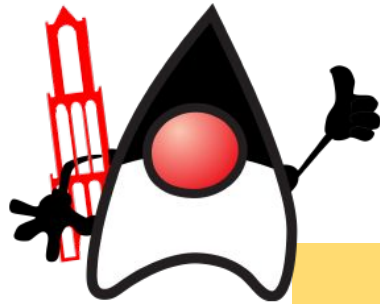




Curso FullStack Python

Codo a Codo 4.0



VUE

VUE.js





VUE.js

Con JavaScript nos encontramos que para hacer varias cosas necesitamos de mucho tiempo, esfuerzo y muchas líneas de código. Así como Bootstrap era un framework que me permitía resolver muy fácilmente cuestiones de estilos y estructuras, Vue me va a permitir resolver varias cuestiones del comportamiento porque es un **framework exclusivo de JS**.

Nos permitirá conectarnos con mi documento HTML en forma sencilla y podremos realizar cosas que desde JS puro y HTML implicarían un mayor esfuerzo.

Resulta un primer paso para otros frameworks de desarrollo de JS como pueden ser Angular y React, que requieren un paso más.

VUE está enfocado para armar aplicaciones de Single Page (esas que permiten navegar todo en una sola página). La idea de VUE es, por ejemplo, que no se tenga que cargar toda la información de una, que no se carguen todos los comentarios en un posteo, sino lo más relevante y a medida que vaya bajando se vayan cargando. Podemos actualizar **partes** del documento, y no toda la página que ahorra tiempo y recursos.

No nos sirve de nada ver un Framework si no tenemos las bases de JS, porque necesitamos los conocimientos previos.

VUE.js

Es un framework de JavaScript. La primera versión se lanzó en febrero de 2014.

Vue (*pronunciado /vju:/, como view*) es un **framework** progresivo de código abierto que se utiliza para desarrollar interfaces Web interactivas. A diferencia de otros frameworks, Vue está diseñado desde cero para ser utilizado incrementalmente y simplificar el desarrollo.

La **librería central** está enfocada solo en la *capa de visualización*, y es fácil de utilizar e integrar con otras librerías o proyectos existentes en desarrollos front-end.

Por otro lado, Vue también es perfectamente capaz de impulsar sofisticadas **Single-Page Applications (SPA)** cuando se utiliza en combinación con herramientas modernas y librerías de apoyo.

Las SPA son como una “sábana” donde a medida que vamos *scrolleando* nos movemos a distintas secciones de la misma página. El contenido no se va a cargar en forma completa, ya que es poco eficiente, repercute en la experiencia de usuario que va a esperar que se cargue todo el contenido cuando en realidad quiere ver lo que ya cargó.

***Ejemplo:** si un influencer tiene 10000 comentarios en cada posteo no los vamos a cargar todos juntos, sino que vamos a dividir por partes el documento HTML de forma tal que la información relevante se la cargue rápido al usuario.*

DOM Virtual

Vue.js utiliza **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.

*Desde JS haremos cambios en las estructuras de datos propias de JS, esa copia la va a comparar con el DOM y **sólo va actualizar los cambios**. Aquí radica la gran ventaja de trabajar con VUE, porque si tuviese que actualizar todo repercutiría en la performance.*

Modelo de enlazado de datos

Es la forma a través de la cual JavaScript se conecta (enlaza, comunica) con Vue.js., permitiendo comunicar el documento HTML con JS.

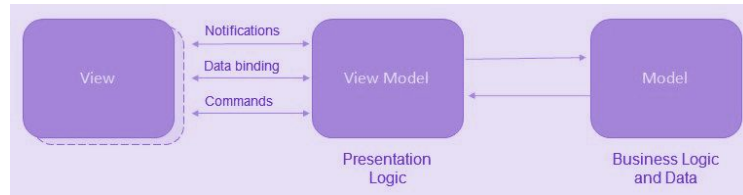
Data binding: A través de la función de data binding podremos manipular o asignar valores a atributos HTML, cambiar el estilo, asignar clases con la ayuda de la **directiva** de enlace **v-bind**.

Modelo de vista (*model-view-viewmodel o MVVM*)

El patrón modelo–vista–modelo de vista es un patrón de **arquitectura de software**. Se caracteriza por tratar de desacoplar lo máximo posible la interfaz de usuario de la lógica de la aplicación.

Sus elementos son:

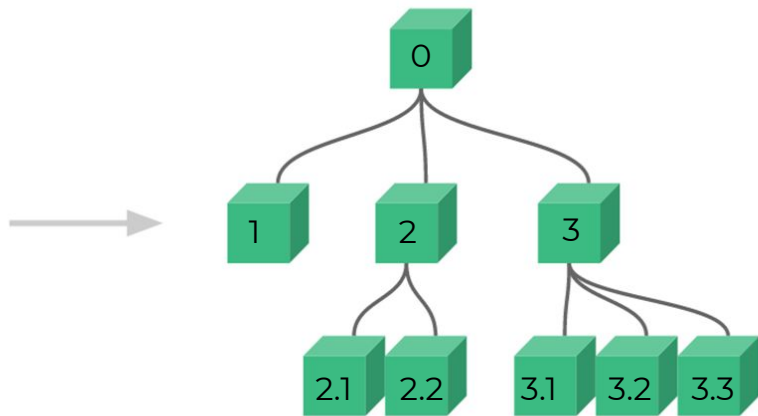
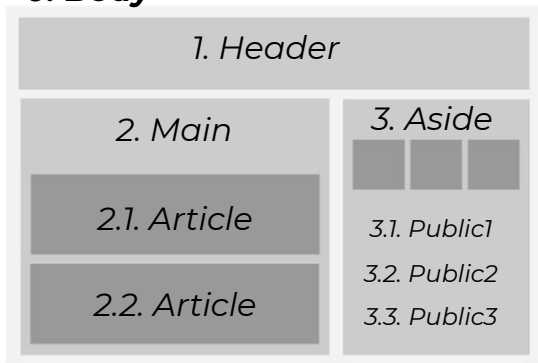
- **La vista:** Representa la información a través de los elementos visuales que la componen. Son activas, contienen comportamientos, eventos y enlaces a datos que, en cierta manera, necesitan tener conocimiento del modelo subyacente.
- **Modelo de vista:** Actor intermediario entre el **modelo** y la **vista**, contiene toda la lógica de presentación y se comporta como una abstracción de la interfaz. La comunicación entre la vista y el viewmodel se realiza por medio de los enlaces de datos.
- **El modelo:** Representa la capa de datos y contiene la información, pero nunca las acciones o servicios que la manipulan. No tiene dependencia con la vista.



Organización de componentes en VUE.js

Es común que una aplicación se organice en un árbol de componentes anidados:

0. Body



Vue te permite tomar una pagina web y dividirla en componentes cada uno con su HTML, CSS y JS necesario para generar esa parte de la página.

Permite hacer una **“intervención por partes”**, por ejemplo para intervenir sobre el header y footer, que es siempre el mismo

Comenzando con VUE.js

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@2.5.16/dist/vue.js"></script>
```

HTML

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

Nota: No recomendamos que los principiantes comiencen con vue-cli.

IMPORTANTE: Debemos colocar la referencia al CDN al final del <body> y antes de la referencia a nuestro archivo .js

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

JS

Esta constante me conecta VUE con mi HTML y tiene un objeto de tipo VUE. Dentro de las llaves voy a tener propiedades y valores.

Más información: https://www.w3schools.com/whatis/whatis_vue.asp
<https://es.vuejs.org/v2/guide/>

Hola mundo con VUE.js

En nuestro primer caso, tendremos en el documento HTML un elemento div con un ID que va a conectar con mi archivo JS

```
<body>
  <div id="app">
    <p>
      {{mensaje}}
    </p>
  </div>
  <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
  <script src="intro-vue.js"></script>
</body>
```

HTML

```
const app= new Vue({
  el: '#app',
  data: {
    mensaje: "Hola Mundo con Vue!",
    nombre: "Juan Pablo"
  }
})
```

JS

Hola Mundo con Vue!

*new Vue es el objeto de tipo VUE y lo que está entre {} es el contenido que yo quiero **cambiar** de la página.*

La conexión desde JS con mi documento HTML a través de VUE se llama **renderización declarativa**. Tiene que ver con enlazar el contenido del HTML que estoy presentando a través de VUE, ya no modificándolo desde JS a través del manejo del DOM, sino a través de VUE.

Ver ejemplo intro-vue (.html y .js)

Renderización declarativa (interpolación)

Nos permite insertar texto en el documento HTML, algún valor, propiedad o atributo. Por ejemplo, podremos agregar algún mensaje dentro de las etiquetas HTML.

VUE utiliza las **llaves dobles** para encerrar el dato que quiere mostrar `{{ }}`, similar a Template String de JS, que lo hace con `${}`

`{{ message }}`

Al partir del uso de la doble llave lo que vamos a estar haciendo es **vincular los datos con el DOM**, reaccionando a esos nuevos valores. Al cambiar esa réplica del DOM (*DOM virtual*) lo voy a ver reflejado en el DOM ya que framework al detectar un cambio lo actualiza.

Los **datos** y el **DOM** ahora están vinculados, y ahora **todo es reactivo** (sólo se modifica ante los cambios). Si cambio el valor de `app.message` a un valor diferente, debería ver que el ejemplo se ha renderizado con el nuevo valor que acaba de ingresar.

Directivas

Vue utiliza **directivas** para aplicar un comportamiento especial al DOM. Las directivas nos permiten enlazar VUE con nuestro HTML pero con los **atributos** de las etiquetas, no solo con el contenido.

Las directivas tienen el prefijo **v-** para indicar que son atributos especiales proporcionados por Vue.

- **v-text:** <https://es.vuejs.org/v2/api/#v-text>
- **v-bind:** <https://es.vuejs.org/v2/api/#v-bind>
- **v-if, v-else, v-elseif:** <https://es.vuejs.org/v2/api/#v-if>
- **v-for:** <https://es.vuejs.org/v2/api/#v-for>
- **v-show:** <https://es.vuejs.org/v2/api/#v-show>
- **v-model:** <https://es.vuejs.org/v2/api/#v-model>

Más directivas: <https://es.vuejs.org/v2/api/#Directivas>

v-bind: permite enlazar (*bindear*) una variable de Vue con un atributo específico de una etiqueta HTML.

```
<a href="#" v-bind:title="mensaje">mail@mail.com</a>
```

HTML

Con **v-bind:title** estamos modificando el atributo `title` dentro de la etiqueta `a`, mostrando el contenido de la propiedad **mensaje**

Más información clic [aquí](#)

[mail@mail.com](#)

Hola Mundo con Vue!

Directivas: v-for (renderización de una lista)

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

```
<ul id="example-1">
  <li v-for="fruta in frutas">
    {{ fruta.nombre }}
  </li>
</ul>
```

HTML

En este caso cargamos ítems desde una lista, pero con datos almacenados en un array desde JavaScript, aprovechando la directiva v-for. El contenido podrá ser dinámico, cargándose en función de algo previamente almacenado.

***fruta** es cada elemento de la lista, mientras que **frutas** es el array en sí.*

```
var example1 = new Vue({
  el: '#example-1',
  data: {
    frutas: [
      {nombre:"naranja"},
      {nombre:"banana"},
      {nombre:"pera"}
    ]
  }
})
```

JS

- naranja
- banana
- pera

Directivas: v-if, v-else, v-elseif

Podemos establecer que el contenido se muestre dependiendo de alguna condición.

Ejemplo: En un listado de productos podemos hacer que aquellos que tengan un stock igual a 0 nos avise de alguna manera:

```
<div id="ejemplo">
  <h1> {{ titulo }}</h1>
  <ul>
    <li v-for="fruta in frutas">
      {{ fruta.nombre }} - {{ fruta.cantidad }}
    </li>
  </ul>
</div>
```

HTML

```
var ejemplo_vIf_vFor = new Vue({
  el: '#ejemplo',
  data: {
    titulo: "Ejemplo v-if y v-for",
    frutas: [
      {nombre:"naranja", cantidad: 10},
      {nombre:"banana", cantidad: 0},
      {nombre:"pera", cantidad: 3}
    ]
  }
})
```

JS

*En este ejemplo iteramos sobre el array de frutas, mostrando de ese objetos dos propiedades: **nombre** y **cantidad**.*

Utilizaremos un condicional **v-if** para determinar qué elementos no tienen stock (= 0).

Ejemplo v-if y v-for

- naranja - 10
- banana - 0
- pera - 3

Directivas: v-if, v-else, v-elseif

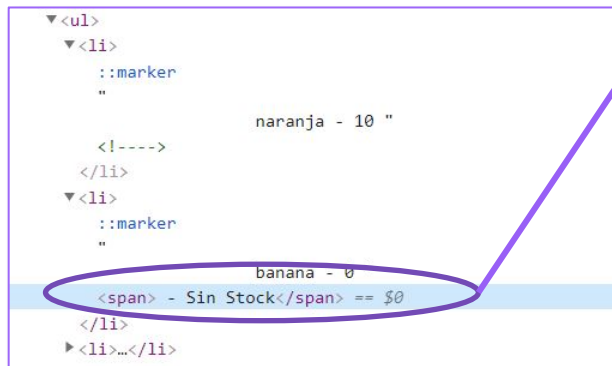
Agregaremos una etiqueta **** que aparecerá solamente en caso de cumplirse una condición:

```
<li v-for="fruta in frutas">
  {{ fruta.nombre }} - {{ fruta.cantidad }} <span v-if="fruta.cantidad===0"> - Sin Stock</span>
</li>
```

HTML

Cuando dentro del array la propiedad cantidad del elemento fruta sea igual a 0 se mostrará el texto "Sin Stock"

- naranja - 10
- banana - 0 - Sin Stock
- pera - 3



*Si inspeccionamos el documento veremos que el **** solo aparece en el segundo ítem, Esto lo resuelve VUE a través de JavaScript.*

Ver ejemplo: v-if for (.html y .js)

Directivas: v-if, v-else, v-elseif

Ampliaremos el ejemplo anterior incorporando más elementos al array y estableciendo otras condiciones:

- Stock = 0: Sin stock
- Stock < 5: Stock bajo
- Stock >=5 Stock alto

Para esto emplearemos v-else-if y v-else:

```
<li v-for="fruta in frutas">
  {{ fruta.nombre }} - {{ fruta.cantidad }}
  <span v-if="fruta.cantidad===0"> - Sin Stock</span>
  <span v-else-if="fruta.cantidad<5"> - Stock Bajo</span>
  <span v-else="fruta.cantidad>=5"> - Stock Alto</span>
</li>
```

HTML

*El **v-for** iterará sobre cada elemento y determinará con los condicionales cuál es la situación de cada elemento (sin Stock Stock Bajo o Stock Alto)*

Ver ejemplo: v-if for2 (.html y .js)

Las directivas v-model y v-on

La directiva **v-model** establece un enlace bidireccional, es decir, vincula el valor de los elementos HTML a los **datos** de la aplicación. La directiva **v-on** permite escuchar eventos DOM y ejecutar algunas instrucciones de JavaScript cuando se activan.

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

HTML

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

JS

Aquí en el input, lo que ingrese el usuario, modifica el message y el message modifica el 1er párrafo. A medida que se escriba en el input ese cambio se verá reflejado en el párrafo.

Hello Vue!

Hello Vue!

```
<div id="example-1">
  <button v-on:click="counter += 1">Add 1</button>
  <p>Se ha hecho clic en el botón de arriba {{ counter }} veces.</p>
</div>
```

HTML

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

JS

*Se asocia la directiva **v-on** al evento clic y se incrementa en 1 el valor de la propiedad counter*

Add 1

Se ha hecho clic en el botón de arriba 3 veces.

Ver ejemplos v-model y v-on (.html y .js)

Agregando elementos a la instancia VUE

Tomaremos el ejemplo del array de frutas y sumaremos una nueva propiedad a la instancia VUE, dentro de la propiedad data:

```
nuevaFruta: '',
```

Así como tenemos datos / información asociada a mi instancia también podemos tener métodos o funciones:

```
methods: {  
  agregarFruta(){  
    this.frutas.push({ nombre: this.nuevaFruta, cantidad: 0 })  
  }  
}
```

JS

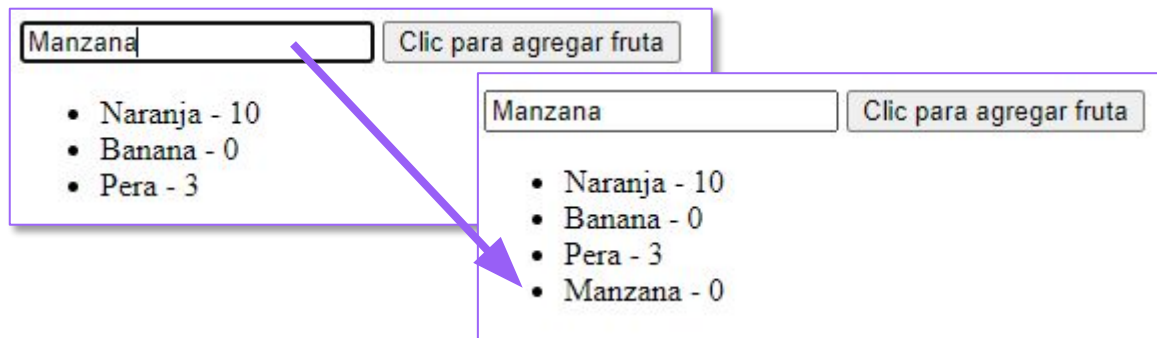
Crearemos un método que utilizará el método `push` para agregar un nuevo elemento al array de objetos (frutas) de la misma manera que fueron agregadas antes, con los pares clave: valor (respetando la estructura). El nombre de la fruta es el de **nuevaFruta** (propiedad de la instancia de VUE) y utilizamos el **this** para hacer referencia, justamente, a esa instancia. Además sumamos que la `nuevaFruta` inicie con cantidad 0.

Agregando elementos a la instancia VUE

En el documento HTML agregaremos un input y un botón para agregar la fruta que deseamos al array de objetos:

```
<input type="text" v-model="nuevaFruta">  
<button v-on:click=agregarFruta()>Clic para agregar fruta</button>
```

HTML



¿Cómo funciona todo esto?

1. El botón "Clic para agregar..." tiene asociado una directiva `v-on` que llama al método `agregarFruta()` de la instancia de VUE creada en JS.
2. En JS el método `agregarFruta()` utilizaba un `push` para agregar esa nueva fruta al array de objetos.
3. Además, a través de la directiva `v-model` se conectan el input con la propiedad de la instancia de VUE. Lo que suceda en el input se va a ver reflejado en la propiedad y viceversa (**comunicación bidireccional**).

Agregando elementos a la instancia VUE

Una mejora que se puede hacer es que se limpie la caja de texto cuando agregamos un elemento. Esto se logra agregando en el método agregarFruta() esta instrucción:

```
this.nuevaFruta= '';
```

JS

Podemos hacer una mejora para que no nos permita agregar una fruta hasta que no se haya completado la caja de texto:

```
agregarFrutaConIF(){  
  if (this.nuevaFruta !== "") {  
    this.frutas.push({ nombre: this.nuevaFruta, cantidad: 0 });  
    this.nuevaFruta= '';  
  }  
}
```

JS

Eventos: Modificadores de teclas

Evento keyUp: Nos permite disparar un método una vez que se levanta una tecla. Por ejemplo Enter:

```
<input type="text" v-model="nuevaFruta" @keyup.enter="agregarFrutaConIF">
```

HTML

Al presionar Enter se dispara el método que me permite agregar una fruta al array de objetos

Ver ejemplo eventos-key (.html y .js)

Fuente: <https://vuejs.org/v2/guide/events.html#Key-Modifiers>

Otros eventos: <https://es.vuejs.org/v2/guide/events.html>

Computados

Me van a permitir que VUE agregue una **función** que realice alguna *operación matemática*. La potencia de los computados es que este método se va a ejecutar cuando haya un cambio en el HTML.

En este ejemplo crearemos una función que realice la sumatoria total de frutas. Va a recorrer el array y sumar las cantidades, luego las va a mostrar.

```
computed: {  
  sumarFrutas() { //Muestra sumatoria total de cantidades de frutas.  
    this.total = 0;  
    for (fruta of this.frutas) {  
      this.total += fruta.cantidad; //acumulador  
    }  
    return this.total;  
  }  
}
```

JS

El **for** recorrerá las cantidades del array **frutas**. Se utiliza **fruta.cantidad** y se va agregando a total con **this.total +=fruta.cantidad**; Además hay un **return** que devuelve el total.

Dentro de **data** necesitamos inicializar esa propiedad (variable) total.

total: 0

Computados

Agregaremos la etiqueta H4 en el HTML para que me muestre el total calculado:

```
<h4>Total: {{sumarFrutas}}</h4>
```

HTML

Debemos tomar el valor de retorno del método, no la variable Total

Ejemplo computados

Clic para agregar fruta

- Naranja - 10
- Banana - 12
- Pera - 5

Total: 27

*Cuando haga un cambio en el documento HTML la función que está dentro de **computed** se va a ejecutar*

Ver ejemplo computados (.html y .js)

Computados

Aprovechando la capacidad de VUE de actualizar solamente aquello que se modifica en el documento HTML podremos desarrollar un proyecto que sume frutas a medida que se van agregando, como si fuera un “carrito de compras”:

```
<ul>
  <li v-for="fruta in frutas">
    {{ fruta.nombre }} - <input type="number" v-model.number="fruta.cantidad">
  </li>
</ul>
<h4>Total: {{sumarFrutas}}</h4>
```

JS

Si hacemos una modificación en el input se modifican los datos y se ven en el HTML (enlace bidireccional). Utilizamos v-model.number porque lo que introduciremos son valores numéricos. Si no lo colocáramos concatenaría los datos al no estar parseados, los guarda como un string y concatena.

Ejemplo computados

Clic para agregar fruta

- Naranja -
- Banana -
- Pera -

Total: 31

Ver ejemplo computados-2 (.html y .js)

Computados

También podremos agregar botones para sumar y restar valores:

```
<li v-for="fruta in frutas">
  {{ fruta.nombre }} - <input type="number" v-model.number="fruta.cantidad">
  <button @click=fruta.cantidad++>+</button>
  <button @click=fruta.cantidad-->-</button>
</li>
```

JS

Utilizamos @click que reemplaza a v-on y agregando el acumulador de la propiedad cantidad del objeto fruta (para sumar o restar) dentro de cada botón.

Ejemplo computados

Clic para agregar fruta

- Naranja -
- Banana -
- Pera -

Total: 30

[Ver ejemplo computados-3 \(.html y .js\)](#)

Creación de componentes

Los componentes nos permiten dividir en partes nuestro código HTML. Por ejemplo podemos tener resuelta nuestra barra de navegación en un componente externo y utilizarla. Hasta ahora sabíamos que podíamos crear un objeto de tipo VUE y utilizar sus propiedades en el documento HTML. Ahora vamos a **crear componentes** que son los que van a contener lo que queremos mostrar en el documento HTML:

```
var app = new Vue({  
  el: '#app',  
  data: {  
    msj: "Hola Mundo!"  
  }  
})
```

JS

```
<div id="app">  
  <h1>{{msj}}</h1>  
</div>
```

HTML

Hola Mundo!

Esto ahora vamos a modificarlo con la creación de componentes.

Importante: Aunque creamos un componente de VUE es requisito mantener esta referencia al ID #app (en este caso), aunque no tenga **data**. En caso contrario no funciona el componente.

Creación de componentes

Vamos a crear un componente, que aprovechando una etiqueta HTML creada por nosotros. Podemos crear una etiqueta, por ejemplo **<saludo>** y utilizarla en nuestro documento. Si bien no es válida en HTML la vamos a hacer válida a partir de su creación.

```
Vue.component('saludo', {  
  template: "<h1>Hola (estático desde template)</h1>"  
})
```

JS

Hola (estático desde template)

```
<div id="app">  
  <saludo></saludo>  
</div>
```

HTML

```
<div id="app">  
  <h1>Hola (estático desde template)</h1>  
</div>
```

*Como primer parámetro le paso el nombre del componente y como segundo parámetro el objeto. **Template** es una de las propiedades más importantes de los componentes.*

Sin embargo, esta referencia es estática. Cuando trabajamos con componentes al **data**, que se agregaba como propiedad en la instancia de VUE, lo vamos a incluir en el propio componente (antes **data** era una propiedad de la instancia de VUE, pero a la vez era un objeto con propiedades).

Creación de componentes

Entonces **data** ahora será un método que retorna un valor, por ejemplo otro saludo:

```
Vue.component('saludodos', {  
  template: "<h1>{{msj}}</h1>",  
  data(){  
    return {  
      msj: 'Hola (dinámica y como componente)'  
    }  
  }  
})
```

JS

En este caso el dato lo toma desde el mismo componente

Hola (estático desde template)

Hola (dinámica y como componente)

```
<div id="app">  
  <saludo></saludo>  
  <saludodos></saludodos>  
</div>
```

```
▼ <div id="app">  
  <h1>Hola (estático desde template)</h1>  
  <h1>Hola (dinámica y como componente)</h1>
```

Utilizar el template de esta forma nos limita a una única línea. Si queremos poner más de una línea en HTML utilizaremos los **backticks** (*comillas invertidas*).

Creación de componentes

Con las comillas invertidas podremos escribir *más de una línea*. Es importante también saber que los templates deben ir dentro de un **contenedor** (en este caso un **div**):

```
Vue.component('saludotres', {  
  template: `  
    <div>  
      <h1>{{msj}}</h1>  
      <h2>{{titulo}}</h2>  
    </div>  
  `,  
  data(){  
    return {  
      msj: 'Hola (dinámica y como componente)',  
      titulo: "Título dinámico"  
    }  
  }  
})
```

JS

```
<div id="app"> == $0  
  <div>  
    <h1>Hola (dinámica y como componente)</h1>  
    <h2>Título dinámico</h2>  
  </div>  
</div>
```

Hola (dinámica y como componente)

Título dinámico

```
<div id="app">  
  <saludotres></saludotres>  
</div>
```

HTML

Con la extensión **es6-string-html** podemos formatear las etiquetas HTML que estén dentro de un string, eso visualmente ayuda mucho.

Creación de componentes

Otro ejemplo de uso de un componente utilizando botones y contadores:

```
Vue.component('contador', {
  template: `
    <div>
      <h3>Cantidad: {{num}}</h3>
      <button @click="num++">+</button>
      <button @click="num--">-</button>
    </div>
  `,
  data() {
    return {
      num: 0
    }
  }
})
```

JS

```
<div id="app">
  <contador></contador>
  <h2>Otra instancia del mismo componente
  (son independientes)</h2>
  <contador></contador>
</div>
```

HTML

Cantidad: 6



Otra instancia del mismo componente (son independientes)

Cantidad: -2



```
<div id="app">
  <div>
    <h3>Cantidad: 6</h3>
    <button>+</button>
    <button>-</button>
  </div>
  <h2>Otra instancia del mismo componente (son independientes)</h2>
  <div>
    <h3>Cantidad: -2</h3>
    <button>+</button>
    <button>-</button>
  </div>
</div>
```

Creamos un componente llamado **contador** que lo vinculamos con la etiqueta homónima. Este componente tiene tres líneas: un **h3** que contiene un texto fijo y un elemento variable que se incrementa de acuerdo a la instrucción que tiene cada botón para incrementar o decrementar de a 1.

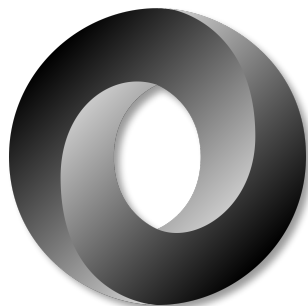
JSON: JavaScript Object Notation

JSON es una sintaxis propia de objetos tipo JavaScript utilizada para **almacenar e intercambiar** datos. Es texto, escrito con notación de objetos JavaScript, con un formato determinado.

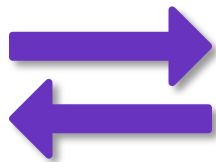
Intercambio de datos

Al intercambiar datos entre un navegador y un servidor, los datos **solo pueden ser texto**.

Dado que JSON trabaja como texto podemos convertir cualquier objeto JavaScript en JSON y enviar JSON al servidor. También podemos convertir cualquier JSON recibido del servidor en objetos JavaScript. De esta forma podemos trabajar con los datos como objetos JavaScript, sin complicados análisis ni traducciones.



JSON



*Cualquier JSON podrá ser
convertido a JS y cualquier texto
con el formato correspondiente
podrá ser convertido a JSON*



JSON: JavaScript Object Notation

Convirtiendo de JavaScript a JSON:

Si tiene datos almacenados en un objeto JavaScript, puede convertir el objeto en JSON con ***JSON.stringify()***:

```
var myObj = { name: "John", age: 31, city: "New York" };  
var myJSON = JSON.stringify(myObj);  
// myJson= {"name":"John","age":31,"city":"New York"}
```

JS

Convirtiendo de JSON a JavaScript:

Si tiene datos almacenados en un JSON, puede convertir el objeto en JavaScript con ***JSON.parse()***:

```
var myObj1=JSON.parse(myJSON);  
//myObj1= { name: "John", age: 31, city: "New York" }
```

JS

JSON: JavaScript Object Notation

Reglas de sintaxis JSON:

La sintaxis JSON se deriva de la sintaxis de notación de objetos de JavaScript:

- Los datos están en pares de nombre / valor
- Los datos están separados por comas
- Las {} contienen objetos
- Los corchetes contienen Array

```
myJson= {"name":"John","age":31,"city":"New York"}
```

JS

En JSON , los valores deben ser uno de los siguientes tipos de datos:

- string
- number
- object (JSON object)
- array
- boolean
- null

El tipo de archivo de los archivos JSON es ".json"

JSON: JavaScript Object Notation

JSON

```
// Ejemplo de JSON
{
  "employees": [
    { "firstName": "John", "lastName": "Doe" },
    { "firstName": "Anna", "lastName": "Smith" },
    { "firstName": "Peter", "lastName": "Jones" }
  ]
}

// Otro ejemplo:
{
  "firstName": "John",
  "lastName": "Doe",
  "middlename": null,
  "edad": 30,
  "Hijos": ["John", "Anna", "Peter"]
}
```

1er ejemplo: En la propiedad “empleados” hay un array de 3 elementos y dentro de cada uno tengo objetos separados por comas. Cada objeto tiene un primer nombre y un apellido.

2do ejemplo: Vemos que el objeto JSON tiene un primer nombre asociado, junto con otras propiedades. Además hay una propiedad Hijos que a su vez tiene un array.

Ver ejemplos [ejemplo.json](#) y [ejemplo2.json](#)

JSON: JavaScript Object Notation

Otro ejemplo JSON: Ingresar aquí:

<https://mdn.github.io/learning-area/javascript/oojs/ison/superheroes.json>

```
{
  "squadName" : "Super Hero Squad",
  "homeTown" : "Metro City",
  "formed" : 2016,
  "secretBase" : "Super tower",
  "active" : true,
  "members" : [
    {
      "name" : "Molecule Man",
      "age" : 29,
      "secretIdentity" : "Dan Jukes",
      "powers" : [
        "Radiation resistance",
        "Turning tiny",
        "Radiation blast"
      ]
    },
    {
      "name" : "Madame Uppercut",
      "age" : 39,
      "secretIdentity" : "Jane Wilson",
      "powers" : [
        "Million tonne punch",
        "Damage resistance",
        "Superhuman reflexes"
      ]
    }
  ]
}
```

Google Chrome



```
squadName:      "Super Hero Squad"
homeTown:       "Metro City"
formed:         2016
secretBase:     "Super tower"
active:         true
members:
  ▼ 0:
    name:        "Molecule Man"
    age:         29
    secretIdentity: "Dan Jukes"
    powers:
      ▼ 0: "Radiation resistance"
      1:  "Turning tiny"
      2:  "Radiation blast"
  ▼ 1:
    name:        "Madame Uppercut"
    age:         39
    secretIdentity: "Jane Wilson"
    powers:
      ▼ 0: "Million tonne punch"
      1:  "Damage resistance"
      2:  "Superhuman reflexes"
```

Firefox Developer



Otro ejemplo JSON: <https://github.com/midesweb/taller-angular/blob/master/11-mi-API/peliculas.json>

JSON: JavaScript Object Notation

API pública Randomuser: <https://randomuser.me/api>

Muestra datos de usuarios aleatorios, se utiliza para hacer pruebas. Es un string de JSON con un formato particular. Devuelve un usuario aleatorio, un array con un solo elemento.

Conviene leerlo desde **Firefox Developer Edition**, ya que la visualización es más simple.

Nosotros podremos **consumir la API**, esto quiere decir leerla y traerla a nuestra aplicación.

Fetch

La API Fetch proporciona una interfaz JavaScript para acceder y manipular partes del canal HTTP, tales como peticiones y respuestas.

También provee un método global **fetch()** que proporciona una forma fácil y lógica de obtener recursos de forma asíncrona por la red.

Más información sobre Fetch: https://developer.mozilla.org/es/docs/Web/API/Fetch_API/Utilizando_Fetch

Una petición básica de fetch es realmente simple de realizar:

```
fetch('https://api.coindesk.com/v1/bpi/currentprice.json')  
  .then(response => response.json())  
  .then(data => console.log(data));
```

JS

Aquí estamos recuperando un archivo JSON a través de red y mostrando en la consola. El uso de **fetch()** más simple toma un argumento (la ruta del recurso que quieres buscar) y devuelve un objeto **Promise** conteniendo la respuesta, un objeto **Response**.

Esto es, por supuesto, una respuesta HTTP no el archivo JSON. Para extraer el contenido en el cuerpo del JSON desde la respuesta, usamos el método **json()**

Fetch

Con Fetch, así como podemos leer información que proviene de una API externa vamos a poder leer un archivo de texto y mostrarlo por consola.

En el siguiente ejemplo utilizamos clases de Bootstrap y dentro del **body** incorporaremos dos etiquetas **divs**: la primera para el título y el botón que me permitirá traer el contenido y la segunda para el contenido en cuestión:

```
<div class="container my-5 text-center">  
  <h1>Ejemplo Fetch</h1>  
  <button class="btn-danger w-100" onclick="traer()">Obtener</button>  
</div>  
<div class="mt-5" id="contenido">  
  <!-- Insertaremos contenido del archivo de texto para utilizar Fetch -->  
</div>
```

HTML

Ejemplo Fetch

Obtener

Importante: Este ejemplo solamente funcionará con LiveServer de VSC

[Ver ejemplo fetch.html](#)

Fetch

Crearemos un script dentro del mismo HTML con el siguiente código:

```
<script>
  var contenido = document.querySelector('#contenido');
  function traer() {
    fetch('texto.txt')
      .then(data => data.text())
      .then(data => {
        console.log(data)
        contenido.innerHTML= `${data}`
      })
  }
</script>
```

JS

Fetch me va a permitir hacer "promesas":

Se denominan promesas porque puede que no se ejecuten, porque por ejemplo si del otro lado no tengo nada el .then no va a ocurrir.

Explicación:

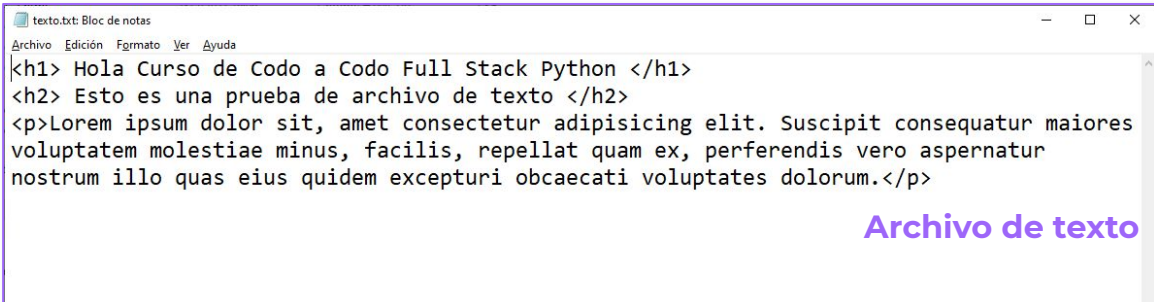
La variable contenido me permite traer el elemento con el ID #contenido.

La función traer() me permite obtener los datos:

- Con **fetch('texto.txt')** hago referencia al archivo que quiero traer.
- Con **.then** establezco que la variable data va a guardar el contenido traído por fetch a través del método .text()
- Con el siguiente **.then** muestro por consola y agrego a mi HTML el contenido de data (traído como **template string**)

Fetch

El resultado final será el siguiente:



```
texto.txt: Bloc de notas
Archivo Edición Formato Ver Ayuda
<h1> Hola Curso de Codo a Codo Full Stack Python </h1>
<h2> Esto es una prueba de archivo de texto </h2>
<p>Lorem ipsum dolor sit, amet consectetur adipisicing elit. Suspendisse
consequatur maiores voluptatem molestiae minus, facilis, repellat quam ex,
perferendis vero aspernatur nostrum illo quas eius quidem excepturi
obcaecati voluptates dolorum.</p>
```

Archivo de texto

Ejemplo Fetch

Ejemplo final

Obtener

Hola Curso de Codo a Codo Full Stack Python

Esto es una prueba de archivo de texto

Lorem ipsum dolor sit, amet consectetur adipisicing elit. Suspendisse consequatur maiores voluptatem molestiae minus, facilis, repellat quam ex, perferendis vero aspernatur nostrum illo quas eius quidem excepturi obcaecati voluptates dolorum.

[Ver ejemplo fetch.html](#)

Fetch: consumir API externa

En este caso consumiremos la API <https://randomuser.me/api> que nos permitirá traer datos escritos en formato tipo JSON:

```
function traer_dos() {  
  //fetch('texto.txt')  
  fetch('https://randomuser.me/api')  
    .then(res => res.json())  
    .then(res => {  
      console.log(res)  
      console.log(res.results[0].email)  
      // contenido.innerHTML= `${res.results[0].email}`  
  
      contenido.innerHTML = `  
          
        <p>Nombre: ${res.results[0].name.first}</p>  
  
        <p>Mail: ${res.results[0].email}</p>  
      `;  
    })  
}
```

JS

Explicación:

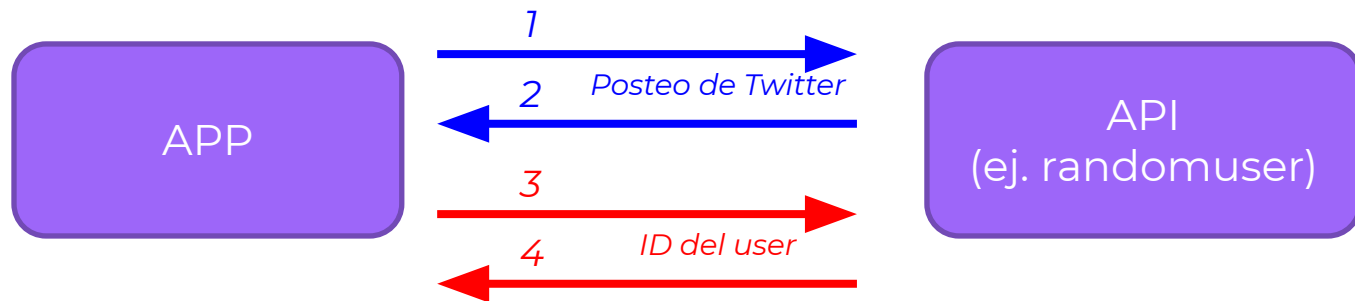
La función traer_dos() me permite obtener los datos, pero esta vez desde una API externa:

- Con **fetch('https:..')** hago referencia a la API externa.
- En **.then** establezco que la variable res va a guardar el dato desde el método json(), que extrae el contenido del archivo **.json**
- Con el siguiente .then muestro por consola todos los resultados y agrego a mi HTML de los resultados en la posición 0 la imagen, el nombre y el mail.

Ver ejemplo fetch.html + inspeccionar para ver los cambios

Otro ejemplo de uso de Fetch con API externa

Este ejemplo es más completo. Consultará con una Api más de una vez, enviará un requerimiento, nos devolverá el dato y en base a esa respuesta voy a volver a enviar esa solicitud.



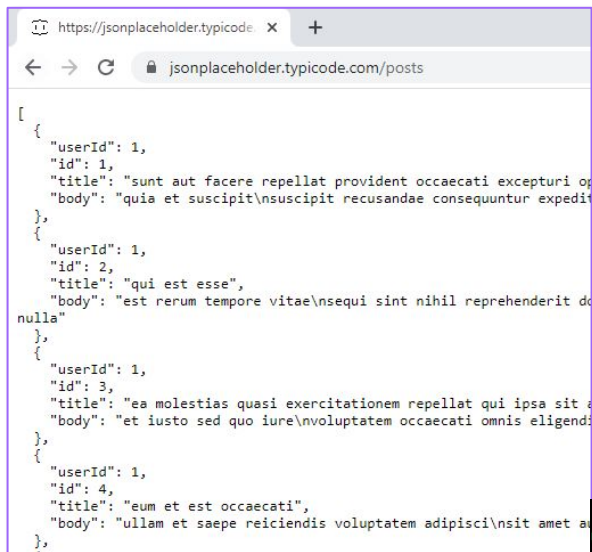
Desde la APP (1) hacemos una solicitud, vamos a traer información de la API externa. Esa API me devuelve información (2). Luego, en función de la respuesta que nos dio la API vamos a hacer otra solicitud (3) y esperar la respuesta (4). Lo que está en **rojo** dependerá de que ocurra lo que está en **azul**. La **segunda** respuesta depende de la **primera**.

Por ejemplo: podríamos pedir información de un posteo de Twitter, que lo devuelva pero luego pedir información del usuario que hizo ese posteo, para lo cual pido el ID de referencia del usuario donde podré ver sus datos.

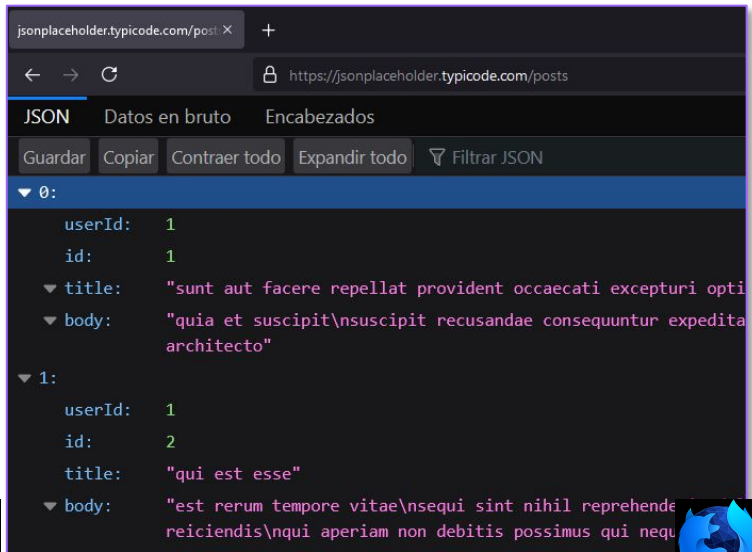
Ver ejemplo fetch-then (.html y .js)

Otro ejemplo de uso de Fetch con API externa

Para este ejemplo aprovecharemos la estructura de tipo JSON que nos ofrece <https://jsonplaceholder.typicode.com/posts>



```
[
  {
    "userId": 1,
    "id": 1,
    "title": "sunt aut facere repellat provident occaecati excepturi opti",
    "body": "quia et suscipit\nsuscipit recusandae consequuntur expedita"
  },
  {
    "userId": 1,
    "id": 2,
    "title": "qui est esse",
    "body": "est rerum tempore vitae\nsequi sint nihil reprehenderit dolor"
  },
  {
    "userId": 1,
    "id": 3,
    "title": "ea molestias quasi exercitationem repellat qui ipsa sit",
    "body": "et iusto sed quo iure\nvoluptatem occaecati omnis eligendi"
  },
  {
    "userId": 1,
    "id": 4,
    "title": "eum et est occaecati",
    "body": "ullam et saepe reiciendis voluptatem adipisci\nsit amet autem"
  }
]
```



```
jsonplaceholder.typicode.com/post: X
https://jsonplaceholder.typicode.com/posts

JSON  Datos en bruto  Encabezados
Guardar Copiar Contraer todo Expandir todo Filtrar JSON

▼ 0:
  userId: 1
  id: 1
  title: "sunt aut facere repellat provident occaecati excepturi opti"
  body: "quia et suscipit\nsuscipit recusandae consequuntur expedita"
  architecto"
▼ 1:
  userId: 1
  id: 2
  title: "qui est esse"
  body: "est rerum tempore vitae\nsequi sint nihil reprehenderit dolor"
  reiciendis\nqui aperiam non debitis possimus qui neque"
▼ 2:
  userId: 1
  id: 3
  title: "ea molestias quasi exercitationem repellat qui ipsa sit"
  body: "et iusto sed quo iure\nvoluptatem occaecati omnis eligendi"
▼ 3:
  userId: 1
  id: 4
  title: "eum et est occaecati"
  body: "ullam et saepe reiciendis voluptatem adipisci\nsit amet autem"
  autem
```



Estas APIs públicas contienen documentación donde te explican cómo utilizarla.

Para acceder a un posteo determinado debemos agregar el número de posteo, ejemplo: <https://jsonplaceholder.typicode.com/posts/2>, me devolverá un objeto que se corresponde con ese post.

Otro ejemplo de uso de Fetch con API externa

Vamos a obtener el ID de usuario de un determinado posteo. Para ello en nuestros archivos HTML y JS haremos lo siguiente:

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

HTML

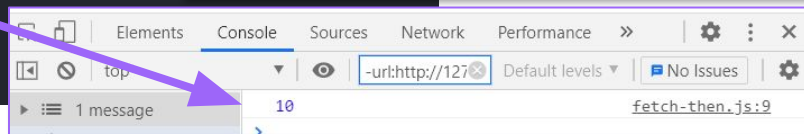
En este caso haremos un Fetch de un id de posteo que pasamos a la función por parámetro (99). Nos retorna ese objeto JSON y pedimos el id de usuario...

```
const getNombre= (idPost) => {
  // hacemos la solicitud a la API...
  fetch(`https://jsonplaceholder.typicode.com/posts/${idPost}`)
  // la API responde en formato JSON
  .then(res=> {
    return res.json()
  })
  // Pedimos el userID de ese posteo
  .then(post => {
    console.log(post.userId)
  })
}

getNombre(99); // llamada a la función
```

JS

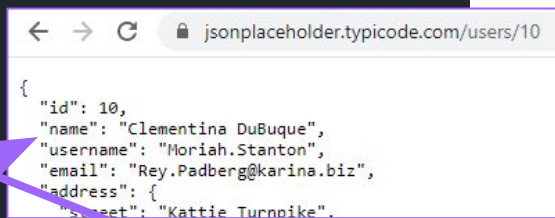
```
{
  "userId": 10,
  "id": 99,
  "title": "temporibus sit alias delectus eligendi possimus magni",
  "body": "quo deleniti praesentium dicta non quod naut est molestias"
}
```



Otro ejemplo de uso de Fetch con API externa

```
const getNombre = (idPost) => {  
  // hacemos la solicitud a la API...  
  fetch(`https://jsonplaceholder.typicode.com/posts/${idPost}`)  
    // la API responde en formato JSON  
    .then(res => {  
      return res.json()  
    })  
  // Pedimos el userID de ese posteo  
  .then(post => {  
    console.log(post.userId)  
    fetch(`https://jsonplaceholder.typicode.com/users/${post.userId}`)  
      .then(res => {  
        return res.json()  
      })  
      .then(user => {  
        console.log(user.name)  
      })  
  })  
}  
getNombre(99); // llamada a la función
```

Pero en realidad lo que deseamos es mostrar la información del usuario que hizo ese post. En el post tengo solamente el id de usuario. La información del usuario podremos ir a consultarla al JSON que contiene los usuarios. <https://jsonplaceholder.typicode.com/users>



Fetch Async Await y manejo de errores

Uno de los inconvenientes que tenemos con este tipo de comunicaciones es que si no recibimos respuesta de la API me va a dar un error y falla al momento de hacer el Fetch y el resto del código no va a ejecutarse.

Este problema se genera porque la comunicación **no es asincrónica**. Para resolver esto vamos a crear una función asincrónica (**async**), con lo cual vamos a tener que esperar (**await**) a que pase algo para ser ejecutada y si no se recibe respuesta que no avance.

Además incorporaremos un **try...catch**, que es una instrucción utilizada para el manejo de errores:

```
try {  
    //Instrucciones que pueden dar error  
} catch (error) {  
    //Acciones en caso de error  
}
```

Estructura básica de try...catch

Más información:

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Statements/async_function

<https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Statements/try...catch>

Fetch Async Await y manejo de errores

JS

```
// Asíncrono  
const getNombre = async(idPost) => {
```

Indicamos que la función será asíncrona

Misma estructura

```
  try {  
    const resPost = await fetch(`https://jsonplaceholder.typicode.com/posts/${idPost}`)  
    const post = await resPost.json()  
    console.log(post.userId);  
  
    const resUser = await fetch(`https://jsonplaceholder.typicode.com/users/${post.userId}`)  
    const user = await resUser.json()  
    console.log(user.name);  
  } catch (error) {  
    console.log('Ocurrió un error grave', error);  
  }  
}  
getNombre(99); // llamada a la función
```

Pedimos que espere la respuesta a la consulta

Se guarda la información del error que ocurra

En caso de colocar una URL equivocada el bloque try...catch permitirá manejar el error

Si no suceden errores el bloque catch **no se ejecuta**.

Comparándolo con la escritura anterior, esta forma es más prolija y más controlable.

[Ver ejemplo fetch-async-await \(.html y .js\)](#)

Librería Axios

Axios es una **librería JavaScript** que puede ejecutarse en el navegador y que nos permite hacer sencillas las operaciones como cliente HTTP, por lo que podremos configurar y realizar solicitudes a un servidor y recibiremos respuestas fáciles de procesar.

Más información [aquí](#).

Para utilizar Axios debemos:

1. Ingresar a <https://cdnjs.com/libraries/axios>
2. Copiar el primer código y agregarlo al final del <body> en el documento HTML, antes del script del archivo externo:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/axios/0.21.1/axios.min.js"></script>  
<script src="async-await-axios.js"></script>
```

Para trabajar con Axios aprovecharemos el ejemplo anterior, donde vamos a tener una función asíncronica, vamos a seguir manejando errores pero a diferencia de la anterior vamos a recibir la respuesta pero no se la vamos a pedir a **Fetch**, sino que se la vamos a pedir a **Axios**.

Librería Axios

JS

```
// Asincrónico con Axios
const getNombre = async (idPost) => {
  try {
    const resPost = await axios(`https://jsonplaceholder.typicode.com/posts/${idPost}`)
    console.log(resPost); //traemos el objeto completo
    console.log(resPost.data.userId); //traemos solo el userID, guardado dentro de data

    const resUser = await axios(`https://jsonplaceholder.typicode.com/users/${resPost.data.userId}`)
    console.log(resUser); //traemos el objeto completo
    console.log(resUser.data.name); //traemos solo el nombre, guardado dentro de data
  } catch (error) {
    console.log('Ocurrió un error grave', error);
  }
}
getNombre(99); // llamada a la función
```

¿Qué cambia con respecto al ejemplo anterior? En primer lugar utilizamos la librería **Axios** en vez de **Fetch** y ya no necesitamos convertir a JSON, sino que directamente vamos a poder acceder al dato a través de la respuesta. Haremos referencia al objeto **data** para obtener el **userID**. Haremos lo mismo para pedir los datos del usuario.

Ver ejemplo [async-await-axios \(.html y .js\)](#)

SPA: Single Page Application

SPA 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**. Aunque parezcan páginas distintas, realmente es la misma página, por eso la respuesta es muchas veces instantánea para pasar de una página a otra. 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".*

Fuente (para ampliar): <https://desarrolloweb.com/articulos/que-es-una-spa.html>

SPA: Single Page Application



Ver ejemplo SPA (La cocina de Juan): <https://dreamy-pike-507001.netlify.app/>

Características importantes del ejemplo:

- Aplican la lógica de tener separada la vista de los datos.
- Navegamos por la Web como si fuera una aplicación de escritorio. La respuesta es muy rápida, no estamos cargando cada página cuando accedemos a otra parte del menú.
- Las páginas se cargan de una vez, cuando estemos navegando vamos a hacerlo sobre contenido que ya está cargado.
- Podríamos hacer que la URL cambie y que se conserve el historial de usuario, además que puedan utilizar los botones de adelante y atrás.
- Así funciona **Gmail**, es una sola página, no es que se cargan todos los mensajes, se cargan los primeros, el resto de la carga es **on demand**. Lo mismo sucede con las páginas de los bancos cuando muestran los movimientos de la cuenta por partes.
- Quien se encarga de gestionar todo esto es JavaScript que le da comportamiento a la página.
- Podríamos comunicar, entonces, el **front** con el **back** a través de una API propia o de terceros.

Para ampliar:

¿Qué es una web SPA? - Single Page Application:
<https://www.youtube.com/watch?v=Fr5QGDJZBV0>

Ejemplos, cursos y guías de VUE.js. APIS

- **Guía de VUE.js:** <https://es.vuejs.org/v2/guide/index.html#>
- **Ejemplos VUE:** <https://vuejsexamples.com/>
- **Escuela VUE:** <https://escuelavue.es/series/>
- **¿Qué son las APIs y para qué sirven?:** <https://youtu.be/u2Ms34GE14U>
- **Curso de Vue JS - Tutorial en Español [Desde Cero]:**
<https://www.youtube.com/playlist?list=PLPl81lqbj-4J-gfAERGDCdOOtVgRhSvIT>
- **VUE Mastery (curso):** <https://www.vuemastery.com/courses/intro-to-vue-is/vue-instance/>
- **It-brain – Tutorial de VUE.js:** https://es.it-brain.online/tutorial/vuejs/vuejs_overview/
- **Lenguaje JS - ¿Qué es VUE?:** <https://lenquajejs.com/vuejs/introduccion/que-es-vue/>