JavaScript

PID_00254185

Jordi Collell Puig
Anna Ferry Mestres

Universitat Oberta de Catalunya

Índice

1. Introducción

JavaScript es un lenguaje de programación que surgió con el objetivo de programar ciertos comportamientos sobre las páginas web, respondiendo a la interacción del usuario y la realización de automatismos sencillos. En este contexto podríamos decir que nació como un «lenguaje de scripting» del lado del cliente, pero hoy JavaScript es mucho más. Las necesidades de las aplicaciones web modernas y el HTML5 han provocado que el uso de JavaScript que encontramos hoy haya llegado a unos niveles de complejidad y prestaciones tan grandes como otros lenguajes de primer nivel.

Hace unos años JavaScript se utilizaba para validar formularios, mostrar cajas de diálogo y poco más. Hoy es el motor de las aplicaciones más conocidas en el ámbito de internet: Google, Facebook, Twitter, Outlook, etc. Absolutamente todas las aplicaciones que utilizamos en nuestro día a día en internet tienen su núcleo realizado con grandes dosis de JavaScript. La web actual se basa en el uso de JavaScript para implementar aplicaciones enriquecidas con la capacidad de realizar todo tipo de efectos, interfaces de usuario y comunicación asíncrona con el servidor por medio de AJAX.

En este módulo veremos el estándar en el que se basa JavaScript, haremos un repaso de las librerías más utilizadas (haciendo especial énfasis en jQuery), y haremos una pequeña aproximación a TypeScript, un lenguaje que compila a JavaScript y que le añade muchas facilidades y ventajas.

2. Ecmascript

Actualmente hay muchas plataformas que interpretan JavaScript, como los navegadores web o NodeJS y son utilizadas para el desarrollo de aplicaciones de diferentes sistemas operativos. Es el estándar ECMAScript el que marca cómo tiene que ser interpretado el lenguaje en cada una de estas tecnologías.

ECMA es el nombre de la asociación europea de fabricantes de ordenadores (European Computer Manufacturer Association), una organización sin ánimo de lucro que se encarga, entre otras cosas, de regular el funcionamiento de muchos estándares de la industria mundial. Uno de estos es el estándar ECMA-262, que también se conoce como ECMAScript.

JavaScript es una de las implementaciones del estándar ECMA-262, en concreto la que se usa en los navegadores. Es decir, ECMAScript es el estándar que define cómo tiene que funcionar el lenguaje, y JavaScript es una implementación concreta de lo que indica la especificación estándar.

Hay otras implementaciones con sus propias extensiones, como por ejemplo ActionScript, el lenguaje que se usaba para programar Flash, o JScript, la implementación de Microsoft de ECMAScript disponible a través de Internet Explorer.

2.1. Historia de la ECMAScript

La primera versión de ECMAScript apareció en 1997 con la intención de estandarizar el lenguaje, de forma que los fabricantes de navegadores no se dedicaran a modificar a su gusto creando versiones incompatibles.

Un año después, en junio de 1998, apareció ECMAScript 2, que realmente no incluía nada nuevo sino que era un cambio de formato en la especificación para alinearla con el estándar internacional ISO/IEC 16262.

En diciembre de 1999 se lanzó ECMAScript 3, que añadió soporte para expresiones regulares, gestión estructurada de excepciones, definición más estricta de errores y otras mejoras.

La versión 4 de ECMAScript no llegó nunca a ver la luz a consecuencia de las luchas internas sobre la complejidad del lenguaje. Se querían añadir demasiadas cosas que hacían que el lenguaje perdiera parte de su propósito inicial. Muchas de las propuestas fueron desestimadas y desaparecieron por siempre, mientras que otras vieron la luz en la siguiente versión o incluso se pospusieron hasta las versiones más recientes.

Así que no fue hasta diciembre del 2009, una década después de la versión anterior, cuando se presentó la siguiente versión del lenguaje: ECMAScript 5. Esta es en realidad la cuarta versión real del lenguaje, por lo cual, oficialmente se tendría que llamar ECMAScript 4, pero no es así y en algunos casos genera confusión.

Esta edición añadió la manera estricta del lenguaje («strict mode»), mejoró la especificación aclarando varias ambigüedades de la tercera edición y que causaban confusión e implementaciones incoherentes, añadió el soporte nativo para JSON, o los getters y setters para propiedades, entre otras pequeñas cosas. A esta versión se la conoce también como Harmony, que es el nombre en clave que se le daba mientras estaba en desarrollo.

En junio del 2011 salió la revisión 5.1, que simplemente alineaba el estándar de ECMA con el formato correspondiente de ISO: ISO / IEC 16262: 2011 pero que no tenía mejoras relevantes.

La sexta edición se lanzó en junio del 2015 con el nombre oficial de ECMAScript 2015. Esta añadía muchas novedades al lenguaje, algunas de las cuales ya se habían planteado para la fallida versión 4. Lleva el lenguaje a un nivel superior, con cambios significativos en la sintaxis para escribir aplicaciones complejas, y también conceptos nuevos, como los símbolos o las lambdas, funciones de flecha, tipo de datos que no existían, estructuras mejoradas por iteración, promesas, etc.

La versión ES5 permaneció durante tanto de tiempo que se convirtió en la más extendida de JavaScript en todo tipo de plataformas. Su alto índice de compatibilidad la convierte en candidata ideal para ser empleada en la escritura de código apto para todo tipo de sistemas o navegadores. De todos modos, actualmente ya son muchos los sistemas capaces de interpretar la versión del 2015 (ES6).

La 7.ª edición, oficialmente conocida como ECMAScript 2016, finalizó en junio del 2016. Resultó ser una versión bastante descafeinada comparada con la anterior, puesto que no incluye grandes mejoras: se introduce el operador de exponenciación (**) y un método nuevo para las matrices que permite comprobar si hay ciertos elementos dentro de estas.

A día de hoy la última versión publicada es la octava, ECMAScript 2017, que finalizó en junio del 2017 con la esperada implementación await/async.

2.2. ECMAScript 6

A pesar de que ya se han publicado las versiones 7 y 8 del estándar ECMAScript, la versión 6 supuso un gran cambio en el lenguaje al introducir muchas mejoras (las posteriores han sido poco significativas en comparación, y todavía no son populares en la comunidad). Por este motivo nos centraremos en los cambios y novedades implementados en ECMAScript 6.

2.2.1. Variables *let* y *const*

Anteriormente, la declaración de una variable se hacía con la palabra clave *var* seguida del nombre de la variable, a declarar de este modo:

```
var variable = "valor";
```

Esta variable podía ser de ámbito global o local. El ámbito depende del lugar donde se declara: para hacer una variable global lo teníamos que declarar simplemente fuera del cuerpo de una función, y las variables locales eran declaradas dentro de una función y solo eran accesibles dentro del código de esta.

A continuación veremos un ejemplo de un código en ES5, en el que una variable está definida dentro de un bloque *if*, pero aun así su ámbito será el de la función:

```
// ES5
function () {
  var variable;
  console.log(variable);
  if (true) {
    variable = "Hola mundo";
  }
  console.log(variable);
};
```

Si lo ponemos en ejecución, el resultado que obtendremos en consola es el siguiente:

undefined
Hola mundo

Este resultado se debe a que en la primera la variable ya existe (se declara previamente) pero no tiene ningún valor definido, y en la segunda ya tiene el valor «Hola mundo». En este caso lo que sería de esperar es que la variable quedara limitada al ámbito donde se ha declarado, que es en el interior de las llaves del *if*, pero se extiende al ámbito de toda la función.

Ahora, con ES6 podemos declarar variables con *let* en lugar de *var* si no queremos que sean accesibles más allá de un ámbito, que será cualquier ámbito expresado por unas llaves. Así, si reescribimos el ejemplo anterior:

```
// ES6
function () {
  console.log(variable);
  if (true) {
    let variable = "Hola mundo";
  }
  console.log(variable);
};
```

Y la salida de la consola sería en este caso:

undefined
undefined

Ahora la segunda llamada devuelve un valor indefinido porque la variable no existe dentro del ámbito fuera del *if*.

Como las variables *let* solo existen dentro del bloque que las contienen, no contaminarán nunca el código a su alrededor. Esto nos evita estar pendientes de sobrescribir variables, porque podemos tener un buen número. Este ejemplo es de un *if*, pero es lo mismo con cualquier bucle, sea un *while*, *for*, etc.

Otra nueva forma de crear variables es *const*. Esto nos crea una variable constante con un valor inalterable a lo largo del código: solo se puede leer, no modificar. Por ejemplo, en un script para contabilidad sería una buena idea poner por ejemplo el IVA como constante, puesto que siempre tendría que ser el mismo.

Por convención se acostumbra a definir en mayúscula los nombres de las constantes en la mayoría de lenguajes de programación para una mejor identificación.

Veamos los siguientes ejemplos:

```
const IVA = 21;
console.log(IVA);

IVA = 25;
console.log(IVA);
```

El primer console.log nos imprimirá 21, que es su valor asignado en la declaración, pero el segundo nos dará un error y nos dirá que su valor continúa siendo 21. El lenguaje no nos permite modificar el valor de una constante.

```
const IVA;
IVA = 21;
console.log(IVA);
```

En este caso hemos declarado IVA y en la siguiente línea le hemos asignado un valor. Quizás parece correcto, pero en realidad no podemos asignarle ningún valor posteriormente a su definición. En realidad, la constante se ha creado con el valor «undefined» y se quedará con este valor para siempre. Es un aspecto que se debe tener muy en cuenta y que puede inducir a error muy fácilmente.

Otra particularidad de *const* está en su uso con objetos. Como es de esperar, se crea un objeto cuyo valor será inalterable, pero quizás no de la manera que pensamos. Por ejemplo:

```
const PERSONA = {
  id: "01",
  nombre: "Jana"
}
PERSONA.nombre = "Gina";
console.log(PERSONA.nombre);
```

Esto nos imprimirá «Gina» puesto que la constante es el propio objeto no sus propiedades, fijaos en que el objeto en sí está con mayúsculas pero sus propiedades no, porque no son constantes. Las propiedades del objeto se pueden modificar, añadir o borrar según necesitemos, lo que nunca podremos cambiar es la referencia a este objeto. Por lo tanto, no podremos hacer algo como: PERSONA = 'Hola mundo'; puesto que sí estamos intentando alterar la referencia al objeto en sí.

La introducción de *let* y *const* no significa que hayamos de dejar de utilizar *var*. En la mayoría de los casos se usará *let* para declarar variables, dado que su ámbito es más restringido y por lo tanto nos puede dar lugar a menos errores debidos a la interferencia entre variables del mismo nombre. Sin embargo, la declaración *var* sigue existiendo y se podrá usar cuando se trata de asignar un ámbito de toda una función, o ámbito global.

Una posible regla sería: utilizar *let* por regla general a no ser que por su ámbito restringido no nos venga bien, y entonces pasaremos a usar *var*.

2.2.2. Funciones flecha (*arrow*)

Como su nombre indica, son funciones definidas usando una flecha =>, pero su comportamiento es un poco diferente a las funciones tradicionales en varios aspectos.

Las funciones flecha tienen una sintaxis más corta que una expresión de función convencional. Además, son anónimas, no están relacionadas con métodos y no pueden ser usadas como constructores.

```
let nuevaFuncion = () => {
  //código de la función
}
```

Como se ve en el ejemplo, no solo se usa la flecha, sino que los paréntesis donde se colocarían los parámetros de la función también se mueven de lado, colocándolos antes de la flecha. La invocación se realizaría del mismo modo que conocemos:

```
nuevaFuncion();
```

El tratamiento de los parámetros se realiza como hasta ahora, simplemente se colocan entre los paréntesis:

```
let saludar = (nombre, tratamiento) => {
  alert('Hola ' + tratamiento + ' ' + nombre)
}

//invocación
saludar('Jana', 'srta.');
```

En caso de pasar solo un parámetro, la función se simplifica, puesto que nos podemos ahorrar los paréntesis como en el ejemplo siguiente:

```
let cuadrado = numero => {
  return numero * numero;
}
```

También nos podemos ahorrar algún carácter extra si solo tenemos una línea de código en nuestra función. En este caso no habría que poner las llaves de apertura y cierre de la función.

let saludar = (nombre, tratamiento) => alert('Hola ' + tratamiento + ' ' + nombre).

Adicionalmente, en caso de tener una única línea de código y que la función devuelva un valor, también nos podemos ahorrar la palabra *return* (además de las llaves, como ya hemos visto). La función del cuadrado quedaría así:

```
let cuadrado = numero => numero * numero;
```

La nueva forma es mucho más compacta que la que teníamos anteriormente. Lo veremos más claro comparando el último fragmento de código con la declaración de la función de la forma tradicional en ES5:

```
// ES5
var cuadrado = function(numero) {
  return numero * numero;
}
```

Si una función no tiene que recibir ningún valor por parámetro, entonces se dejan los paréntesis solo. Por ejemplo:

```
let getBienvenida = () => "Hola mundo";
```

Que sería lo mismo que hacer esto con ES5:

```
var getBienvenida = function () {
  return "Hola mundo";
};
```

2.2.3. Clases

Ahora JavaScript tiene clases, muy parecidas a las funciones constructoras de objetos con las que se trabajaba en el estándar anterior, pero ahora bajo el paradigma de clases y con todo lo que esto comporta, como por ejemplo, herencia.

Así pues, a partir de ahora disponemos de la palabra reservada *class*, que convertirá la declaración de una clase en algo mucho más natural.

El método constructor es un método especial para crear e inicializar un objeto creado con una clase. Solo puede haber un método especial con el nombre «constructor» en una clase. Si esta contiene más de un método constructor, se producirá un Error SyntaxError.

Al igual que en otros lenguajes de programación como Java, un constructor puede usar la palabra reservada *super* para llamar al constructor de una superclase.

Si queremos ver un ejemplo de cómo sería la clase Persona, podemos hacer lo siguiente:

```
class Persona
{
  constructor(nombre)
  {
    this.nombre = nombre;
  }
}

var p1 = new Persona("Carles");
console.log(p1.nombre);
```

Podemos completar la clase anterior añadiendo getters y setters, también del mismo modo como se hace con Java, para encapsular nuestro código:

```
class Persona
{
    constructor(nombre,apellidos)
    {
        this._nombre = nombre;
        this._apellidos = apellidos;
    }
 get nombre(){
        return this._nombre;
    }

 set nombre(nombre){
        this._nombre = nombre;
    }

    get apellidos(){
        return this._apellidos;
    }

 set apellidos(apellidos){
        this._apellidos = apellidos;
    }

}

var p1 = new Persona("Carles","Santamaria");
console.log(p1.nombre);
console.log(p1.apellidos);
```

También se pueden definir métodos estáticos para una clase con la palabra clave *static*. Los métodos estáticos pueden ser llamados sin instanciar la clase. A menudo se usan para crear funciones de utilidad para una aplicación.

Por ejemplo:

```
class Punto {
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }

    static distancia(a, b) {
        const dx = a.x - b.x;
        const dy = a.y - b.y;

        return Math.sqrt(dx*dx + dy*dy);
    }
}

const p1 = new Punto(5, 5);
const p2 = new Punto(10, 10);

console.log(Punto.distancia(p1, p2));
```

2.2.4. Variable *this*

Algunos de los errores más comunes en JavaScript ocurren por la asignación que tiene la palabra clave *this*. El valor de *this* puede cambiar dentro de una función dependiendo del ámbito donde es ejecutada y es muy posible afectar por error a un objeto cuando la intención es afectar a otro.

Anteriormente había que capturar el valor porque *this* solo hace referencia al contexto en el que nos encontramos. Así pues, cuando queríamos pasar una función como *callback*[1]en la cual se tenía que usar alguna propiedad del objeto, había que asignar *this* a una variable, típicamente llamada «that», «self» o «this», o utilizar el método *bind* para que esta nueva función tuviera el mismo contexto que nuestro objeto.

(1)En programación, un callback («devolución de llamada») es una función A a la que se le envía por parámetro otra función B (el callback) esperando que la función A se encargue de ejecutar la función callback.

Las funciones flecha no tienen un valor *this* propio. Si la función flecha está dentro de una función contenedora tradicional, el valor de *this* será igual a la función contenedora; en caso contrario, el valor de *this* será undefined.

```
// ES5
function Persona() {
  this.edad = 18;
  var that = this;
```

```
  setTimeout(function() {
    console.log(that.edad);
  }, 5000)
}
var p = new Persona();
```

Con ES6 no hay que coger el valor de *this* dentro de la variable *that* porque la función flecha del **callback** lo cogería correctamente, por lo tanto, el código pasado a ES6 quedaría de la siguiente manera:

```
// ES6
function Persona() {
  this.edad = 18;

  setTimeout(() => {
    console.log(this.edad);
  }, 5000)
}
var p = new Persona();
```

2.2.5. Template strings

Las template strings, o plantillas de cadenas de texto, ayudan a producir un código JavaScript más claro a la hora de trabajar con cadenas de caracteres. Facilita el mantenimiento de los programas gracias a que su lectura es más sencilla con un simple vistazo.

En JavaScript, y en cualquier lenguaje de programación en general, es normal crear cadenas en las que tenemos que juntar el contenido de literales de cadena con los valores tomados desde las variables. A esto lo llamamos interpolar.

```
var nombre = "Josep Maria";
var saludar = "Estimado/da" + nombre;
```

Esto es muy fácil de leer, pero a medida que en una cadena tenemos que interpolar el contenido de varias variables, el código empieza a ser más enrevesado.

```
var nombre = "Josep Maria";
var apellidos = "Puig"
var profesion = "profesor";
var localidad = "Barcelona";
var descripcion = "<strong>" + nombre + "" + apellidos +  "</strong>es" + profesion + " a " +
 localidad;
```

El código está bien y es del todo correcto, pero podría ser mucho más amigable si usamos las template strings.

Para crear una template string se usa un carácter diferente como apertura y cierre de la cadena: el del acento grave. Así, el ejemplo que hemos visto antes quedaría de la siguiente manera:

```
var nombre = "Josep Maria";
var apellidos = "Puig"
var profesion = "profesor";
var localidad = "Barcelona";
var descripcion = `<strong>${nombre} ${apellidos}</strong> es ${profesion} a ${localidad}`;
```

Otra novedad es que, hasta ahora, si queremos hacer una cadena con un salto de línea teníamos que usar el carácter de escape «contrabarra n» (\n), así:

```
//ES5
console.log("línea 1 de la cadena de texto\n\
línea 2 de la cadena de texto");
```

Para obtener el mismo efecto con cadenas de texto multilínea, con ES6 es posible escribir:

```
//ES6
console.log(`línea 1 de la cadena de texto
línea 2 de la cadena de texto`);
```

2.2.6. Desestructuración (*destructuring*)

La desestructuración es un nuevo método ES6 para extraer datos de objetos y matrices JavaScript utilizando una sintaxis mucho más ajustada y más clara que la proporcionada por ES5.

La desestructuración es lo contrario a la construcción de datos: en lugar de construir un nuevo objeto o matriz, toma los datos de un objeto o matriz existente y los destruye para extraer solo los valores que nos interesa.

Echemos un vistazo al problema que resuelve la desestructuración de JavaScript. A veces, necesitaremos variables de nivel superior como:

```
const persona = {
  nombre: "Eduard",
  apellidos: "Punset",
  twitter: "epunset",
  web: "eduardpunset.es",
};
```

```
const nombre = persona.nombre;
const apellidos = persona.apellidos;
```

A menudo se da el caso de que tenemos que hacer una nueva variable de una cosa que se encuentra dentro de un objeto o dentro de una matriz. En lugar de crear dos variables «nombre» y «apellidos» y asignar el valor de cada parte del objeto separadamente, lo que se puede hacer ahora con ES6 es:

```
const { nombre, apellidos } = persona;
```

El código anterior va a buscar dentro del objeto «persona» una variable llamada «nombre» y una variable llamada «apellidos» y añade los datos a dos nuevas variables que estarán sujetas al bloque principal.

Supongamos por ejemplo que tenemos algunos datos profundamente anidados procedentes de una API JSON:

```
const eduard = {
  nombre: "Eduard",
  apellidos: "Punset",
  links: {
    social: {
      twitter: 'https://twitter.com/epunset',
      facebook: 'https://facebook.com/eduardpunset',
    },
    web: {
      blog: 'http://www.eduardpunset.es'
    }
  }
};
```

De este objeto queremos coger el Twitter y el Facebook. Lo podemos hacer de la manera tradicional y larga:

```
const twitter = eduard.links.social.twitter;
const facebook = eduard.links.social.facebook;
```

O lo podemos hacer de una manera más elegante con la desestructuración:

```
const { twitter, facebook } = eduard.links.social;
console.log(twitter, facebook);
```

Fijaos en que tomamos «eduard.links.social» y no solo «eduard». Esto es importante porque estamos destruyendo algunos niveles de profundidad.

2.2.7. Valores por defecto

Otra novedad es asignar valores por defecto a las variables que se pasan por parámetro en las funciones. Anteriormente hacía falta comprobar si la variable ya tenía un valor, pero con ES6 se puede asignar en el momento de crear la función.

```
//ES5
function(valor) {
    valor = valor || "hola";
}

//ES6
function(valor = "hola") {...};
```

2.2.8. Módulos

Un módulo en ES6 es un archivo que tiene código e información y que puede ser aislado, ayudándonos a organizar y agrupar nuestro código. Todo el código e información contenidos dentro del archivo tiene su propio alcance (*scope*), es decir, no es accesible desde fuera del módulo. Para compartir el contenido se tiene que usar la palabra *export*.

```
// archivo lib/persona.js

modulo "persona" {
    export function hola(nombre) {
        return nombre;
    }
}
```

Y podemos importarlo a otro fichero con:

```
// archivo: app.js

importe { hello } from "persona";
var app = {
    foo: function() {
        hola("Jana");
    }
}
export app;
```

Fijaos en que cuando hacemos la importación de persona solo ponemos el nombre del archivo sin la extensión «.js».

2.3. Transpiladores

La palabra *transpilador* no la encontramos en el diccionario (de momento). Se ha acuñado recientemente a raíz de las necesidades de traducción de diversas versiones de ECMAScript y dialectos y lenguajes derivados. Transpilador en realidad es una adaptación del término *transpiler*, que a su vez viene de *translator* (traductor) y *compiler* (compilador).

Los transpiladores son por lo tanto una especie de compiladores que transforman nuestro código ES6 a ES5, para así ser totalmente compatible con todos los navegadores.

Básicamente, los transpiladores se emplean en la fase de desarrollo del proyecto, donde el desarrollador escribe el código y estos posteriormente hacen su trabajo traduciéndolo, compilándolo y adaptándolo. Este nuevo código es el que se despliega para llevarlo a producción y se forma únicamente de código JavaScript comprensible por todos los navegadores.

Gracias a los transpiladores, lenguajes como Flow, TypeScript y CoffeeScript se traducen a JavaScript ES5, siendo de este modo compatibles con cualquier plataforma.

Uno de los más conocidos y usados es Babel, una herramienta que nos permite transformar nuestro código JavaScript de última generación (o con funcionalidades extras) a un JavaScript que cualquier navegador o versión de Node.js entienda, ayudando a utilizar la última versión del estándar más reciente, e incluso algunas funciones experimentales o que no son parte del estándar (ni siquiera propuestas), pero que nos ayudan a trabajar más fácilmente.

Podéis consultar (y descargar) en los formatos HTML y PDF de manera gratuita una versión del estándar de ecma-international.org:

- http://www.ecma-international.org/ecma-262/6.0/index.html
- http://www.ecma-international.org/ecma-262/6.0/ecma-262.pdf

3. jQuery

jQuery es una de las librerías de JavaScript más populares y usadas por su rapidez, sencillez y la amplia gama de posibilidades que ofrece. Permite la manipulación de eventos, animaciones, carga dinámica de contenidos, etc., con unas pocas líneas de código. Además funciona en los navegadores más utilizados, como IE, Safari, Mozilla o Chrome.

3.1. Obtener jQuery

Hay varias maneras de obtener jQuery. Podemos descargar jQuery de su web (http://jquery.com/), donde disponemos de dos versiones diferentes (Production, minified and gziped), versión preparada por entornos de producción, con el código comprimido y optimizado para ocupar muy pocos Kb de descarga o la versión de desarrollo. Solo descargaremos esta última si lo que queremos es revisar y leer el código de la propia librería y ver cómo está hecho, puesto que la versión de producción es completamente utilizable.

También podemos obtener jQuery mediante NPM (*node package manager*), un gestor de paquetes JavaScript que instala las librerías dentro de nuestro proyecto tan solo escribiendo un comando de código. jQuery es un paquete registrado en NPM, así que nos podemos instalar la última versión escribiendo el comando:

```
$ npm install jquery
```

Esto instalará la librería dentro de la carpeta node_modules.

Por otro lado, si queremos podemos utilizarla desde un CDN (*content delivery network*) de Google, la manera recomendada por la comunidad. Un CDN es una red de distribución de contenidos a nivel global con servidores geolocalizados de forma óptima para mejorar los tiempos de descarga. De este modo, si enlazamos la librería desde aquí, cuando el cliente de web intenta descargarla, muchas veces ya se encuentra que la tiene en la memoria caché.

Sea como fuere, hará falta que enlacemos la librería en nuestro documento HTML. Para hacerlo, utilizaremos la etiqueta estándar de HTML <script> haciendo:

```
<script src="jquery.js"></script>
```

O bien, utilizaremos la etiqueta con una dirección CDN:

```
<script src="http://ajax.googleapis.com/ajax/libs/jquery/x.x.x/jquery.min.js"></script>
```

Una vez hecho esto, ya podremos empezar a utilizar la librería completa para realizar nuestras aplicaciones.

3.2. Cómo funciona jQuery: El Objeto $

Generalmente los frameworks de JavaScript envuelven a partir de un objeto global. De este modo el espacio de memoria queda limpio y podemos implementar la librería en espacios complejos con otras muchas funciones.

jQuery, utiliza el símbolo $() como función que nos permitirá interactuar con la librería. De hecho, el símbolo $ es un simple sinónimo de la verdadera función window.jQuery. Así:

```
var jQuery = window.jQuery = window.$
```

A partir de esta función se envuelven todas las funcionalidades de la librería, y esta nos sirve para realizar cualquier operación. Para compatibilizar con otras librerías (Prototype utiliza el mismo símbolo), jQuery nos ofrece la función jQuery.noConflict(); que desactiva el símbolo $(), dejándolo con jQuery().

3.3. Interactuando con el DOM

La finalidad de jQuery es interactuar con el DOM. La función $() nos permite «seleccionar» referencias a elementos del DOM y a partir de aquí, empezar a interactuar con el lenguaje de programación. No es convencional, puesto que no sigue la estructura usual de programación de aplicaciones, pero sí que es muy práctico, puesto que el 80 % de lo que programamos en el navegador se hace empleando algún elemento del DOM.

Para aclarar un poco el tema, lo mejor es ver un ejemplo.

```
<html>
<head>
<title>Ejercicio jQuery</title>
<script src="https://ajax.googleapis.com/ajax/libs/jquery/x.x.x/jquery.min.js"></script>
</head>
<body>
   <p id="p1">¡Hola Mundo!</p>

< script>
$(function(){ // 1
      var resultado = $('#p1').texto(); // 2
   alert(resultado)
});
</script>
```

```
</body>
</html>
```

El código que aparece en **1** todavía no es relevante, pero sirve para programar el evento «*DOM disponible*». Si nos fijamos en la parte de **2**, con *$('#p1')*, seleccionamos el elemento con id igual a «p1». De hecho, lo que nos devuelve la ***función $*** es una instancia de la librería jQuery inicializada con el elemento p del DOM. El método texto() nos devuelve el contenido del nodo como cadena de texto, lo guardamos en la variable resultado y la mostramos haciendo un alert.

3.4. Selectores, filtros y CSS

Como se puede ver en el ejemplo anterior, la función $() recibe un selector CSS y nos devolverá la instancia del objeto. La cuestión es ver qué selectores podemos utilizar.

En el ejemplo, el primer selector era el #, igual que con CSS, que nos permite seleccionar elementos por ID. Del mismo modo podemos utilizar selectores con clases CSS. Así, la llamada *$('.estilo1')* seleccionará todos los elementos que tengan asociada la clase estilo1. Como vemos, utilizamos el mismo lenguaje que ya conocemos con CSS a la hora de interactuar con el DOM.

Ambas llamadas devuelven una instancia de la librería lista para ser utilizada, con la diferencia de que en el primer caso nos devuelve la instancia inicializada con un solo elemento y en la segunda, con un conjunto de elementos.

Casi siempre podemos aplicar las mismas reglas de CSS a los selectores de jQuery:

$('*')	Todos los elementos
$('#nombre')	Elemento con id=nombre
$('.clase')	Elementos que tengan la .clase CSS
$('img')	Elementos
$('img, p, selectorN')	Combina los elementos img, p, selectorN

Igual que en CSS3 podemos seleccionar por atributo:

$('a[rel="nofollow"]')	Todos los a con el atributo rel="nofollow"
$('[atributo|="valor"]')	Atributo sea o empiece por valor
$('[attribute~="value"]')	Atributo que contenga valor delimitado por espacios.
$('[atributo!="valor"]')	Que el atributo no tenga el siguiente valor
$('[atributo^="valor"]')	Que el valor del atributo empiece por valor

$('[atributo$="valor"]')	Que el valor del atributo acabe con valor
$('[atributo]')	Que el elemento contenga el atributo
$('[atributo1="1"][atributo2="2]')	Que contengan todos los atributos con valores

Y como CSS3, también disponemos de filtros basados en la posición que ocupan los elementos:

$('tr:first-child')	La primera fila de una tabla. También podemos emplear :last-child, y :nth-child(4n)
$("tr:even")	Fila impar. También se puede llamar a las pares con *odd*
$('tr:eq(3)')	El tercer elemento del conjunto seleccionado. En este caso, la tercera fila del conjunto de todas las tablas de la página. También podemos emplear • *gt(n)* conjunto mayor que n • *lt(n)* conjunto menor que n • *not(n)* todos, excepto n
$('tr:first')	El primer elemento. También podemos con el último con *:last*

También podemos filtrar elementos en función del contenido que tienen, así:

$("div:contains('Libro')")	Seleccionará los elementos div que en su contenido aparezca la palabra *Libro*
$("div:has(p)")	Elementos div que contenga otro elemento p
$("td:pariente")	Elementos que son padres de otro nodo, incluido nodos de texto

Finalmente, para completar el apartado de selectores, hay que ver una nueva funcionalidad y detallar una diferencia en los objetos que nos devuelve.

Cuando seleccionamos múltiples objetos, como por ejemplo *$('img')* (para seleccionar todas las imágenes), el objeto que nos devuelve jQuery es una instancia normal aumentada con las funciones de lista (*iterable*). jQuery cuenta con una función *.each* que permite iterar por los diferentes elementos que hemos seleccionado. Así podríamos escribir el siguiente código:

```
$('img').each(function(index, elemento) {
console.log( elemento.src )
});
```

El ejemplo anterior iterará por todas las imágenes del documento. Para saber si un selector nos ha devuelto un conjunto o solo un elemento, podemos consultar la propiedad *length*:

```
$('img').length; // devolverá el total de imágenes que contiene el documento.
```

La función **$()** admite un segundo parámetro que permite definir el contexto en el cual queremos que se realice la busca del selector. Por defecto, si el parámetro no existe, la función realiza la busca en todo el documento, pero si se indica un parámetro, la busca se limitará a este contexto:

```
<div><p>Content</p></div>
<div id="p1">
<p>Content 2</p>
< /div>

$('p',  $('#p1'));
```

La selección anterior devolverá solo el segundo p, puesto que es el único que encontramos dentro del contexto de #p1.

3.5. Manipulando el DOM

Ahora que ya sabemos seleccionar elementos de un documento, podemos ir un paso más allá y empezar a manipularlos. Podemos cambiar las propiedades CSS, leer y modificar sus atributos, copiarlos, eliminarlos y añadir otros nuevos.

3.5.1. Propiedades CSS

Seleccionado un conjunto de elementos, podemos manipular sus propiedades CSS con el método .css de la siguiente manera:

```
$('p').css('color', 'red');
```

Cambiará el color del texto de todos los párrafos a rojo. Pero también se pueden cambiar muchas propiedades a la vez:

```
$('p').css({ 'color': 'grey',
'padding': '5px',
'background-color': 'yellow' });
```

La mayoría de los métodos del objeto jQuery que pueden admitir un par de parámetros también pueden admitir un objeto y así hacer mucho más trabajo de una sola vez.

De la misma manera que los podemos cambiar, también los podemos consultar, preguntando la propiedad. El siguiente código nos devolverá el color del elemento p:

```
$('p').css('color')
```

Hay que decir que existen algunos métodos en modo de acceso directo, como por ejemplo:

.show()	Nos mostrará el elemento si estaba escondido
.hide()	Esconde un elemento
.toggle()	Actúa a modo de interruptor. Si el elemento está escondido, lo muestra y si el elemento es visible, lo esconde

También podemos añadir una nueva clase CSS a un elemento seleccionado, con el comando:

```
$('#id').addClass('nombredelaclase')
```

o podemos consultar si tiene asignada la clase:

```
$('#id').hasClass('nombredelaclase')
```

o bien, la podemos eliminar:

```
$('#id').removeClass('nombredelaclase')
```

También podemos manipular y consultar la altura y anchura de un elemento mediante los métodos *.height()* y *.width()*, que sirven tanto para asignar una altura y anchura, como para consultar la altura y anchura reales.

3.5.2. Atributos de los nodos

Del mismo modo que podemos consultar o modificar propiedades CSS, jQuery nos facilita una fórmula para interactuar con los atributos de cada nodo.

```
<img width="34" />
$('img').attr('width', '44')
```

Como con el método anterior, admite un objeto como parámetro. En su defecto, si solo le enviamos un parámetro que es una propiedad, pues nos devolverá el valor de esta.

3.5.3. Añadir contenido al DOM

Para insertar contenido dentro del DOM disponemos de tres fórmulas básicas en relación con la selección correspondiente:

1) Añadir envolviendo

Cuando hacemos un *.wrap* de un selector lo que se hace es envolverlo con la etiqueta inserta:

```
<p id="uno">Una prueba</p>
$('.inner').wrap('<div class="new" />');
<div class="new">
   <p id="uno">Una prueba</p>
< /div>
```

Del mismo modo podemos hacer un *.unwrap()*, que tiene el efecto contrario. Y también *.wrapAll*, que envolverá todo el conjunto de selectores en un solo div.

2) Añadir dentro

Nos añadirá el contenido pasado como parámetro en diferentes posiciones del selector, en función del método que usemos:

.append('contento')	Añade el contenido al final del selector
.prepend()	Añade el contenido al principio del selector
.html()	Recupera el contenido de un selector o se lo asigna. Siempre contenido html
.texto()	Recupera el texto de un selector, sin etiquetas. También lo asigna

3) Añadir fuera

$().after(),*$().before()* inserta los elementos seleccionados justo en el nodo antes o después del seleccionado. Si la selección nos devuelve diferentes nodos, hará la operación para cada uno de ellos.

3.5.4. Borrar nodos

Dado cualquier selector lo podemos borrar simplemente ejecutando el método .remove(). También podemos utilizar el método .empty(), que lo que hace es vaciar el contenido de un nodo, incluido texto plano.

3.6. Funciones generales

Como librería o marco de trabajo, jQuery nos ofrece una serie de extensiones, métodos o funciones, que nos permiten añadir pequeñas utilidades y funcionalidades al JavaScript:

$.each(collection, callback(indexInArray, valueOfElement))

Es una función que nos permite programar iteraciones, supongamos:

```
a = [1,2,3,4,5]
resultado = 0
$.each(a, function(index, valor) {  resultado += a }
alert('el resultado es' + a)
```

A collection le podemos pasar cualquier elemento que sea iterable en JavaScript, como por ejemplo un array o el resultado de un selector. De hecho, un patrón común cuando queremos hacer cosas con un conjunto sería:

```
$('.item').each(function(ind) {
   $(ind).text()
})
```

En el ejemplo anterior iteraríamos por la colección de nodos que tienen asignado el estilo item. En este patrón es muy interesante saber que *$(ind)* es el nodo actual.

También podríamos iterar sobre las propiedades de un objeto o hash:

```
$('.item').each(function(ind) {
   $(ind).text()
})
```

$.extend({}, objeto1, objeto2)

El método extend nos permite añadir propiedades a un objeto definido previamente:

```
a = {uno: 'hola', dos:'dos' }
b = {uno:'si', tres:'quepasa'}
$.extend(a, b)
a == {uno:'si', tres:'quepasa', dos:'dos'}
```

Es de gran utilidad para escribir código modular, puesto que nos permite ampliar objetos, con propiedades y métodos de otros, de una forma muy fácil. Una aplicación práctica es la creación de mixins, objetos que amplían la funcionalidad de otros. Como por ejemplo:

```
var a = {
   'hola': function() { return 1; },
   'adios': function() { return 2; } }
var mixin = { 'hola': function() { return 2; } }
$.extend(a, mixin)
console.info( a.hola() == 2 )
console.info( a.adios() == 2 )
```

Si nos fijamos en el ejemplo, disponemos de un objeto A que tiene una función hola y que devuelve 1. Pero después le aplicamos el objeto *mixin* con el método *$.extend*, sobrescribiendo la función **a.**

Al final del ejemplo podemos ver cómo *a.hola*() ya no devuelve 1, sino que devuelve 2 porque ha sido sobrescrita (extendida).

Si revisamos el código fuente de jQuery veremos que la mayoría de los métodos nuevos los instalan de este modo. También lo podemos hacer extendiendo la cadena de prototipaje:

```
$.extend(a.prototype, mixin)
```

La línea anterior nos generaría métodos públicos del objeto A. Mientras que el A del ejemplo simplemente es un paquete de funciones (*namespace*).

$.inArray(valor, array)

Busca valor en lista. Si no lo encuentra, devuelve -1. Si lo encuentra, devuelve la posición >0.

$.isArray(p)

Devuelve true si p es un array.

$.merge(a, b)

Fusiona el contenido de a y b devolviendo un nuevo array con la combinación de ambos.

Podéis encontrar muchas más utilidades en la dirección: http://api.jquery.com/category/utilities/.

3.7. Sistema de eventos (*events*)

La gestión de los eventos es el otro gran pilar de las librerías de JavaScript y es tan importante como poder manipular fácilmente el DOM desde nuestros programas. Aquellos que han querido hacer frente a esta tarea sin utilizar ningún framework ya sabrán que la cosa no es trivial, dado que cada navegador nos ofrece un sistema de eventos diferente.

La forma genérica como se escucha un **event** en jQuery es mediante el comando *.on*. Así, para poder capturar los clics en todos los enlaces, podemos escribir:

```
$('a').on('click', function() {
   contador += 1
```

```
})
```

Siempre sigue el mismo patrón: seleccionamos el nodo al cual queremos asignar el event y con el comando *on*, le podemos asignar el event que deseamos. Por ejemplo, para asignar un eventchange a un campo desplegable (*select*) escribiríamos:

```
$('#idselect').on('change', manipulador)
```

Donde *manipulador* puede ser una función anónima como en el caso anterior, o puede ser el nombre de una función (la variable que contiene el objeto función).

A la función *on* le podemos asignar un objeto que contenga una lista de eventos mapeados, como por ejemplo:

```
$('a').on({
   'mousenter': function(evt) {
   $(this).css('color', 'black')
},
   'click': function(evt) {
$(this).toggle();
}
});
```

Es importante ver cómo en la función manipuladora del event la variable *this* apunta al nodo del DOM que la desencadena y que el comando *$(this)* selecciona el propio nodo.

La función recibe un parámetro que contiene un objeto de tipo event. Este parámetro a menudo es omitido, pero con él podemos acceder a información adicional en el momento de generación del evento y durante su propagación. Con jQuery el sistema de eventos sigue una regla de propagación en burbuja (*bubbling*) por la cual, cuando un evento es generado en un nodo, se propaga hacia sus padres. Imaginemos el siguiente ejemplo:

```
<script>
$(function(){
      $('tr').click(function(){
         alert('clic a tr');})
         .height(30)
         .css( 'background-color', '#eaeaea');
      $('a').click(function(e){
         alert('clic a a');
         e.stopPropagation()
      });
   });
```

```
</script>
</head>
<body>

<table>
   <tr>
      <td><a>Hola</a></td>
   </tr>
</table>
```

Si hiciéramos clic en el enlace **a**, lo capturaríamos desde la función del evento, pero como el event seguiría su proceso de propagación, también ejecutaría el manipulador asignado al nodo padre tr y generaría un doble alert(): el del elemento **a** y el del elemento **tr**. O sea, que se ejecutarían ambos eventos. Para impedir esto y romper la cadena de propagación, utilizamos el objeto event que recibe el manipulador y llamaríamos al método .stopPropagation() tal y como se ha hecho en el ejemplo.

En el objeto event también podemos encontrar otros datos, como las propiedades pageX y pageY, que son las coordenadas de ratón, donde se ha desencadenado el event. También dentro de este objeto encontramos propiedades como target, que es el objeto que genera el evento, o bien currentTarget, que es el objeto desde donde se ha iniciado la propagación.

Del mismo modo que asignamos un event con el método on, lo podemos desasignar con el método .off. Así, haciendo:

```
$('tr').off('click');
```

Los eventos pueden ser ejecutados de forma manual empleando el comando *.trigger*('nomevent'), así si hacemos:

```
$('tr').trigger('click');
```

Se ejecutará el manipulador (*handler)* asignado al tal event. (Siempre y cuando se haya asignado *a priori*). Y del mismo modo, podemos generar nuestros propios events.

Imaginemos que tenemos un reloj y queremos que nos notifique el tiempo cada segundo. Podemos generar un event de tipo 'segundo' (el nombre lo elegimos nosotros) desde la función que controla el tiempo que pasa, como en el ejemplo siguiente:

```
var segundos = 0
setInterval(function(){
$('#p1').text( ++segundos )
      .trigger('segundo', [segundos])
```

```
}, 1000)

$('#p1').on('segundo', function(event, data){
   if(data==10) segundos = 0
})
```

Utilizamos el elemento #p1 para que nos genere un event de segundo, después lo capturamos y hacemos que la variable que cuenta los segundos que pasan se inicialice cuando llega a 10. Para que esto funcione, necesitamos tener un nodo con id #p1 en nuestro html.

3.7.1. Tabla de eventos

blur	Cuando en un campo de formulario perdemos el foco del teclado
focus	Cuando un elemento de un formulario recibe un clic del ratón
resize	La ventana cambia de tamaño, pertenece al window
scroll	Estamos haciendo scroll en la ventana o en un elemento div
click	Hacemos clic sobre un nodo
dblclick	Hacemos doble clic sobre un nodo
mouseover	Pasamos el ratón por encima del elemento
mouseout	El ratón sale del elemento
change	El valor de un campo de formulario cambia
submit	Un formulario es enviado hacia el servidor
keydown, keyup	Pulsamos una tecla del teclado. La tecla que se ha pulsado la podemos encontrar en la propiedad which, del objeto event pasado al manipulador
error	Se desencadena, por ejemplo, cuando desde el servidor una imagen no se carga

Existen accesos directos a la mayoría de las propiedades que podemos utilizar con un *on*, con su nombre de función directamente. Así, podemos llamar a los métodos de un selector *.click*, *.change*, *.error*,...

Encontraréis la lista completa de eventos en la documentación oficial de jQuery.

3.8. Formularios

Otra de las tareas clave en la programación de aplicaciones web para el JavaScript es la manipulación, construcción y, sobre todo, validación de formularios. jQuery nos ofrece toda una serie de métodos especialmente diseñados para consultar y validar formularios. Podemos acceder al valor de cualquier campo de formulario a partir de su selector, haciendo:

```
$('#id').val()
```

El mismo método nos sirve para asignar un valor, enviándole como parámetro el valor a asignar.

También podemos capturar el momento de enviar el formulario y programar un método que decida si se puede enviar o no en función del contenido de los campos, utilizando el evento submit. Así, podemos hacer:

```
$('formulario#un').on('submit', function(){
   // si validación correcta
   return true;
   // si hay errores en la validación
   // los podemos mostrar y se tiene que devolver false
   return false;
}
```

Existen también métodos para trabajar con AJAX, que nos permiten serializar de una vez todo el contenido del formulario, haciendo: *$('form').serialize()*, que nos generará el formulario preparado para ser enviado por GET. De lo contrario y según nos interese, podemos utilizar *.serializeArray()* y nos permitirá obtener el formulario dentro una lista.

3.9. AJAX

Asyncronimous JavaScript and xml consiste en una técnica mediante la cual podemos hacer llamadas al servidor sin recargar el contenido de nuestra página, es decir, podemos enviar y recibir datos desde el servidor de forma interactiva desde JavaScript. Cada navegador implementa AJAX de una forma diferente, pero por suerte, jQuery nos facilita una interfaz unificada.

Para hacer una petición contra el servidor, disponemos de la función *.ajax* de tipo genérico y que admite muchos parámetros diferentes, o dos alternativas mucho más directas que nos permiten solicitar documentos con *GET* o *POST*.

```
$.get( url, [ data ], success(data))
```

url: Será la dirección a solicitar al servidor.

data: Es un objeto JavaScript con los parámetros que le queramos enviar al servidor.

success: Es un manipulador que se ejecutará con la respuesta que el servidor nos envía.

Así, un ejemplo sencillo, donde lo que hacemos será cargar un documento extra desde el servidor, puede quedar así:

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title>Ejercicio de carga de ajax</title>
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.5.1/jquery.min.js"></script>
<script>
$(function(){
   $('#bt').click(function(){ // 1
      $.get('archivo.txt', {hola:3, mundo:4}, function(data){ // 2
         $('#desti').html(data); // 3
      });
   })
});
</script>
<style>
   #desti { color:blue; }
</style>
</head>
<body>
<p>Al apretar el botón, cargaremos dentro del párrafo azul, el contenido del archivo txt,
    del servidor.</p>
< input type="button" id="bt" value="Carga" />
<p id="desti">Aquí cargaremos un contenido de con AJAX desde el servidor</p>
< /body>
</html>
```

En el ejemplo, programamos (1) en el evento clic del botón, `#bt` que lance una petición AJAX, por método GET (2) `$.get`, descargando el `archivo.txt` y enviándole dos parámetros (hola y mundo). Como la petición es asincrónica (no sabemos cuándo nos responderá el servidor), le enviamos una función para que lo ejecute en este momento (2). La función tiene un parámetro data, que es el contenido de la petición, y que finalmente en (3) lo colocamos en el nodo `#desti`.

Podemos ver el ejemplo funcionando en:

http://multimedia.uoc.edu/~jcollell/ejercicio_ajax.html

También podemos hacer peticiones al servidor a través de *POST* con la función *$.post* y los mismos parámetros.

Podemos consultar en línea la documentación porque las tres funciones nos ofrecen multitud de opciones más para simplificar el trabajo.

3.10. Componentes (*widgets*)

jQuery internamente no ofrece ningún sistema de componentes, pero sí que existe un proyecto paralelo que ofrece determinados *widgets*. El proyecto se denomina jQueryUI (User Interface) y lo podemos revisar en http://jqueryui.com/. A pesar de estar muy ligado a jQuery, funciona de manera independiente con su propio calendario de versiones.

Fundamentalmente ofrecen componentes visuales, componentes interactivos, efectos y utilidades. En cuanto a componentes visuales, tienen un pequeño *widget* de calendario, un campo de autocompletado (mientras escribes filtra resultados de una lista), unas pestañas, acordeones, barras de progreso, entre otros.

Aparte de los componentes visuales, jQueryUI también ofrece una serie de componentes funcionales, como por ejemplo los draggables y sortables. Los primeros permiten programar objetos arrastrables por la pantalla y los segundos, crear objetos ordenables con drag&drop.

jQueryUI supuso una revolución para la interactividad de las páginas web hace unos años, a pesar de que actualmente ha quedado un poco en segundo plano desplazado por librerías tan completas como Bootstrap, que cuenta con muchos de los *widgets* existentes en jQueryUI, con una estética y una experiencia mejorada. Pero los elementos de interacción como draggable y sortable siguen siendo de gran valor.

3.11. Patrones de uso

Un framework o librería, además de una serie de instrucciones, objetos y métodos para facilitarnos la vida a la hora de trabajar, también implica ciertos patrones de trabajo a modo de convenciones que harán que la ejecución de aplicaciones sea más robusta y más convencional para la mayoría de los programadores que lo utilicen.

```
$(documento).ready(function() {
   // código de programación
})
```

Este primer patrón emplea el evento ready, un evento especial que nos notifica que el DOM del documento HTML ya está listo para ser empleado, es decir, que podemos emplear de forma segura cualquiera de los elementos del documento desde nuestro programa. Como en las últimas versiones ha quedado obsoleto, actualmente habría que escribirlo de este modo, con el mismo significado:

```
$(function() {
   // código de programación
})
```

El evento estándar para hacer esto en JavaScript es el *body.onload*. Con la técnica de jQuery, el evento se ejecuta sin haber descargado las imágenes asociadas con el DOM, mientras que con el método convencional se tiene que esperar al resto, así que es algo menos eficiente. Aparte se pueden crear llamadas ilimitadas *documento.ready* en todo el documento, mientras que de *window.onload* solo podía haber una.

Otro patrón de uso muy interesante es la capacidad de jQuery de encadenar métodos. A nivel funcional no supone mucho, pero sí en cuanto a legibilidad. Sobre un mismo selector podemos ir aplicando diferentes métodos, de forma que resulta habitual encontrar construcciones como esta:

```
$('#lelemtn')
   .css('color', 'black')
   .on('click', alferclic)
   .on('mouseover', rollover)
```

4. Frameworks JavaScript

JavaScript es el lenguaje de programación utilizado en el desarrollo de aplicaciones web por el lado del cliente. Recordando brevemente su historia, JavaScript nace en 1995 gracias a Netscape Corporation, que lo incorpora como lenguaje de script en su primera versión de navegador. Paralelamente, Microsoft inicia el desarrollo de Internet Explorer y copia el lenguaje de Netscape, le cambia el nombre y lo denomina JScript. Realmente los dos lenguajes son muy parecidos pero con matices.

Desde el principio se generan diferencias en el uso, con la forma como se interactúa con el DOM (*document object model*), el sistema de eventos y otras muchas pequeñas peculiaridades que los hacen diferentes. Es un lenguaje similar pero que tiene que interactuar con modelos de clases diferentes y utiliza sistemas de eventos distintos.

Al principio la programación de cliente era terriblemente difícil puesto que había que trabajar con cada una de las especificaciones para los diferentes navegadores y esto hacía que el código que se generaba fuera poco robusto y mantenible. Era habitual desarrollar pequeñas funciones con dos condicionales para cada especificación de navegador.

Para solucionar estos problemas de interacción del lenguaje con los navegadores nacieron frameworks con el objetivo de conseguir una API (*application programming interface*) común en los diferentes navegadores. Técnicamente un framework es:

> Una estructura conceptual y tecnológica de asistencia definida, normalmente con artefactos o módulos de software concreto, a partir de la cual otros proyectos de software pueden ser organizados y desarrollados. Wikipedia

De este modo, en este capítulo expondremos los que, a nuestro entender, son los principales frameworks existentes en el trabajo de JavaScript en el desarrollo de aplicaciones web.

4.1. Angular

AngularJS es un framework JavaScript de desarrollo de aplicaciones web en el lado cliente, creado por el equipo de Google. En el año 2015 anunciaron, en la conferencia «ng-conf 2015», que estaban trabajando en una nueva versión de Angular radicalmente diferente, escrita desde cero, y que no sería compa-

tible con versiones anteriores. Naturalmente, esto provocó el desconcierto y la indignación de muchos desarrolladores, que temían por el mantenimiento futuro de las aplicaciones que habían desarrollado en la primera versión de este framework.

A AngularJS se le denomina a menudo Angular 1, a pesar de que a partir de la versión 2 pasó a denominarse simplemente Angular. Por un lado se dejó de utilizar «JS» porque pasó a estar escrito mayoritariamente en TypeScript en lugar de en JavaScript, y por otro lado pasaron a utilizar la nomenclatura SEMVER (*semantic versioning*).

¿Qué es el semantic versioning?

El sistema SEMVER es un conjunto de reglas para proporcionar un significado claro y definido a las versiones de un proyecto de software. Este se compone de tres números siguiendo la estructura XYZ, donde:

- X (Major): indica cambios rupturistas
- Y (Minor): indica cambios compatibles con la versión anterior
- Z (Patch): indica resoluciones de bugs (compatibles)

Básicamente, cuando se arregla un bug se incrementa el patch, cuando se introduce una mejora se incrementa el minor y cuando se introducen cambios que no son compatibles con la versión anterior, se incrementa el major.

Así pues, si sale una actualización donde el mayor se ha incrementado, sabréis que tendréis que ensuciaros las manos con el código para pasar el proyecto a la nueva versión.

4.1.1. Características de Angular

1) Multiplataforma

- **Aplicaciones web progresivas**: Permiten utilizar las capacidades modernas de las aplicaciones sobre plataforma web para ofrecer experiencias similares en aplicaciones de escritorio. Estas permiten una instalación rápida, en pocos pasos y sin necesitar una conexión a internet.

- **Aplicaciones nativas**: Podemos escribir aplicaciones móviles nativas combinando Angular con Ionic Framework, NativeScript y React Native, entre otras.

- **Escritorio**: También permite crear aplicaciones instalables para escritorios de Mac, Windows y Linux utilizando los mismos métodos de Angular empleados para desarrollar sobre plataformas web. Por otro lado, también posee la capacidad de acceder a las API nativas del sistema operativo.

2) Rendimiento

- **Generación de código**: Angular convierte nuestras plantillas en código altamente optimizado para las máquinas virtuales de JavaScript de hoy en día, ofreciéndonos todas las ventajas del código escrito a mano con la productividad de un framework.

- **Universal**: Ejecuta la primera vista de tu aplicación en NODE.JS, .NET, PHP, y otros servidores para su renderizado de forma casi instantánea obteniendo solo HTML y CSS. También abre posibilidades para la optimización del SEO de la aplicación, algo que en la versión anterior resultaba más problemático.

- **División del código**: Las aplicaciones de Angular se cargan rápidamente gracias al nuevo encaminador de componentes. Este ofrece una división automática de códigos para que los usuarios solo carguen el código necesario para procesar la vista que piden.

- **Reactivo**: Todas las mejoras en el diseño mejoran el rendimiento general, pero la más crítica es la detección de cambios en la vista que antes se hacía con un ciclo de enviado a diario que consumía muchos ciclos de CPU, y ahora se implementa con un sistema reactivo que supone una muy importante mejora de rendimiento.

3) Productividad

- **Plantillas**: Permite crear rápidamente vistas de interfaz de usuario con una sintaxis de plantilla simple y potente.

- **Angular CLI**: Las herramientas de línea de comandos nos permitirán empezar a desarrollar rápidamente, añadir componentes y realizar tests, así como previsualizar de forma instantánea nuestra aplicación.

- **IDE**: Ofrece sugerencias de código inteligente, detección de errores y otras informaciones cuando se integra en la mayoría de los editores populares e IDE.

4) Historia completa del desarrollo

- **Testing**: Utiliza Karma y Jasmine para las pruebas de unidad (*unit tests*) para saber si se han roto cosas cada vez que guardamos nuestros cambios. También es posible implementar tests de aceptación de selenium con Protractor para hacer que nuestras pruebas de escenarios corran más rápido y de manera estable.

- **Animación**: Permite crear animaciones complejas y de alto rendimiento con muy poco código a través de la intuitiva API de Angular.

- **Accesibilidad**: Posee características para crear aplicaciones accesibles con los componentes disponibles por ARIA[2] (MDN web docs).

- **Integración con otras tecnologías**: Una de sus fortalezas es que se integra a la perfección con otras tecnologías que permiten crear web componentes como: React (Facebook), Polymer (Google) y X-Tag (Microsoft).

(2)ARIA, o *accesible rich internet applications*, define cómo realizar contenido web y aplicaciones web (especialmente las desarrolladas con AJAX y JavaScript) más accesibles a personas con discapacidades. Por ejemplo, ARIA posibilita puntos de navegación accesibles, *widgets* JavaScript, sugerencias en formularios y mensajes de error, actualizaciones en directo, etc.

4.2. VueJS

Vue es un framework JavaScript creado en 2014 por Evan You (desarrollador de Google). Lo define como un framework progresivo porque se encuentra dividido en diferentes librerías acotadas con una responsabilidad específica, de forma que el desarrollador incluya los diferentes módulos según las necesidades del contexto en el que se encuentre. No hay que incluir toda la funcionalidad desde el principio como en el caso de otros frameworks, como por ejemplo Angular, y se evita cargar la aplicación de código innecesario.

La librería se enmarca dentro de las arquitecturas de componentes con una gestión interna de modelos basada en el patrón MVVM (*Model-View-ViewModel*). Esto quiere decir que los componentes, internamente, tienen mecanismos de doble data-binding para manipular el estado de la aplicación.

Patrón MVVM

En este patrón de diseño se separan los datos de la aplicación, la interfaz de usuario no controla manualmente los cambios en la vista o en los datos, sino que estos se actualizan directamente cuando sucede un cambio en ellos, por ejemplo, si la vista se actualiza un dato que está presentando, se actualiza el modelo automáticamente y viceversa.

Pero ¿qué define a VueJS? ¿Qué lo diferencia o lo asemeja al resto de alternativas? ¿Por qué se está poniendo tan de moda? Intentamos explicar algunas de sus características para que vosotros mismos veáis si el framework tiene la potencia que nos dicen:

- **Proporciona componentes visuales de forma reactiva**. Piezas de UI bien encapsuladas que exponen una API con propiedades de entrada y emisión de eventos. Los componentes reaccionan ante eventos masivos sin que el rendimiento se vea perjudicado.

- Cuenta con conceptos de **directivas**, **filtros y componentes** bien diferenciados.

- **La API es pequeña y fácil de utilizar**.

- **Utiliza Virtual DOM**. Las operaciones más costosas en JavaScript suelen ser las que manipulan el DOM. VueJS por su naturaleza reactiva necesita

hacer cambios constantemente, pero para agilizar esta tarea cuenta con una copia virtual que se encarga de ir cambiando las partes necesarias.

- **Externaliza** el encaminamiento y la gestión de estado en otras librerías.

- **Renderiza templates a pesar de soportar JSX**. JSX es el lenguaje que usa React para renderizar la estructura de un componente. Es una especie de HTML + JS + extras que permite escribir plantillas HTML con más potencia. VueJS apoya a JSX, pero entiende que es mejor usar plantillas puras en HTML por su legibilidad y por la posibilidad de usar herramientas de terceros que trabajen con estas plantillas más estándar. De este modo, el desarrollador puede escoger el sistema que más le convenga.

- Permite focalizar CSS para un componente específico. Permite crear contextos específicos para los componentes sin perder potencia en cuanto a las reglas de CSS.

- Cuenta con un sistema de efectos de transición y animación.

- Permite renderizar componentes para entornos nativos (Android e iOS).

- **Sigue un flujo one-way data-binding** para la comunicación entre componentes.

- **Sigue un flujo doble-way data-binding** para la comunicación de modelos dentro de un componente aislado.

- **Tiene soporte para TypeScript**. Cuenta con decoradores y tipos definidos de manera oficial y son descargados junto con la librería. +

- **Tiene soporte para ES6**.

- Tiene soporte a partir de Internet Explorer 9.

- **Permite renderizar las vistas en servidor**. Los SPA y los sistemas de renderizado de componentes en JavaScript tienen el problema de que muchas veces son difíciles de utilizar por robots como los de Google, por lo tanto, el SEO de nuestra web o aplicación se puede ver perjudicado. VueJS permite mecanismos para que los componentes puedan ser renderizados en tiempos de servidor.

- **Es extensible**. Vue se puede extender mediante plugins.

4.3. React

React es una librería de JavaScript creada por Facebook, usada por ellos mismos en el desarrollo de su red social. Está más enfocada al desarrollo de interfaces de usuario, aunque con varios complementos puede llegar a alcanzar tanto como Angular 2. Su característica fundamental es que las aplicaciones desarrolladas tienen un gran rendimiento.

Dentro del contexto de Facebook, donde necesitaban herramientas para un desarrollo rápido, y a la vez estar focalizadas a un mayor rendimiento que otras alternativas existentes en el mercado, detectaron que el típico marco de binding y doble binding retrasaba su aplicación debido a la cantidad de conexiones entre las vistas y los datos. Por este motivo crearon una nueva dinámica de funcionamiento, en la cual optimizaron la forma como las vistas se renderizaban cuando había un cambio en los datos de la aplicación.

Veamos algunas de sus características:

- **Composición de componentes**: En programación funcional se pasan funciones como parámetros para resolver problemas más complejos, creando lo que se conoce como composición funcional. En React podemos aplicar este patrón mediante la composición de componentes, unos componentes que encapsulan un comportamiento, una vista y un estado. Crearemos componentes para resolver pequeños problemas, que para ser pequeños son más fáciles de resolver y más adelante resultan más fáciles de visualizar y comprender. Después, unos componentes se apoyarán en otros para resolver problemas mayores y al final la aplicación será un conjunto de componentes que trabajan entre sí. Este modelo es fácil de mantener, de depurar, de escalar, etc.

- **Desarrollo declarativo**: Con librerías más sencillas como jQuery se realiza un estilo de programación imperativo, es decir, se realizan scripts que paso por paso tienen que informar sobre qué acciones se tienen que realizar en el DOM, especificando con detalle cada uno de los cambios. La forma imperativa de declarar obliga a escribir mucho código, porque cada pequeño cambio se tiene que definir en un script y cuando el cambio puede ser provocado desde muchos lugares. El estilo de React es más declarativo, en él contamos con un estado de la aplicación, y sus componentes reaccionan ante el cambio de este estado. Los componentes tienen una funcionalidad dada y cuando cambia una de sus propiedades, ellos producen un cambio.

- **Flujo de datos unidireccional**: Esta es otra acción que facilita React, aunque no es exclusivo. En este modelo de funcionamiento, los componentes de orden superior propagan datos a los componentes de orden inferior. Los de orden inferior trabajarán con estos datos y cuando cambia su estado

podrán propagar eventos hacia los componentes de orden superior para actualizar sus estados.

- **Mejor rendimiento gracias al DOM virtual**: El rendimiento a la hora del renderizado de la aplicación se consigue mediante el DOM virtual. No es que React no opere con el DOM real del navegador, pero sus operaciones las realiza antes sobre el DOM virtual, que es mucho más rápido. El DOM virtual está cargado en memoria y gracias a la herramienta que diferencia entre este y el real se actualiza el DOM del navegador, permitiendo actualizaciones de hasta 60 frames por segundo, y por lo tanto, producen aplicaciones muy fluidas, con movimientos suavizados.

- **Isomorfismo**: Es la capacidad de ejecutar el código tanto en el cliente como en el servidor. También se conoce como «JavaScript Universal». Sirve principalmente para solucionar problemas de posicionamiento tradicionales de las aplicaciones JavaScript.

- **Elementos y JSX**: ReactJS no devuelve HTML. El código embebido dentro de JavaScript parece HTML pero realmente es JSX. Son como funciones JavaScript, pero expresadas mediante una sintaxis propia de React llamada JSX. Lo que produce son elementos en memoria y no elementos del DOM tradicional, con lo cual las funciones no ocupan tiempos al producir pesados objetos del navegador sino simplemente elementos de un DOM virtual.

- **Componentes con y sin estado**: React permite crear componentes de varias maneras, pero hay una diferencia entre componentes con y sin estado. Los componentes stateless son los componentes que no tienen estado (no guardan datos en su memoria). Esto no quiere decir que no puedan recibir valores de propiedades, pero estas propiedades siempre las llevarán en las vistas sin producir un estado dentro del componente. Estos componentes sin estado se pueden escribir con una sencilla función que devuelve el JSX que el componente tiene que representar en la página. Por otro lado, los componentes statefull son algo más complejos porque son capaces de guardar un estado y mantienen lógica de negocio generalmente. Su principal diferencia es que se escriben en el código de una manera más compleja, generalmente por medio de una clase ES6, en la que podemos tener atributos y métodos para realizar todo tipo de operaciones.

- **Ciclo de vida de los componentes**: React implementa un ciclo de vida para los componentes. Son métodos que se ejecutan cuando pasan cosas comunes con el componente, que nos permiten subscribir acciones cuando se produce una inicialización, se recibe la devolución de una promesa, etc.

- **Comportamiento con otras librerías**: A pesar de que React no se encarga de todas las partes necesarias para hacer una aplicación web compleja, la

serie de componentes y herramientas basadas en React nos permiten encontrar una alternativa capaz de hacer cualquier cosa que podríamos hacer con otro framework más complejo. Por otro lado, React solapa completamente las funcionalidades de jQuery, por lo que resulta una evolución natural para todos los sitios que usan esta librería. Podrían convivir pero no es demasiado necesario y a la vez recargaría un poco la página, de forma que tampoco sería muy recomendable.

4.4. Vanilla JavaScript

Realmente, Vanilla JS o Vanilla JavaScript **no es ningún framework**. Precisamente se utilizan estos términos para referirse a código JavaScript sencillo que no está extendido para ningún framework ni ninguna biblioteca adicional. Lo hemos querido incluir en este apartado para aclarar el significado del término.

Las funciones nativas de JavaScript son realmente muy potentes, puesto que se ejecutan a un nivel más bajo que el de cualquier librería y esto les proporciona mayor velocidad a la hora de analizar los nodos de una web y seleccionar los elementos de la forma correcta.

Por otro lado, las mejoras que incorporan los estándares y su adopción por parte de los navegadores hacen que en muchos proyectos, y cada vez más, se vuelva al JavaScript sin florituras.

5. Typescript

TypeScript es un lenguaje de programación de código abierto desarrollado y presentado por Microsoft en 2012. Es un superset de JavaScript que esencialmente añade capacidades de programación orientada a objetos, como el tipado estático y objetos basados en clases.

Extiende la sintaxis de JavaScript por medio de un lenguaje propio que compila ficheros en lenguaje JavaScript original, asegurando la compatibilidad con todos los navegadores, servidores y sistemas operativos.

Es importante resaltar que, al igual que sucede con los preprocesadores de CSS, no se trabaja con el fichero JavaScript, sino que este es el resultado de la compilación del fichero TypeScript.

Asimismo, este es un lenguaje tipado, por lo tanto se tiene que definir el objeto que se está utilizando para poder obtener las propiedades o métodos que soporta. Así, en el ejemplo se identifica el objeto con el ID «nombre_usuario» como un objeto del tipo Input() para poder acceder a su value. En caso de no hacer esta especificación el compilador daría un error y no permitiría continuar.

Otra cosa que facilita mucho la vida a los que utilizamos este lenguaje es que aprovecha todas las capacidades de compilación y depuración de Visual Studio tal como se hace en cualquier lenguaje .NET. Esto quiere decir que podemos poner un punto de control en cualquier lugar y hacer un resumen de inspección u obtener la pila de llamadas de un código JavaScript.

5.1. Superset

Un superset es un lenguaje escrito por encima de otro lenguaje, o mejor dicho, que compila a otro lenguaje. En el caso de TypeScript es un lenguaje que compila a JavaScript y que le añade muchas facilidades y ventajas.

Los lenguajes como JavaScript basados en un estándar a menudo evolucionan de manera más lenta que las necesidades que tienen los desarrolladores. Es por este motivo por lo que surgen empresas y/o comunidades que deciden expandir un lenguaje, aportando todas las herramientas de que carecen para poder desarrollar en las mejores condiciones.

Hay dos supersets especialmente populares para JavaScript: CoffeeScript y TypeScript. La diferencia principal es que mientras que CoffeeScript nos aleja del lenguaje, con TypeScript escribimos en un lenguaje muy similar al propio Ja-

vaScript. Por este motivo, los programadores interesados en utilizar un superset se han decantado principalmente por TypeScript, y por este motivo también es el lenguaje que veremos en detalle.

5.2. Instalación

El compilador de TypeScript está desarrollado en NodeJS, así que si no tenemos este entorno instalado en nuestro equipo, lo primero que tendremos que hacer es ir al sitio web de NodeJS, donde encontraremos las opciones para la instalación en nuestro sistema operativo mediante un instalador con asistente.

Después necesitamos el TSC (*Command-line TypeScript Compiler*), la herramienta que nos permite compilar un archivo TypeScript a JavaScript nativo. Este software es el que está realizado con NodeJS y su instalación se realiza vía NPM con el siguiente comando:

```
npm install -g typescript
```

5.3. Crear y compilar un archivo ts

Los archivos TypeScript se pueden crear y editar desde cualquier editor de texto y tienen la extensión «.ts».

Cualquier código JavaScript compila en TypeScript. Esto quiere decir que en nuestro código TypeScript podemos incluir fragmentos de JavaScript sin que suponga ningún problema.

Para compilar utilizamos el mencionado compilador TSC y mediante el siguiente comando se convertirá en JavaScript nativo:

```
tsc nombredelfichero.ts
```

El anterior es un ejemplo muy sencillo de compilación, pero TSC incluye también un método «watch», que permite vigilar cambios en los archivos TS, automatizando así la compilación.

También tenemos la opción de configurar el archivo «tsconfig.json» donde podemos definir todos los parámetros de compilación que nos interesen, como por ejemplo, el estándar ECMAScript al que compilaremos el código, el tipo de informe de errores que se desea, las rutas donde colocar los archivos compilados, etc.

5.4. Playground

Dentro de la página oficial de TypeScript encontramos un rincón interesante para probar nuestro código sin necesidad de instalar nada, o para revisar rápidamente cómo un código TS compilaría a JavaScript, es el *playground* (o «zona de recreo»):

https://www.typescriptlang.org/play/index.html

Encontramos allí dos cajas de texto: la primera es para escribir código TypeScript y en tiempo real veremos cómo este código compila a JavaScript en la caja de la derecha. Además encontramos ayudas indicando en qué partes de nuestro código tenemos problemas y por qué motivos.

Bibliografía

Angular (Web oficial)

https://angular.io/

AngularJS (Web oficial)

https://angularjs.org/

ARIA (MDN web docs)

https://developer.mozilla.org/es/docs/Web/Accessibility/ARIA

BabelJS

https://babeljs.io/

Documentación TypeScript

https://www.typescriptlang.org/docs/handbook/basic-types.html

ECMAScript 2015 Language Specification (Ecma Internactional – junio del 2015)

http://www.ecma-international.org/ecma-262/6.0/index.html

ECMAScript 2015 Language Specification PDF (Ecma Internactional – junio del 2015)

http://www.ecma-international.org/ecma-262/6.0/ECMA-262.pdf

Jasmine (Github)

https://jasmine.github.io/

Karma (Github)

https://karma-runner.github.io/1.0/index.html

Semanting versioning (SEMVER)

http://semver.org/