Vue. Contenidos Generales:

Vue.js: ¿Qué es? Componentes Reutilizables. Instalación. Modificación del DOM, eventos, esquema de componentes.

Aplicaciones Reactivas. Enviar y pedir datos a un servidor. SPA. Consumo de Servicios con JavaScript/Vue/Axios/Fetch. Ejemplo SPA con Vue.

Introducción a Vue

Vue (pronunciado */vjuː/*, como view) es un framework para construir interfaces de usuario. A diferencia de otros frameworks, Vue está diseñado desde cero para ser utilizado incrementalmente. La librería central está enfocada solo en la capa de visualización, y es fácil de utilizar e integrar con otras librerías o proyectos existentes. Por otro lado, Vue también es perfectamente capaz de impulsar sofisticadas Single-Page Applications cuando se utiliza en combinación con herramientas modernas y librerías de apoyo.

¿Qué es Vue.js?

Vue.js es un framework JavaScript progresivo de código abierto que se utiliza para desarrollar interfaces web interactivas. Es uno de los marcos famosos que se utilizan para simplificar el desarrollo web. Vue.JS se centra en la capa de vista. Se puede integrar fácilmente en grandes proyectos para el desarrollo front-end sin ningún problema.

La instalación de Vue.JS es muy fácil. Cualquier desarrollador puede comprender y crear interfaces web interactivas fácilmente en cuestión de tiempo. Vue.JS fue creado por Evan You, un ex empleado de Google. La primera versión de Vue.JS se lanzó en febrero de 2014.

Virtual DOM

Vue.JS hace uso del DOM virtual, que también es utilizado por otros frameworks como React, Ember, etc. Los cambios no se realizan en el DOM, sino que se crea una réplica del DOM que está presente en forma de estructuras de datos JavaScript. . Siempre que se deben realizar cambios, se realizan en las estructuras de datos de JavaScript y esta última se compara con la estructura de datos original. Luego, los cambios finales se actualizan al DOM real, que el usuario verá cambiar. Esto es bueno en términos de optimización, es menos costoso y los cambios se pueden realizar a un ritmo más rápido.

Data Binding

La función de data binding ayuda a manipular o asignar valores a atributos HTML, cambiar el estilo, asignar clases con la ayuda de la directiva de enlace llamada **v-bind** disponible en Vue.JS.

Components

Los componentes son una de las características importantes de Vue.JS que ayudan a crear elementos personalizados, que se pueden reutilizar en HTML.

Event Handling

v-on es el atributo agregado a los elementos DOM para escuchar los eventos en Vue.JS.

Computed Properties

Esta es una de las características más importantes de Vue.JS. Ayuda a escuchar los cambios realizados en los elementos de la interfaz de usuario y realiza los cálculos necesarios. No hay necesidad de codificación adicional para este fin.

Templates

Vue.JS proporciona plantillas basadas en HTML que unen el DOM con los datos de la instancia de Vue. Vue compila las plantillas en funciones virtuales DOM Render. Podemos hacer uso de la plantilla de las funciones de render y para hacerlo tenemos que reemplazar la plantilla con la función de render.

Directives

Vue.JS tiene directivas integradas como **v-if, v-else, v-show, v-on, v-bind** y **v-model**, que se utilizan para realizar varias acciones en la interfaz.

Watchers

Los Watchers se aplican a los datos que cambian. Por ejemplo, elementos de input en formularios. Aquí, no tenemos que agregar ningún evento adicional. Los Watchers se encargan de manejar cualquier cambio de datos haciendo que el código sea simple y rápido.

Liviano

Vue.JS es muy ligero y el rendimiento también es muy rápido.

Hay muchas formas de instalar Vue.JS. Algunas de las formas de cómo realizar la instalación se comentan a continuación.

Fuente: tutorialspoint.com / platzi.com

Instalación y uso de Vue

La forma más fácil de comenzar a usar Vue.js es crear un archivo index.html e incluir Vue con:
<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>

La página de instalación https://es.vuejs.org/v2/guide/installation.html proporciona más opciones de instalación de Vue.

Usando la etiqueta <script> en el documento HTML.

Descargar desde https://vuejs.org/v2/guide/installation.html la versión necesaria: La versiónde desarrollo no está minificada, mientras que la versión de producción sí.

```
<html>
  <head>
      <script type = "text/javascript" src = "vue.min.js"></script>
  </head>
  <body></body>
</html>
```

Usando CDN

```
<!-- development version, includes helpful console warnings -->
<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
```

or:

```
<!-- production version, optimized for size and speed -->
<script src="https://cdn.jsdelivr.net/npm/vue"></script>
```

Fuente: tutorialspoint.com / platzi.com

Modificación del DOM

Ejemplo

```
<html>
  <head>
      <title>VueJs Introduction</title>
      <script type = "text/javascript" src = "js/vue.js"></script>
  </head>
  <body>
      <div id = "intro">
        <h1>{{ message }}</h1>
      </div>
      <script type = "text/javascript">
        const vue_det = new Vue({
           el: '#intro',
           data: {
              message: 'My first VueJS Task'
           }
        });
      </script>
  </body>
</html>
```

Output

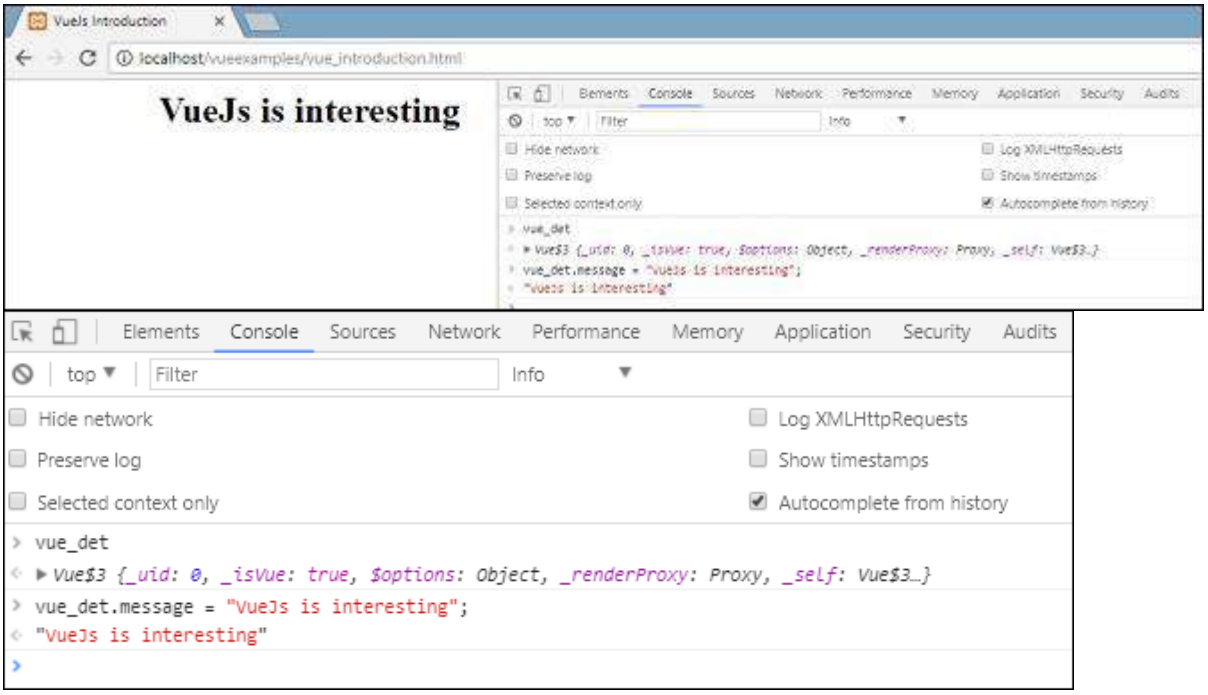
Para este ejemplo se incluyó el import del archivo js de Vue.JS dentro de la etiqueta <head>.El div presente tiene un mensaje en una interpolación {{}}. Esto interactúa con Vue.JS y muestra los datos en el navegador. Para mostrar el valor del mensaje en el DOM, se crea una instancia de Vue.JS.

```
<div id = "intro">
  <h1>{{ message }}</h1>
</div>

const vue_det = new Vue({
  el: '#intro',
  data: {
      message: 'My first VueJS Task'
  }
```

});

Se llama a la instancia de Vue, la cual toma el ID del elemento del DOM, por ejemplo '#intro'.Hay datos dentro de dicha instancia con el mensaje al cual se le asigna el valor 'My first VueJS Task'. Vue.JS interactúa con el DOM y cambia el valor en el DOM {{message}} con el mensaje 'My first VueJS Task'.

También es posible cambiar el valor del mensaje en consola:



Detalles en consola

Sintaxis

```
const app = new Vue({
  // options
})
```

```
<html>
  <head>
      <title>VueJs Instance</title>
      <script type = "text/javascript" src = "js/vue.js"></script>
  </head>
  <body>
      <div id = "vue_det">
        <h1>Firstname : {{firstname}}</h1>
        <h1>Lastname : {{lastname}}</h1>
        <h1>{{mydetails()}}</h1>
      </div>
      <script type = "text/javascript" src = "js/vue_instance.js"></script>
  </body>
```

```
</html>
```

vue_instance.js

```
const  vm = new Vue({
  el: '#vue_det',
  data: {
      firstname : "Ria",
      lastname  : "Singh",
      address    : "Mumbai"
  },
  methods: {
      mydetails : function() {
        return "I am "+this.firstname +" "+ this.lastname;
      }
  }
});
```

Hay un parámetro llamado **el**. Toma el id del elemento a modificar en el DOM.

```
<div id = "vue_det"></div>
```

Lo que escribamos dentro de los {{}} afectará únicamente al div seleccionado y a nada másfuera de él. Posteriormente se definen los datos, nombre, apellido y dirección.

```
<div id = "vue_det">
  <h1>Firstname : {{firstname}}</h1>
  <h1>Lastname : {{lastname}}</h1>
  <h1>Address : {{address}}</h1>
</div>
```

El valor del nombre se reemplazará dentro de la primera interpolación y así sucesivamentecon los demás parámetros.

Posteriormente se define una función llamada mydetails() que devuelve un valor que seráasignado a la interpolación siguiente:

```
<h1>{{mydetails()}}</h1>
```

Output

Podemos también mostrar en el DOM un template HTML generado dentro de una instanciade Vue.JS

```
<html>
  <head>
      <title>VueJs Instance</title>
      <script type = "text/javascript" src = "js/vue.js"></script>
  </head>
  <body>
      <div id = "vue_det">
        <h1>Firstname : {{firstname}}</h1>
        <h1>Lastname : {{lastname}}</h1>
        <div>{{htmlcontent}}</div>
      </div>
      <script type = "text/javascript" src = "js/vue_template.js"></script>
  </body>
</html>
```

vue_template.js

```
const vm = new Vue({
  el: '#vue_det',
  data: {
      firstname : "Ria",
      lastname  : "Singh",
      htmlcontent : "<div><h1>Vue Js Template</h1></div>"
  }
});
```

Esto no es correcto, dado que el resultado en el navegador sería el siguiente:

Si necesitamos insertar código HTML en el dom desde Vue.JS se utiliza la directiva **v-html**. Desde el momento que se usa Vue.JS ya sabe que tiene que mostrar dicho contenido comoHTML en el navegador.

```
<html>
  <head>
      <title>VueJs Instance</title>
      <script type = "text/javascript" src = "js/vue.js"></script>
  </head>
  <body>
      <div id = "vue_det">
        <h1>Firstname : {{firstname}}</h1>
        <h1>Lastname : {{lastname}}</h1>
        <div v-html = "htmlcontent"></div>
      </div>
      <script type = "text/javascript" src = "js/vue_template.js"></script>
  </body>
</html>
```

```
const vm = new Vue({
  el: '#vue_det',
  data: {
      firstname : "Ria",
      lastname  : "Singh",
      htmlcontent : "<div><h1>Vue Js Template</h1></div>"
  }
```

Para añadir **atributos** a los elementos del DOM utilizamos la directiva v-bind, que en estecaso se orienta al atributo src.

```
<html>
  <head>
      <title>VueJs Instance</title>
      <script type = "text/javascript" src = "js/vue.js"></script>
  </head>
  <body>
      <div id = "vue_det">
         <h1>Firstname : {{firstname}}</h1>
         <h1>Lastname : {{lastname}}</h1>
         <div v-html = "htmlcontent"></div>
         <img v-bind:src = "imgsrc" width = "300" height = "250" />
      </div>
      <script type = "text/javascript" src = "js/vue_template1.js"></script>
  </body>
</html>
```

En la instancia Vue se agrega en el apartado **data** la siguiente líneaimgsrc:

"img/paisaje.jpg"

v-once

Todo elemento que posea la directiva **v-once** será renderizado sólo una vez.

<span v-once>Mensaje: {{ msg }}

Componentes

```
<html>
  <head>
      <title>VueJs Instance</title>
      <script type = "text/javascript" src = "js/vue.js"></script>
  </head>
  <body>
      <div id = "component_test">
        <testcomponent></testcomponent>
      </div>
      <div id = "component_test1">
        <testcomponent></testcomponent>
      </div>
      <script type = "text/javascript" src = "js/vue_component.js"></script>
  </body>
</html>
```

vue_component.js

```
Vue.component('testcomponent',{
  template : '<div><h1>This is coming from component</h1></div>'
});
const vm = new Vue({
  el: '#component_test'
});
const vm1 = new Vue({
  el: '#component_test1'
});
```

Se crean dos div con id component_test y component_test1, junto con dos instancias Vue que hacen referencia a dichos ids. Se crea un componente en común que será asignado a ambas instancias.

Para crear un componente se utiliza la siguiente sintaxis:

```
Vue.component('nombredelcomponente',{ // options});
```

```
<div id = "component_test">
  <testcomponent></testcomponent>
</div>
<div id = "component_test1">
  <testcomponent></testcomponent>
</div>
```

Dentro del componente se agrega un template el cual tiene asignado código HTML. Esta esla manera de registrar globalmente un componente en Vue.JS, que puede ser reutilizado encualquier instancia Vue.

```
Vue.component('testcomponent',{
  template : '<div><h1>This is coming from component</h1></div>'
});
```

El nombre personalizado de las etiquetas será reemplazado por el código del template en escrito en la creación del componente. En el navegador no existe ningún componente conetiquetas <testcomponent></testcomponent>.

También se puede modificar el comportamiento de un componente global dentro de unainstancia Vue.

```
const vm = new Vue({
  el: '#component_test',
  components:{
      'testcomponent': {
          template : '<div><h1>This is coming from component</h1></div>'
      }
  }
});
```

Example

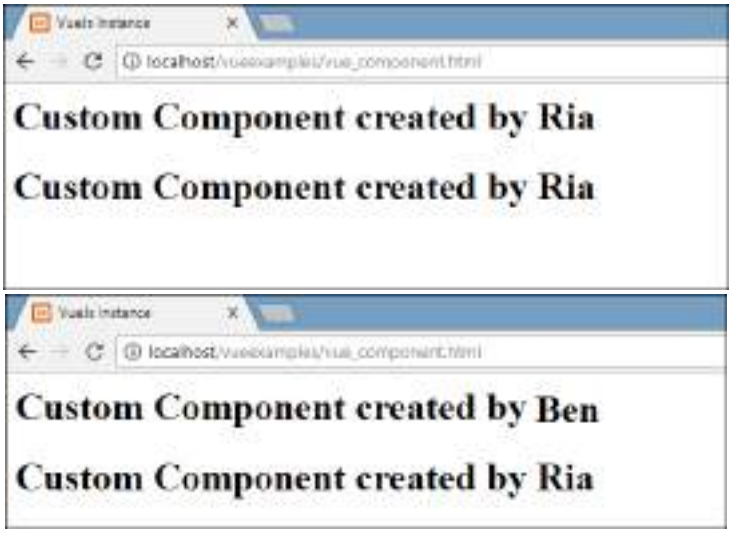
```
<html>
   <head>
      <title>VueJs Instance</title>
      <script type = "text/javascript" src = "js/vue.js"></script>
   </head>
   <body>
      <div id = "component_test">
         <testcomponent></testcomponent>
      </div>
      <div id = "component_test1">
         <testcomponent></testcomponent>
      </div>
      <script type = "text/javascript" src = "js/vue_component.js"></script>
   </body>
</html>
```

vue_component.js

```
Vue.component('testcomponent',{
   template : '<div v-on:mouseover = "changename()" v-on:mouseout =
"originalname();"><h1>Custom Component created by <span id =
"name">{{name}}</span></h1></div>',
   data: function() {
      return {
         name : "Ria"
      }
   },
   methods:{
      changename : function() {
         this.name = "Ben";
      },
      originalname: function() {
         this.name = "Ria";
      }
   }
});
const vm = new Vue({
   el: '#component_test'
});
const vm1 = new Vue({
   el: '#component_test1'
```

```
});
```

En el ejemplo anterior, el template dentro del componente hace que cuando el mouse se posicione encima del mismo ejecute la función changename() y cuando el mismo salga del componente se ejecute la función originalname(). Ambas funciones trabajan sobre la propiedad name del componente, cambiandola.



Componentes dinámicos

Los componentes dinámicos se crean con la etiqueta <component></component> y se ligancon un elemento del DOM de la siguiente manera.

```
<html>
  <head>
      <title>VueJs Instance</title>
      <script type = "text/javascript" src = "js/vue.js"></script>
  </head>
  <body>

      <div id = "databinding">
        <component v-bind:is = "view"></component>
      </div>

<script type = "text/javascript">
       const vm = new Vue({
           el: '#databinding',
           data: {
             view: 'component1'
           },
           components: {
             'component1': {
                template: '<div><span style = "font-size:25;color:red;">Dynamic
Component</span></div>'
```

```
        }
      }
    });
</script>
</body>
</html
>
```

Los componentes dinámicos son creados de la siguiente manera:

```
<component v-bind:is = "view"></component>
```

El componente posee una directiva **v-bind:is**, cuyo valor es “view” y un valor asignado a ese
view. View está definido en la instancia Vue.

```
var vm = new Vue({
  el: '#databinding',
  data: {
      view: 'component1'
  },
  components: {
      'component1': {
        template: '<div><span style = "font-size:25;color:red;">Dynamic
Component</span></div>'
      }
  }
});
```

Propiedades computadas

Las propiedades computadas son similares a los métodos, pero las mismas no aceptan parámetros y su invocación no lleva () por que se ha vuelto una propiedad de la instancia Vue. Además, los métodos son llamados en todo momento cuando se actualice la vista, mientras que las propiedades computadas analizan el código y solo se actualizan si las propiedades involucradas en este cómputo cambian su valor.

```
<html>

  <head>
      <title>VueJs Instance</title>
      <script type = "text/javascript" src = "js/vue.js"></script>
  </head>
  <body>
```

```
<div id = "computed_props">
```

```
        FirstName : <input type = "text" v-model = "firstname" /> <br/><br/>
        LastName : <input type = "text" v-model = "lastname"/> <br/><br/>
        <h1>My name is {{firstname}} {{lastname}}</h1>
        <h1>Using computed method : {{getfullname}}</h1>
      </div>
      <script type = "text/javascript" src = "js/vue_computedprops.js"></script>
  </body>
</html>
```

vue_computeprops.js

```
const vm = new Vue({
  el: '#computed_props',
  data: {
      firstname :"",
      lastname :"",
      birthyear : ""
  },
  computed :{
      getfullname : function(){
        return this.firstname +" "+ this.lastname;
      }
  }
});
```

Se crea el documento con dos elementos input tipo caja de texto, los cuales están ligados ala instancia Vue con las directivas **v-model**.

Posteriormente tenemos la propiedad computada getfullname, que devuelve el firstname yel lastname que ingresó el usuario concatenados.

```
computed :{
  getfullname : function(){
      return this.firstname +" "+ this.lastname;
  }
}
```

Lo que se escribe en los textboxes es lo mismo que devolverá la función, cuando los valoresde los textboxes cambien así también lo hará lo devuelto por la función, gracias a utilizar las propiedades computadas que se llaman automáticamente cuando los textboxes cambien.

Acceder a los elementos del DOM utilizando $refs

Todo el tiempo necesitamos acceder a elementos del DOM utilizando JavaScript.
Vue.JS nos permite hacerlo de una manera supremamente sencilla.
Las instancias de Vue.JS cuentan con diversas propiedades, una de ellas es $refs.
Visto en código sería algo como:

app.$refsvm.$refs

Donde “app” o “vm” representan la instancia misma de Vue (por convención se utiliza “app”o “vm” para nombrar a la instancia de Vue, pero puedes utilizar el nombre que desees, también puedes utilizar la palabra reservada “this” para referirte a la instancia. En éste ejemplo utilizaremos el nombre de la instancia, en cuyo caso es “app”) y $refs sería una propiedad propia de la instancia.

Ahora bien, ¿qué es examente “$refs”?
Es un objeto, dentro de él se van a almacenar todos los elementos del DOM que cuentencon el atributo “ref”. El atributo “ref” vendría a ser algo así como darle un ID al elemento.
<input ref="entrada"></input>

Podemos tener todos los elementos que deseemos con el atributo “ref”, siempre y cuando elvalor del atributo sea diferente para cada elemento.
<input ref="entrada"></input>
<input ref="entrada2"></input>

Para acceder al objeto que almacena estos elementos bastaría con llamarlo de la siguientemanera:
app.$refs

Y para acceder al elemento dentro del objeto:
app.$refs.entrada

Vamos a aplicar lo anterior con una aplicación sencilla
Lo que hace la aplicación es añadir a un párrafo el texto que escribamos en una entrada:

Lo primero es crear el HTML y la instancia de Vue.HTML:

```
<div id="app">
<h1>Accediendo a Elementos del DOM utilizando vm.$refs</h1>
<h2>Añade texto a la página</h2>
<input type="text">
<br>
<br>
<button type="button">Guardar</button>
<button type="button">Borrar</button>
<p></p>
</div>
```

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

Vamos colocarle los atributos “ref” a los elementos cuyas propiedades queremos acceder,que en este caso son el input y el párrafo.

```
<div id="app">
<h1>Accediendo a Elementos del DOM utilizando vm.$refs</h1>
<h2>Añade texto a la página</h2>
<input type="text" ref="text">
<br>
<br>
<button type="button">Guardar</button>
<button type="button">Borrar</button>
<p ref="textField"></p>
</div>
```

Ahora vamos a escribir los métodos **addText()** y **deleteText()** de los botones en la instanciade Vue.

```
const app = new Vue({
el : '#app', methods : {
```

```
addText () {
const text = app.$refs.text.value const textField = app.$refs.textField
textField.innerHTML = textField.innerHTML + '<br />' + text
},
deleteText () {
const textField = app.$refs.textFieldtextField.innerHTML = ''
}
}
});
```

Y por último ponemos los botones a la escucha de los métodos.

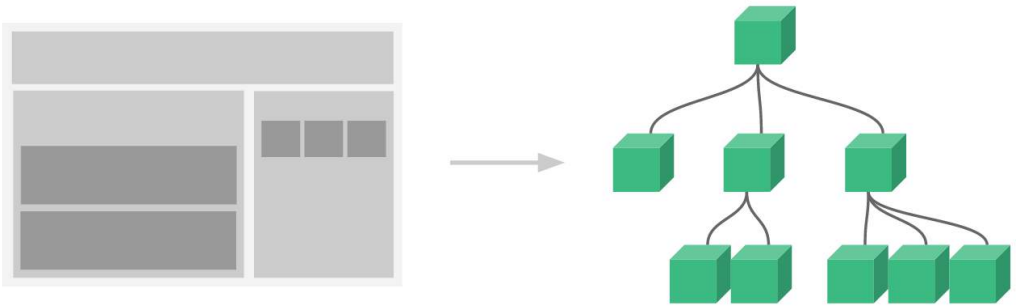
```
<div id="app">
<h1>Accediendo a Elementos del DOM utilizando vm.$refs</h1>
<h2>Añade texto a la página</h2>
<input type="text" ref="text">
<br>
<br>
<button type="button" @click="addText">Guardar</button>
<button type="button" @click="deleteText">Borrar</button>
<p ref="textField"></p>
</div>
```

Fuente: tutorialspoint.com / platzi.com

Componentes

Esquema de componentes

El sistema de componentes es otro concepto importante en Vue, porque es una abstracción que nos permite crear aplicaciones a gran escala compuestas de componentes pequeños, independientes y, a menudo, reutilizables. Si lo pensamos, 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.

Ejemplo Componentes

```
<html>
  <head>
      <title>VueJs Instance</title>
      <script type = "text/javascript" src = "js/vue.js"></script>
  </head>
  <body>
      <div id = "component_test">
        <testcomponent></testcomponent>
      </div>
      <div id = "component_test1">
        <testcomponent></testcomponent>
      </div>
      <script type = "text/javascript" src = "js/vue_component.js"></script>
  </body>
</html> vue_component.js

Vue.component('testcomponent',{
  template : '<div><h1>This is coming from component</h1></div>'
});
const vm = new Vue({
  el: '#component_test'
});
```

```
const vm1 = new Vue({
  el: '#component_test1'
});
```

Se crean dos div con id component_test y component_test1, junto con dos instancias Vue que hacen referencia a dichos ids. Se crea un componente en común que será asignado a ambas instancias.

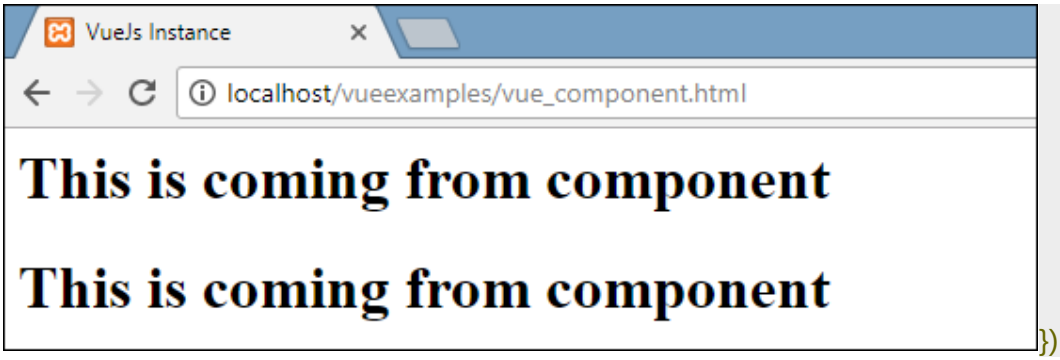
Para crear un componente se utiliza la siguiente sintaxis:

```
Vue.component('nombredelcomponente',{ // options});
```

```
<div id = "component_test">
  <testcomponent></testcomponent>
</div>
<div id = "component_test1">
  <testcomponent></testcomponent>
</div>
```

Dentro del componente se agrega un template el cual tiene asignado código HTML. Esta esla manera de registrar globalmente un componente en Vue.JS, que puede ser reutilizado encualquier instancia Vue.

```
Vue.component('testcomponent',{
  template : '<div><h1>This is coming from component</h1></div>'
```



});

El nombre personalizado de las etiquetas será reemplazado por el código del template en escrito en la creación del componente. En el navegador no existe ningún componente conetiquetas <testcomponent></testcomponent>.

También se puede modificar el comportamiento de un componente global dentro de unainstancia Vue.

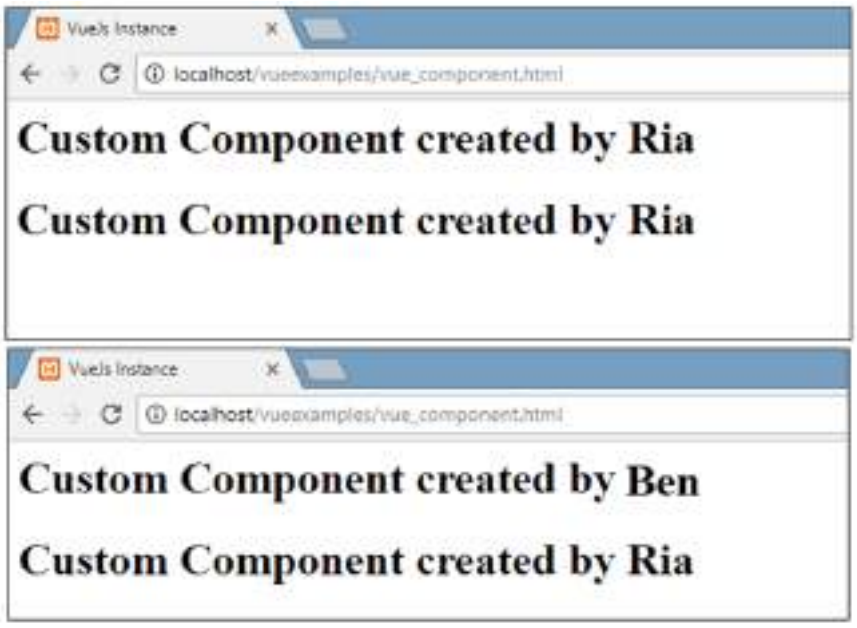
```
const vm = new Vue({
   el: '#component_test',
   components:{
        'testcomponent': {
           template : '<div><h1>This is coming from component</h1></div>'
        }
   }
});
```

Ejemplo

```
<html>
   <head>
        <title>VueJs Instance</title>
        <script type = "text/javascript" src = "js/vue.js"></script>
   </head>
   <body>
        <div id = "component_test">
           <testcomponent></testcomponent>
        </div>
        <div id = "component_test1">
           <testcomponent></testcomponent>
        </div>
        <script type = "text/javascript" src = "js/vue_component.js"></script>
   </body>
</html> vue_component.js

Vue.component('testcomponent',{
   template : '<div v-on:mouseover = "changename()" v-on:mouseout =
"originalname();"><h1>Custom Component created by <span id =
"name">{{name}}</span></h1></div>',
   data: function() {
        return {
           name : "Ria"
        }
   },
   methods:{
        changename : function() {
           this.name = "Ben";
        },
        originalname: function() {
           this.name = "Ria";
        }
   }
});
```

```
const vm = new Vue({
  el: '#component_test'
});
const vm1 = new Vue({
  el: '#component_test1'

});
```

En el ejemplo anterior, el template dentro del componente hace que cuando el mouse se posicione encima del mismo ejecute la función changename() y cuando el mismo salga del componente se ejecute la función originalname(). Ambas funciones trabajan sobre la propiedad name del componente, cambiandola.



Fuente: tutorialspoint.com / platzi.com

Componentes dinámicos

Los componentes dinámicos se crean con la etiqueta <component></component> y se ligan con un elemento del DOM de la siguiente manera.

```
<html>

  <head>
      <title>VueJs Instance</title>
      <script type = "text/javascript" src = "js/vue.js"></script>
  </head>
  <body>


      <div id = "databinding">
        <component v-bind:is = "view"></component>
      </div>

<script type = "text/javascript">
       const vm = new Vue({
           el: '#databinding',
           data: {
             view: 'component1'
           },
           components: {
             'component1': {
               template: '<div><span style = "font-size:25;color:red;">Dynamic
Component</span></div>'

   }
}
});
</script>
</body>
</html>
```

Los componentes dinámicos son creados de la siguiente manera:

```
<component v-bind:is = "view"></component>
```

El componente posee una directiva **v-bind:is**, cuyo valor es “view” y un valor asignado a ese **view**. View está definido en la instancia Vue.

```
var vm = new Vue({
   el: '#databinding',
   data: {
        view: 'component1'
   },
   components: {
        'component1': {
           template: '<div><span style = "font-size:25;color:red;">Dynamic
Component</span></div>'
        }
   }
});
```

Fuente: tutorialspoint.com / platzi.com

Propiedades computadas

Las propiedades computadas son similares a los métodos, pero las mismas no aceptan parámetros y su invocación no lleva () por que se ha vuelto una propiedad de la instancia Vue. Además, los métodos son llamados en todo momento cuando se actualice la vista, mientras que las propiedades computadas analizan el código y solo se actualizan si las propiedades involucradas en este cómputo cambian su valor.

```
<html>

  <head>
      <title>VueJs Instance</title>
      <script type = "text/javascript" src = "js/vue.js"></script>
  </head>
  <body>
      <div id = "computed_props">
         FirstName : <input type = "text" v-model = "firstname" /> <br/><br/>
         LastName : <input type = "text" v-model = "lastname"/> <br/><br/>
         <h1>My name is {{firstname}} {{lastname}}</h1>
         <h1>Using computed method : {{getfullname}}</h1>
      </div>
      <script type = "text/javascript" src = "js/vue_computedprops.js"></script>
  </body>
</html>
```

```
vue_computeprops.jsconst vm = new Vue({
  el: '#computed_props',
  data: {
      firstname :"",
      lastname :"",
      birthyear : ""
  },
  computed :{
      getfullname : function(){
         return this.firstname +" "+ this.lastname;
      }
  }
});
```

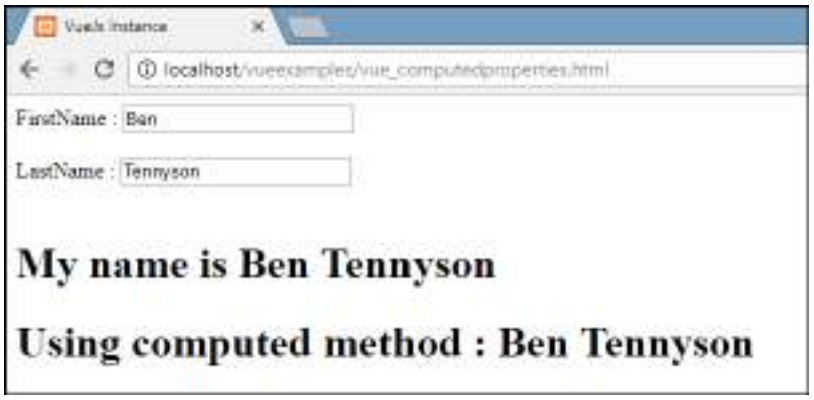
Se crea el documento con dos elementos input tipo caja de texto, los cuales están ligados ala instancia Vue con las directivas **v-model**.

Posteriormente tenemos la propiedad computada getfullname, que devuelve el firstname yel lastname que ingresó el usuario concatenados.

```
computed :{
  getfullname : function(){
      return this.firstname +" "+ this.lastname;
```

```
  }
}
```

Lo que se escribe en los textboxes es lo mismo que devolverá la función, cuando los valoresde los textboxes cambien así también lo hará lo devuelto por la función, gracias a utilizar las propiedades computadas que se llaman automáticamente cuando los textboxes cambien.



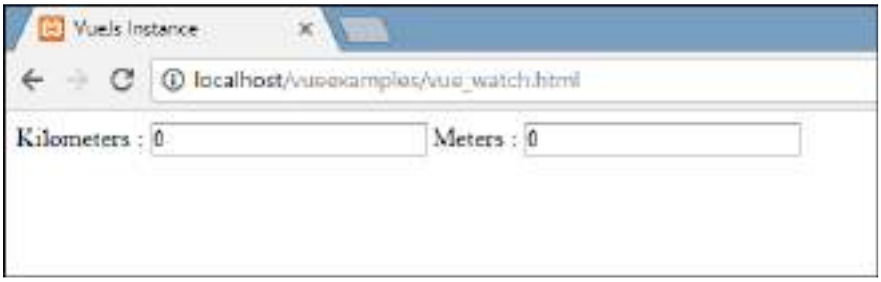
Watchers

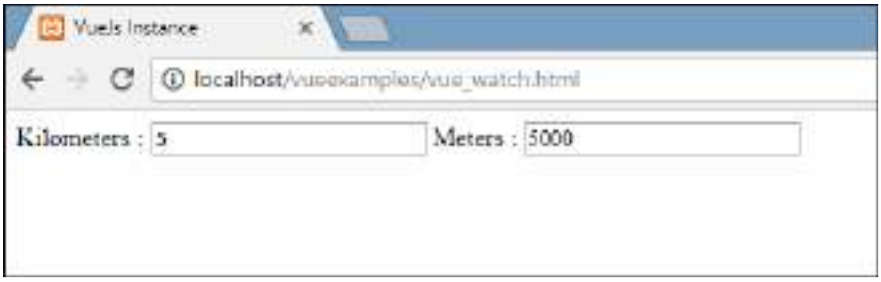
```
<html>
  <head>
     <title>VueJs Instance</title>
     <script type = "text/javascript" src = "js/vue.js"></script>
  </head>
  <body>
     <div id = "computed_props">
       Kilometers : <input type = "text" v-model = "kilometers">
       Meters : <input type = "text" v-model = "meters">
     </div>
     <script type = "text/javascript">
       const vm = new Vue({
         el: '#computed_props',
         data: {
           kilometers : 0,
           meters:0
         },
         methods: {
         },
         computed :{
         },
         watch : {
           kilometers:function(val) {
             this.kilometers = val;
             this.meters = val * 1000;
```

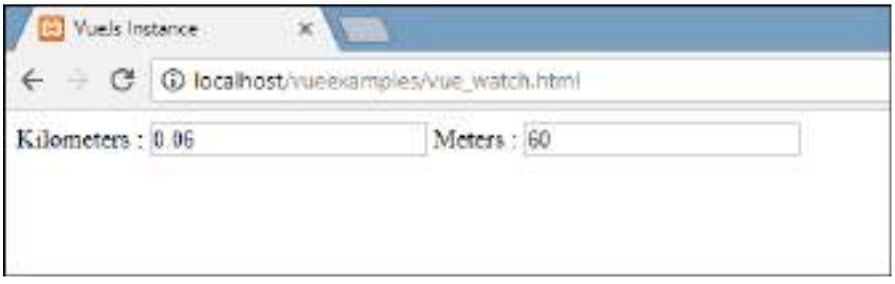
```
            },
            meters : function (val) {
              this.kilometers = val/ 1000;
              this.meters = val;
            }
          }
        });
    </script>
  </body>
</html>
```

Se crean dos textboxes, uno con kilómetros y otro con metros. En **data** ambas propiedades son inicializadas en cero. Existe dentro de la instancia Vue un objeto **watch** que se crea condos funciones, cuyo objetivo es convertir de kilómetros a metros y viceversa.

Cada vez que se ingrese algún valor en los textboxes, **watch** se encarga de actualizarlos calculando lo que está declarado en las funciones.







Fuente: tutorialspoint.com / platzi.com / https://es.vuejs.org/v2/guide/computed.html

Acceder a los elementos del DOM utilizando $refs

Todo el tiempo necesitamos acceder a elementos del DOM utilizando JavaScript.
Vue.JS nos permite hacerlo de una manera supremamente sencilla.
Las instancias de Vue.JS cuentan con diversas propiedades, una de ellas es $refs.
Visto en código sería algo como:

app.$refs
vm.$refs

Donde “app” o “vm” representan la instancia misma de Vue (por convención se utiliza “app”o “vm” para nombrar a la instancia de Vue, pero puedes utilizar el nombre que desees, también puedes utilizar la palabra reservada “this” para referirte a la instancia. En éste ejemplo utilizaremos el nombre de la instancia, en cuyo caso es “app”) y $refs sería una propiedad propia de la instancia.

Ahora bien, **¿qué es examente “$refs”?**
Es un objeto, dentro de él se van a almacenar todos los elementos del DOM que cuentencon el atributo “ref”. El atributo “ref” vendría a ser algo así como darle un ID al elemento.
<input ref="entrada"></input>

Podemos tener todos los elementos que deseemos con el atributo “ref”, siempre y cuando elvalor del atributo sea diferente para cada elemento.
<input ref="entrada"></input>
<input ref="entrada2"></input>

Para acceder al objeto que almacena estos elementos bastaría con llamarlo de la siguientemanera:
app.$refs

Y para acceder al elemento dentro del objeto:
app.$refs.entrada

Vamos a aplicar lo anterior con una aplicación sencilla
Lo que hace la aplicación es añadir a un párrafo el texto que escribamos en una entrada:

Lo primero es crear el HTML y la instancia de Vue.HTML:

```
<div id="app">
<h1>Accediendo a Elementos del DOM utilizando vm.$refs</h1>
<h2>Añade texto a la página</h2>
<input type="text">
<br>
<br>
<button type="button">Guardar</button>
<button type="button">Borrar</button>
```

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

Vamos colocarle los atributos “ref” a los elementos cuyas propiedades queremos acceder,que en este caso son el input y el párrafo.

<h1>Accediendo a Elementos del DOM utilizando vm.$refs</h1>
<h2>Añade texto a la página</h2>
<input type="text" ref="text">

<button type="button">Guardar</button>
<button type="button">Borrar</button>

Ahora vamos a escribir los métodos **addText()** y **deleteText()** de los botones en la instanciade Vue.
const app = new Vue({
el : '#app', methods : {
addText () {
const text = app.$refs.text.value const textField = app.$refs.textField
textField.innerHTML = textField.innerHTML + '
' + text
},
deleteText () {
const textField = app.$refs.textFieldtextField.innerHTML = ''
}
}
});
Y por último ponemos los botones a la escucha de los métodos.

<h1>Accediendo a Elementos del DOM utilizando vm.$refs</h1>
<h2>Añade texto a la página</h2>
<input type="text" ref="text">

<button type="button" @click="addText">Guardar</button>
<button type="button" @click="deleteText">Borrar</button>

Fuente: tutorialspoint.com / platzi.com

Más sobre manipulación de atributos

```
<html>
  <head>
      <title>VueJs Instance</title>
      <script type = "text/javascript" src = "js/vue.js"></script>
  </head>
  <body>
      <div id = "databinding">
        {{title}}<br/>
        <a href = "hreflink" target = "_blank"> Click Me </a> <br/>
        <a href = "{{hreflink}}" target = "_blank">Click Me </a>  <br/>
        <a v-bind:href = "hreflink" target = "_blank">Click Me </a>  <br/>
      </div>
      <script type = "text/javascript">
        const vm = new Vue({
          el: '#databinding',
          data: {
            title : "DATA BINDING",
            hreflink : "http://www.google.com"
          }
        });
      </script>
  </body>
</html>
```

En el ejemplo anterior se presentan tres supuestas formas de enlazar el atributo **href** a los enlaces. Pero sólo una es correcta al revisar el ejemplo en el navegador.

Como fue demostrado en anteriores ejemplos, para enlazar atributos desde una instancia Vue al DOM se utiliza la directiva **v-bind:atributo**. En este caso trabajaremos sobre **href.**

```
<a v-bind:href = "hreflink" target = "_blank">Click Me </a>
```

Vue.JS nos brinda un atajo a dicha directiva eliminando **v-bind.** Ninguna de las propiedades de Vue.JS será mostrada en el inspector del navegador.

```
<a :href = "hreflink" target = "_blank">Click Me </a>
```

Enlace Clases y Estilos

Una necesidad común de data binding es manipular la lista de clases de un elemento y sus estilos en línea. Como ambos son atributos, podemos usar **v-bind** para manejarlos: solo necesitamos crear una cadena de texto con nuestras expresiones. Sin embargo, concatenar cadenas de texto puede llegar a ser incómodo y propenso a errores. Por esta razón, Vue proporciona mejoras cuando se utiliza **v-bind** con **class** y **style**. Además de las cadenas de texto, las expresiones también pueden evaluar objetos o matrices.

Enlazando clases HTML

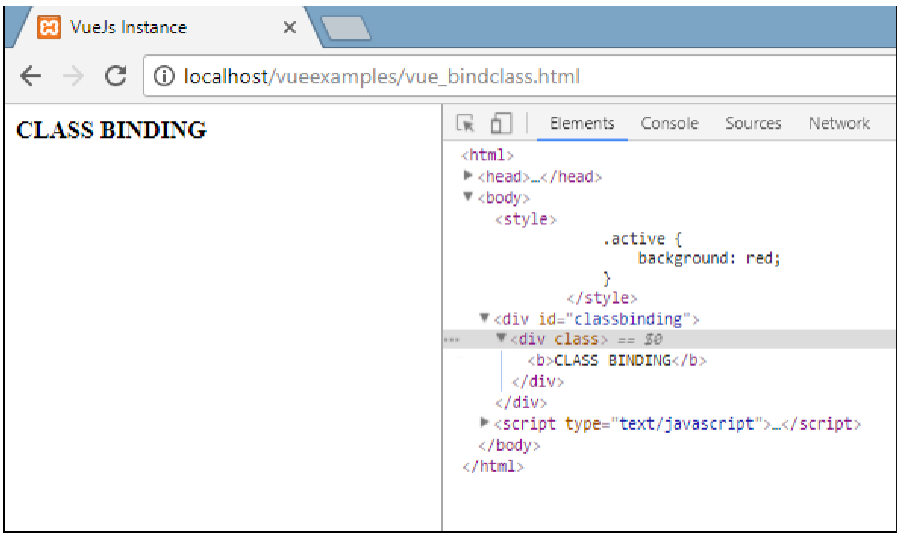
Para enlazar clases se utiliza también la directiva **v-bind**, pero utilizando el atributo **class**.

```
<html>
  <head>
      <title>VueJs Instance</title>
      <script type = "text/javascript" src = "js/vue.js"></script>
  </head>
  <body>
      <style>
         .active {
            background: red;
         }
      </style>
      <div id = "classbinding">
         <div v-bind:class = "{active:isactive}"><b>{{title}}</b></div>
      </div>
      <script type = "text/javascript">
         const vm = new Vue({
            el: '#classbinding',
            data: {
               title : "CLASS BINDING",
               isactive : true
            }
         });
      </script>
  </body>
</html>
```

isactive es una variable booleana, la cual puede contener en su interior true o false lo cual hará que se aplique o no la clase **active** al div enlazado a la instancia Vue. La clase active cambia el fondo del elemento a un color rojo.

En el siguiente ejemplo a dicha variable se le asigna el valor false.

```
<html>
  <head>
     <title>VueJs Instance</title>
     <script type = "text/javascript" src = "js/vue.js"></script>
  </head>
  <body>
     <style>
        .active {
           background: red;
        }
     </style>
     <div id = "classbinding">
        <div v-bind:class = "{active:isactive}"><b>{{title}}</b></div>
     </div>
     <script type = "text/javascript">
        const vm = new Vue({
           el: '#classbinding',
           data: {
              title : "CLASS BINDING",
              isactive : false
           }
        });
     </script>
  </body>
</html>
```



El siguiente ejemplo enlaza múltiples clases al elemento HTML.

```
<html>
  <head>
     <title>VueJs Instance</title>
     <script type = "text/javascript" src = "js/vue.js"></script>
  </head>
  <body>
     <style>
        .info {
```

```
            color: #00529B;
            background-color: #BDE5F8;
        }
        div {
            margin: 10px 0;
            padding: 12px;
        }
        .active {
            color: #4F8A10;
            background-color: #DFF2BF;
        }
        .displayError{
            color: #D8000C;
            background-color: #FFBABA;
        }
    </style>

<div id = "classbinding">
        <div class = "info"  v-bind:class = "{ active: isActive, 'displayError': hasError }">
            {{title}}
        </div>
    </div>
    <script type = "text/javascript">
        const vm = new Vue({
            el: '#classbinding',
            data: {
                title : "This is class binding example",
                isActive : false,
                hasError : false
            }
        });
    </script>
  </body>
</html>
```

Con ambas variables con valores false el elemento HTML se ve de la siguiente manera:

Con la variable isactive en true:

Con ambas variables en true:

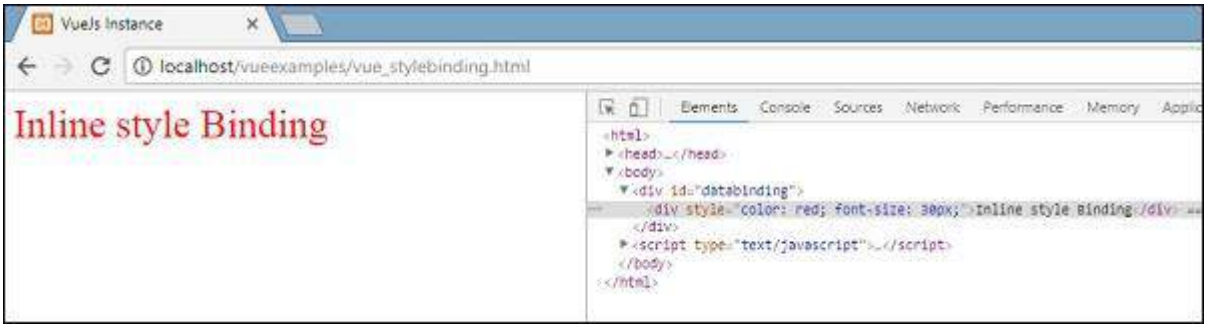
El siguiente ejemplo aplica **v-bind** para clases en un componente.

```
<html>
  <head>
     <title>VueJs Instance</title>
     <script type = "text/javascript" src = "js/vue.js"></script>
  </head>
  <body>
     <style>
        .info {
           color: #00529B;
           background-color: #BDE5F8;
        }
        div {
           margin: 10px 0;
           padding: 12px;
           font-size : 25px;
        }
        .active {
           color: #4F8A10;
           background-color: #DFF2BF;
        }
        .displayError{
           color: #D8000C;
           background-color: #FFBABA;
        }
     </style>
     <div id = "classbinding">
        <new_component class = "active"></new_component>
```

```
    </div>
    <script type = "text/javascript">
      var vm = new Vue({
        el: '#classbinding',
        data: {
          title : "This is class binding example",
          infoclass : 'info',
          errorclass : 'displayError',
          isActive : false,
          haserror : true
        },
 components:{
 'new_component' : {
            template : '<div class = "info">Class Binding for component</div>'
          }
        }
      });
    </script>
  </body>
</html>
```

Enlazar estilos inline

```
<html>
  <head>
    <title>VueJs Instance</title>
    <script type = "text/javascript" src = "js/vue.js"></script>
  </head>
  <body>
    <div id = "databinding">
      <div v-bind:style = "{ color: activeColor, fontSize: fontSize + 'px' }">{{title}}</div>
    </div>
    <script type = "text/javascript">
      const vm = new Vue({
        el: '#databinding',
        data: {
          title : "Inline style Binding",
          activeColor: 'red',
          fontSize :'30'
        }
      });
    </script>
  </body>
</html>
```



Fuente: tutorialspoint.com / https://es.vuejs.org/v2/guide/class-and-style.html

Binding en Formularios

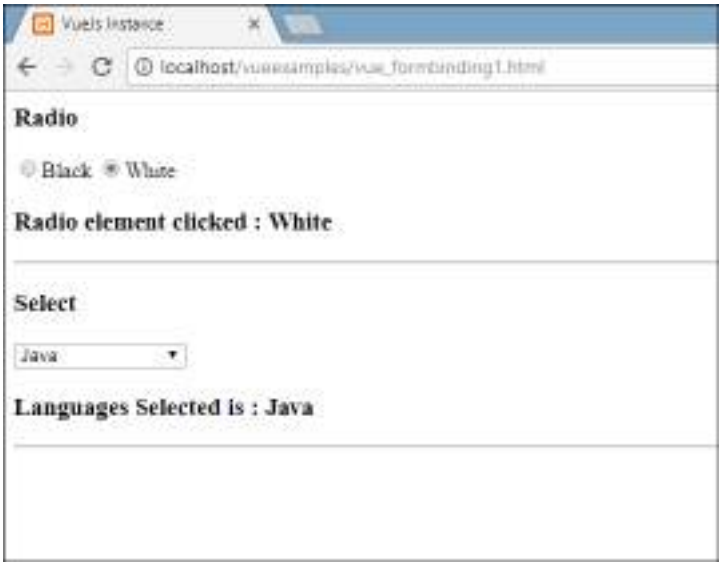
Enlazar elementos input de un formulario

```
<html>
  <head>
      <title>VueJs Instance</title>
      <script type = "text/javascript" src = "js/vue.js"></script>
  </head>
  <body>
      <div id = "databinding">
        <h3>TEXTBOX</h3>
        <input  v-model = "name" placeholder = "Enter Name" />
        <h3>Name entered is : {{name}}</h3>
        <hr/>
        <h3>Textarea</h3>
        <textarea v-model = "textmessage" placeholder = "Add Details"></textarea>
        <h1><p>{{textmessage}}</p></h1>
        <hr/>
        <h3>Checkbox</h3>
        <input type = "checkbox" id = "checkbox" v-model = "checked"> {{checked}}
      </div>
      <script type = "text/javascript">
        const vm = new Vue({
          el: '#databinding',
          data: {
            name:'',
            textmessage:'',
            checked : false
          }
        });
      </script>
  </body>
</html>
```

Lo que escribamos en el textbox se mostrará debajo. **v-model** tiene asignado el valor de la propiedad **name** y dicho nombre es mostrado en {{name}}, que muestra lo que tenga escrito en su interior el textbox, lo mismo sucede con el checkbox y el textarea.

Radio y Select

```
<html>
  <head>
      <title>VueJs Instance</title>
      <script type = "text/javascript" src = "js/vue.js"></script>
  </head>
  <body>
      <div id = "databinding">
        <h3>Radio</h3>
        <input type = "radio" id = "black" value = "Black" v-model = "picked">Black
        <input type = "radio" id = "white" value = "White" v-model = "picked">White
        <h3>Radio element clicked : {{picked}} </h3>
        <hr/>
        <h3>Select</h3>
        <select v-model = "languages">
          <option disabled value = "">Please select one</option>
          <option>Java</option>
          <option>Javascript</option>
          <option>Php</option>
          <option>C</option>
          <option>C++</option>
        </select>
        <h3>Languages Selected is : {{ languages }}</h3>
        <hr/>
      </div>
      <script type = "text/javascript">
        var vm = new Vue({
          el: '#databinding',
          data: {
            picked : 'White',
            languages : "Java"
          }
        });
      </script>
  </body>
</html>
```



Modificadores

Veremos tres tipos: **trim**, **number**, y **lazy**.

```
<html>
  <head>
      <title>VueJs Instance</title>
      <script type = "text/javascript" src = "js/vue.js"></script>
  </head>
  <body>
      <div id = "databinding">
         <span style = "font-size:25px;">Enter Age:</span> <input v-model.number = "age"
type = "number">
         <br/>
         <span style = "font-size:25px;">Enter Message:</span> <input v-model.lazy = "msg">
         <h3>Display Message : {{msg}}</h3>
         <br/>
         <span style = "font-size:25px;">Enter Message : </span><input v-model.trim =
"message">
         <h3>Display Message : {{message}}</h3>
      </div>
      <script type = "text/javascript">
         var vm = new Vue({
            el: '#databinding',
            data: {
               age : 0,
               msg: '',
               message : ''
            }
         });
      </script>
  </body>
</html>
```

El modificador **number** sólo permite el ingreso de números, no tomará otra entrada salvo que sean números.

```
<span style = "font-size:25px;">Enter Age:</span> <input v-model.number = "age" type = "number">
```

El modificador **lazy** mostrará el contenido presente en el textbox cuando el usuario abandone el mismo.

```
<span style = "font-size:25px;">Enter Message:</span> <input v-model.lazy = "msg">
```

El modificador **trim** eliminará los espacios al principio y al final de lo ingresado en el textbox.

```
<span style = "font-size:25px;">Enter Message : </span><input v-model.trim = "message">
```

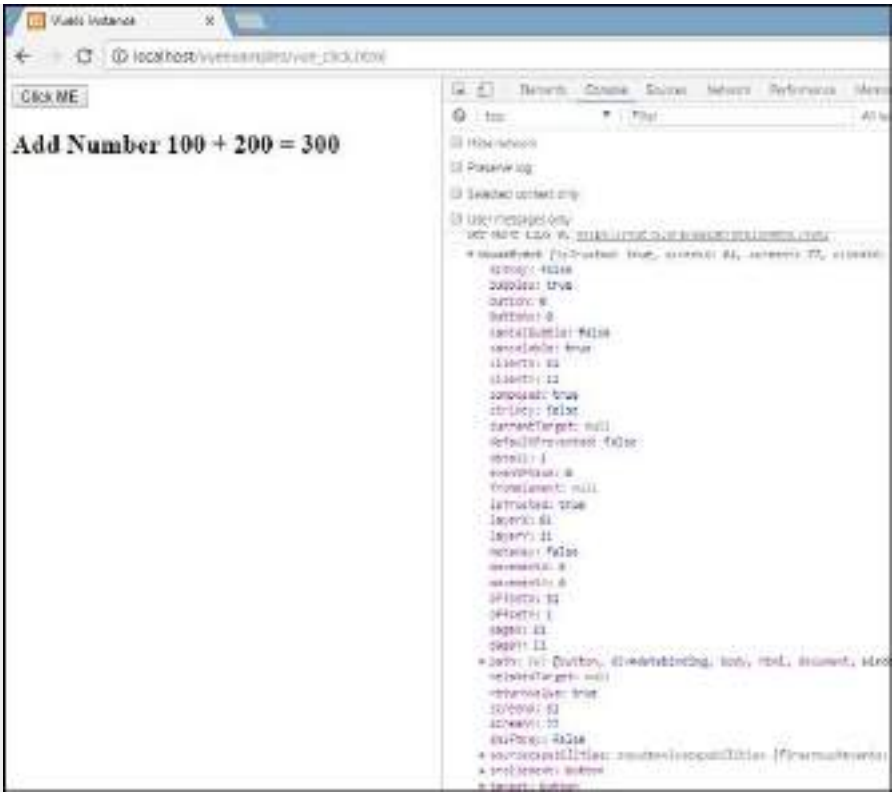
Fuente: tutorialspoint.com / https://es.vuejs.org/v2/guide/forms.html

Manejo de eventos

Utilizaremos la directiva **v-on** en los elementos HTML del DOM para escuchar a los eventos.

Evento de Click

```
<html>
  <head>
      <title>VueJs Instance</title>
      <script type = "text/javascript" src = "js/vue.js"></script>
  </head>
  <body>
      <div id = "databinding">
        <button v-on:click = "displaynumbers">Click ME</button>
        <h2> Add Number 100 + 200 = {{total}}</h2>
      </div>
      <script type = "text/javascript">
       const vm = new Vue({
          el: '#databinding',
          data: {
           num1: 100,
             num2 : 200,
             total : ''
          },
          methods : {
             displaynumbers : function(event) {
                console.log(event);
                return this.total =  this.num1+ this.num2;
             }
          },
       });
      </script>
  </body>
</html>
```



El siguiente código asignará un evento de click al elemento del DOM.

<button v-on:click = "displaynumbers">Click ME</button>Vue.JS nos brinda una propiedad atajo para los eventos:

<button @click = "displaynumbers">Click ME</button>

Cuando se cliquee el botón, se llamará al metodo displaynumbers.

Eventos mouseover, mouseout

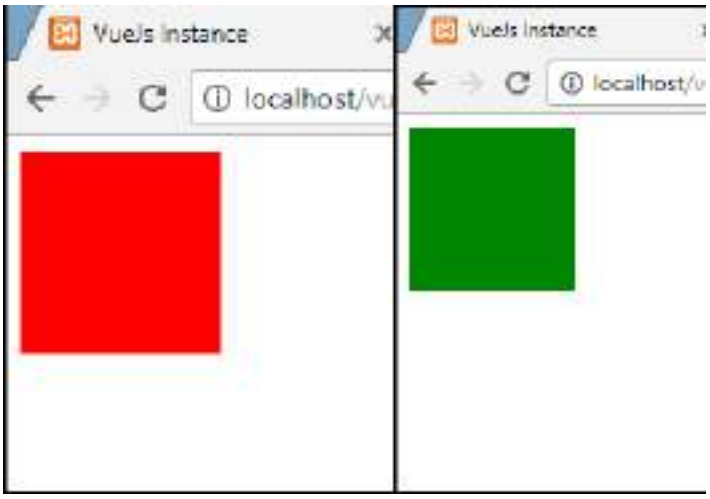
```
<html>
  <head>
      <title>VueJs Instance</title>
      <script type = "text/javascript" src = "js/vue.js"></script>
  </head>
  <body>
      <div id = "databinding">
        <div v-bind:style = "styleobj" v-on:mouseover = "changebgcolor" v-on:mouseout =
"originalcolor"></div>
      </div>
      <script type = "text/javascript">
        const vm = new Vue({
          el: '#databinding',
          data: {
            styleobj : {
              width:"100px",
              height:"100px",
              backgroundColor:"red"
            }
          },
          methods : {
```

```
            changebgcolor : function() {
              this.styleobj.backgroundColor = "green";
            },
            originalcolor : function() {
              this.styleobj.backgroundColor = "red";
            }
          },
        });
    </script>
  </body>
</html>
```

Se crea un div con ancho y alto de 100 px. Se le asigna un background de color rojo. Cuando el mouse se posiciona sobre él se cambiará dicho fondo al color verde y cuando el mouse salga del elemento volverá al color rojo mediante los métodos changebgcolor y originalcolor.

```
<div v-bind:style = "styleobj" v-on:mouseover = "changebgcolor" v-on:mouseout = "originalcolor"></div>
```

```
changebgcolor : function() {
  this.styleobj.backgroundColor = "green";
}
originalcolor : function() {
  this.styleobj.backgroundColor = "red";
}
```



Modificadores de eventos

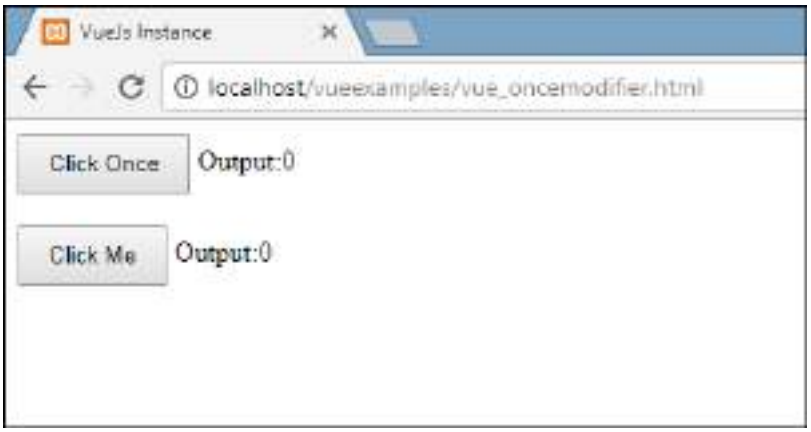
Vue.JS tiene disponibles modificadores de eventos en la directiva **v-on**.

.once

Permite que el evento se ejecute sólo una vez.

```
<button v-on:click.once = "buttonclicked">Click Once</button>
<html>
  <head>
    <title>VueJs Instance</title>
    <script type = "text/javascript" src = "js/vue.js"></script>
  </head>
  <body>
```

```
      <div id = "databinding">
        <button v-on:click.once = "buttonclickedonce" v-bind:style = "styleobj">Click
Once</button>
        Output:{{clicknum}}
        <br/><br/>
        <button v-on:click = "buttonclicked"  v-bind:style = "styleobj">Click Me</button>
        Output:{{clicknum1}}
      </div>
      <script type = "text/javascript">
        const vm = new Vue({
          el: '#databinding',
          data: {
            clicknum : 0,
            clicknum1 :0,
            styleobj: {
              backgroundColor: '#2196F3!important',
              cursor: 'pointer',
              padding: '8px 16px',
              verticalAlign: 'middle',
            }
          },
         methods : {
            buttonclickedonce : function() {
              this.clicknum++;
            },
            buttonclicked : function() {
              this.clicknum1++;
            }
          }
        });
      </script>
  </body>
</html>
```



En el anterior ejemplo se crean dos botones. Uno que posee el modificador **.once** y otro no. Uno podrá ejecutar tantas veces como el usuario desee el evento y el otro (con el modificador .once) no.

```
<button v-on:click.once = "buttonclickedonce" v-bind:style = "styleobj">Click Once</button>
```

```
<button v-on:click = "buttonclicked" v-bind:style = "styleobj">Click Me</button>
buttonclickedonce : function() {
  this.clicknum++;
},
buttonclicked : function() {
  this.clicknum1++;
}
```

.prevent

```
<a href = "http://www.google.com" v-on:click.prevent = "clickme">Click Me</a>
```

```
<html>
  <head>
     <title>VueJs Instance</title>
     <script type = "text/javascript" src = "js/vue.js"></script>
  </head>
  <body>
     <div id = "databinding">
        <a href = "http://www.google.com" v-on:click = "clickme" target = "_blank" v-bind:style
= "styleobj">Click Me</a>
     </div>
     <script type = "text/javascript">
        const vm = new Vue({
           el: '#databinding',
           data: {
              clicknum : 0,
              clicknum1 :0,
              styleobj: {
                 color: '#4CAF50',
                 marginLeft: '20px',
                 fontSize: '30px'
              }
           },
           methods : {
              clickme : function() {
                 alert("Anchor tag is clicked");
              }
           }
        });
     </script>
  </body>
```

```
</html>
```

Si cliqueamos el link nos aparecerá un alert() que nos informa que el link fué cliqueado.

Si el link no posee el modificador **.prevent** se abrirá el enlace al que apunta el link. Dicho comportamiento es el que tienen por defecto los enlaces al ser clickeados. Con .prevent evitamos dicho comportamiento y ejecutamos la función especificada en las comillas.

```
<a href = "http://www.google.com" v-on:click.prevent = "clickme" target = "_blank" v-bind:style = "styleobj">Click Me</a>
```

Modificadores de teclas

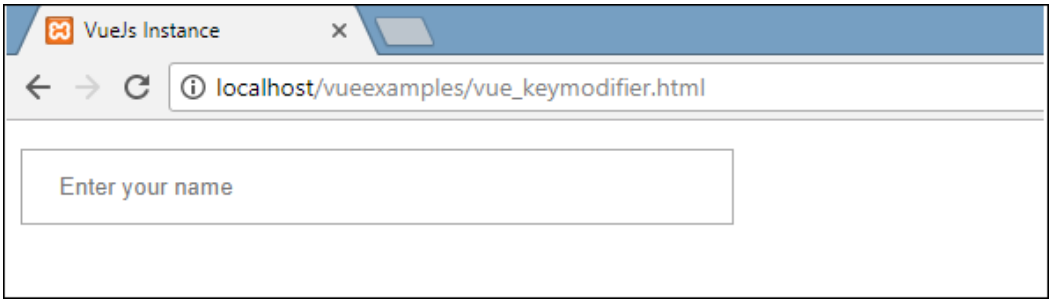
Vue.JS ofrece modificadores de teclas mediante los cuales se manejan los eventos. Si necesitaramos que un textbox llame a un método definido en una instancia Vue sólo cuando se pulsa la tecla enter deberíamos agregar un modificador.

```
<input type = "text"  v-on:keyup.enter = "showinputvalue"/>
```

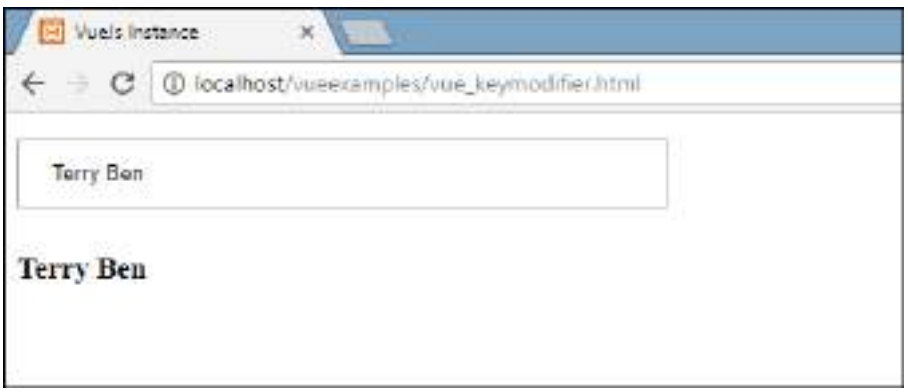
Podemos utilizar múltiples teclas. Por ejemplo: V-on.keyup.ctrl.enter

```
<html>
  <head>
      <title>VueJs Instance</title>
```

```
    <script type = "text/javascript" src = "js/vue.js"></script>
  </head>
  <body>
    <div id = "databinding">
      <input type = "text" v-on:keyup.enter = "showinputvalue" v-bind:style = "styleobj"
placeholder = "Enter your name"/>
      <h3> {{name}}</h3>
    </div>
    <script type = "text/javascript">
      const vm = new Vue({
        el: '#databinding',
        data: {
          name:'',
          styleobj: {
            width: "30%",
            padding: "12px 20px",
            margin: "8px 0",
            boxSizing: "border-box"
          }
        },
        methods : {
          showinputvalue : function(event) {
            this.name=event.target.value;
          }
        }
      });
    </script>
  </body>
</html>
```



Al presionar Enter:

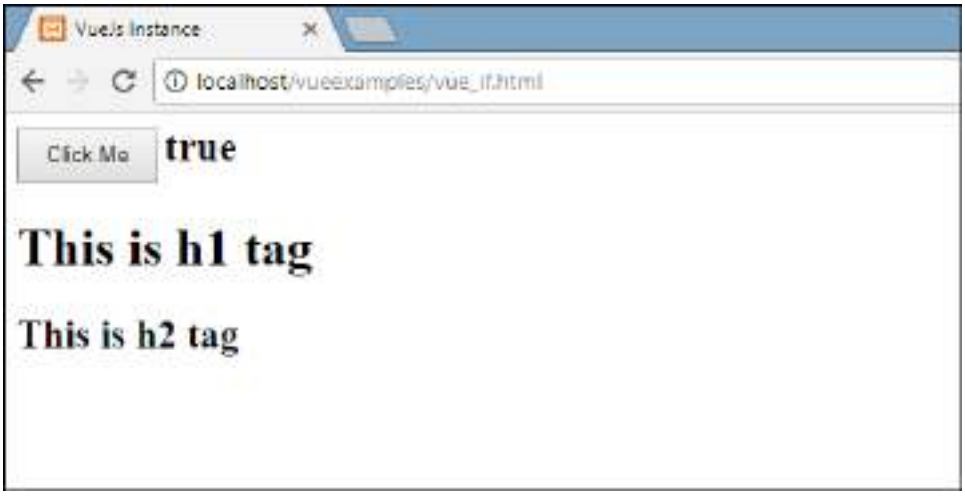


Fuente: tutorialspoint.com / https://es.vuejs.org/v2/guide/events.html

Renderización Condicional

v-if

```
<html>
  <head>
      <title>VueJs Instance</title>
      <script type = "text/javascript" src = "js/vue.js"></script>
  </head>
  <body>
      <div id = "databinding">
        <button v-on:click = "showdata" v-bind:style = "styleobj">Click Me</button>
        <span style = "font-size:25px;"><b>{{show}}</b></span>
        <h1 v-if = "show">This is h1 tag</h1>
        <h2>This is h2 tag</h2>
      </div>
      <script type = "text/javascript">
        const vm = new Vue({
          el: '#databinding',
          data: {
            show: true,
            styleobj: {
              backgroundColor: '#2196F3!important',
              cursor: 'pointer',
              padding: '8px 16px',
              verticalAlign: 'middle',
            }
          },
          methods : {
            showdata : function() {
              this.show = !this.show;
            }
          },
        });
      </script>
  </body>
</html>
```



Se crea un botón con dos etiquetas de encabezados con un mensaje en el interior de ellas. Una variable llamada **show** es declarada e inicializada con un valor **true**. Su valor se muestra cerca del botón. Cada vez que se clickea el botón se llama al método **showdata** que cambia el estado de dicha variable de false a true y viceversa

```
<button v-on:click = "showdata" v-bind:style = "styleobj">Click Me</button>
<h1 v-if = "show">This is h1 tag</h1>
```

Si el valor de la variable show es falso el tag h1 no se mostrará.

v-else

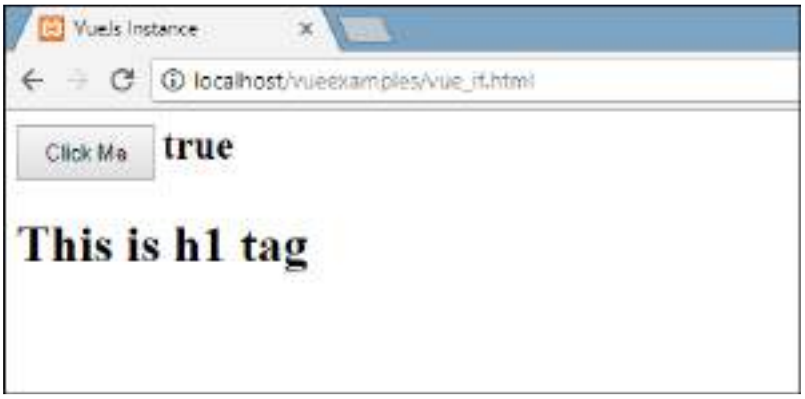
```
<html>
  <head>
     <title>VueJs Instance</title>
     <script type = "text/javascript" src = "js/vue.js"></script>
  </head>
  <body>
     <div id = "databinding">
        <button v-on:click = "showdata" v-bind:style = "styleobj">Click Me</button>
        <span style = "font-size:25px;"><b>{{show}}</b></span>
        <h1 v-if = "show">This is h1 tag</h1>
        <h2 v-else>This is h2 tag</h2>
     </div>
     <script type = "text/javascript">
        const vm = new Vue({
           el: '#databinding',
           data: {
              show: true,
              styleobj: {
                 backgroundColor: '#2196F3!important',
                 cursor: 'pointer',
                 padding: '8px 16px',
                 verticalAlign: 'middle',
              }
           },
```

```
            methods : {
               showdata : function() {
                  this.show = !this.show;
               }
            },
         });
      </script>
   </body>
</html>
```

v-else se añade utilizando la siguiente sintaxis:

```
<h1 v-if = "show">This is h1 tag</h1>
<h2 v-else>This is h2 tag</h2>
```

Ahora, si **show** es true, se mostrará “This is h1 tag”, si es false, se mostrará “This is h2 tag”



v-show

v-show se comporta igual que v-if. También muestra y oculta los elementos basado en una condición. La diferencia es que v-if los elimina del DOM si la condición es falsa y v-show sólo los oculta.

```
<html>
   <head>
      <title>VueJs Instance</title>
      <script type = "text/javascript" src = "js/vue.js"></script>
   </head>
   <body>
      <div id = "databinding">
```

```
        <button v-on:click = "showdata" v-bind:style = "styleobj">Click Me</button>
        <span style = "font-size:25px;"><b>{{show}}</b></span>
        <h1 v-if = "show">This is h1 tag</h1>
        <h2 v-else>This is h2 tag</h2>
        <div v-show = "show">
          <b>V-Show:</b>
          <img src = "images/img.jpg" width = "100" height = "100" />
        </div>
      </div>
      <script type = "text/javascript">
        const vm = new Vue({
          el: '#databinding',
          data: {
            show: true,
            styleobj: {
              backgroundColor: '#2196F3!important',
              cursor: 'pointer',
              padding: '8px 16px',
              verticalAlign: 'middle',
}
},
methods : {
showdata : function() { this.show = !this.show;
}

});

</body>
</html>
```

Se asigna v-show a un elemento HTML de la siguiente manera:

```
<div v-show = "show"><b>V-Show:</b><img src = "images/img.jpg" width = "100" height = "100" /></div>
```

Si la variable **show** es true se muestra la imagen, si no, la misma se oculta. Si inspeccionamos el DOM vemos que sigue estando, nada más que está ocultada.

Fuente: tutorialspoint.com / https://es.vuejs.org/v2/guide/conditional.html

Renderizado de listas

v-for

```
<html>
  <head>
      <title>VueJs Instance</title>
      <script type = "text/javascript" src = "js/vue.js"></script>
  </head>
  <body>
      <div id = "databinding">
        <input type = "text" v-on:keyup.enter = "showinputvalue"
          v-bind:style = "styleobj" placeholder = "Enter Fruits Names"/>
        <h1 v-if = "items.length>0">Display Fruits Name</h1>
        <ul>
          <li v-for = "a in items">{{a}}</li>
        </ul>
      </div>
      <script type = "text/javascript">
        const vm = new Vue({
          el: '#databinding',
          data: {
            items:[],
            styleobj: {
              width: "30%",
              padding: "12px 20px",
              margin: "8px 0",
              boxSizing: "border-box"
            }
          },
          methods : {
            showinputvalue : function(event) {
              this.items.push(event.target.value);
            }
          },
        });
      </script>
  </body>
</html>
```

Se declara como array una variable llamada **items**. En la sección de métodos hay uno que se llama showinputvalue, el cual es asignado al textbox y recibe el nombre de las frutas. En el método dichos nombres serán agregados al array mediante el siguiente código.

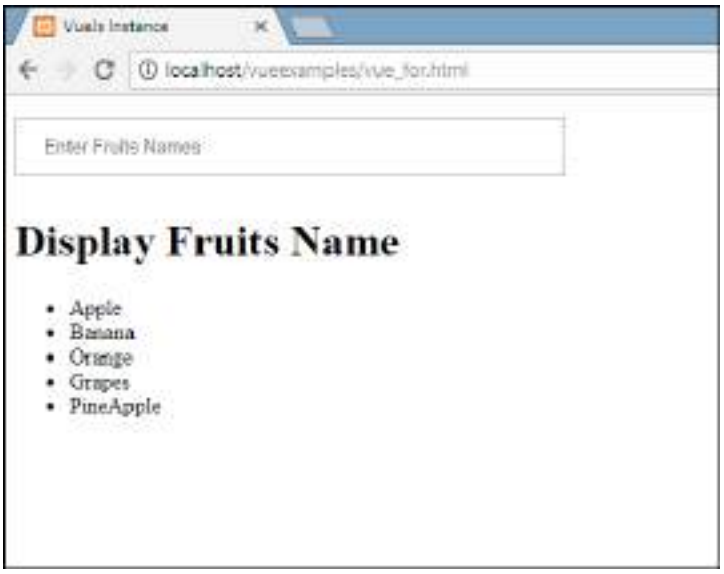
```
showinputvalue : function(event) {
  this.items.push(event.target.value);
}
```

Usaremos **v-for** para mostrar las frutas ingresadas al array mediante el siguiente código. v-for sirve para iterar sobre los elementos presentes en el array.

```
<ul>
  <li v-for = "a in items">{{a}}</li>
```

```
</ul>
```

Para iterar sobre el array en bucle utilizamos v-for = ”a in items” en donde **a** tendrá en su interior cada uno de los valores del array de forma secuencial y los mostrará mientras que queden ítems restantes que recorrer.



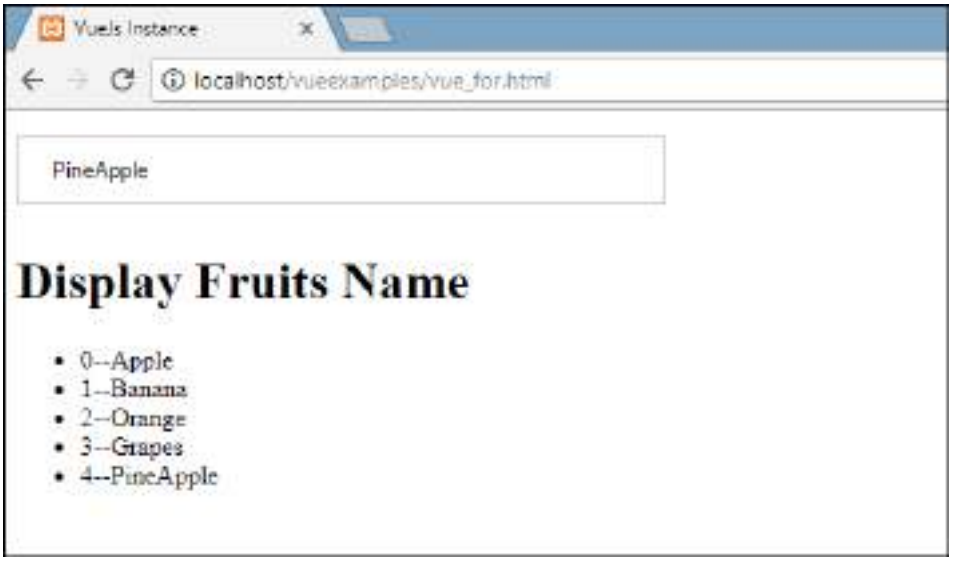
Si quisieramos mostrar el índice del array se debería utilizar el siguiente código.

```
<html>
  <head>
     <title>VueJs Instance</title>
     <script type = "text/javascript" src = "js/vue.js"></script>
  </head>
  <body>
     <div id = "databinding">
       <input type = "text" v-on:keyup.enter = "showinputvalue"
         v-bind:style = "styleobj" placeholder = "Enter Fruits Names"/>
       <h1 v-if = "items.length>0">Display Fruits Name</h1>
       <ul>
         <li v-for = "(a, index) in items">{{index}}--{{a}}</li>
       </ul>
     </div>
     <script type = "text/javascript">
       const vm = new Vue({
         el: '#databinding',
         data: {
```

```
            items:[],
            styleobj: {
              width: "30%",
              padding: "12px 20px",
              margin: "8px 0",
              boxSizing: "border-box"
            }
          },
          methods : {
            showinputvalue : function(event) {
              this.items.push(event.target.value);
            }
          },
        });
    </script>
  </body>
</html>
```

Para conseguir el índice se agrega entre paréntesis y separado por una coma de **a**, la palabra **index**, cuyo valor es posteriormente mostrado en {{index}}.

`<li v-for = "(a, index) in items">{{index}}--{{a}}</li>`En **(a, index)**, **a** es el valor e **index** es la llave.



Fuente: tutorialspoint.com / https://es.vuejs.org/v2/guide/list.html

Asincronía

La **asincronía** es uno de los conceptos principales que rige el mundo de JavaScript. Cuando comenzamos a programar, normalmente realizamos tareas de forma **síncrona**, llevando a cabo tareas secuenciales que se ejecutan una detrás de otra, de modo que elorden o flujo del programa es sencillo y fácil de observar en el código:

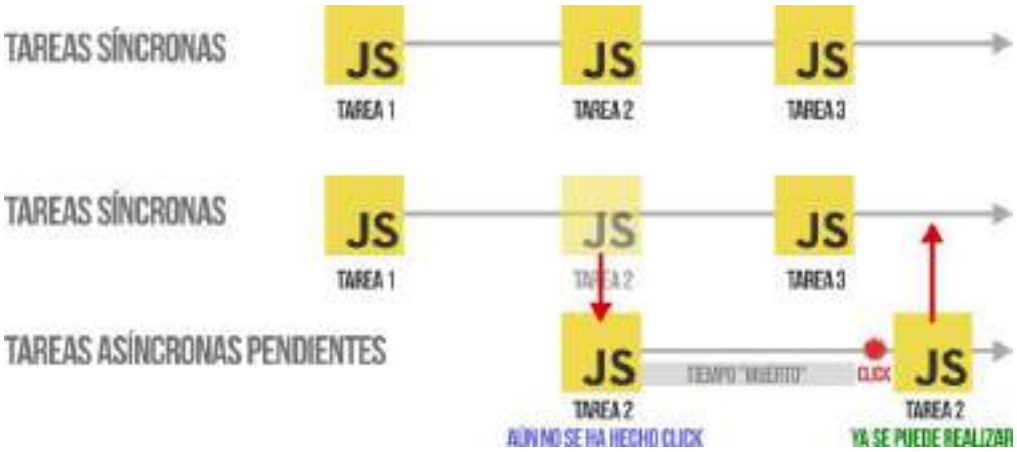
```
primera_funcion();          // Tarea 1: Se ejecuta primero
segunda_funcion(); // Tarea 2: Se ejecuta cuando termina primera_funcion()
tercera_funcion();          // Tarea 3: Se ejecuta cuando termina segunda_funcion()
```

Sin embargo, en el mundo de la programación, tarde o temprano necesitaremos realizar operaciones **asíncronas**, especialmente en ciertos lenguajes como JavaScript, donde tenemos que realizar tareas **que tienen que esperar a que ocurra un determinado suceso** que no depende de nosotros, y reaccionar realizando otra tarea sólo cuando dichosuceso ocurra.

Lenguaje no bloqueante

Cuando hablamos de JavaScript, habitualmente nos referimos a él como un lenguaje **no bloqueante**. Con esto queremos decir que las tareas que realizamos no se quedan bloqueadas esperando ser finalizadas, y por consiguiente, evitando proseguir con el restode tareas.

Imaginemos que la **segunda_funcion()** del ejemplo anterior realiza una tarea que depende de otro factor, como por ejemplo un click de ratón del usuario. Si hablasemos de un **lenguaje bloqueante**, hasta que el usuario no haga click, JavaScript no seguiría ejecutandolas demás funciones, sino que se quedaría bloqueado esperando a que se terminase esa segunda tarea:



Pero como JavaScript es un **lenguaje no bloqueante**, lo que hará es mover esa tarea auna lista de **tareas pendientes** a las que irá «*prestando atención*» a medida que lo necesite, pudiendo continuar y retomar el resto de tareas a continuación de la segunda.

¿Qué es la asincronía?

Pero esto no es todo. Ten en cuenta que pueden existir **múltiples** tareas asíncronas, dichastareas puede que terminen realizándose correctamente (*o puede que no*) y ciertas tareas pueden depender de otras, por lo que deben respetar un cierto orden. Además, es muy habitual que no sepamos previamente **cuánto tiempo** va a tardar en terminar una tarea, porlo que necesitamos un mecanismo para controlar todos estos factores.

¿Cómo gestionar la asincronía?

Teniendo en cuenta el punto anterior, debemos aprender a buscar mecanismos para dejarclaro en nuestro código JavaScript, que ciertas tareas tienen que procesarse de forma asíncrona para quedarse a la espera, y otras deben ejecutarse de forma síncrona.

En JavaScript existen varias formas de gestionar la **asincronía**, donde quizás las más populares son las siguientes:

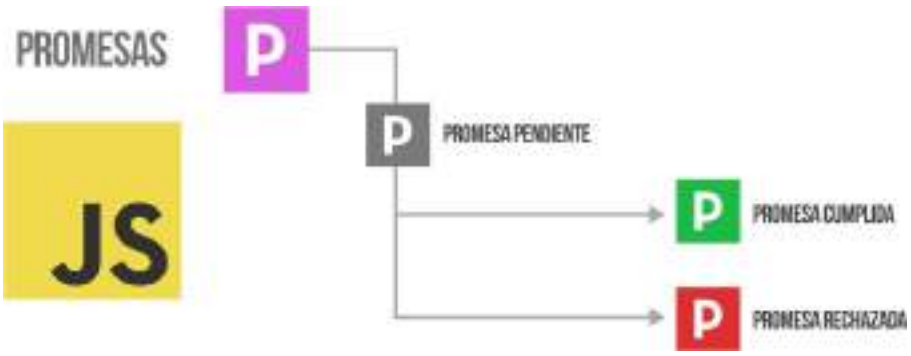
Método	Descripción
Mediante **callbacks**	Probablemente, la forma más clásica de gestionar la asincroníaen JavaScript.
Mediante **promesas**	Una forma más moderna y actual de gestionar la asincronía.
Mediante **async**/**await**	Seguimos con promesas, pero con **async**/**await** añadimos más azúcar sintáctico.

Fuente: lenguajejs.com / programacionymas.com

Promesas

Las **promesas** son un concepto para resolver el problema de asincronía de una forma mucho más elegante y práctica.

Como su propio nombre indica, una **promesa** es algo que, en principio pensamos que secumplirá, pero en el futuro pueden ocurrir varias cosas:



- La promesa **se cumple** (*promesa resuelta*)
- La promesa **no se cumple** (*promesa se rechaza*)
- La promesa se queda en un **estado incierto** indefinidamente (*promesa pendiente*)

Básicamente, las Promesas son similares a los Eventos, con las siguientes diferencias:

- Una promesa sólo puede tener éxito o fracasar una única vez. No puede tener éxito o fallar por una 2da vez, ni cambiar de éxito a fallo posteriormente, o viceversa.
- Si una promesa ha sido exitosa o ha fallado, y más adelante (recién)registramos un callback de success o failure, la función de callback correspondiente será llamada (incluso si el evento tuvo lugar antes).

Esto resulta muy útil para operaciones asíncronas, porque más allá de capturar el momentoexacto en que ocurre algo, nos enfocamos en reaccionar ante lo ocurrido.

Terminología asociada a las Promesas

Tenemos muchos términos relacionados a lo que son Promesas en Javascript. A continuaciónveamos lo más básico.

Una promesa puede presentar los siguientes estados:

- **fulfilled** - La acción relacionada a la promesa se llevó a cabo con éxito
- **rejected** - La acción relacionada a la promesa falló
- **pending** - Aún no se ha determinado si la promesa fue fulfilled o rejected
- **settled** - Ya se ha determinado si la promesa fue fulfilled o rejected

También se suele usar el término **thenable**, para indicar que un objeto tiene disponible unmétodo "then" (y que por tanto está relacionado con Promesas).

Promesas en JavaScript

Las **promesas** en JavaScript se representan a través de un **object**, y cada **promesa** estaráen un estado concreto: **pendiente**, **aceptada** o **rechazada**. Además, cada **promesa** tiene los siguientes métodos, que podremos utilizar para utilizarla:

Métodos	Descripción
.then(resolve)	Ejecuta la función callback **resolve** cuando la promesa secumple.
.catch(reject)	Ejecuta la función callback **reject** cuando la promesa serechaza.
.then(resolve,reject)	Método equivalente a las dos anteriores en el mismo **.then()**.
.finally(end)	Ejecuta la función callback **end** tanto si se cumple como si se rechaza.

Cómo crear una Promesa en JS

Un objeto Promise representa un valor, que no se conoce necesariamente al momento de crear la promesa.

Esta representación nos permite realizar acciones, con base en el valor de éxito devuelto, o la razón de fallo.

Es decir, los métodos asíncronos producen valores que aún no están disponibles. Pero la idea ahora es que, en vez de esperar y devolver el valor final, tales métodos devuelven un objeto Promise (que nos proveerá del valor resultante en el futuro).

Hoy en día, muchas bibliotecas JS se están actualizando para hacer uso de Promesas, en vez de simples funciones callback.

Nosotros también podemos crear nuestras promesas, basados en esta sintaxis:

```
new Promise(function(resolve, reject) { ... });
```

Este constructor es usado principalmente para envolver funciones que no soportan el uso dePromesas.

- El constructor espera una función como parámetro. A esta función se leconoce como **executor**.
- Esta función executor recibirá 2 argumentos: **resolve** y **reject**.
- La función executor es ejecutada inmediatamente al implementar el objeto Promise, recibiendo las funciones resolve y reject para su uso correspondiente. Esta función executor es llamada incluso antes que el constructor Promise devuelva el objeto creado.
- Las funciones resolve y reject, al ser llamadas, "resuelven" o "rechazan" la promesa. Es decir, modifican el estado de la promesa (como hemos visto antes, inicialmente es pending, pero posteriormente puede ser fulfilled o rejected).
- Normalmente el executor inicia alguna operación asíncrona, y una vez que ésta se completa, llama a la función resolve para resolver la promesa o bienreject si ocurrió algo inesperado.
- Si la función executor lanza algún error, la promesa también es rejected.
- El valor devuelto por la función executor es ignorado.

```
let promise = new Promise(function(resolve, reject) {function sayHello() {
  resolve('Hello World!')
  }
  setTimeout(sayHello, 3000)
  });
  console.log(promise);
```

Lo que ha de ocurrir es lo siguiente:

- Se ejecuta la función executor y se crea nuestro objeto Promise.
- Se llama al método then, expresando qué es lo que queremos hacer con elvalor que devolverá la promesa.
- Se imprime por consola [object Promise].
- A los 3 segundos tras ejecutar la función executor, se resuelve la promesa, y terminamos mostrando por consola el mensaje "Hello World!".

Ejemplo con .then(), .catch() y .finally()

```
let promesa = new Promise(function(resolve,reject){
if(true){
resolve(`Funcionó!`);
```

```
}
else{
reject(`Hay un error`);
}
});

promesa.then(function(respuesta){
console.log(`Respuesta: ${respuesta}`);
})
.catch(function(error){
console.log(`Error: ${error}`);
})
.finally(function(){
console.log(`Esto se ejecuta siempre`);
});
```

Objeto Promise

El objeto **Promise** de Javascript tiene varios métodos que podemos utilizar en nuestro código. Todos devuelven una promesa y son los que veremos en la siguiente tabla:

Métodos	Descripción
Promise.all([array]list)	Acepta sólo si **todas** las promesas del **array** secumplen.
Promise.allSettled([array]list)	Acepta sólo si **todas** las promesas del **array** secumplen o rechazan.
Promise.any([array]list)	Acepta con el valor de la primera promesa del **array** que se cumpla.
Promise.race([array]list)	Acepta o rechaza dependiendo de la primera promesaque se procese.
Promise.resolve(value)	Devuelve un valor envuelto en una promesa que secumple directamente.
Promise.reject(value)	Devuelve un valor envuelto en una promesa que se rechaza directamente.

En los siguientes ejemplos, vamos a utilizar la función **fetch()** para realizar varias peticionesy descargar varios archivos diferentes que necesitaremos para nuestras tareas.

Promise.all()

El método **Promises.all()** funciona como un «**todo o nada**»: devuelve una promesa que secumple cuando todas las promesas del **array** se cumplen. Si alguna de ellas se rechaza, **Promise.all()** también lo hace.
A **Promise.all()** le pasamos un **array** con las promesas individuales. Cuando **todas y cadauna** de esas promesas se cumplan favorablemente, **entonces** se ejecutará la función callback de su **.then()**. En el caso de que alguna se rechace, no se llegará a ejecutar.

Promise.allSettled()

El método **Promises.allSettled()** funciona como un «**todas procesadas**»: devuelve unapromesa que se cumple cuando todas las promesas del **array** se hayan procesado, independientemente de que se hayan cumplido o rechazado. });
Esta promesa nos devuelve un campo **status** donde nos indica si cada promesa individual ha sido cumplida o rechazada, y un campo **value** con los valores devueltos por la promesa.

Promise.any()

El método **Promise.any()** funciona como «**la primera que se cumpla**»: Devuelve una promesa con el valor de la primera promesa individual del **array** que se cumpla. Si todas laspromesas se rechazan, entonces devuelve una promesa rechazada.
Promise.any() devolverá una respuesta de la primera promesa cumplida.

Promise.race()

El método **Promise.race()** funciona como una «**la primera que se procese**»: la primerapromesa del **array** que sea procesada, independientemente de que se haya cumplido o rechazado, determinará la devolución de la promesa del **Promise.race()**. Si se cumple, devuelve una promesa cumplida, en caso negativo, devuelve una rechazada.
De forma muy similar a la anterior, **Promise.race()** devolverá la promesa que se resuelvaprimero, ya sea cumpliéndose o rechazándose.

Async/Await

La palabra clave async

En **ES2017** se introducen las palabras clave **async**/**await**, que no son más que una forma de **azúcar sintáctico** para gestionar las promesas de una forma más sencilla. Con **async**/**await** seguimos utilizando promesas, pero abandonamos el modelo de encadenamiento de **.then()** para utilizar uno en el que trabajamos de forma más tradicional.

En primer lugar, tenemos la palabra clave **async**. Esta palabra clave se colocará previamente a **function**, para definirla así como una **función asíncrona**, el resto de la función no cambia:

```
async function funcion_asincrona() {
  return 42;
}
```

En el caso de que utilicemos **arrow function**, se definiría como vemos a continuación, colocando el **async** justo antes de los parámetros de la arrow function:

```
const funcion_asincrona = async () => 42;
```

Al ejecutar la función veremos que ya nos devuelve una que ha sido cumplida, con el valordevuelto en la función (*en este caso, 42*). De hecho, podríamos utilizar un **.then()** para manejar la promesa:

```
funcion_asincrona().then(function(value){
  console.log("El resultado devuelto es: ", value);
});
```

Sin embargo, veremos que lo que se suele hacer junto a **async** es utilizar la palabra clave
await, que es donde reside lo interesante de utilizar este enfoque.

La palabra clave await

Cualquier función definida con **async**, o lo que es lo mismo, cualquier **Promise** puede utilizarse junto a la palabra clave **await** para manejarla. Lo que hace **await** es esperar a quese resuelva la promesa, mientras permite continuar ejecutando otras tareas que puedan realizarse:

```
const funcion_asincrona = async function(){return 42;}
```

```
const value = funcion_asincrona();                    // Promise { <fulfilled>: 42 }const
asyncValue = await funcion_asincrona(); // 42
```

Observa que en el caso de **value**, que se ejecuta sin **await**, lo que obtenemos es el

valor devuelto por la función, pero «envuelto» en una promesa que deberá utilizarse con **.then()** para manejarse. Sin embargo, en **asyncValue** estamos obteniendo un tipo de dato **Number**
, guardando el valor directamente ya procesado, ya que **await** espera a que se resuelva lapromesa de forma asíncrona y guarda el valor.

> Esto hace que la forma de trabajar con **async/await**, aunque se sigue trabajando exactamente igual con promesas, sea mucho más fácil y trivial para usuarios que no estén acostumbrados a las promesas y a la asincronía en general, ya que el código «parece» síncrono.

> Recuerda que en el caso de querer controlar errores o promesas rechazadas, siempre podrás utilizar bloques **try/catch**.

¿Qué es una petición HTTP?

Un **navegador**, durante la carga de una página, suele realizar múltiples **peticiones HTTP** aun servidor para solicitar los archivos que necesita renderizar en la página. Es el caso de, en primer lugar, el documento **.html** de la página (*donde se hace referencia a múltiples archivos*) y luego todos esos archivos relacionados: los ficheros de estilos **.css**, las imágenes **.jpg**, **.png**, **.webp** u otras, los scripts **.js**, las tipografías **.ttf**, **.woff** o **.woff2**, etc.

Una **petición HTTP** es como suele denominarse a la acción por parte del navegador de solicitar a un servidor web un documento o archivo, ya sea un fichero **.html**, una imagen, una tipografía, un archivo **.js**, etc. Gracias a dicha petición, el navegador puede descargar ese archivo, almacenarlo en un **caché temporal de archivos del navegador** y, finalmente,mostrarlo en la página actual que lo ha solicitado.

HTTP define una gran cantidad de métodos que son utilizados para diferentes circunstancias:

- **GET:** Es utilizado únicamente para **consultar información** al servidor, muy parecidoa realizar un SELECT a la base de datos.
- **POST:** Es utilizado para solicitar la **creación de un nuevo registro**, es decir, algoque no existía previamente, es equivalente a realizar un INSERT en una base de datos.
- **PUT**: Se utiliza para **actualizar por completo un registro existente**, es decir, esparecido a realizar un UPDATE en la base de datos.
- **PATCH**: Este método es similar al método PUT, pues permite actualizar un registro existente, sin embargo, este se utiliza cuando **es necesario actualizar solo un fragmento del registro** y no en su totalidad, es equivalente a realizar un UPDATE ala base de datos.
- **DELETE**: Este método se utiliza para **eliminar un registro existente**, es similar aDELETE en la base de datos.
- **HEAD**: Este método se utiliza para **obtener información sobre un determinado recurso** sin retornar el registro. Este método se utiliza a menudo para probar la validez de los enlaces de hipertexto, la accesibilidad y las modificaciones recientes.

Peticiones HTTP mediante AJAX

Con el tiempo, aparece una nueva modalidad de realizar peticiones, denominada **AJAX** (*Asynchronous Javascript and XML*). Esta modalidad se basa en que la **petición HTTP** serealiza desde Javascript, de forma transparente al usuario, descargando la información y pudiendo tratarla **sin necesidad de mostrarla directamente en la página**.

Esto produce un interesante cambio en el panorama que había entonces, puesto que podemos hacer actualizaciones de contenidos **de forma parcial**, de modo que se actualice una página «**en vivo**», sin necesidad de recargar toda la página, sino solamente actualizadouna pequeña parte de ella, pudiendo utilizar Javascript para crear todo tipo de lógica de apoyo.

Originalmente, a este sistema de realización de peticiones HTTP se le llamó **AJAX**, dondela **X** significa **XML**, el formato ligero de datos que más se utilizaba en aquel entonces.
Actualmente, sobre todo en el mundo Javascript, se utiliza más el formato **JSON**, aunque por razones fonéticas evidentes (*y evitar confundirlo con una risa*) se sigue manteniendo eltérmino **AJAX**.

Métodos de petición AJAX

Existen varias formas de realizar **peticiones HTTP mediante AJAX**, pero las principalessuelen ser **XMLHttpRequest** y **fetch** (*nativas, incluidas en el navegador por defecto*), además de librerías como **axios** o **superagent**:

Peticiones HTTP con fetch

Fetch es el nombre de una nueva API para Javascript con la cual podemos realizar peticiones HTTP asíncronas utilizando promesas y de forma que el código sea un poco más sencillo. La forma de realizar una petición es muy sencilla, básicamente se trata de llamar a **fetch** y pasarle por parámetro la URL de la petición a realizar:

```
// Realizamos la petición y guardamos la promesaconst request = fetch("/robots.txt");

// Si es resuelta, entonces... request.then(function(response) { ... });
```

El **fetch()** devolverá una que será aceptada cuando reciba una respuesta y sólo será rechazada si hay un fallo de red o si por alguna razón no se pudo completar la petición. Elmodo más habitual de manejar las promesas es utilizando **.then()**. Esto se suele reescribirde la siguiente forma, que queda mucho más simple:

```
fetch("/robots.txt")
  .then(function(response) {
    /** Código que procesa la respuesta **/
  });
```

Al método **.then()** se le pasa una función callback donde su parámetro **response** es el objeto de respuesta de la petición que hemos realizado. En su interior realizaremos la lógica que queramos hacer con la respuesta a nuestra petición. A la función **fetch(url, options)** sele pasa por parámetro la **url** de la petición y, de forma opcional, un objeto **options** con opciones de la petición HTTP.

Vamos a examinar un código donde veamos un poco mejor como hacer la petición con **fetch**:

```
// Opciones de la petición (valores por defecto)const options = {
  method: "GET"
};

// Petición HTTP fetch("/robots.txt", options)
  .then(function(response){ return response.text();
  })
  .then(function(data){
    /** Procesar los datos **/
  });
```

Vamos a centrarnos ahora en el parámetro opcional **options** de la petición HTTP. En esteobjeto podemos definir varios detalles:

Campo	Descripción

method	Método HTTP de la petición. Por defecto, **GET**. Otras opciones: **HEAD**, **POST**, etc...
body	Cuerpo de la petición HTTP (El cuerpo del mensaje HTTP son los bytesde datos transmitidos en un mensaje de transacción HTTP inmediatamente después de los encabezados, si los hay). Puede ser devarios tipos: **String**, **FormData**, **Blob**, etc.
headers	Cabeceras o encabezados HTTP (parámetros que se envían en una petición o respuesta HTTP al cliente o al servidor para proporcionar información esencial sobre la transacción en curso). Por defecto, **{}**.
credentials	Modo de credenciales. Por defecto, **omit**. Otras opciones: **same-origin** e **include**.

Lo primero, y más habitual, suele ser indicar el método HTTP a realizar en la petición. Por defecto, se realizará un **GET**, pero podemos cambiarlos a **HEAD**, **POST**, **PUT** o cualquier otro tipo de método. En segundo lugar, podemos indicar objetos para enviar en el **body** dela petición, así como modificar las cabeceras en el campo **headers**:

```
const options = {
  method: "POST",
  headers: {
    "Content-Type": "application/json"
  },
  body: JSON.stringify(jsonData)
};
```

Por último, el campo **credentials** permite modificar el modo en el que se realiza la petición.Por defecto, el valor **omit** hace que no se incluyan credenciales en la petición, pero es posible indicar los valores **same-origin**, que incluye las credenciales si estamos sobre el mismo dominio, o **include** que incluye las credenciales incluso en peticiones a otros dominios.

Recuerda que estamos realizando peticiones relativas al **mismo dominio**. En el caso

de realizar peticiones a dominios diferentes obtendremos un problema de CORS (*Cross-OriginResource Sharing*) similar al siguiente:

> Access to fetch at 'https://otherdomain.com/file.json' from origin 'https://domain.com/' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource. If an opaque response serves your needs, set the request's mode to 'no-cors' to fetch the resource with CORS disabled.

Cabeceras (Headers)

Aunque en el ejemplo anterior hemos creado las cabeceras como un objeto genérico deJavascript, es posible crear un objeto **Headers** con el que trabajar:

```
const headers = new Headers(); headers.set("Content-Type", "application/json");
headers.set("Content-Encoding", "br");
```

Para ello, a parte del método **.set()** podemos utilizar varios otros para trabajar con cabeceras, comprobar su existencia, obtener o cambiar los valores o incluso eliminarlos:

Método	Descripción
.has(name)	Comprueba si la cabecera **name** está definida.
.get(name)	Obtiene el valor de la cabecera **name**.
.set(name, value)	Establece o modifica el valor **value** a la cabecera **name**.
.append(name, value)	Añade un nuevo valor **value** a la cabecera **name**.

.delete(name)	Elimina la cabecera **name**.

Respuesta de la petición HTTP

Si volvemos a nuestro ejemplo de la petición con **fetch**, observaremos que en el primer **.then()** tenemos un objeto **response**. Se trata de la respuesta que nos llega del servidorweb al momento de recibir nuestra petición:

```
// Petición HTTP fetch("/robots.txt", options)
  .then(function(response){ return response.text();
})
  .then(function(data){
    /** Procesar los datos **/
  });
```

El objeto **response** tiene una serie de propiedades y métodos que pueden resultarnos útilesal implementar nuestro código.

Propiedad	**Descripción**
.status	Código de error HTTP de la respuesta (100-599).
.statusText	Texto representativo del código de error HTTP anterior.
.ok	Devuelve **true** si el código HTTP es **200** (o empieza por **2**).
.headers	Cabeceras de la respuesta.

.url	URL de la petición HTTP.

Las propiedades **.status** y **statusText** nos devuelven el **código de error HTTP** de la respuesta en formato numérico y cadena de texto respectivamente.

Tenemos una propiedad **.ok** que nos devuelve **true** si el código de error de la respuesta esun valor del rango **2xx**, es decir, que todo ha ido correctamente. Así pues, tenemos una forma práctica y sencilla de comprobar si todo ha ido bien al realizar la petición:

```
fetch("/robots.txt")
  .then(function(response) {
    if (response.ok)
            return response.text();
  })
```

Por último, tenemos la propiedad **.headers** que nos devuelve las cabeceras de la respuestay la propiedad **.url** que nos devuelve la URL completa de la petición que hemos realizado.

Métodos de procesamiento

Por otra parte, la instancia **response** también tiene algunos **métodos** interesantes, la mayoría de ellos para procesar mediante una promesa los datos recibidos y facilitar el trabajo con ellos:

Método	Descripción
.text()	Devuelve una promesa con el texto plano de larespuesta.
.json()	Idem, pero con un objeto **json**. Equivalente a usar **JSON.parse()**.

.blob()	Idem, pero con un objeto **Blob** (binary large object).
.arrayBuffer()	Idem, pero con un objeto **ArrayBuffer** (buffer binariopuro).
.formData()	Idem, pero con un objeto **FormData** (datos deformulario).
.clone()	Crea y devuelve un clon de la instancia en cuestión.
Response.error()	Devuelve un nuevo objeto **Response** con un error de red asociado.
Response.redirect(url,code)	Redirige a una **url**, opcionalmente con un **code** de error.

Observa que en los ejemplos anteriores hemos utilizado **response.text()**. Este método indica que queremos procesar la respuesta como datos textuales, por lo que dicho métododevolverá un **string** con los datos en texto plano, facilitando trabajar con ellos de forma manual:

```
fetch("/robots.txt")
  .then(function(response){return response.text();})
  .then(function(data){console.log(data)});
```

Observa que en este fragmento de código, tras procesar la respuesta con **response.text()**,devolvemos una con el contenido en texto plano. Esta se procesa en el segundo **.then()**, donde gestionamos dicho contenido almacenado en **data**.

Ten en cuenta que tenemos varios métodos similares para procesar las respuestas. Por ejemplo, el caso anterior utilizando el método **response.json()** en lugar de **response.text()**sería equivalente al siguiente fragmento:

```
fetch("/contents.json")
  .then(function(response){ return response.text();})
  .then(function(data){
    const json = JSON.parse(data);
    console.log(json);
  });
```

Como se puede ver, con **response.json()** nos ahorraríamos tener que hacer el **JSON.parse()** de forma manual, por lo que el código es algo más directo.

Ejemplo utilizando promesas

Lo que vemos a continuación sería un ejemplo un poco más completo de todo lo que hemosvisto hasta ahora:

- Comprobamos que la petición es correcta con **response.ok**
- Utilizamos **response.text()** para procesarla
- En el caso de producirse algún error, lanzamos excepción con el código de error
- Procesamos los datos y los mostramos en la consola
- En el caso de que la sea rechazada, capturamos el error con el **catch**

```
// Petición HTTP fetch("/robots.txt")
  .then(function(response){
    if (response.ok)
            return response.text()
    else
            throw new Error(response.status);
  })
  .then(function(data){
    console.log("Datos: " + data);
  })
  .catch(function(err){
    console.error("ERROR: ", err.message)
  });
```

Sin embargo, aunque es bastante común trabajar con promesas utilizando **.then()**, también podemos hacer uso de **async/await** para manejar promesas, de una forma un poco más directa.

Ejemplo utilizando Async/await

Utilizar **async/await** no es más que lo que se denomina **azúcar sintáctico**, es decir, utilizar algo visualmente más agradable, pero que por debajo realiza la misma tarea.

Para ello, lo que debemos tener siempre presente es que un **await** sólo se puede ejecutar si esta dentrode una función definida como **async**.

En este caso, lo que hacemos es lo siguiente:

- Creamos una función **request(url)** que definimos con **async**
- Llamamos a **fetch** utilizando **await** para esperar y resolver la promesa
- Comprobamos si todo ha ido bien usando **response.ok**
- Llamamos a **response.text()** utilizando **await** y devolvemos el resultado

```
const request = async function(url){
  const response = await fetch(url);
  if (!response.ok)
    throw new Error("WARN", response.status);
  const data = await response.text();
  return data;
}

const resultOk = await request("/robots.txt");
const resultError = await request("/nonExistentFile.txt");
```

Una vez hecho esto, podemos llamar a nuestra función **request** y almacenar el resultado, usando nuevamente **await**. Al final, utilizar **.then()** o **async/await** es una cuestión de gustosy puedes utilizar el que más te guste.

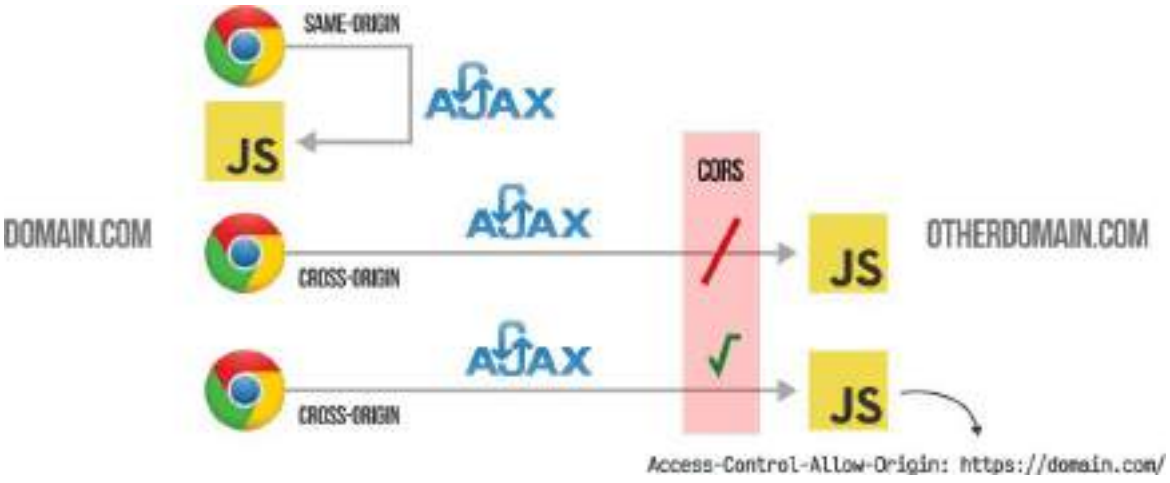
¿Qué es CORS?

Cross Origin (*origen cruzado*) es la palabra que se utiliza para denominar el tipo de peticiones que se realizan a un dominio diferente del dominio de origen desde donde se realiza la petición. De esta forma, por una parte tenemos las peticiones de **origen cruzado**(*cross-origin*) y las peticiones del **mismo origen** (*same-origin*).

CORS (*Cross-Origin Resource Sharing*) es un mecanismo o política de seguridad que permite controlar las peticiones HTTP asíncronas que se pueden realizar desde un navegador a un servidor con un dominio diferente de la página cargada originalmente. Estetipo de peticiones se llaman **peticiones de origen cruzado** (*cross-origin*).

Por defecto, los navegadores permiten enlazar hacia documentos situados en todo tipo de dominios si lo hacemos desde el HTML o desde Javascript utilizando la API DOM (*que a suvez está construyendo un HTML*). Sin embargo, no ocurre lo mismo cuando se trata de **peticiones HTTP asíncronas** mediante Javascript (*AJAX*), sea a través de **XMLHttpRequest**, de **fetch** o de librerías similares para el mismo propósito.

Utilizando este tipo de peticiones asíncronas, los recursos situados en dominios diferentesal de la página actual **no están permitidos** por defecto. Es lo que se suele denominar **protección de CORS**. Su finalidad es dificultar la posibilidad de añadir recursos ajenos enun sitio determinado.



Si intentamos realizar una petición asíncrona hacia otro dominio diferente, probablementeobtendremos un error de CORS similar al siguiente:

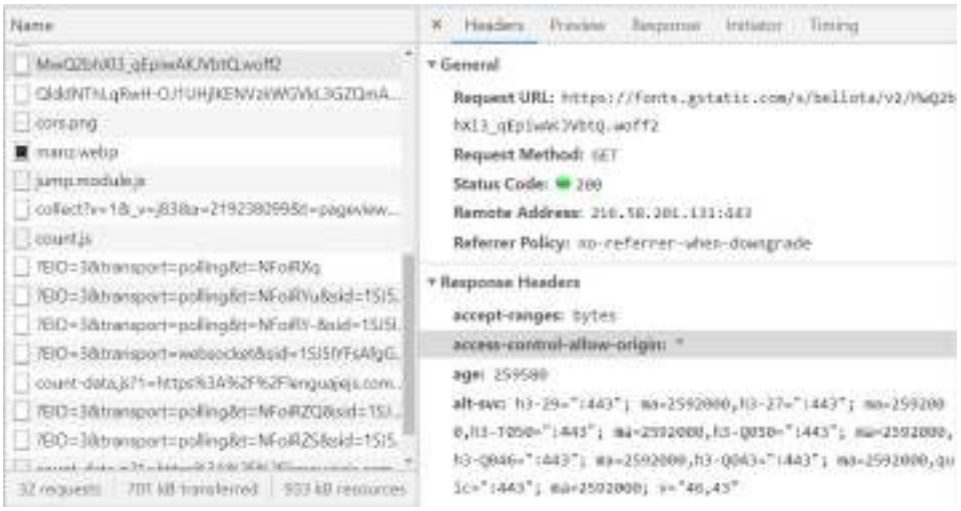
> Access to fetch at 'https://otherdomain.com/file.json' from origin 'https://domain.com/' has been blocked by **CORS policy**: No 'Access-Control-Allow-Origin' header is present on the requested resource. If an opaque response serves your needs, set the request's mode to 'no-cors' to fetch the resource with CORS disabled.

Access-Control-Allow-Origin

Como hemos comentado, las peticiones HTTP asíncronas de origen cruzado no están permitidas, pero existen formas de permitirlas. La más básica, probablemente, sea la de incluir una cabecera **Access-Control-Allow-Origin** en la respuesta de la petición, dondedebe indicarse el dominio al que se le quiere dar permiso:

```
Access-Control-Allow-Origin: https://domain.com/
```

De esta forma, el navegador comprobará dichas cabeceras y si coinciden con el dominio deorigen que realizó la petición, esta se permitirá. En el ejemplo anterior, la cabecera tiene el valor **https://domain.com/**, pero en algunos casos puede interesar indicar el valor *****.



> El asterisco ***** indica que se permiten peticiones de origen cruzado **a cualquier dominio**, algo que puede ser interesante cuando se tienen API públicas a las que quieres permitir el acceso al público en general, casos como los de Google Fonts o JSDelivr, por citar un ejemplo.

Estas cabeceras puedes verlas fácilmente accediendo a la pestaña **Network** del **DeveloperTools** del navegador. En esta sección te aparecerá una lista de peticiones realizadas por elnavegador en la página actual. Si seleccionamos una petición y accedemos al apartado de cabeceras (*Headers*), podremos examinar si existe la cabecera **Access-Control-Allow- Origin**:

CORS en entornos de desarrollo

Otra opción sencilla y rápida para no tener que lidiar con CORS **temporalmente** es la de instalar la **extensión Allow CORS**, disponible tanto Allow CORS para Chrome como AllowCORS para Firefox. Esta extensión deshabilita la **política CORS** mientras está instalada yactivada. Esta elección es equivalente a que todas las respuestas a las peticiones asíncronas realizadas tengan la mencionada cabecera con el valor *****. Obviamente, es importante recalcar que es una opción que **sólo nos funcionará en**

nuestro equipo y navegador, pero puede ser muy práctica para simplificar el trabajo en desarrollo.

Desactivar CORS en JavaScript: Agregar dentro de la propiedad **headers:**

```
'Access-Control-Allow-Origin' : 'https://sitio.com'
```

Ejemplo Async & Await en Javascript con Fetch y Axios:

En el siguiente ejemplo veremos cómo consumir un servicio externo de forma tal de poder traer y mostrar información en nuestro navegador mediante JavaScript. Para ello utilizaremos diferentes formas pero el objetivo será el mismo.

Algunos recursos que utilizaremos a lo largo del ejemplo:

CDN Axios:

https://cdnjs.com/libraries/axios

https://cdnjs.cloudflare.com/ajax/libs/axios/0.20.0/axios.min.js

{JSON} Placeholder

Recursos:

http://jsonplaceholder.typicode.com/

http://jsonplaceholder.typicode.com/posts

http://jsonplaceholder.typicode.com/users

http://jsonplaceholder.typicode.com/posts/1

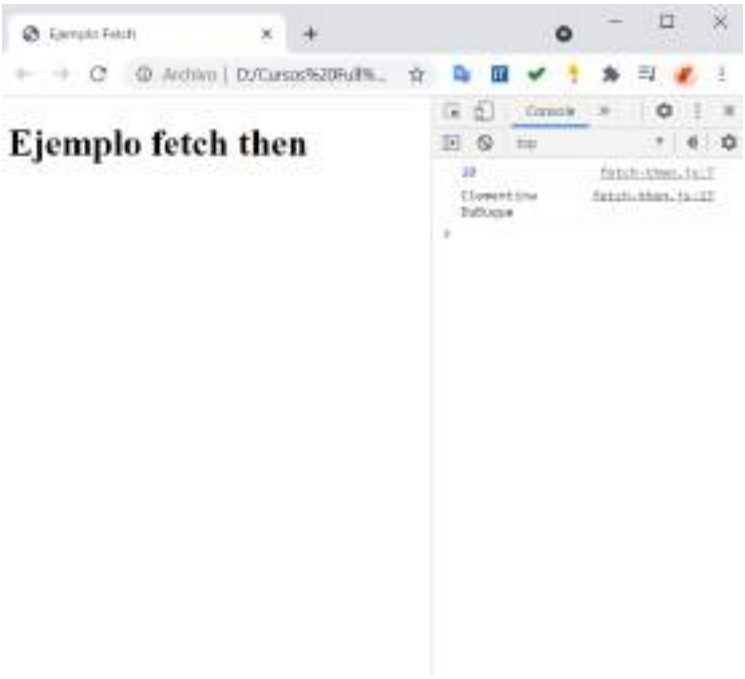
Ejemplo 1: uso de fetch y then

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Ejemplo Fetch</title>
</head>

<body>
    <h1>Ejemplo fetch then</h1>
    <script src="fetch-then.js"></script>
</body>

</html>
```

```
const getNombre = (idPost) => {
    fetch(`http://jsonplaceholder.typicode.com/posts/${idPost}`)
```

```
        .then(res => {
            return res.json()
        })
        .then(post => {
            console.log(post.userId);
            fetch(`http://jsonplaceholder.typicode.com/users/${post.us
erId}`)
                .then(res => {
                    return res.json()
                })
                .then(user => {
                    console.log(user.name)
                })
        })
}

getNombre(99);
```

Salida por la consola del navegador:



Ejemplo 2: uso de fetch con async & await

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Ejemplo Async Await</title>
</head>
```

```
<body>
    <h1>Ejemplo Async Await</h1>
    <script src="fetch-async-await.js"></script>
</body>

</html>
```

```
// Asincrónico
const getNombre = async(idPost) => {
    try {
        const resPost = await fetch(`http://jsonplaceholder.typicode.com/posts/${idPost}`)
        const post = await resPost.json()
        console.log(post.userId);

        const resUser = await fetch(`http://jsonplaceholder.typicode.com/users/${post.userId}`)
            // const resUser = await fetch(`http://jonplaceholder.typicode.com/users/${post.userId}`)
        const user = await resUser.json()
        console.log(user.name);
    } catch (error) {
        console.log("Ocurrió un error grave: ", error);
        // console.log(error);
        // console.log(error);
    }
}

getNombre(99);
```

La salida por consola es la misma que en el ejemplo anterior.

Ejemplo 3: uso de axios con async & await

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Ejemplo Axios</title>
</head>

<body>
    <h1>Ejemplo Axios</h1>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/axios/0.20.0/axios.min.js"></script>
    <script src="axios-async-await.js"></script>
```

```
</body>

</html>
```

```
// Asincrónico con Axios
const getNombre = async(idPost) => {
    try {
        const resPost = await axios(`http://jsonplaceholder.typicode.com/posts/${idPost}`)
        console.log(resPost);
        console.log(resPost.data);
        console.log(resPost.data.userId);

        const resUser = await axios(`http://jsonplaceholder.typicode.com/users/${resPost.data.userId}`)
            // const resUser = await axios(`http://jonplaceholder.typicode.com/users/${post.userId}`)
        console.log(resUser.data.name);
    } catch (error) {
        console.log("Ocurrió un error grave: ", error);
    }
}

getNombre(99);
```

La salida por consola es la misma que en los ejemplos anteriores.

Video “Ejemplo Async & Await en JavaScript con Fetch y Axios”:
https://drive.google.com/file/d/18wyg5_H1HlcIKuKi_SSuQXVxwZJQfITR/view?usp=sharing

Single Page Application (SPA)

En pocas palabras, SPA son las siglas de Single Page Application. Es un tipo de aplicación web donde todas las pantallas las muestra en la misma página, sin recargar el navegador.

Técnicamente, una SPA es un sitio donde existe un único punto de entrada, generalmente el archivo index.html. En la aplicación no hay ningún otro archivo HTML al que se pueda acceder de manera separada y que nos muestre un contenido o parte de la aplicación, toda la acción se produce dentro del mismo index.html.

Varias vistas, no varias páginas

Aunque solo tengamos una página, lo que sí tenemos en la aplicación son varias vistas, entendiendo por vista algo como lo que sería una pantalla en una aplicación de escritorio. En la misma página, por tanto, se irán intercambiando vistas distintas, produciendo el efecto de que tienes varias páginas, cuando realmente todo es la misma, intercambiando vistas.

El efecto de las SPA es que cargan muy rápido sus pantallas. De hecho aunque parezcan páginas distintas, realmente es la misma página, por eso la respuesta es muchas veces instantánea para pasar de una página a otra. Otra característica es que suele comunicar con un servidor y éste le envía los datos que debe mostrar “en crudo”. En crudo e refiere a que manda los datos puros, no mezclados con HTML u otro lenguaje para definir su presentación.

Para aclarar posibles malentendidos, SPA no quiere decir que no cambie la dirección de la barra de direcciones, es decir, la URL a la que accedes con el navegador. De hecho, es normal que al interaccionar con una SPA la URL que se muestra en la barra de direcciones del navegador vaya cambiando también. La clave es que, aunque cambie esta URL, la página no se recarga nunca. El hecho de cambiar esa URL es algo importante, ya que el propio navegador mantiene un historial de pantallas entre las que el usuario se podría mover, pulsando el botón de “atrás” en el navegador o “adelante”. Con todo ello, se consigue que el usuario pueda usar el historial como lo hace en una página normal, siendo que en una SPA la navegación entre la secuencia de páginas del historial se realiza realmente entre vistas de la aplicación.

Por qué una SPA ofrece una experiencia de usuario tan agradable

Al pesar muy poco los datos, mucho menos que si estuvieran mezclados dentro de un complejo código HTML y CSS para definir su presentación, las transmisiones son muy rápidas y las comunicaciones entre cliente y servidor se realizan muy fluidas. Nuevamente ayuda a que las páginas respondan muy velozmente al visitante, creando una experiencia de usuario muy agradable.

Nota: *El lenguaje con el que habitualmente comunicas los datos crudos desde el servidor hacia el cliente es JSON. De todos modos, nada impide usar otro lenguaje como XML, también muy popular en los Web Services. Aunque JSON se ha establecido como un estándar y el preferido por la mayoría de los desarrolladores, debido a diversos motivos. Entre ellos está que JSON es una notación de objeto Javascript, por lo que es algo muy cercano a la web. Es ligero y tiene soporte en la totalidad de los lenguajes usados en la web.*

Las páginas de gestión, o administración de cualquier tipo de servicio, paneles de control y cosas así son muy adecuadas para las SPA. El resultado es una aplicación web se comporta muy parecido a una aplicación de escritorio.

Lenguajes y tecnologías para producir una SPA

Una SPA se realiza en Javascript. No existe ningún otro tipo de lenguaje en el que puedas realizar una SPA, ya que básicamente es una aplicación web que se ejecuta del lado del cliente.

Nota: *Obviamente, también se realiza usando HTML + CSS, para la presentación, ya que son los lenguajes que entiende el navegador.*

Ya luego, dentro de Javascript, existen diversas librerías y frameworks que facilitan mucho el desarrollo de una SPA, entre los que podemos mencionar:

- Angular
- React
- Vue.js

Ahora bien, si quieres también expandir esta pregunta y abarcar la parte del lado del servidor, el backend, ahí tienes un nuevo abanico de posibilidades. Aunque creo que no hace falta ni entrar en ello porque realmente cualquier lenguaje backend te serviría para producir la parte del servidor, creando lo que sería un API REST que devuelve el JSON necesario para alimentar de datos a una SPA. Lo que debe quedar claro es que, cualquier SPA es agnóstica a cómo se desarrolle en el backend. Son dos proyectos independientes. Dicho de otra forma, en el lado del cliente resulta totalmente indiferente cómo se construyen del lado del servidor los datos recibidos.

Fuente:

https://escuelavue.es/

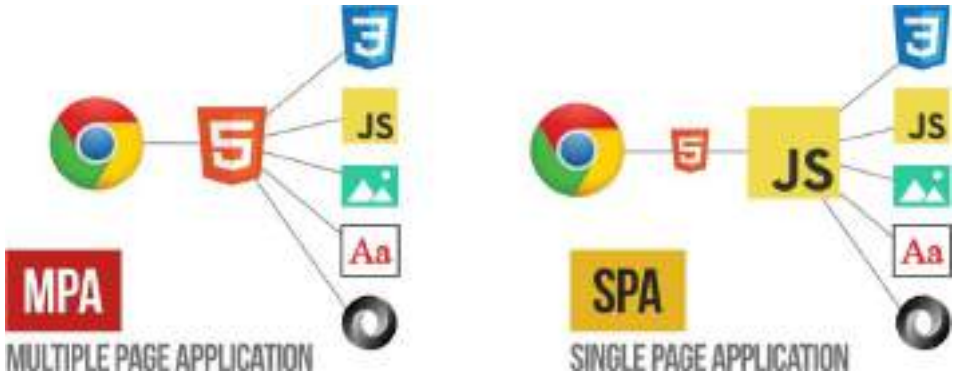
https://lenguajejs.com/javascript/peticiones-http/ajax/#mpa-multiple-page-application

https://lenguajejs.com/javascript/peticiones-http/ajax/#spa-single-page-application

MPA vs SPA

MPA: Multiple Page Application

Tradicionalmente, el sistema que se seguía para crear páginas o aplicaciones web se enmarcaba dentro de la categoría de páginas **MPA** (*Multiple Page Application*). Bajo este sistema, el navegador se descarga el fichero **.html**, lo lee y luego realiza las peticiones de los restantes archivos relacionados que encuentra en el documento HTML. Si el usuario pulsa en algún enlace, se descarga el **.html** de dicho enlace (*recargando la página completa*) y se repite el proceso.



Este sistema es el que se observa en páginas en las que vamos navegando mediante enlaces, y al hacer click en ellos, se recarga la página completa. Generalmente, es el que se utiliza frecuentemente en sitios web más tradicionales, los que usan mayoritariamente **backend**.

SPA: Single Page Application

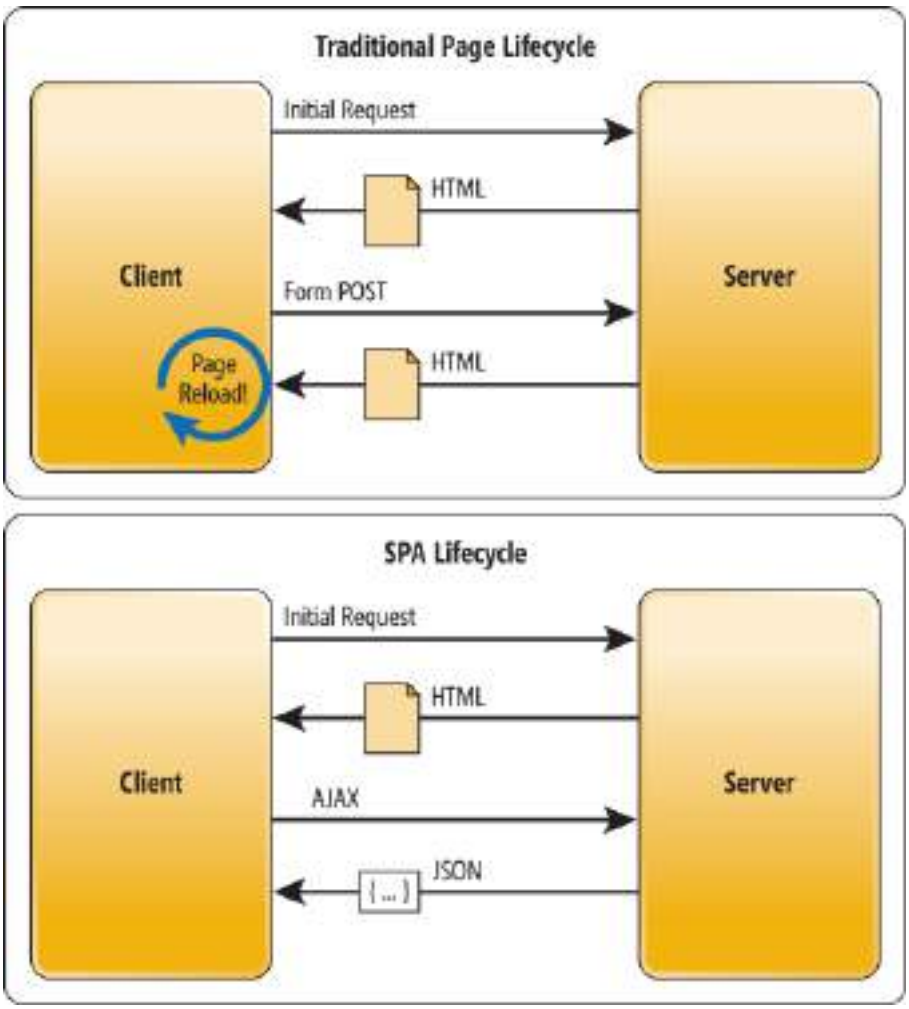
En el lado opuesto se encuentran las páginas de tipo **SPA** (*Single Page Application*). Se trata de un enfoque más moderno, donde el navegador descarga una versión mínima de

.html junto a un **.js** que se encargará de controlar toda la web. Realizará peticiones de los archivos relacionados junto a peticiones a archivos **.json** o **.js** con más información o nuevos contenidos, que mostrará en el navegador parcial o completamente, pero sin la necesidad obligatoria de recargar la página completamente.

Este sistema se utiliza mayoritariamente para construir aplicaciones web como dashboards o sitios de gestión en los que no necesitamos «navegar» a través de una serie de páginas.

Por ejemplo, páginas como WhatsApp Web, Twitter o Google Drive podrían ser ejemplos de **SPA**.

> Las páginas de tipo **SPA** son las que utilizan en la mayoría de los frameworks de Javascript,como por ejemplo, **React**, **Vue** o **Angular**.



Fuente:

https://escuelavue.es/

https://lenguajejs.com/javascript/peticiones-http/ajax/#mpa-multiple-page-application

https://lenguajejs.com/javascript/peticiones-http/ajax/#spa-single-page-application

Ejemplo SPA con Vue

Ejemplo para ver on-line: https://dreamy-pike-507001.netlify.app/

Ejemplos para descargar: AP11-vue-1-spa-basico.zip y AP11-vue-2-spa-con-estilos.zip

Fuente:

https://escuelavue.es/

https://lenguajejs.com/javascript/peticiones-http/ajax/#mpa-multiple-page-application

https://lenguajejs.com/javascript/peticiones-http/ajax/#spa-single-page-application