente y vuelva a representar la lista completa; afortunadamente, ese no es el caso. Vue implementa algunas heurísticas inteligentes para maximizar la reutilización de elementos DOM, por lo que reemplazar una matriz con otra matriz que contenga objetos superpuestos es una operación muy eficiente.

Advertencias

Debido a las limitaciones de JavaScript, existen tipos de cambios que Vue **no puede detectar** con matrices y objetos.

Visualización de resultados filtrados / ordenados

A veces queremos mostrar una versión filtrada u ordenada de una matriz sin mutar o restablecer los datos originales. En este caso, puede crear una propiedad calculada que devuelva la matriz filtrada u ordenada.

Por ejemplo:

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

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

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

```
<ul v-for="set in sets">
  <li v-for="n in even(set)">{{ n }}</li>
</ul>

data: {
  sets: [[ 1, 2, 3, 4, 5 ], [6, 7, 8, 9, 10]]
},
methods: {
  even: function (numbers) {
    return numbers.filter(function (number) {
      return number % 2 === 0
    })
  }
}
```

v-for con una gama

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

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

Resultado:

1 2 3 4 5 6 7 8 9 10

v-for en un <template>

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

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

v-for con v-if

> Tené en cuenta que **no se** recomienda usar v-if y v-for juntos.

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

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

Lo anterior solo muestra los todos que no están completos.

Si, en cambio, tu intención es omitir condicionalmente la ejecución del bucle, podés colocar el v-if en un elemento contenedor (o **<template>**). Por ejemplo:

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

```
<p v-else>No todos left!</p>
```

Vinculaciones de clase y estilo

Una necesidad común para el enlace de datos es manipular la lista de clases de un elemento y sus estilos en línea. Dado que ambos son atributos, podemos usarlos `v-bind` para manejarlos: solo necesitamos calcular una cadena final con nuestras expresiones. Sin embargo, entrometerse con la concatenación de cadenas es molesto y propenso a errores. Por esta razón, Vue proporciona mejoras especiales cuando se usa `v-bind` con `class` y `style`. Además de las cadenas, las expresiones también pueden evaluar objetos o matrices.

Vinculación de clases HTML
Sintaxis del objeto

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

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

La sintaxis anterior significa que la presencia de la clase `active` estará determinada por la **veracidad** de la propiedad de los datos `isActive`.

Podés alternar varias clases si tiene más campos en el objeto. Además, la directiva `v-bind:class` también puede coexistir con el atributo `class` simple . Así que dada la siguiente plantilla:

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

Y los siguientes datos:

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

Rendirá:

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

Cuando `isActive` o `hasError` cambie, la lista de clases se actualizará en consecuencia. Por ejemplo, si `hasError` se convierte en `true`, la lista de clases se convertirá en `"static active text-danger"`.

El objeto vinculado no tiene que estar en línea:

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

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

Esto producirá el mismo resultado. También podemos vincularnos a una **propiedad calculada** que devuelve un objeto. Este es un patrón común y poderoso:

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

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

Sintaxis de matriz

Podemos pasar una matriz `v-bind:class` para aplicar una lista de clases:

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

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

Que rendirá:

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

Si también deseas alternar una clase en la lista condicionalmente, podés hacerlo con una expresión ternaria:

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

Esto siempre se aplicará `errorClass`, pero solo se aplicará `activeClass` cuando `isActive` sea veraz.

Sin embargo, esto puede ser un poco detallado si tiene varias clases condicionales. Es por eso que también es posible usar la sintaxis de objeto dentro de la sintaxis de matriz:

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

Consumo de un Web Service

Tras haber realizado una primera introducción a los conceptos básicos de Vue.js, vamos a continuar presentando otras funciones y tareas que se pueden llevar a cabo utilizando este framework.

En este caso veremos cómo se pude invocar de una forma muy sencilla a servicios web y APIs de terceros.

Uno de los aspectos clave en los desarrollos web actuales, además de los aspectos estéticos o de la usabilidad, es que la web en cuestión contemple algún mecanismo de integración. Como desarrolladores habrá ocasiones en las que tendremos que crear un producto capaz de acceder a los contenidos ofrecidos por otros sitios, o que sea nuestro producto el que pueda ofrecer contenidos a terceros.

En este caso vamos a ocuparnos de consumir servicios web. Es decir, vamos a crear un componente Vue.js que pueda acceder a un sitio remoto para solicitar y obtener información del mismo, para posteriormente mostrar dicha información al usuario. A día de hoy, el mecanismo estándar para realizar solicitudes a sitios de terceros se basa en la utilización de APIs de tipo RESTful.

En el ejemplo que presentamos, vamos a utilizar como proveedor del servicio remoto el sitio jsonplaceholder, que ofrece a los desarrolladores la posibilidad de realizar llamadas a un servicio web (API) con diferentes parámetros para obtener distintos tipos de resultados en forma totalmente gratuita, precisamente para que estos desarrolladores puedan realizar todo tipo de pruebas. En nuestro caso, el componente que vamos a desarrollar solicitará a este sitio remoto una lista de usuarios.

Para llevar a cabo esta invocación al proveedor de datos remotos, se va a realizar una solicitud mediante la librería Axios. Se trata de la librería más utilizada en javascript para actuar como cliente que invoca servicios web remotos. Por lo tanto, en nuestro código de ejemplo, en la sección inicial en la que se cargan las librerías externas, en esta ocasión además de cargar la librería javascript correspondiente a Vue.js, cargaremos también la librería correspondiente a Axios.
Finalmente, éste es el código de la aplicación resultante:

```
<html>
<head>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/vue/2.6.10/vue.min.js"></script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/axios/0.19.0/axios.min.js"></script>
</head>

<body>
<div id="vueapp">
  <remote-users></remote-users>
</div>

<script>
const REMOTE_WS = axios.create({
  baseURL: 'https://jsonplaceholder.typicode.com/'
})

Vue.component('RemoteUsers', {
  template: `
  <ul>
```

```
    <li v-for="user in users" :key="user.id">{{ user.name }}</li>
  </ul>
  `,
  data: function() {
    return {
      users: []
    }
  },
  created() {
    console.log("Getting remote users")
    this.getRemoteUsers()
  },
  methods: {
    getRemoteUsers() {
      REMOTE_WS.get('users/')
        .then(response => {
          this.users = response.data
          // console.log(response.data)
        })
        .catch(err => {
          console.error('err')
        })
    }
  }
})

const vueApp = new Vue({
  el: '#vueapp'
})
</script>
</body>
</html>
```

Comentamos a continuación algunos detalles y aspectos a destacar en el código de la aplicación. En primer lugar, se instancia un cliente axios en el que se define la URL base del servicio web remoto. Dicho cliente será el que se utilice en las posteriores invocaciones (en este caso se realiza una única invocación). En la aplicación se referencia al cliente a través de REMOTE_WS:

```
const REMOTE_WS = axios.create({
  baseURL: 'https://jsonplaceholder.typicode.com/'
})
```

A continuación, en la parte de datos de nuestro componente RemoteUsers se define la variable users, que será de tipo vector y almacenará la lista de usuarios obtenidos al realizar la consulta al servicio web. Inicialmente se le asigna como valor un vector vacío.

```
data: function() {
  return {
    users: []
  }
}
```

Por otra parte, la plantilla para mostrar el contenido deberá de recorrer el vector de usuarios y mostrar los detalles de cada uno de los usuarios (en este caso se muestran los atributos username y name). Para poder iterar en el conjunto resultante, se utiliza la expresión v-for, que nos permite recorrer los elementos tanto de vectores como de objetos JSON. Para cada uno de los elementos existentes en el vector users se creará un elemento de tipo <li> y en dicho elemento se mostrarán los campos username y name de cada uno de los usuarios. Con respecto al campo :key, en este caso no es estrictamente necesario, pero es una buena práctica utilizar una clave que pueda identificar de forma única a cada ítem al iterar mediante v-for

```
<ul>
  <li v-for="user in users" :key="user.id">
    {{ user.username }}: {{ user.name }}
  </li>
</ul>
```

Una novedad en este ejemplo es la utilización de la función created()
Tanto la aplicación principal como cada componente en Vue.js tienen un ciclo de vida en el que se invocan unos métodos predefinidos en determinadas etapas del componente. Podemos aprovechar estas llamadas a estos métodos y crear, como en este caso, funciones que realicen una tarea de inicialización en el momento en que el componente ha sido creado.

```
created() {
  console.log("Getting remote users")
  this.getRemoteUsers()
}
```

En nuestro caso, la tarea inicial que se ejecuta una vez que el componente ha sido creado, es la solicitud del servicio web remoto mediante el cliente axios anteriormente definido. Se solicita al servicio mediante una llamada de tipo GET que nos entregue la lista de usuarios. Si la operación se realiza de forma satisfactoria, la lista resultante se almacenará en la variable users de nuestro componente. En caso contrario, se mostrará por consola la información relacionada con el error que pudiera haberse producido.

```
getRemoteUsers() {
  REMOTE_WS.get('users/')
    .then(response => {
      this.users = response.data
      // console.log(response.data)
    })
    .catch(err => {
      console.error(err)
    })
}
```