

Tema 3: Desarrollo en el cliente con Javascript estándar

Parte 1: Introducción. Eventos. Manipulación del HTML

3.1

Javascript en el cliente: conceptos básicos

Versiones de JS en el navegador

- Los navegadores actuales implementan casi en su totalidad la versión 6 de JS (también llamada **ES6**, o ES2015)

Versiones de JS en el navegador

- Los navegadores actuales implementan casi en su totalidad la versión 6 de JS (también llamada **ES6**, o ES2015)
- No obstante **hay funcionalidades importantes todavía no soportadas**, por ejemplo los **módulos**

Transpilación ES6->ES5

- Como solución al soporte parcial de ES6 se usan compiladores ([*transpiladores*](#)) que **traducen Javascript ES6 a ES5** (sí soportado al 100% nativamente). Así podemos desarrollar en ES6 con compatibilidad "asegurada".
- El transpilador más usado actualmente es [**Babel**](#)

Sebastian McKenzie, *lead developer* de Babel

sebmck



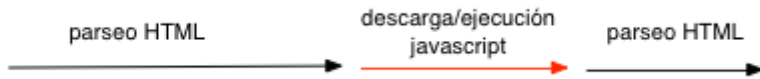
Insertar JS en el HTML

- En etiquetas `<script>`
- El ámbito de las variables y funciones definidas es la *página*. Pero las definiciones no se pueden compartir entre páginas
- Por defecto el JS se *parsea* y ejecuta conforme se va leyendo

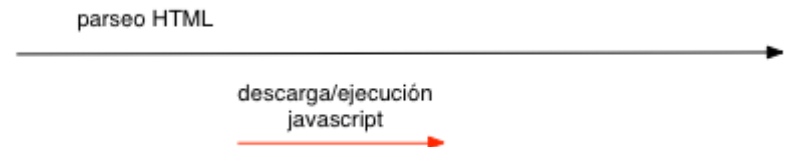
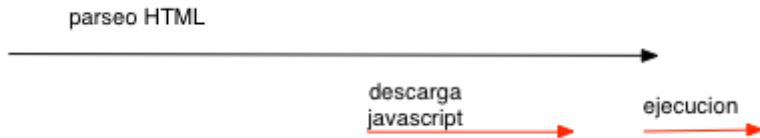
```
<html>
<head>
  <script>
    //esto define la función pero no la llama todavía
    function ahora() {
      var h = new Date();
      return h.toLocaleString(); }
    var verFecha = true;
  </script>
  <!-- podemos cargar JS externo con un tag vacío y su URL en el src -->
  <script src="otroscript.js">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

Carga de *scripts* externos

Por defecto al cargar un *script* se interrumpe la carga del HTML hasta que se acabe de *parsear* y ejecutar el *script*. Por ello típicamente se recomienda *colocar los scripts al final*, así el HTML ya será visible.



Con *scripts* externos podemos usar los atributos `defer` (izq.) o `async` (der.)



Módulos ES6

JAVASCRIPT:

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

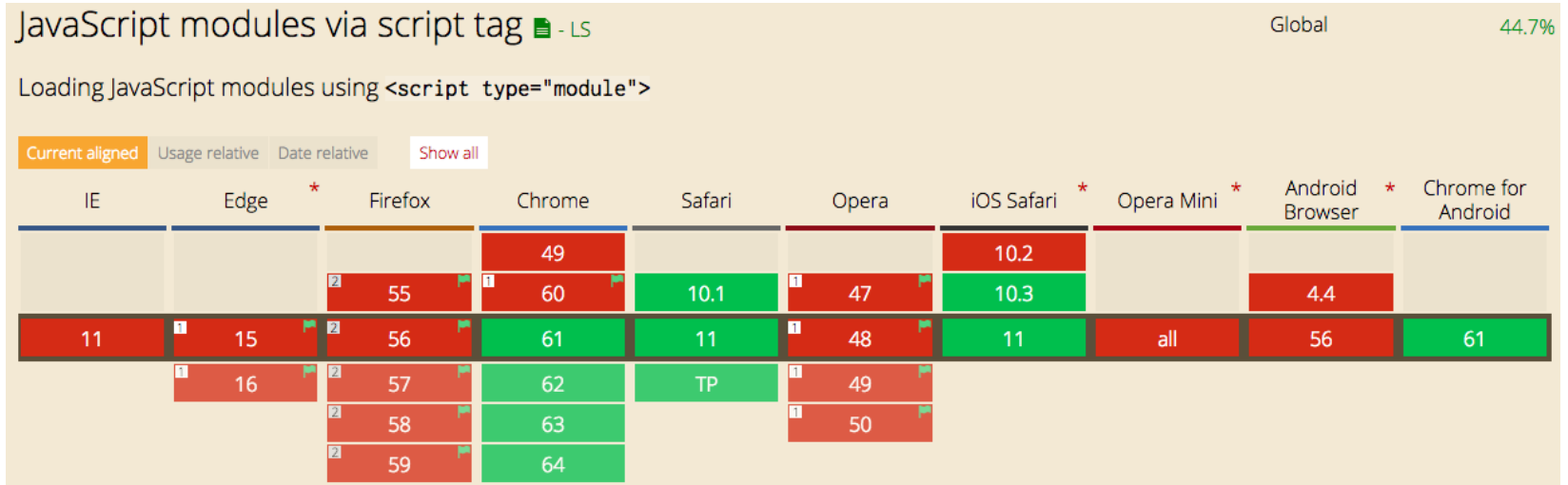
```
//archivo main.js
import {saludar} from './modulo_saludo.js'
console.log(saludar('Pepe'))
```

HTML:

```
<script type="module" src="main.js"></script>
```

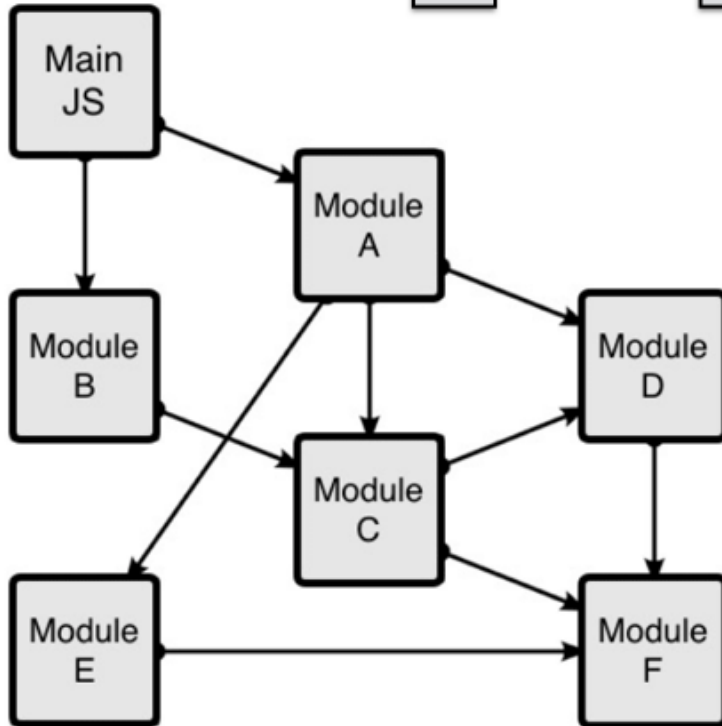
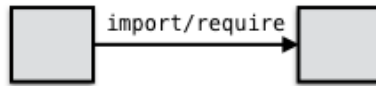
Soporte módulos ES6 en navegadores

A fecha de hoy (octubre 2017) la mayoría de navegadores no los implementan



Bundlers

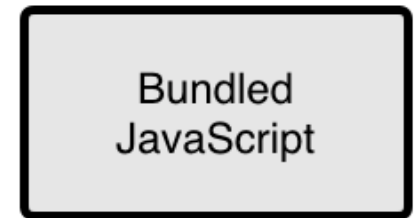
- Herramientas que a partir de un conjunto de módulos resuelven las dependencias y **concatenan todo el código en un único .js** que el navegador puede cargar con un simple `<script src="">`
- Normalmente el *bundler* permite operaciones adicionales como:
 - Transpilar el código de ES6 a ES5
 - *minificar* el código
 - copiar los *assets* (jpg, png, ...)
 - ...
- Ejemplos: webpack, jspm, browserify



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



jspm.io



1 único archivo .js
bundle.js

3.2

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 `addListener` 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">¡No 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 ehheh')
}))
```

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

Event handlers

Forma *legacy* de definir *listeners*. Además de la sintaxis la diferencia más importante 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">¡No 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*

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  
<button onclick=mensaje()>¡No me pulses!</button>
```

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

Tiene "mala prensa" porque mezcla JS y HTML

3.3

Manipulación del HTML

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](#)

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 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)");
```


jQuery

La *librería* que popularizó la idea de seleccionar nodos con sintaxis de selectores CSS

```
//Podéis probar esto en cualquier web que use jQuery (p.ej. la UA)
//desde la consola del desarrollador
$('img') //Muy similar a document.querySelectorAll('img')
  .fadeOut(2000) //Hacer transparente de modo gradual en 2 s.
  .addClass('miImagen') //Añadir una clase CSS
```

- Se trabaja **implícitamente con todos los resultados** devueltos por \$ sin tener que recorrer explícitamente un array de resultados
- Se pueden **encadenar** operaciones
- Además de ser un **API más sencillo** que el estándar podemos usar **el mismo API** en cualquier navegador. jQuery se ocupa de gestionar las incompatibilidades

Modificar/crear nodos

La idea de modificar los nodos o crear otros nuevos 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. Aun así, "por debajo" se están modificando los nodos

`innerHTML`: propiedad de lectura/escritura que refleja el HTML dentro de una etiqueta. Estandarizado en HTML5.

```
<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* 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, Ember,...
- En otros simplemente **redibujamos el HTML entero** y el *framework* se encarga de modificar solo las partes que cambian. Ejemplos: React, Vue, Riot, ...

8.4

Plantillas

Concatenar cadenas == el Infierno

Con `innerHTML` O `insertAdjacentHTML` acabamos concatenando cadenas que mezclan confusamente HTML+JS

```
document.getElementById('miDiv').innerHTML = '<p> Bienvenido, ' +  
    nombre + '</p> <a href="profile?user=' + login + '">Ver perfil</a>'
```

En código así es muy fácil cometer errores

Lenguajes de plantillas

Los formatos de **plantillas** nos permite especificar de manera mucho más cómoda **texto con variables interpoladas**. Los lenguajes más complejos tienen condicionales e iteradores.

- *Plantillas en el servidor*: PHP, ASP, JSP,... no son más que lenguajes de plantillas. Y prácticamente cualquier *framework* web del lado del servidor tiene su formato propio, o lo toma prestado de otros.
- *Plantillas en el cliente*: Uno de los formatos más difundidos es [Mustache](#). No solo hay implementaciones en JS, también en muchos otros lenguajes.

Una **plantilla** Mustache

```
Hello {{name}}  
You have just won {{value}} dollars!  
{{#in_ca}}  
Well, {{taxed_value}} dollars, after taxes.  
{{/in_ca}}
```

Combinada con **datos**

```
{  
  "name": "Chris",  
  "value": 10000,  
  "taxed_value": 10000 - (10000 * 0.4),  
  "in_ca": true  
}
```

Genera una **cadena** como resultado

```
Hello Chris  
You have just won 10000 dollars!  
Well, 6000.0 dollars, after taxes.
```

Ejemplo: Handlebars

Handlebars es una implementación de Mustache en JS

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

```
<!-- Truco: almacenamos el *template* en el HTML como un *script* con un tipo
      desconocido para el navegador para que el *parser* ignore su contenido.
      También se podría usar la etiqueta <template> en navegadores compatibles -->
<script id="miTemplate" type="text/x-handlebars-template">
  <!-- # y / definen una "seccion", que se repite el número de veces necesario,
      si el objeto es una colección. Es como un bucle -->
  {{#posts}}
    <div class="post">
      <h1>{{titulo}}</h1>
      <div class="body">{{texto}}</div>
    </div>
  {{/posts}}
</script>
```

Ejemplo: Handlebars (II)

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

```
var datos = {
  posts : [
    { titulo: 'Mi primer post', texto: 'Mola, eh?...' },
    { titulo: 'Mi segundo post', texto: 'Ahora ya me está aburriendo un poco' }
  ],
  numero: 2
}

var tpl_texto = document.getElementById('miTemplate').innerHTML
var tpl = Handlebars.compile(tpl_texto)
var resultado = tpl(datos)
document.getElementById('miBlog').innerHTML = resultado
```

Plantillas en ES6

Cadenas de texto delimitadas por *backticks* (``...``):

- Pueden ser multilínea
- Permiten interpolar variables definidas previamente
- Podemos usar expresiones, incluso llamar a funciones

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

```
var nombre = 'Pepe'
var plantilla = `
<div>
  <p>Hola ${nombre.toUpperCase()}, a que no sabías que 2+2 es ${2+2}</p>
</div>
`
console.log(plantilla)
```

Muchos *frameworks* tienen formatos de plantillas propios. Por ejemplo en Angular 1 se usa interpolación con `{{}}` y atributos propios en HTML (*directivas*) para por ejemplo iterar por listas

```
<!-- de https://docs.angularjs.org/tutorial/step_10 -->
...
<h1>{{$ctrl.phone.name}}</h1>
<p>{{$ctrl.phone.description}}</p>
<ul class="phone-thumbs">
  <li ng-repeat="img in $ctrl.phone.images">
    
  </li>
</ul>
<ul class="specs">
  <li>
    <span>Availability and Networks</span>
    <dl>
      <dt>Availability</dt>
      <dd ng-repeat="availability in $ctrl.phone.availability">{{availability}}
    </dl>
  </li>
  ...

```