

# **Tema 4: Javascript en clientes web:**

## **Parte I: introducción, eventos y API del DOM**

## 4.1

# Javascript en el cliente: conceptos básicos


En el principio Javascript y el desarrollo *frontend* era **esto**

```
<h1>Mi web</h1>
<script>
  alert("¡¡Bienvenido a mi web!!")
</script>
```

La web de los 90s: <https://sophieswebsite1999.neocities.org/>

Pero el desarrollo *frontend* actual no es trivial



**Yehuda Katz**   
@wycats · [Follow](#)



Front end software development is:

- real-time (instant load, 60fps)
- distributed, incremental (synchronize remote data as needed)
- asynchronous
- reactive (react to user actions in realtime)

Front end is the hardest kind of dev I do. The folks who do it every day are heroes.

4:53 PM · Nov 14, 2017 

---

 **4.2K**  **Reply**  **Share**

[Read 79 replies](#)

<https://x.com/wycats/status/930463710941872128?s=20>

# Versiones de JS en el navegador

Javascript (también llamado ECMAScript o ES) es un lenguaje en rápida evolución

- La última gran "revolución" en JS fue ES2015, a.k.a. ES6
- Cada año hay un nuevo estándar: ES2021, ES2022,...
- Importa el [soporte de funcionalidades](#) más que de versiones

# Transpilación

- Si el navegador objetivo no soporta las funcionalidades que necesitamos se pueden usar compiladores (*transpiladores*) que **traduzcan de las versiones nuevas de Javascript a código más antiguo**
- El transpilador más usado actualmente es **Babel**
- Se empezaron a usar para transformar ES6->ES5, en general ya no necesario. Se sigue usando para dar soporte a navegadores *legacy* y para poder emplear funcionalidades recientes
- También se usa para traducir otros formatos a JS como *.jsx*, propio de React



Como curiosidad: Babel lo empezó a escribir Sebastian McKenzie a los 17 años mientras estaba en el instituto. Podéis leer [la historia de esta época](#) contada por él mismo.

# Insertar JS en el HTML

- En etiquetas `<script>`
- El ámbito de las variables y funciones definidas es la *página*
- Por defecto el JS se *parsea* y ejecuta conforme se va leyendo

```
<html>
<head>
  <script>
    //esto define la función pero no la llama todavía
    function ahora() {
      var h = new Date();
      return h.toLocaleString(); }
    var verFecha = true;
  </script>
  <!-- podemos cargar JS externo con un tag vacío y su URL en el src -->
  <script src="otroscript.js"></script>
</head>
<body>
  <script>
    //la variable es visible por estar definida antes en la misma página
    if (verFecha)
      alert("Fecha y hora: " + ahora());
  </script>
</body>
</html>
```



# Carga de *scripts* externos

Forma "clásica": con el atributo `src` en un `<script>` vacío conseguimos una especie de "include". Todo lo que incluimos está en el mismo "espacio de nombres"

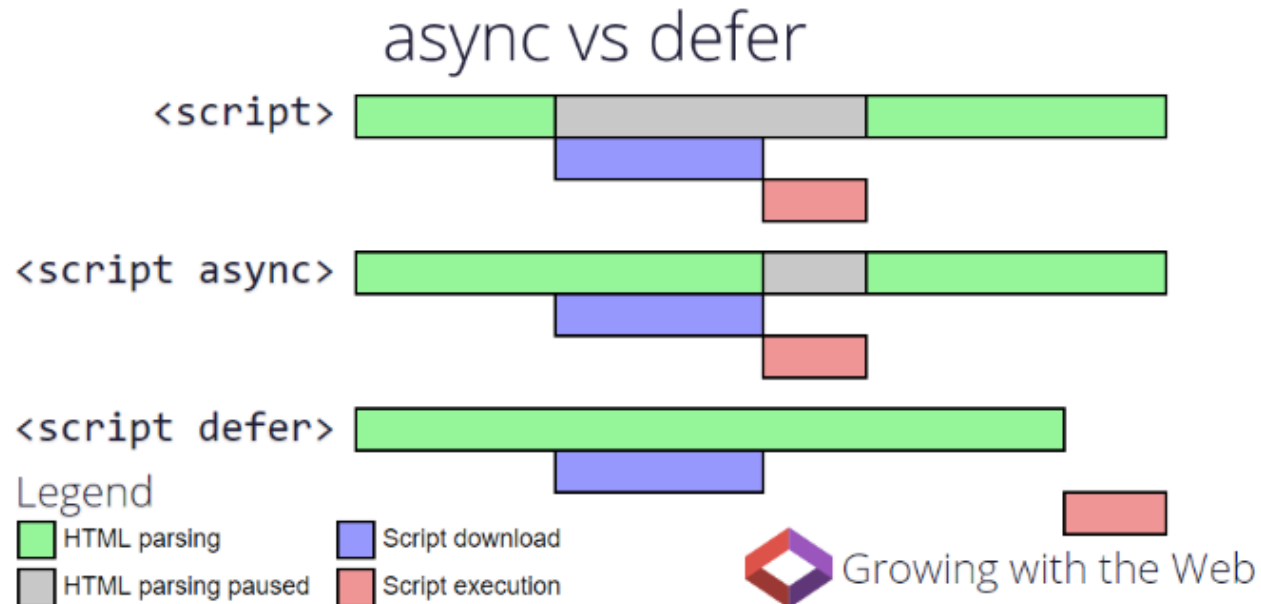
```
<!-- Ejemplo completo en https://jsbin.com/wiyocat/edit?html,js,output -->
<script src="https://threejs.org/build/three.js"></script>
<script>
  // Crear una escena
  var scene = new THREE.Scene();
  // Crear una cámara
  var camera = new THREE.PerspectiveCamera(75, window.innerWidth/window.innerHeight,
  //... el ejemplo continúa...
</script>
```

- Típicamente cada `<script src="">` define una o más clases, funciones o vars. que son **globales**
- Con muchas dependencias externas, esta forma se vuelve tediosa (por la cantidad de `script src`) y problemática (por colisiones en los nombres o tener que gestionar el orden de las dependencias, si hay relaciones entre ellas)

Por defecto al encontrar un *script* se interrumpe la carga del HTML hasta que se acabe de cargar,*parsear* y ejecutar el *script*. Por ello típicamente **se recomendaba colocar los scripts al final**, así el usuario no ve una página en blanco.

Con *scripts* externos podemos usar los atributos **defer** o **async**

<https://www.growingwiththeweb.com/2014/02/async-vs-defer-attributes.html>



# Módulos en JS

Claramente, los `<script src="">` no son una buena solución al **problema de la modularidad**, ya que lo único que estamos haciendo es juntar todo el código en un "espacio global".

En JS han ido surgiendo distintos sistemas de módulos, algunos estándares oficiales y otros "de facto", en la actualidad quedan

- **CommonJS** (originario de Node)
- **Módulos ES6 o ESM** (diseñados para los navegadores, también en Node desde 2020)

# Módulos CommonJS

```
//Archivo "modulo_saludo.js"  
function saludar(nombre) {  
    return "Hola qué tal, " + nombre  
}  
  
module.exports = saludar
```

```
//Archivo que hace uso de "modulo_saludo"  
let s = require('./modulo_saludo')  
console.log(s("Pepe"))
```

# Módulos ESM

```
//archivo modulo_saludo.js
function saludar(nombre) {
  return "Hola qué tal, " + nombre
}
export {saludar}
```

```
//archivo main.js (hace uso del modulo_saludo)
import {saludar} from './modulo_saludo.js'
console.log(saludar('Pepe'))
```

Hay muchas formas de `import`

```
<!-- en el HTML -->
<script type="module" src="main.js"></script>
```

# Un problema de los módulos ESM

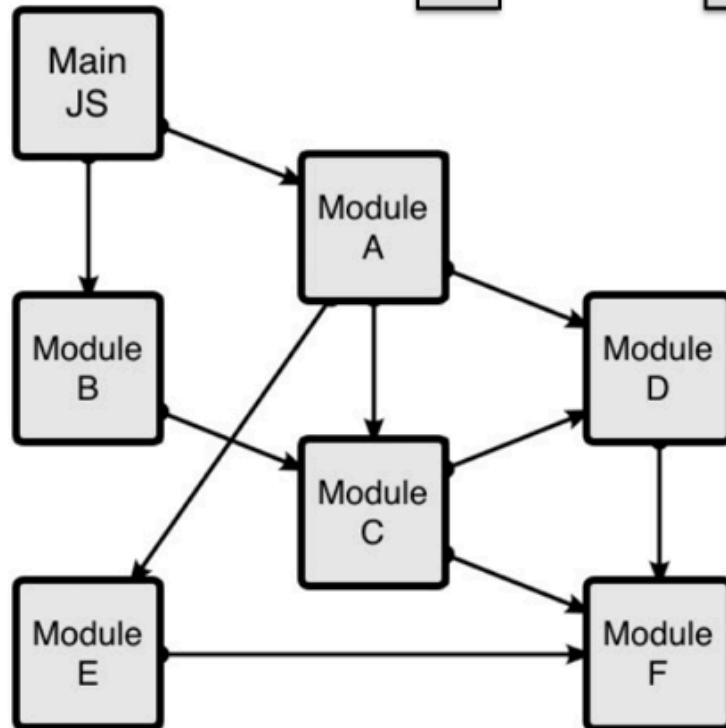
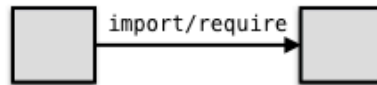
- Aunque a fecha de hoy todos los navegadores [los implementan](#), esto es relativamente reciente (desde 2018). **La necesidad de usar módulos en *frontend* surgió antes de que ESM se implementara en los navegadores más usados**
- A alguien se le ocurrió que se podía añadir soporte de CommonJS al navegador con una herramienta externa que "transformara" el módulo en algo que se pueda incluir con un `script src=""` (esta herramienta se llamó *bundler*)
- Como resultado, desde hace unos años **muchas dependencias de terceros se distribuyen** con npm, **en** formato **CommonJS** (no soportado nativamente por los navegadores)

# Bundlers

- Herramientas que a partir de un conjunto de módulos resuelven las dependencias y **concatenan todo el código en un único .js (*bundle*)** que el navegador puede cargar con un simple `<script src="">`
- Típicamente ofrecen compatibilidad con módulos ESM y CommonJS
- Además el *bundler* puede realizar operaciones adicionales como:
  - Llamar a un transpilador para traducir el código de ES6 a ES5
  - *minificar* el código
  - copiar los *assets* (jpg, png, ...)
  - ...
- Ejemplos: webpack, vite, parcel, rollup, esbuild ...
- Veremos su uso en prácticas







js/  
main.js  
moduleA.js  
moduleB.js  
...



1 único archivo .js  
bundle.js

# ¿Siguen siendo necesarios los *bundlers* en el 2023?

- Teóricamente no deberían, ya que todos los navegadores soportan ESM
- Pero...
  - En producción es más eficiente descargar un solo *bundle* que muchos módulos separados (demasiadas peticiones HTTP)
  - Además del *bundle* realizan otras muchas tareas
- Hay *bundlers* modernos, como [Vite](#), que generan *bundles* compatibles con ESM

# Acceso a los APIs nativos del navegador

- El navegador incluye "de serie" multitud de APIs, para: gestión de eventos, manipulación del HTML, comunicación con el servidor, guardar datos en local, dibujar gráficos,...
- Hay una serie de "objetos globales predefinidos" de los que "cuelgan" estos APIs, por ejemplo
  - **window**: el objeto global por defecto, todo lo que definimos está dentro de él.
  - **document**: la página actual
  - **navigator**: el navegador

Vamos a ver a continuación con más detalle los **APIs estándar** que nos permiten programar **con JS en el navegador**. Para que sea un poco más llevadero, lo haremos en forma de *demo*



Every Owen Wilson Wow In Chronological Order



- API: <https://owen-wilson-wow-api.onrender.com/>
- Plantilla de ejemplo: <https://stackblitz.com/edit/vitejs-vite-vykgjp>

## 4.2

# Eventos

# Eventos y *listeners*

- Casi todo el código Javascript incluido en un HTML se va a ejecutar de modo **asíncrono**, en respuesta a **eventos**
- Los eventos pueden responder directamente a *acciones* del usuario (p.ej. **click** con el ratón) o bien a *sucesos* "externos" (p. ej. la página ha acabado de cargarse).
- A **cada evento le podemos asociar una o más funciones JS** que se ejecutarán cuando se dispare. Genéricamente esto se conoce como *callbacks*. En el contexto de eventos, son llamados *listeners*



# Definir un listener

con `addEventListener` se añade un *listener* que responde a un *evento* sobre un *elemento* del HTML

- Cada evento tiene un **nombre estándar**: 'click', 'mouseover', 'load', 'change'
- Algunos eventos son aplicables prácticamente a cualquier elemento HTML ('click', 'mouseover'). Otros solo a algunos ('change' o 'keydown' solo a campos de entrada de datos)
- Un mismo elemento y evento pueden tener asociados **varios *listener***

# Ejemplo de *listener*

HTML:

```
<button id="miBoton">iNo me pulses!</button>
```

JS:

```
//El listener recibirá automáticamente un objeto Event con info sobre el evento
//https://developer.mozilla.org/en-US/docs/Web/API/Event
function miListener(evento) {
    alert('Te dije que no lo hicieras!, pero has clicado en '
        + evento.clientX + ',' + evento.clientY)
}
var boton = document.getElementById('miBoton')
//cuando se haga click sobre el objeto "boton", se llamará a "miListener"
boton.addEventListener('click', miListener)
//Otra forma: definimos el listener como una función anónima
boton.addEventListener('click', function() {
    console.log('no espíes la terminal ehhe')
})
```

<https://jsbin.com/funizen/edit?html,js,output>

# Delegación de eventos

Los eventos sobre una etiqueta HTML *suben* hacia arriba en la jerarquía de etiquetas (*bubbling up*), de modo que podemos capturarlos también en niveles superiores.

```
<body>
  <button id="boton">Pulsa aquí</button>
  <p>Hola, aquí también puedes pulsar</p>
</body>
```

```
document.getElementById('boton').addEventListener('click', function(e) {
  console.log('en el listener del botón')
  //si ponemos esto, paramos el bubbling
  //e.stopPropagation()
})
//Aquí recibiríamos también los clicks sobre el "button" y el "p"
document.addEventListener('click', function(e){
  //En un listener, this es el objeto al que está vinculado el evento. Aquí document
  console.log("this es " + this.nodeName) //document
  //target es el "objetivo" del evento. P.ej. si clicamos en el boton será este
  console.log('click sobre ' + e.target.nodeName)
})
```



La delegación de eventos es útil cuando queremos asignar un comportamiento similar a varios elementos sin tener que repetir el *event listener* para cada uno

Ejemplo "teclado simulado"

# Eventos que no dependen del usuario

- Evento **DOMContentLoaded** (sobre **document**): indica que se ha parseado y cargado completamente el HTML
- Evento **load** (sobre **window**, **document** o **script**): indica que se ha cargado completamente un recurso y sus dependientes (por ejemplo si es sobre **window** no solo el HTML sino también los *scripts*, imágenes)

```
document.addEventListener('DOMContentLoaded', function() {  
    alert("Ahora ya puedo marearte con este bonito anuncio")  
})
```

# Event handlers

Forma *legacy* de definir *listeners*. Además de la sintaxis, la diferencia principal es que **solo puede haber un handler** para un evento y un elemento HTML dados

Los *handler* tienen como nombre 'onXXX', donde 'XXX' es el nombre del evento: 'onclick', 'onmouseover', 'onload',...

# Ejemplo de *handler*

HTML:

```
<button id="miBoton">iNo me pulses!</button>
```

JS:

```
var boton = document.getElementById('miBoton')
boton.onclick = function() {
    console.log('has hecho click')
}
//CUIDADO, este handler SUSTITUIRÁ al anterior!!!
boton.onclick = function() {
    alert('has hecho click')
}
```



# Manejadores de evento *inline*

La forma más antigua de definir *handlers*: en el propio HTML, con un atributo **onXXX**:

```
<!-- nótese que decimos que hay que INVOCAR la función, y no la  
referenciamos simplemente como hasta ahora. Esto es porque aquí  
podemos poner código arbitrario. Pero un listener debía ser una función -->  
<button onclick="mensaje()">iNo me pulses!</button>
```

```
function mensaje() {  
  console.log('mira que eres pesadito/a')  
}
```

Tiene "mala prensa" porque mezcla JS y HTML. Curiosamente en los *frameworks* web modernos se tiende a usar esta forma de programar 🤔.

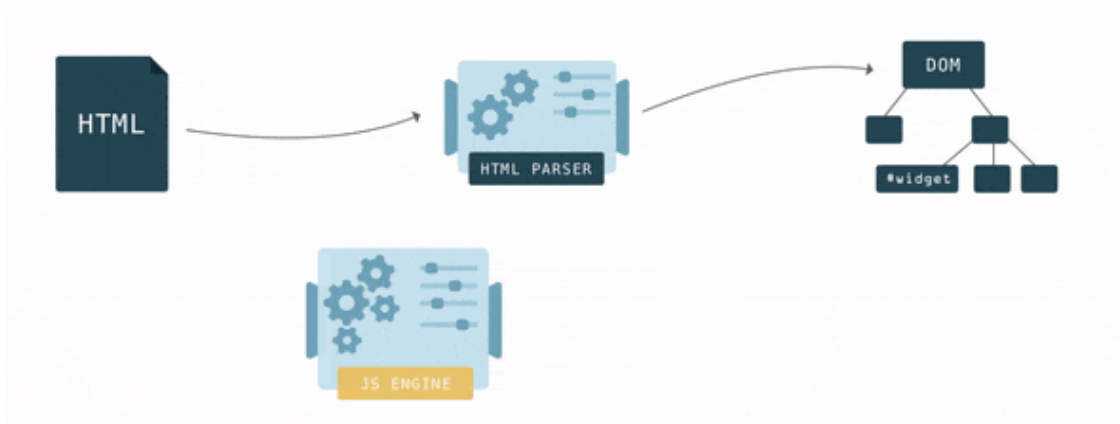
## 4.3

# Manipulación del HTML: el API DOM

**DOM** (*Document Object Model*): por cada etiqueta o componente del HTML actual hay en memoria un objeto Javascript equivalente.

Los objetos JS forman un árbol en memoria, de modo que un nodo del árbol es "hijo" de otro si el elemento HTML correspondiente está *dentro* del otro.

**API DOM:** conjunto de APIs que nos permite acceder al DOM y manipularlo. Al manipular los objetos JS estamos cambiando indirectamente el HTML *en vivo*



# El árbol del DOM



# Live DOM Viewer

Markup to test ([permalink](#), [save](#), [upload](#), [download](#), [hide](#)):

```
<!DOCTYPE html>
<html>
<head>
<title>Ejemplo de DOM</title>
</head>
<body>
<!-- es un ejemplo un poco simple -->
<p style="color:red">Bienvenidos al <b>DOM</b></p>
</body>
</html>
```

DOM view ([hide](#), [refresh](#)):

```
└ DOCTYPE: html
└ HTML
  └ HEAD
    └ #text:
    └ TITLE
      └ #text: Ejemplo de DOM
    └ #text:
  └ #text:
  └ BODY style="zoom: 1;"
    └ #text:
    └ #comment: es un ejemplo un poco simple
    └ #text:
    └ P style="color:red"
      └ #text: Bienvenidos al
        └ B
          └ #text: DOM
    └ #text:
```

# Acceder a un nodo

**Por id.** "marcamos" con un **id** determinado aquellas partes de la página que luego queremos manipular dinámicamente

```
var noticias = document.getElementById("noticias")
```

**Por etiqueta:** accedemos a todas las etiquetas de determinado tipo

```
//Ejemplo: reducir el tamaño de todas las imágenes a la mitad
//getElementsByTagName devuelve un array
var imgs = document.getElementsByTagName("img");
for(var i=0; i<imgs.length; i++){
    //por cada atributo HTML hay una propiedad JS equivalente
    imgs[i].width /= 2;
    imgs[i].height /= 2;
}
```

# Acceder a un nodo (II)

Con **selectores CSS**:

```
//querySelector: obtener el 1er nodo que cumple la condición
//este ejemplo sería equivalente a getElementById
var noticias = document.querySelector('#noticias')
//aunque puede haber varios divs solo obtendremos el 1o
var primero = document.querySelector("div");
//querySelectorAll: obtenerlos todos (en un array)
var nodos = document.querySelectorAll("div");
//Cambiamos la clase. Nótese que el atributo es "className", no "class"
//al ser "class" una palabra reservada en JS
for (var i=0; i<nodos.length; i++) {
    nodos[i].className = "destacado";
}
//selectores un poco más complicados
var camposTexto = document.querySelectorAll('input[type="text"]');
var filasPares = document.querySelectorAll("tr:nth-child(2n)")
```

# "Datos básicos" de un nodo

En un nodo tenemos su nombre (`nodeName`) y su valor (`nodeValue`)

- Para las etiquetas, el nombre es la etiqueta en mayúsculas y sin `<` `>` y el valor `null`
- Para los nodos de texto, el nombre siempre es `#text` y el valor su contenido
- En los campos de formulario editables, el contenido no está en `nodeValue` sino en `value`



# Relaciones "familiares" entre nodos

Una vez accedemos a un nodo podemos acceder a su "padre", "hijos" o "hermanos" con una serie de propiedades:

- Padre: `parentNode`
- Hijos
  - Solo los *tags*: array `children`. Primero=>`firstElementChild`, último=> `lastElementChild`
  - Todos (incl. *whitespace nodes*): array `childNodes`. Primero=>`firstChild`, Último=>`lastChild`
- Hermanos:
  - Solo los *tags*: siguiente `nextElementSibling`, anterior `previousElementSibling`
  - Todos (incl. *whitespace nodes*): `nextSibling`, `previousSibling`

# Añadir/eliminar/crear nodos

La idea de poder añadir/eliminar/crear nodos para que cambie el HTML es muy **potente**, pero el API es **tedioso** de utilizar

```
<input type="button" value="Añadir párrafo" id="boton"/>
<div id="texto"></div>
<script>
  document.getElementById("boton").addEventListener('click', function() {
    var texto = prompt("Introduce un texto para convertirlo en párrafo");
    /* Nótese que la etiqueta <p> es un nodo, y el texto que contiene es OTRO
       nodo, de tipo textNode, hijo del nodo <p> */
    var par = document.createElement("P");
    var nodoTexto = document.createTextNode(texto);
    par.appendChild(nodoTexto);
    document.getElementById('texto').appendChild(par);
  })
</script>
```

<http://jsbin.com/gaxehayeni/edit?html,js,output>

# Manipular directamente el HTML

Insertar/eliminar directamente una **cadena HTML** en determinado punto (internamente se siguen añadiendo/eliminando nodos como antes, pero ahora es automático)

**innerHTML**: propiedad de lectura/escritura que refleja el HTML dentro de una etiqueta.

```
<input type="button" value="Pon texto" id="boton"/>
<div id="texto"></div>
<script>
  document.getElementById("boton").addEventListener('click', function() {
    var mensaje = prompt("Dame un texto y lo haré un párrafo")
    var miDiv = document.getElementById("texto")
    miDiv.innerHTML += "<p>" + mensaje + "</p>"
  })
</script>
```

Nótese que el **+=** de este ejemplo es ineficiente, ya que estamos *reevaluando* el HTML ya existente

# Insertar directamente HTML

`insertAdjacentHTML(posicion, cadena_HTML)`: método llamado por un nodo, inserta HTML en una posición relativa a él. `posicion` es una cte. con posibles valores `"beforebegin"`, `"afterbegin"`, `"beforeend"`, `"afterend"`

```
<div id="texto">Hola </div>
<button id="boton">Añadir</button>
```

```
document.getElementById("boton").addEventListener('click', function() {
    var nodoTexto = document.getElementById("texto");
    nodoTexto.insertAdjacentHTML("beforeend", "<b>mundo</b>");
    nodoTexto.insertAdjacentHTML("afterend", "<div>más texto</div>");
})
```

<http://jsbin.com/romewolidi/edit?html,output>

La mayoría de *frameworks Javascript* nos liberan de la necesidad de modificar el DOM directamente

- En algunos podemos **vincular** elementos HTML con partes del modelo, de manera que se **actualicen automáticamente** (*binding*). Ejemplos: Knockout, Angular, Svelte...
- En otros simplemente **especificamos el HTML deseado** y el *framework* se encarga de modificar solo las partes que cambian. Por ejemplo React, Vue,...

Lo veremos con detalle en el tema siguiente, de momento un ejemplo sencillo...

# Ejemplo con Vue

<https://jsbin.com/nahikon/edit?html,js,output>

```
<div id="app">
  Tu nombre: <input type="text" v-model="nombre"><br>
  <button v-on:click="generarPuesto">Generar puesto</button>
  <div class="tarjeta">
    {{nombre}} <br> {{puesto}}
  </div>
</div>
```

```
var app = new Vue({
  el: '#app',
  data: {
    nombre: '',
    puesto: ''
  },
  methods: {
    generarPuesto: function() {
      this.puesto = faker.name.jobTitle()
    }
  }
})
```

