JavaScript Documentation

Versión 1.0

José María García Pérez

20 de junio de 2017

Índice general

CAPÍTULO 1

Objetivo

El objetivo es aprender Javascript (que no es un lenguaje especialmente bonito) para crear aplicaciones tanto cliente como servidor, que estén bien estructuradas (AngularJS) y testeables. Para ello usaremos frameworks adecuados para obtener las depedencias.

El objetivo final será el tener el poder hacer un deployment sencillo de la herramienta creada con todas sus dependencias.

CAPÍTULO 2

Contenido

Arquitectura

Introducción

Planteamos la aplicación puramente javascript, tanto para el servidor como para el cliente:

- Servidor: javascript mediante nodejs. Hay frameworks como *express*. Ya veremos la foto.
- Cliente: HTML+CSS+JavaScript + (jquerymobile o dojo o bootstrap o similar). En el cliente interesará usar frameworks como *angularjs*.

La aplicación cliente puede usarse en el móvil con sistemas como *phonegap*.

Infraestructura

Para la descarga de paquetes y gestión de dependencias se usa típicamente:

- **npm**: para el lado servidor
- **bower**: para el lado cliente

Aplicación Cliente

Introducción

Para crear una aplicación cliente, usaremos yeoman. Sirve para estructurar nuestra aplicación, es decir, genera una estructura de directorios estándar. Yeoman se instala globalmente mediante:

```
# npm install -g yo
```

y se actualiza mediante:

```
# npm update -g yo
```

Haremos:

```
$ mkdir new_app
$ yo angular
Would you like to use Sass (with Compass)? (Y/n) Y
Would you like to include Bootstrap? (Y/n) Y
Would you like to use the Sass version of Bootstrap? (Y/n) Y
Which modules would you like to include? <intro>
```

tras bastante tiempo, usa bower para descargar las dependencias y grunt para la ejecución de tareas. La estructura generada tiene la siguiente pinta:

```
./app/
./bower_components/
bower.json
.bowerrc
.editorconfig
.gitattributes
.gitignore
Gruntfile.js
.jshintrc
./node_modules/
package.json
./test/
.travis.yml
```

y dentro del directorio **app** tenemos:

```
404.html
favicon.ico
images/
index.html
robots.txt
scripts/
styles/
views/
```

Pero hay diferentes tipos de generadores, tanto para casos particulares de angular (view, service, ...) o para phonegap (ej. angular-phonegap:app, ...) que sirve para hacer aplicaciones para el móvil.

Queremos que la aplicación sea testeable. Para ello disponemos de jasmine y de karma.

Workflow

Lo típico es crear la aplicación tal y como hemos hecho anteriormente. Después gestionamos las depedencias con bower:

```
$ bower search raphael
$ bower install raphael --save
```

en donde **–save** sirve para que actualice **bower.json**. Lo anterior nos instala Raphaël.

Imprescindible leer esto.

Ejecutamos lo que tenemos mediante:

```
$ grunt serve
```

Qué queremos ver

Empezamos con un pequeño borrador que muestra lo que queremos ver. Por ejemplo, un formulario y una lista.

Tutorial de Javascript

Muy inspirado en Eloquent Javascript.

Contenido:

Introducción a Javascript

JavaScript está basado en ECMA-262, Edición 3. Es un desarrollo de Mozilla. Un matiz interesante es que JavaScript no es orientado a objetos, sino "prototyped-based programming".

JavaScript se usa principalmente en el lado cliente, pero también en el lado servidor. También se usa para desarrollar aplicaciones de escritorio.

HTML y CSS están muy bién, pero JavaScript es quien da la lógica y permite hacer cosas como el drag'n'drop.

Con SpiderMonkey (interprete de Mozilla) que podemos hacer:

```
$ js fichero.js
```

O podemos usar directamente el intérprete:

```
$ js
js> quit()
$
```

En los ficheros podemos poner el "shebang":

```
#!/usr/bin/env js
print("Hola mundo")
```

y podremos ejecutarlo mediante:

```
$ js ex01.js
```

o bien:

```
$ chmod +x ex01.js
$ ./ex01.js
```

Otro javascript shells.

También podemos ejecutarlo dentro de HTML:

```
<script language="javascript" type="text/javascript">
<!--
codigo javascript
// -->
```

```
</script>
<noscript>
Su navegador no soporta Javascript
</noscript>
```

o bien referenciando el código:

```
<script src="http://mundogeek.net/miarchivo.js"></script>
```

Sevidor o cliente

Si es el servidor es el que realiza todo el trabajo tiene las ventajas:

- El código está en el servidor (menor "espionaje")
-

El que el trabajo lo haga el cliente tiene la ventaja de la distribución de la carga de trabajo sobre los clientes.

Básico

Introducción

El propósito de este capítulo es enseñar los aspectos básicos de Javascript como lenguaje. Para practicar alguno de los elementos de este tutorial, podremso usar:

```
$ node
>
```

Comentarios

Son trozos de código que no se ejecutan:

```
// Esto no se ejecuta

/* Esto
   tampoco se ejecuta */
```

Asignación de variables

Haremos:

```
var miVariable=7, otraVariable="mundo";
miVariable="Hola Mundo";
const pi = 3.14159;
```

Nota: vemos que todas las líneas terminan con ";".

Advertencia: "var" no es un elemento opcional para declarar variables. Define el "scope" de la variable. Define un nuevo scope para la variable. Si no aparece "var", se asume que la variable es global.

Nota: "const" es un modificador.

Tipos de datos

Tenemos:

- Números: 5
- Cadenas: "Hola\nqué tal"
- Booleanos: true, false

Además tenemos **undefined** que se usa cuando las variables han sido declaradas pero no asignadas:

```
> var a;
undefined
> a
undefined
> k
ReferenceError: k is not defined
...
```

Tenemos **null** para variables declaradas y se les ha asignado el valor **null** (se asegura que no contiene nada):

```
> var b;
undefined
> b
null
```

Nota: como vemos, la diferencia entre **undefined** y **null** es muy sutil (y poco útil). ¿Puede tener más sentido con funciones?

Tenemos **NaN**.

También tenemos diccionarios:

```
> var a = { nombre: "José María", apellidos: "García Pérez" }
undefined
> a.nombre
'José María'
> a['apellidos']
'García Pérez'
```

Y también arrays:

```
> var primos= [1,3,5,7,11,13]
undefined
> primos[0]
1
> primos[1]
```

```
3
> primos[5]
13
```

Operadores

Aritméticos

operador	significado
+	Suma
-	Resta
*	Multiplicación
/	División
%	Módulo
++	Incremento
–	Decremento
-	Negación (unario)

Relacionales

Operador	Significado
==	Devuelve true si los dos operandos son iguales (si son de distinto tipo se hace una conversión primero)
===	Devuelve true si los dos operandos son iguales y son del mismo tipo
!=	Devuelve false si los dos operandos son iguales (si son de distinto tipo se hace una conversión primero)
!==	Devuelve false si los dos operandos son iguales y son del mismo tipo
>	Mayor que
>=	Mayor o igual que
<	Menor que
<=	Menor o igual que

Advertencia: conviene evitar "==" y "!=" dado que existen conversiones implícitas que no son obvias. Es mejor usar "===" y "!==" que son versiones más estrictas.

Condicionales

Operador	Significado
&&	And (devuelve true si ambos operandos evalúan a true)
||	Or (Devuelve true si alguno de los operandos evalúa a true)
!	Not (unario, devuelve true si la expresión evalúa a false)

Nivel de bit

Operador	Significado
&	And
|	Or
^	Xor
~	Not
<<	Desplazamiento a la izquierda
>>	Desplazamiento a la derecha
>>>	Desplazamiento a la derecha sin signo

Control de flujo

Condicionales

Tenemos "if", "else:

```
if (condicion) {
  expr1;
} else if {
  expr2;
} else {
  expr3;
}
```

Tenemos "switch":

```
switch (expresion){
  case valor1 :
    expr1;
    break;
  case valor2 :
    expr2;
    break;
  ...
  default : exprN;
}
```

Bucles

Tenemos "for":

```
for (inicializacion; condicion; incremento) {
  exprs
}
```

y también:

```
for (variable in objeto) {
  exprs
}
```

Por ejemplo:

```
> var primos= [1,3,5,7,11,13]
undefined
> for (i in primos) {console.log(i)}
1
2
3
4
5
```

Tenemos “while”:

```
while (condicion) {
  exprs
}
```

Tenemos “do...while”:

```
do {
  exprs
} while (condicion);
```

Funciones

Introducción

Definición:

```
function nombre (<argumentos>) {
   <cuerpo de la funcion>
}
```

por ejemplo, la siguiente función pura (no tiene “side-effects”):

```
function multiplica (x, y) {
   return x * y;
}
```

y se usa:

```
// miVar toma como valor el producto de 3 por 2, 6
var miVar = multiplica(3,2);
// ahora miVar vale 60
miVar = multiplica(miVar,10);
```

Lexical Scope

Las variables declaradas fuera de funciones son “globales”.

Las variables declaradas dentro de las funciones serán locales.

Por ejemplo:

```
var x = "fuera";

var f1 = function() {
```

```
  var x = "dentro de f1"; // Usa "var": declaración local
};
f1();
console.log(x);
// Imprimirá: "fuera"

var f2 = function() {
  x = "dentro de f2";  // No usa "var": usa la variable global
};
f2();
console.log(x);
// Imprimirá: "dento de f2"
```

Closure

El scope puede ser anidado, lo que posibilita la definición de "closures": generador parametrizada de funciones:

```
var gen_fun = function gen_adder( param ) {
   // param: es local a "gen_adder"
   return function(x) {  // Anónima
     return x+param
   }
}
```

En donde:

- param: es local a gen_adder y está disponible a todas las funciones anidadas.
- Devuelve un función anónima.

Advertencia: no develver funciones dentro de "if".

Notación

Lo siguiente:

```
var square = function() {...}
```

es equivalente:

```
function square() {...}
```

Argumentos

Las funciones admiten cualquier número de argumentos. Aquellos que no se usan, simplemente se ignoran. Esto se puede aprovechar para hacer que las funciones acepten argumentos opcionales:

```
function power(base, exponent) {
  if (exponent == undefined)
    exponent = 2;
  var result = 1;
  for (var count = 0; count < exponent; count++)
```

```
    result *= base;
  return result;
}
```

Funciones Puras / Funciones con side-effects

Las funciones puras son aquellas que típicamente devuelven un valor. Funciones con side-effects no devuelven ningún valor pero tienen efectos: imprimir en pantalla, modificar variables globales.

Programación funcional

Recursiva

Ejemplo:

```
function map (funcion, matriz) {
  if (matriz.length==0) {
    return matriz;
  else {
    return new Array().concat(car = funcion(matriz.shift())).concat(map(funcion,
↪matriz));
  }
}

function cuadrado (x) {
return x * x;
}

var miArray = new Array(1,2,3,4,5,6);
/*map devolvera una matriz cuyos elementos son el cuadrado de los elementos de␣
↪miArray*/
var resultado = map(cuadrado,miArray);
```

Advertencia: la recursión, aunque elegante, puede resultar hasta 10 veces más lento.

Composición

Ejemplo:

```
function componer (f, g, x) {
   return f(g(x));
}
```

Orden superior

Devolvemos funciones:

```
function operar (operacion) {
switch (operacion){
case "rep":
return reponer;
case "ven":
return vender;
}
}

function reponer (cantidad) {
dinero = dinero - (cantidad * 5);
unidades = unidades + cantidad;
}

function vender (cantidad) {
dinero = dinero + (cantidad * 10);
unidades = unidades - cantidad;
}

var dinero = 1000;
var unidades = 100;
//Ahora tenemos 990 euros y 102 unidades
operar("rep")(2);
//y despues de vender 50, 1490 euros y 52 unidades
operar("ven")(50);
```

Estructuras de Datos

Datasets

Esto son arrays. Pueden contener tipos diferentes:

```
> a = [17, 21, false, "pero"]
[ 17, 21, false, 'pero' ]
> a[2]
false
> a[0]
17
```

Objetos

Se definen mediante:

```
var obj1 = { propiedad_1: valor_1,
             ...,
             propiedad_n: valor_n }
```

en donde "valor_i" puede ser cualquier cosa (función, objeto, ...).

Por ejemplo:

```
> var empleado = { nombre: "José María",
                   edad:   38,
                   roles: ["alfarero", "pintor"],
                   activo: false,
```

```
                    ids: {dni:123456,
                          passport: 9876543}
                    activate: function () {this.activo = true }
                  }
undefined
> empleado["nombre"]
'José María'
> empleado.edad
38
> empleado.edad = 39
39
> empleado.activate()
undefined
> empleado.activo
true
```

En este ejemplo, vemos que el objeto tiene diferentes tipos de datos e incluso una función. Hablaremos de:

- propiedades: cuando contienen valores
- métodos: funciones definidas dentro de un objeto.

Vemos que tenemos diferentes formas de acceder a las propiedades y métodos:

```
obj["nombre"]
obj.nombre
obj["metodo"]()
obj.metodo()
```

En JavaScript todo salvo **null** y **undefined** tiene propiedades:

```
> var a="Hola"
undefined
> a.length
4
> a.toUpperCase()
'HOLA'
> var b = []
undefined
> b.push("Hola", "mundo", "cruel")
3
> a.join("--")
'Hola--mundo--cruel'
> a.pop(1)
'mundo'
> a
[ 'Hola','cruel']
```

Si hacemos referencia a métodos inexistentes:

```
> var a = {hola:"mundo"}
undefined
> a.hola
'mundo'
> a.adios
undefined
```

Nota: aquí puede tener más sentido la diferenciación entre null y undefined.

Podemos borrar propiedades de un objeto y comprobar si existen ciertas propiedades en un objeto:

```
> var a = {hola:"mundo",adios:"trabajo"}
undefined
> "adios" in a
true
> delete a.adios
true
> "adios" in a
false
```

Nota: un array es un objeto especializado en almacenar secuencias de datos.

Mutabilidad

Una instancia podrá hacer referencia a otra instancia y para JavaScript será como si ambas fueran la misma. Sin embargo, dos instancias con las mismas propiedades y métodos se considerarán objetos diferentes:

```
> var obj1 = { edad: 25 }
undefined
> var obj2 = obj1
undefined
> var obj3 = { edad:25 }
undefined
> obj1 == obj2
true
> obj1 == obj3
false
> obj2.edad = 30
30
> obj1.edad
30
```

El comportamiento del resto de tipos no es análogo:

```
> var a = 25
> var b = a
> var c = 25
> a == b
true
> a == c
true
> b = 30
30
> a
25
```

Nota: los valores son inmutables.

Iterar

Podemos iterar en los objetos (y como consecuencia en los arrays):

```
> var a = {hola:"jose",adios:"pedro",callme:function(){console.log("calling")}}
> for (var i in a) { console.log(i) }
hola
adios
callme
undefined
```

Nota: ya vimos un ejemplo con array.

Arrays - Métodos

Tenemos:

- push: añade elementos al final de un array.
- pop: quita el último elemento de un array (o el del índice indicado).
- shift/unshift: son los equivalentes pero en el comienzo del array.
- indexOf: para sacar el índice del primer elemento que se encuentra en un array.
- lastIndexOf: lo mismo pero encuentra el último índice.
- slice: extrae un trozo de array. slice(firstIndex, lastIndex) el primero se incluye y el segundo no.
- concat: pega dos arrays.

Strings - Métodos

Por ejemplo:

- length
- toUpperCase
- toLowerCase
- slice
- indexOf
- trim: elimina los espacios en blanco de una cadena.
- trimRight
- trimLeft
- charAt

Funciones

Objeto “arguments”

Dijimos que una función admite cualquier número de parámetros:

```
> function f1() {}
undefined
> f1("a",31) // funciona sin problemas
undefined
> function f2(a,b,c) {}
undefined
> f2() // funciona sin problemas
undefined
```

Dentro del scope local de cada función existe el objeto **arguments**:

```
> function f() { return arguments.length }
undefined
> f("hola",35,true)
3
> function f() { return arguments }
> f("hola",35,true)
{ '0': hola',
  '1': 35,
  '2': true }
> function f() { return arguments[1] }
undefined
> f("hola",35,true)
35
```

Objeto - Math

El objeto Math agrupa una serie de funciones matemáticas. Tiene los métodos:

- max
- min
- sqrt
- cos / sin / tan / acos / asin / atan
- PI
- random
- floor
- ceil
- round

Objeto - global

Lo que se almacena en el namespace global es accesible en el objeto **global**:

```
> global
{ ...
  ... }
```

Nota: en los navegadores, el namespace global se representa mediante el objeto **window**.

Advertencia: no conviene contaminar con variables el namespace "global". Aumenta el riesgo de sobreescribir algo que no debamos. **JavaScript no avisa si vamos a sobreescribir una variable que ya existe**.

Funciones de orden superior

Introducción

Funciones que llaman a otras funciones o que devuelven funciones (closures), se conocen como funciones de orden superior.

Sirven para esconder el detalle, es decir, proporcionan un mayor nivel de abstracción, permitiéndonos pensar a un mayor nivel de abstracción.

Advertencia: el coste de la elegancia es la eficiencia. Llamar funciones en JavaScript es costoso.

Ejemplos típicos

Hacer forEach, filter, map, reduce, every, some, ...

Por ejemplo:

```
> [NaN,NaN,NaN].every( isNaN )
true
> [5,NaN,false].some(isNaN)
true
```

Composición

Las funciones pueden componerse.

Recursión

Una función puede llamarse a sí misma.

Binding

Todas las funciones tienen el método **bind**. Ello permite crear una nueva función fijando alguno de los parámetros:

```
> function f(a,b) { return a + b}
undefined
> var addtwo = f.bind(a,2)
undefined
> addtwo(4)
6
```

JSON

JSON.stringify

JSON.parse

Bugs / Error Handling

Introducción

JavaScript no ayuda mucho a la hora de encontrar problemas. Podemos activar el modo estricto para intentar que nos ayude un poquito más:

```
> function canYouSpotTheProblem() {
    "use strict";
    for (counter = 0; counter < 10; counter++)
       console.log("Happy happy");
  }
> canYouSpotTheProblem();
ReferenceError: counter is not defined
```

Nota: normalmente, JavaScript habría creado "counter" en el espacio global.

Dado que JavaScript no ayuda y normalmente es mucho más difícil encontrar fallos que programar, la alternativa es realizar mucho testing. Para esto tenemos "testing frameworks".

Podemos usar "console.log(...)".

Podemos usar Firebug en Firefox. El statement **debugger** en el código debería hacer que la herramienta de depuración del navegador pare en ese punto.

Algunas vez el error es inevitable y lo que hay que decidir es qué hacer ante ese valor.

Excepciones

Podemos generar excepciones:

```
> throw new Error("Invalid Position: lat/long should be...")
```

También podemos capturar las excepciones:

```
> try {
     pepe();
  } catch (error) {
     console.log("Ocurrió el error: " + error);
  }
Occurrió el error: ReferenceError: pepe is not defined
```

Matrices

Crear

Creación:

```
// creamos una matriz vacia
a = new Array();
// también podemos crear una matriz vacia pero reservar espacio para n elementos
b = new Array(10);
// o especificar sus elementos a la hora de crear el array
personajes = new Array("Ricenwind", "Mort", "Tata Ogg");
```

Leer y modificar

Leer y modificar:

```
/* Obtenemos el valor del primer elemento de la matriz y lo metemos en la variable␣
↪protagonista */
var protagonista = personajes[0];
/*Para modificar o añadir valores se usa también []*/
personajes[3] = "La Muerte";
```

Las matrices son objetos

Propiedades

“length”:

```
/*Tenemos 4 elementos en personajes, por lo tanto el tamaño de la matriz es 4*/
tamanyo = personajes.length;
```

Métodos

concat

Devuelve el resultado de concatenar las matrices que se pasan como argumento a la matriz sobre el que se llama, sin afectar a ninguna de las matrices involucradas:

```
/*ahora tendriamos la matriz que tiene como elementos los elementos de personajes dos␣
↪veces*/
var personajes = personajes.concat(personajes);
```

pop

Retira el el primer elemento de la matriz y devuelve este como valor de retorno:

```
/*ultimo contendria "La Muerte" y personajes pasaria a ser la matriz con los␣
↪elementos que tenía antes menos el último elemento*/
var ultimo = personajes.pop();
```

push

Añade los elementos que se pasan como argumento a la función a la matriz:

```
/*Ahora personajes contiene al final de la matriz los numeros 1,2 y 3*/
personajes.push(1,2,3);
```

shift

Similar a pop, pero en este caso elimina y devuelve el primer elemento de la matriz, no el último

toString

Devuelve la representación de la matriz como una cadena de texto:

```
/* cadena contendría "Rincewind, Mort, Tata Ogg, La Muerte, Rincewind, Mort, Tata Ogg,
↪ 1, 2, 3 */
var cadena = personajes.toString();
```

Fecha

Date

La fecha de hoy:

```
> var hoy = new Date()
Sun Aug 10 2014 13:42:38 GMT+0200 (CEST)
> hoy.getFullYear()
2014
> hoy.getYear()
114
> hoy.getMonth()
7
> hoy.getDate()
10
> hoy.getHours()
13
> hoy.getMinutes()
42
> hoy.getSeconds()
38
```

También sirve para crear una fecha:

```
> // año, mes, día, horas, minutos, segundos, milisegundos
undefined
> var fecha = new Date(2009, 11, 9, 12, 59, 59, 999)
undefined
> fecha
Wed Dec 09 2009 12:59:59 GMT+0100 (CET)
> fecha.getTime()  // Unix Time: contador de milisegundos
1260359999999
> new Date(1260359999999)
Wed Dec 09 2009 12:59:59 GMT+0100 (CET)
```

Advertencia: en JavaScript, los meses comienzan en “0” y Diciembre es “11”.

Expresiones regulares

Crear RegEx

Haremos:

```
> var patron = /pat[aeo]/;
undefined
```

o:

```
> var patron = new RegExp(“pato”);
```

Advertencia: cuando se usa RegExp, el carácter **** se representa mediante “\\”.

¿Contiene una cadena la expresión regular?:

```
> var a = /abc/
undefined
> a.test("holabcaro")
true
```

Mini recordatorio de RegEx

Por ejemplo:

```
/[0-9]/        // Cualquier número
/[aoc]/        // Las letras 'a', 'o', 'c'
/\d/           // Any digit character
/\w/           // An alphanumeric character (“word character”)
/\s/           // Any whitespace character (space, tab, newline, and similar)
/\D/           // A characters that is not a digit
/\W/           // A non-alphanumeric character
/\S/           // A non-whitespace character
/./            // Any character except for newline
/[^01]/        // Cualquier carácter excepto 0 ó 1.
/\d+/          // Uno o más dígitos
/\d*/          // 0 o más dígitos
/casas?/       // 0 o 1 's'.
/\d{1,2}/      // 1 ó 2 dígitos
{,5}           // De 0 a 5 veces
{5,}           // 5 o más veces
/ca(ro)+/      // Agrupa "ro". Acepta: "ca" ó "caro"
/Hola/i        // Insensible al caso
/^\d+$/        // Comienzo y fin de cadena.
/(gato|pez)/   // Machea: gato o pez.
/.../g         // Reemplaza todo cuando se usa con replace.
```

Advertencia: "w" no captura caracteres internacionales.

Match

Devuelve lo que encuentra:

```
> /\d+/.exec("Las edades son 33, 25 y 48.")
[ '33',
  index: 15,
  input: 'Las edades son 33, 25 y 48.' ]
> "Las edades son 33, 25 y 48.".match(/\d+/)
[ '33',
  index: 15,
  input: 'Las edades son 33, 25 y 48.' ]
```

Cuando hay grupos, aparece el match completo y después el de cada grupo:

```
> "Las edades son 33, 25 y 48.".match(/a(d+)e(s+)/)
[ 'ades',
      'd',
      's',
      index: 6,
      input: 'Las edades son 33, 25 y 48.' ]
```

lastIndex

No existe una forma adecuada de decir "busca desde". Como alternativa, podemos hacer:

```
> var pattern = /y/g;
undefined
> pattern.lastIndex = 3
3
> pattern.exec("xyzzy");
[ 'y', index: 4, input: 'xyzzy' ]
```

Es un uso "visioso" de:

```
> var pattern = /y/g;
undefined
> pattern.exec("xyzzy");
[ 'y', index: 1, input: 'xyzzy' ]
> pattern.exec("xyzzy");
[ 'y', index: 4, input: 'xyzzy' ]
> pattern.exec("xyzzy");
null
```

Replace

Hacemos:

```
> "papa".replace("p","m")
'mapa'
```

pero el primer argumento puede ser una expresión regular:

```
> "paco".replace(/[ao]/,"e")
'peco'
> "paco".replace(/[ao]/g,"e")
'pece'
```

Nota: no existe un "replaceAll". Usaremos **/.../g**.

La potencia aquí reside en la capacidad de backtracking:

```
> "Garcia Perez, Jose\nPerez, Maria Luisa".replace( /([\w ]+), ([\w ]+)/g, "$2 $1" )
'Jose Garcia Perez\nMaria Luisa Perez'
```

Podríamos usar **$&** para la cadena completa.

La segunda parte de **replace** admite una función:

```
> "the cia and fbi".replace(/\b(fbi|cia)\b/g, function(str) { return str.
↪toUpperCase();});
'the CIA and FBI'
```

Search

Es algo parecido a "indexOf" (devuelve el primer índice o -1 si no encuentra nada):

```
> "  word".search(/\S/)
2
> "      ".search(/\S/)
-1
```

Boundary Markers

Mediante **\b** (límite de palabra).

Greedy

(+, *, ?, and {}): son greedy

(+?, *?, ??, {}?): se convierten en non-greedy.

Ejemplo de grupos

Convertir una cadena de texto en fecha:

```
> function findDate(string) {
    var dateTime = /(\d{1,2})-(\d{1,2})-(\d{4})/;
    var match = dateTime.exec(string);
    return new Date(Number(match[3]),
                    Number(match[2]),
                    Number(match[1]));
  }
```

```
> console.log(findDate("30-1-2003"));
Sun Mar 02 2003 00:00:00 GMT+0100 (CET)
```

Uso

Ejemplo:

```
<script>
var patron = /pat[ao]{2}/;
document.write("patopata".search(patron));
document.write("patoa".search(patron));
patron = /(pat[ao]){2}/;
document.write("patopata".search(patron));
document.write("patoa".search(patron));
</script>
```

Otros elementos a tener en cuenta son:

- \d un dígito. Equivale a [0-9]
- \D cualquier caracter que no sea un dígito.
- \w Cualquier caracter alfanumérico. Equivalente a [**a-zA-Z0-9_**].
- \W cualquier caracter no alfanumérico
- \s espacio
- \t tabulador

Modulos

Introducción

Nos ayudan a estructurar el código. Espondremos únicamente aquellos elementos del código que nos interesen. Evitaremos contaminar el espacio de nombres global.

JavaScript no trae nada de fábrica para construir módules.

Si existe una base de datos online con módulos compartidos: http://npmjs.org que controla, además, versiones y dependencias.

Espacio de nombres

Dado que las funciones son los únicos elementos en JavaScript que crean un nuevo espacio de nombre, será lo que usemos para dar la separación de espacio de nombres:

```
> var mimodulo = function () {
    var name="jose";
    return function (cadena) {console.log(cadena + name)}
    }();
undefined
> mimodulo
[Function]
> mimodulo("Mi nombre es: ")
```

```
Mi nombre es: jose
undefined
```

Lo siguiente aisla nuestro código del mundo exterior:

```
> ( f() {...<todo el código>...} )()
```

Nota: Why did we wrap the namespace function in a pair of parentheses? This has to do with a quirk in JavaScript's syntax. If an expression starts with the keyword function, it is a function expression. However, if a statement starts with function, it is a function declaration, which requires a name and, not being an expression, cannot be called by writing parentheses after it. You can think of the extra wrapping parentheses as a trick to force the function to be interpreted as an expression.

Objetos como Interfaces

Si en lugar de devolver una función anónima, devolvemos un objeto con métodos:

```
> var mimodulo = function () {
     var name="jose";
     return {
       printName: function (str) {console.log(str + name)},
       printToday: function () {console.log(new Date}
       }
     }();
undefined
> mimodulo.printName("Mi nombre: ")
Mi nombre: jose
> mimodulo.printToday()
Sun Aug 10 2014 19:23:28 GMT+0200 (CEST)
```

No "mola" el devolver todo el objeto al final de la función. Como alternativa, usamos por *convenio* el objeto **exports** (patrón muy común en browsers):

```
> (function(exports) {
    var names = ["Sunday", "Monday", "Tuesday", "Wednesday",
                 "Thursday", "Friday", "Saturday"];

    exports.name = function(number) {
      return names[number];
    };
    exports.number = function(name) {
      return names.indexOf(name);
    };
  })(this.weekDay = {});
> weekDay.name(2)
'Tuesday'
```

De esta forma, sólo se añade al objeto que se pasa como parámetro (en este caso, "this.weekDay" que contiene "{}" se pasa a "exports").

Nota: dado que "this.weekDay" está fuera de la función, "this" hará referencia al scope global.

Ejemplo

```
var weekDay = function() {
    var names = ["Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday"];
    return { name: function(number) { return names[number]; }, number: function(name) { return names.indexOf(name); }
        };
    }();
```

There are two popular, well-defined approaches to such modules. One is called CommonJS Modules, and revolves around a require function that fetches a module by name and returns its interface. The other is called AMD, and uses a define function that takes an array of module names and a function, and, after loading the modules, runs the function with their interfaces as arguments.

CommonJS

Viene incluido en **nodejs** e implementan una función **require**. Permite importar directamente el código sin la necesidad de "envolverlo". Sólo se exporta aquello que se añade al objeto **exports**.

Este método se podría usar en el navegador, pero con cada **require** se quedaría la página esperando a que cargase del servidor remoto: LENTO. Existe Browserify que descarga la información de fuera. Coge lo que necesita y lo guarda en un único fichero.

AMD (Asynchronous Module Definition)

Carga las dependencias de fondo:

```
define([module_names_dependencies], function (module_names_dependencies) {...})

define(["weekDay", "today"], function(weekDay, today) {
  console.log(weekDay.name(today.dayNumber()));
});
```

Nota: **define** carga las dependencias si todavía no se han cargado.

Por ejemplo RequireJS.

Nota: siempre que se pueda, diseñar funciones puras y sencillas.

NODEJS

Introducción

Nos permite ejecutar JavaScript fuera del navegador. Fue concevido para ejecutar I/O asíncrono (se ejecuta una tarea y cuando ésta termina, se ejecuta una función callback).

Instalamos node:

```
$ yaourt -S nodejs
```

y lo ejecutamos:

```
$ node
```

Tenemos las variables accesibles glboalmente **console** y **process**:

```
> console
{...}
> process
{...}
```

Podemos salir:

```
> process.exit(0)
```

Podemos ver los argumentos con los que se llamó:

```
> process.argv
```

Advertencia: **document** y **alert** no están en el entorno de node.

Módulos

Se basan en CommonJS (usando **require**):

```
> require(<path>)
```

en donde *<path>* puede ser una ruta absoluta o relativa. Si es un nombre, se buscará el módulo en:

- **node_modules/**

Los módulos se pueden instalar mediante **npm**. Los modulos se pueden instalar localmente:

```
$ npm install <nombre>
```

o globalmente:

```
$ npm install -g <nombre>
```

Si la carpeta contiene el fichero **package.json**, instalará todas las dependencias del programa.

Ejemplo en que creamos nuestro nuevo módulo

Contenido de **main.js**:

```
var garble = require("./garble");

// Index 2 holds the first actual command line argument
var argument = process.argv[2];

console.log(garble(argument));
```

Contenido de **garble.js**:

```
module.exports = function(string) {
  return string.split("").map(function(ch) {
    return String.fromCharCode(ch.charCodeAt(0) + 5);
  }).join("");
};
```

Y se ejecuta:

```
$ node main.js JavaScript
Of{fXhwnuy
```

require("fs")

Se usa para interactuar con el sistema de ficheros. Normalmente vienen en dos variantes: asíncrona y síncrona.

Asíncronas

Por ejemplo, para leer un fichero:

```
var fs = require("fs");
fs.readFile("file.txt", "utf8", function(error, text) {
  if (error)
    throw error;
  console.log("The file contained:", text);
});
```

Para leer ficheros como si fueran binarios:

```
var fs = require("fs");
fs.readFile("file.txt", function(error, buffer) {
  if (error)
    throw error;
  console.log("The file contained", buffer.length, "bytes.",
              "The first byte is:", buffer[0]);
});
```

Síncronas

Leer un fichero de texto de forma síncrona:

```
var fs = require("fs");
console.log(fs.readFileSync("file.txt", "utf8"));
```

Mientras ocurre esta acción, el programa estará parado por completo.

require("http")

Se usa para hacer servidor HTTP. Por ejemplo:

```
var http = require("http");
var server = http.createServer(function(request, response) {
  response.writeHead(200, {"Content-Type": "text/html"});
  response.write("<h1>Hello!</h1><p>You asked for <code>" +
                 request.url + "</code></p>");
  response.end();
});
server.listen(8000);
```

El cliente apuntará a http://localhost:8000/hello para recibir el contenido.

La función que se pasa como argumento a *createServer*, se ejecuta siempre que un cliente hace una petición al servidor.

Por otro lado:

- *request* representa petición.
- *response* representa la respuesta.

La respuesta va típicamente como:

```
response.writeHead(<status>, <header>)
response.write(...)
...
response.write(...)
response.end(...)
```

Por último, *server.listen* "arranca el servidor" y comienza a esperar por peticiones.

Este módulo también proporciona la función **request** que permite comportarse como un cliente:

```
var http = require("http");
var request = http.request({
  hostname: "eloquentjavascript.net",
  path: "/20_node.html",
  method: "GET",
  headers: {Accept: "text/html"}
}, function(response) {
  console.log("Server responded with status code",
              response.statusCode);
});
request.end();
```

require("https")

Para comunicaciones seguras.

Streams

Los streams pueden ser de lectura o de escritura. Todos los de escritura tendrán los métodos:

- **write**: escribe datos en el stream.
- **end**: cierra el stream.

Como argumentos a estos métodos, se podrá pasar:

- Una *cadena* o un *Buffer*.

- Un callback opcional (una vez haya terminado la acción).

Por ejemplo, con ficheros:

```
> var fs = require('fs')
undefined
> var fp = fs.createWriteStream('prueba.txt', 'utf-8')
undefined
> fp.write("Hola")
true
> fp.write(" José")
true
> fp.end("\n")
undefined
```

Desde el punto de vista del servidor HTTP:

- La variable *request* será de lectura.
- La variable *response* será de escritura.

Desde el punto de vista del cliente HTTP:

- La variable *request* es de escritura.
- La variable *response* es de lectura.

Eventos

En nodejs, los objetos que emiten eventos, tienen un método llamado **on**:

```
<object>.on( <nombre_del_evento>, <callback> )
```

Los streams de lectura, tienen los eventos:

- 'data'
- 'end'
- 'open'
- 'error'

Para crear un stream de lectura, creamos el fichero:

```
> var fs = require('fs')
> var readStream  = fs.createReadStream('prueba.txt', 'utf-8')
> readStream.pipe( process.stdout )
```

O en un fichero **copy.js**:

```
var fs = require('fs');
console.log(process.argv[2], '->', process.argv[3]);

var readStream = fs.createReadStream(process.argv[2]);
var writeStream = fs.createWriteStream(process.argv[3]);

readStream.on('data', function (chunk) {
  writeStream.write(chunk);
});
```

```
readStream.on('end', function () {
  writeStream.end();
});

//Some basic error handling
readStream.on('error', function (err) {
  console.log("ERROR", err);
});

writeStream.on('error', function (err) {
  console.log("ERROR", err);
});
```

y ejecutamos:

```
$ node copy.js file1.txt file2.txt
```

Nota: cuando *chunk* es un binary buffer, podemos convertirlo a una cadena utf-8 mediante **toString()**.

JavaScript en el Navegador

Introducción

La mayoría de los navegadores incluyen un intérprete de JavaScript.

Ejecutar código

Código embebido

Por un lado es fácil ejecutar javascript en el navegador. Basta con crear un fecho **demo.html**:

```
<!DOCTYPE HTML>
<html>
<body>
<span onclick="alert('Hello World!');">Click Here</span>
</body>
</html>
```

Mostrará el texto "Click Here", y al hacer 'click' mostrará una alerta con el texto "Hello World".

Referencias a ficheros

Podemos crear un fichero externo, referenciarlo en la página web y llamar a las funciones que en él aparecen.

Fichero con el código JavaScript **codigo.js**:

```
function showAlert () {
   window.alert('Hello World!')
}
```

Y lo referenciamos en el HTML:

```
<!DOCTYPE HTML>
<html>
<body>
<span onclick="showAlert();">Click Here</span>
</body>
</html>
```

Consola

En Firefox podemos abrir una consola asociada a una página web mediante *Ctrl+May+K*. En dicha consola podemos ver algunas de las variables globales asociadas:

```
>> window
Window → file:///C:/Users/C08937/Documents/src/javascript/ex01/02.html
```

Si hacemos click en Window, Firefox nos mostrará todos los métodos y parámetros asociados. De los muchos que hay, podemos probar:

```
>> window.alert('Pepe')
```

Pero, sobretodo, destacamos:

```
>> window.document
```

Este último es el que contiene la estructura DOM (Document Object Model).

Peculiaridades

La cantidad de cosas que el código JavaScript puede hacer en el navegador es muy limmitada para minimizar los riesgos de seguridad. A esto se le llama sandboxing.

No todos los navegadores cumplen en igual medida los estándares. Programar en el navegador es un ambiente hostil.

DOM (Document Object Model)

La página web es un documento. Daremos una visión rápida por culturilla, aunque en la práctica tenderemos a usar la librería **jQuery**.

En la consola del navegador:

```
>> window.document.documentElement
<html>
```

A partir de aquí, tenemos una estructura jerárquica que tendrá una pinta parecida a la siguiente:

- html
  - head

    -title
  - body
    - h1
    - div

- p
- a
- ...

Cada nodo tendrá un *nodeType*. Dichos tipos vienen definidos en la propia definición del documento:

```
>> window.document.TEXT_NODE
3
>> window.document.COMMENT_NODE
8
```

Advertencia: el DOM es una interfaz genérica y no está especialmente bien integrado con JavaScript. Por ejemplo *childNodes* es una instancia de *NodeList* que no contiene métodos como *slice* o *forEach*.

Existen métodos que permiten obtener referencias a nodos próximos:

- childNodes
- firstChild
- previousSibling
- nextSibling
- lastChild
- parentNode

Encontrar elementos

Por ejemplo:

```
>> window.document.getElementsByTagName('a')
HTMLCollection [....]
>> window.document.getElementsByTagName('a')[0].href
"file:///C:/Users/C08937/AppData/Roaming/Mozilla/Firefox/Profiles/3870q4ru.default/
↪ScrapBook/data/20140808180713/index.html"
```

De la misma forma tenemos métodos como:

- *window.document.getElementById*
- *getElementsByClassName*

Modificar el documento

Por ejemplo:

- removeChild
- appendChild
- insertBefore
- replaceChild
- createTextNode

Atributos

Normalmente son accesibles como propiedades de la instancia del nodo.

El contenido textual del nodo viene:

- textContent

También se puede usar:

- getAttribute
- setAttribute

Layout

Los elementos de una página web ocupan una posición en el navegador. Podemos obtener la información asociada mediante:

- offsetHeight: altura que ocupa el elemento
- offsetWidth: ancho que ocupa el elemento
- clientHeight: altura disponible en el interior del elemento
- clientWidth: anchura disponible en el interior del elemento
- getBoundingClientRect: devuelve el bounding box en la pantalla (top, bottom, left, right).
- pageXOffset / pageYOffset: nos devuelve el offset asociado al scrolling.

Estilos

Por ejemplo:

```
<p><a href=".">Normal link</a></p>
<p><a id="para" href="." style="color: green">Green link</a></p>
```

Podemos editarlo en JavaScript mediante:

```
var para = document.getElementById("para");
console.log(para.style.color);
para.style.color = "magenta";
```

CSS

Query Selectors

querySelectorAll

Posicionar y animar

Events

Usaremos *addEventListener(<evento>, <callback>)*:

```
var button = document.querySelector("button");
button.addEventListener("click", function(event) {
  console.log("Button clicked.");
});
```

También tenemos *removeEventListener*.

Por otro lado, tenemos *onclick* para los nodos.

Los callback reciben como parámetro el objeto **event**:

Propagación

Los padres reciben eventos producidos en los hijos.

Un *event handler* puede llamar al método *stopPropagation*:

```
event.stopPropagation()
```

También se puede inspeccionar *target* que contiene el nodo en el que se originó el evento:

```
event.target
```

Podemos evitar que ocurra el comportamiento predefinido:

```
event.preventDefault()
```

Advertencia: no conviene hacer esto salvo que tengamos un buen motivo para hacerlo.

Eventos de teclado

Tenemos:

- keydown
- keyup
- keypress

Después inspeccionamos el objeto *event* para ver si su contenido está asociado a las teclas que nos interesan. Para ello veremos el contenido de *keyCode* y *ctrlKey*:

```
addEventListener("keydown", function(event) {
  if (event.keyCode == 32 && event.ctrlKey)
    console.log("Continuing!");
});
```

Eventos del ratón

Por ejemplo:

- mouseup
- mousedown

- click
- dblclick

El evento puede generar:

```
event.pageX
event.pageY
```

El movimiento del ratón:

- mousemove
- mouseover
- mouseout

Advertencia: estos eventos también se propagan. Es útil *relatedTarget* en este caso.

Scroll events

Tenemos el evento:

- scroll

Y podemos analizar los valores de:

```
document.body.scrollHeight
innerHeight
innerWidth
pageXOffset
pageYOffset
...
```

Focus events

Los eventos:

- focus
- blur

son lanzados cuando un elemento tiene el focus.

Load event

Este evento se lanza mientra carga la página:

- load
- beforeunload: antes de cerrar una página.

Ejecución de un script

Qué lanza la ejecución de un script:

- Encontrar <*script*>
- Que se dispare un evento
- La función *requestAnimationFrame* (llama a una función antes del *redraw* de otra página)

> **Advertencia:** los script nunca se ejecutan en paralelo. Si un script tarda mucho en ejecutarse, la página se congela. Existen los **Web Workers**, que permiten generar un entorno de ejecución aislado.

Web workers funcionan mediante el envío de mensajes:

```
var squareWorker = new Worker("code/squareworker.js");
squareWorker.addEventListener("message", function(event) {
  console.log("The worker responded:", event.data);
});
squareWorker.postMessage(10);
squareWorker.postMessage(24);
```

Se puede poner timeout a la ejecución de funciones:

```
var bombTimer = setTimeout(function() {
  console.log("BOOM!");
}, 500);  // 500ms
```

Los timeout también pueden eliminarse:

```
clearTimeout(bombTimer);
```

De la misma forma que para *requestAnimationFrame* existe *cancelAnimationFrame*.

Para funciones que deben ejecutarse cada cierto periodo de tiempo tenemos *setInterval* y *clearInterval*:

```
var ticks = 0;
var clock = setInterval(function() {
  console.log("tick", ticks++);
  if (ticks == 10) {
    clearInterval(clock);
    console.log("stop.");
  }
}, 200);
```

Debouncing

Es la técnica por la que evitamos cierto comportamiento con eventos que se disparan muy rápido.

Objetos en el lado cliente

Objeto DOM

ejemplo:

```
// DOM Nivel 1
document.getElementById('miElemento');
// IE
document.all['miElemento'];
```

Objeto Window

ejemplo:

```
"window.alert("Esto es un ejemplo del metodo alert");
window.alert("Encantado de conocerte " + window.prompt("Y este de prompt, para␣
↪preguntarte por ejemplo como te llamas","Este texto será el que se verá en la caja␣
↪de texto por defecto"));
/*Cuando los métodos o propiedades pertenecen al objeto global no hace falta poner el␣
↪nombre del objeto antes. Podemos escribir simplemente el nombre del método o␣
↪propiedad*/
if(confirm("¿Sabías que para window se puede omitir el objeto sobre el que se hace la␣
↪llamada? Esto es así porque window es el objeto global"))
alert("¿Si lo sabías? Que inteligente");
else
alert("Bueno, pues ya lo sabes.");
```

También:

```
window.open(URL, nombre de la ventana, características, reemplazar entrada actual del␣
↪historial);
```

HTML DOM

El HTML DOM

Servidor

Introducción

Aquí veremos las herramientas útiles para programación en el lado servidor mediante javascript. Tenemos nodejs que es un servidor web. Tenemos express, que es un framework de aplicaciones. Tenemos npm que es un gestor de paquetes.

Contenido

Node.js

Introducción

Es un servidor web escrito es javascript. Se instala mediante:

```
yaourt -S nodejs
```

Instalado ocupa unos 13Mbytes.

Podemos usar *npm* para instalar paquetes, por ejemplo:

```
npm install zmq
```

Las instalaciones son locales al directorio en el que se ejecuta. Existen infinidad de módulos.

Ejemplo básico

Creamos el fichero *ex01.js*:

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
}).listen(1337, '127.0.0.1');
console.log('Server running at http://127.0.0.1:1337');
```

y lo ejecutamos con:

```
$ node ex01.js
```

y vemos su comportamiento con firefox o con curl:

```
$ curl http://127.0.0.1:1337
Hello World
```

Tutorial progresivo

Nodejs + Express + MongoDB

Introducción

Típica three-tier-architecture:

- Client
- Application
- Database

Basado en esto.

Instalaciones

Para ello, en el directorio donde va a residir nuestra aplicación:

```
$ npm install express
```

(o *npm install -g express* si queremos que se instale globalemente)

Creamos un proyecto. Como yo lo he hecho localmente:

```
$ ./node_modules/express/bin/express --sessions nodetest1
```

Modificamos *nodetest1/package.json*

Este paquete controla las dependencias, la versión de la aplicación, ...:

```
 {
    "name": "application-name",
    "version": "0.0.1",
    "private": true,
    "scripts": {
        "start": "node app.js"
    },
    "dependencies": {
        "express": "3.4.4",
        "jade": "*"
    }
}
```

Las dependencias se instalan mediante:

```
$ cd nodetest1 && npm install
```

Si instalamos "express" localmente, ahora volverá a instalarlo dentro de nodetest1 y si lo teníamos instalado globalmente, no será necesario.

Podemos ejecutar la aplicación mediante:

```
$ node app
```

y mostrará:

```
Express

Welcome to Express
```

Es bueno curiosear el contenido del fichero *app.js*. En ésta web se explica.

Nuestra web

Si añadimos en *app.js* lo siguiente:

```
app.get('/helloworld', routes.helloworld);
```

y reiniciamos el servidor, nos dará un error porque no hemos modificado la ruta para que maneja la request.

Para ello, añadimos en *routes/index.js*:

```
exports.helloworld = function(req, res){
   res.render('helloworld', { title: 'Hello, World!' });
};
```

y creamos el template al que estamos llamando *views/helloworld.jade*:

```
extends layout

block content
   h1= title
   p Welcome to #{title}
```

Instalamos MongoDB

Para ello:

```
yaourt -S mongodb
```

ocupa 180Mbytes instalado. ¡Ojo! La versión de 32bits está limitada a unos 2Gbytes de datos.

Creamos la base de datos de nuestra aplicación en uno de los directorios:

```
$ mkdir data
$ mongod --dbpath ./data
[...]
[initandlisten] waiting for connections on port 27017
[...]
```

Nota: Usaríamos lo típico de "systemctl start mongodb" y luego enable si quisiéramos que funcionase todo el rato.

Mientras, en otro shell creamos nuestra base de datos:

```
$ use nodetest1
> db.usercollection.insert({ "username" : "testuser1", "email" :
↪"testuser1@testdomain.com" })
> db.usercollection.find().pretty()
{
   "_id" : ObjectId("5335e58e9d20312c8ea80d75"),
   "username" : "testuser1",
   "email" : "testuser1@testdomain.com"
}
```

Nota: interesa curiosear "mongoDB schema design".

Añadimos más:

```
> newstuff = [{ "username" : "testuser2", "email" : "testuser2@testdomain.com" }, {
↪"username" : "testuser3", "email" : "testuser3@testdomain.com" }]
> db.usercollection.insert(newstuff);
```

Página web que muestra los datos

Queremos que la web muestre lo siguiente:

```
<ul>
   <li><a href="mailto:testuser1@testdomain.com">testuser1</a></li>
   <li><a href="mailto:testuser2@testdomain.com">testuser2</a></li>
```

```
   <li><a href="mailto:testuser3@testdomain.com">testuser3</a></li>
</ul>
```

y para ello necesitaremos un template que tendrá la siguiente pinta:

```
extends layout

block content
   h1.
      User List
   ul
      each user, i in userlist
         li
             a(href="mailto:#{user.email}")= user.username
```

El template lo guardamos en *views/userlist.jade*.

Por último generamos, la conexión a la base de datos y routing.

Modificamos *app.js* para conectar con la base de datos:

```
// New Code
var mongo = require('mongo');
var monk = require('monk');
var db = monk('localhost:27017/nodetest1');
```

añadiéndolo justo después del resto de "require".

Obviamente esto requiere modificar *package.json*:

```
"dependencies": {
       "express": "3.4.4",
       "jade": "*",
       "mongo":"*",
       "monk":"*"
   }
```

y ejecutar de nuevo:

```
$ npm install
```

para que se instalen los modulos "mongo" y "monk".

Añadimos el routing a *app.js*:

```
app.get('/userlist', routes.userlist(db));
```

en donde vemos que pasamos la base de datos como parámetro.

Creamos la ruta en *index.js*:

```
exports.userlist = function(db) {
  return function(req, res) {
      var collection = db.get('usercollection');
      collection.find({},{},function(e,docs){
          res.render('userlist', {
              "userlist" : docs
          });
      });
```

```
  };
};
```

Añadir datos a la base de datos

Asociamos la ruta "newuser" en *app.js*:

```
app.get('/newuser', routes.newuser);
```

y creamos la ruta en *index.js*:

```
exports.newuser = function(req, res){
  res.render('newuser', { title: 'Add New User' });
};
```

Finalmente creamos el template para "newuser":

```
extends layout

block content
   h1= title
   form#formAddUser(name="adduser",method="post",action="/adduser")
      input#inputUserName(type="text", placeholder="username", name="username")
      input#inputUserEmail(type="text", placeholder="useremail", name="useremail")
      button#btnSubmit(type="submit") submit
```

que como vemos, una vez presionamos "submit" aplica el métido "post" en la ruta "adduser".

Finalmente, creamos la ruta que recibirá los datos del formulario. Para ello, modificamos *app.js*:

```
app.post('/adduser', routes.adduser(db));
```

y en *index.js* creamos la lógica:

```
exports.adduser = function(db) {
 return function(req, res) {

     // Get our form values. These rely on the "name" attributes
     var userName = req.body.username;
     var userEmail = req.body.useremail;

     // Set our collection
     var collection = db.get('usercollection');

     // Submit to the DB
     collection.insert({
         "username" : userName,
         "email" : userEmail
     }, function (err, doc) {
         if (err) {
             // If it failed, return error
             res.send("There was a problem adding the information to the database.");
         }
         else {
             // If it worked, set the header so the address bar doesn't still say /
↪adduser
```

```
            res.location("userlist");
            // And forward to success page
            res.redirect("userlist");
        }
    });

  }
}
```

Seguir probando con un ejemplo RESTFUL: http://cwbuecheler.com/web/tutorials/2014/restful-web-app-node-express-mongodb/

Node - ZMQ

Introducción

¿Por qué?

Minitutorial

Para usar el paquete:

```
var zmq = require('zmq');
```

Para crear un socket:

```
var sock = zmq.socket("<SOCKET_TYPE>");
```

en donde tenemos múltiples tipos de socket:

- ZMQ_REQ
- ZMQ_REP
- ZMQ_DEALER
- ZMQ_ROUTER
- ZMQ_PUSH
- ZMQ_PULL
- ZMQ_PUB
- ZMQ_SUB
- ZMQ_XSUB
- ZMQ_XPUB

Hacemos el binding, connect y close:

- Bind:

  sock.bind("tcp://10.0.0.1:5555", function(err) { ... });

- Connect:

  sock.connet("tcp://10.0.0.1:5555");

- Close:

  sock.close();

- BindSync:

  sock.bindSync("tcp://10.0.0.1:5555"); // Se ejecuta una vez el socket es binded, pero se rompe ante un error

Se pueden usar múltiples transportes:

- TCP: mediante "tcp://<address>:<port>", por ejemplo:

  sock.bind("tcp://192.168.1.100:5555"); // IP sock.bind("tcp://eth0:5555"); // Interface eth0 sock.bind("tcp://*:5555"); // Todas las interfaces

- IPC: mediante "ipc://<address>":

  sock.bind("ipc:///tmp/mysocket"); // Requiere permisos r/w en el path

- In-process Communication (en memoria): "inproc://<address>"

  sock.bind("inproc://queue"); // bind antes de conectar // address_name < 256 chars

- PGM (multicast): uso "pgm://<address>;<multicas_address>:<port>":

  sock.bind("pgm://192.168.1.100;239.192.1.1:3055"); // slide 22!

- Encapsulated PGM (multicast)

Enviar y recibir:

- Enviar:

```
sock.send("Mi mensaje");
```

- Recibir:

```
sock.on("message", function (msg) {    // msg = buffer object
   console.log(msg.toString());
});
```

Nota: los mensajes son atómicos. Un mensaje puede tener una o varias partes (sólo limitado por memoria). Asegura que se recibe el mensaje entero o nada.

Envío de mensajes multiparte:

```
sock.send("Primera parte", zmq.ZMQ_SNDMORE);
sock.send("parte final");
```

y la recepción:

```
sock.on("message", function() {
   for (var key in arguments) {
      console.log("Part" +key+": "+arguments[key]);
   };
});
```

Patrones

Publicamos a varios transportes simultáneamente:

```
var zmq = require('zmq'),
    pub = zmq.socket('pub');

pub.bindSync('tcp://127.0.0.1:5555'); // TCP Publisher
pub.bindSync('ipc:///tmp/zmq.sock');  // IPC Publisher

setInterval( function() {
   pub.send("Envia mensajes a ambos publicadores");
), 1000); // cada 1000ms

sub = zmq.socket('sub');  // socket subscriptor
sub.connect('ipc:///tmp/zmq.sock');  // al IPC publisher
sub.subscribe('');
sub.on('message', function(msg) {
   console.log("Recibido: " + msg );
});
```

en otro programa, un subscriptor al transporte TCP:

```
var zmq = require('zmq'),
    sub = zmq.socket('sub');

sub.connect('tcp://127.0.0.1:5555');
sub.subscribe('');

sub.on('message', function(msg) {
   console.log("Recibido via TCP: ' + msg );
});
```

Socket options

Dos formas de hacerlo:

- Mediante “sock.setsockopt(<option>, <value>);”:

  sock.setsockopt(‘identity’, ‘monitor-1’);

- Mediante “sock.<option> = <value>;”:

  sock.identity = ‘monitor-1’;

Esto se usa por ejemplo para filtrar:

```
subscriber.subscribe('MUSIC');
ó
subscriber.setsockopt('subcribe', 'MUSIC');
```

o por ejemplo:

```
subscriber.unsubscribe('MUSIC');
subscriber.setsockopt('subscribe', 'POP');
```

Patrones

Distribución de tareas síncrona

Se usa cuando todo mensaje tiene que ser "contestado". Gestiona sólo un mensaje cada vez. Ciclo enviar-recibir estricto.

Si se envía y no se recibe respuesta, el socket que hace la request no se recuperará.

Ver ejemplo en slide 36.

El Dealer

Reparte juego entre varios.

Cliente

Introducción

Para programar en el lado cliente tenemos frameworks MVC como angularjs o backbonejs o emberjs. También tenemos toolkits como pueden ser dojo (que incluye un framework MVC).

Recomiendo usar AngularJS.

Para el HMI parece que ionic framework está hecho a la medida de AngularJS.

Contenido

AngularJS

Contenido:

AngularJS

Enlaces

Por ejemplo:

- AngularJS en w3schools.

Introducción

AngularJS es un framework para el desarrollo de SPA (Single Page Applications). El objetivo es poder desarrollar aplicaciones modulares y testeables. No es un toolkit gráfico.

Se puede usar para crear una página web, pero también para hacer una aplicación móvil (usando por ejemplo PhoneGap).

Como toolkit para generar HMI para móviles, puede usarse Unity Framework.

AngularJS es un framework para el lado cliente de las aplicaciones.

Ahora la tendencia es MEAN (MongoDB/Mongoose + ExpressJS + AngularJS + NodeJS). Todo en JavaScript.

contexto.

AngularJS es un framework MVC. Alternativa a Dojo (que es un toolkit, que incluye un framework MVC), backbone, ember, ... Tiene la ventaja de tener a Google detrás. Y por lo que dicen, parece bien pensado.

Implementa MVW: Model-View-Whatever. En otras palabras, permite implementar diferentes patrones de diseño: Model-View-Controller, Model-View-Presenter, Model-View-View-Controller, ... En donde:

- Model: contiene los datos por ejemplo en la base de datos.
- View: muestra la interfaz con el usuario. Puede haber una o varias para un mismo set de datos.
- Controller: es quien está en medio controlando la interacción entre ambos. Típicamente, esta capa se ejecuta en el cliente.

Ejemplo sencillo

Por ejemplo:

```
<!DOCTYPE html>
<html>

<body>

<div ng-app="">
  <p>Name: <input type="text" ng-model="name"></p>
  <p ng-bind="name"></p>
</div>

<script src="//ajax.googleapis.com/ajax/libs/angularjs/1.2.15/angular.min.js"></
↪script>

</body>
</html>
```

Nota: interesa poner el script al final, para que cargue la página rápidamente y no se quede esperando a que cargue el script.

Directivas

Permiten extender el HTML. Son atributos en los tags HTML (como se puede ver en la página anterior). Por ejemplo:

- ng-app: indica que es una aplicación de Angular.
- ng-model: envía valores de alguna entrada de la vista HTML a la variable *$scope* del controlador.
- ng-bind: hace lo opuesto a *ng-model*, es decir, trae una variable de *$scope* a la página HTML.
- ng-init: sirve para inicializar datos de la aplicación. (Sería el valor por defecto). Se usa poco, porque es parte de la responsabilidad de los controladores que veremos después.
- ng-repeat: sirve para iterar en una lista.

También tenemos:

- ng-disabled: como *ng-bind*, afectando el atribut HTML "disable".
- ng-show: como *ng-bind*, afectando el atribut HTML "show".

También tenemos eventos:

- ng-click: define que es lo que hace cuando se hace click. Puede aparecer una expresión de AngularJS que veremos a continuación.

Nota: aunque las directivas comienzan con **ng-**, HTML5 puede extenderse. Para hacerlo compatible con HTML5, hay que hacer **data-ng-**.

Las últimas directiva que veremos son:

- ng-controller: ésta la dejamos para más tarde donde hablaremos de controladores y de *$scope*.
- ng-view: es sustituido en función del routing que indiquemos. Lo veremos posteriormente.

Expresiones y Data Binding

Dentro de la página HTML, podemos añadir expresiones, que son como pequeños trozos de código JavaScript. El resultado de su ejecución aparecerá en el HTML. Aparecen como:

```
{{ <codigo> }}
```

Hablamos de Data Binding a la referencia a variables dentro de *$scope*. Por ejemplo:

```
{{ empleado.nombre }}
```

Las expresiones pueden tener la pinta:

```
{{ 5 + edad + 8 }}
{{ firstName + " " + lastName[0] + "" + lastName[1] }}
```

Filtros

Dentro de las expresiones, podemos añadir filtros. Por ejemplo:

```
{{ person.lastName | uppercase }}
{{ person.name | lowercase }}
{{ (cantidad * precio) | currency }
```

También hay directivas que aceptan filtros, como a **ng-repeat**:

```
<li ng-repeat="x in names | filter:surname  | orderBy:'country'" >
```

Routing

Especificamos las rutas:

```
angular.module('F1FeederApp', [
  'F1FeederApp.services',
  'F1FeederApp.controllers',
  'ngRoute'
]).
config(['$routeProvider', function($routeProvider) {
  $routeProvider.
```

```
    when("/drivers", {
       templateUrl: "partials/drivers.html",
       controller: "driversController"}).
    when("/drivers/:id", {
       templateUrl: "partials/driver.html",
       controller: "driverController"}).
    otherwise({
      redirectTo: '/drivers'});
}]);
```

En la generación del módulo, usamos el método **config**.

Y la página principal podría tener la pinta:

```
<!DOCTYPE html>
<html>
<head>
  <title>F-1 Feeder</title>
</head>

<body ng-app="F1FeederApp">
  <ng-view></ng-view>
  <script src="bower_components/angular/angular.js"></script>
  <script src="bower_components/angular-route/angular-route.js"></script>
  <script src="js/app.js"></script>
  <script src="js/services.js"></script>
  <script src="js/controllers.js"></script>
</body>
</html>
```

Views

Las vistas funcionan como si fueran templates (me recuerda un poco a jinja2). Es básicamente un fichero HTML que será compilado (¿en el cliente?):

<!DOCTYPE html> <html ng-app> <head>

<title>Hola Mundo</title> <script type="text/javascript"

src="http://ajax.googleapis.com/ajax/libs/angularjs/1.0.7/angular.min.js"></script>

</head> <body>

Escribe un nombre: <input type="text" ng-model="sometext" />

<h1>Hola {{ sometext }}</h1>

</body> </html>

Este fichero se puede ver perfectamente en un navegador. Hay que notar que:

- ng-app: indica cuál es el scope de la aplicación.
- script: hace referencia a angular.min.js.
- ng-model: esto es una directiva de AngularJS. Básicamente define una variable (en este caso "sometext"), de forma que puede ser referenciado con *{{ sometext }}*. Conforme se edita el texto en "input", se modifica allí donde se referencia.

Controllers

El controller viene a ser un "intermediario" que adapta la información entre su origen y la vista.

En la práctica, un controlador es simplemente una función que tiene como parámetro el objeto **$scope**:

```
function ($scope) {...}
```

El objeto *$scope* contiene las variables y métodos con los que interactúan las directivas con las que hablábamos con anterioridad.

Un ejemplo mínimo sería:

```
function personController($scope) {
    $scope.person = {
        firstName: "José",
        lastName: "García"
        fullName: function () {
           return $scope.person.firstName + " " + $scope.person.lastName;
           }
    };
}
```

Sin embargo, la funcion anterior se crea en el espacio de nombres global (y eso NO mola). Lo que haremos es incorporarlo a un módulo de AngularJS:

```
var app = angular.module("myApp", []);

app.controller("myCtrl", function($scope) {
    $scope.firstName = "José";
    $scope.lastName = "García";
});
```

Ello crea un módulo llamado "myApp" y añade el controlador "myCtrl" y define la función asociada.

Servicios

Lo normal no será que los datos estén *hardcoded* en el controlador. Lo normal será que se lean los datos de un servidor o de una base de datos. Para ello existen los servicios.

Tenemos el servicio **$http**:

```
var promesa = $http.get( <url> )
```

que devuelve una "promesa". Dada la naturaleza asíncrona de *$http*, se devuelve un objeto que contiene las funciones *success* y *failure*:

```
promesa.success( function (response) {
  <hacemos algo con la información recibida>
  }
)
```

Angular admite cualquier arquitectura. Podemos hacer que el servicio forme parte del controlador. Algo así:

```
function customersController($scope,$http) {
    $http.get( 'http://server.com/service.php');
```

```
        .success(function(response) {$scope.names = response;});
}
```

que como vemos, lee información de un servidor y la incorpora a *$scope*.

Sin embargo, es deseable la separación de "concerns".

En ese sentido podemos crear nuestro propio servicio que implementará la API con la que se comunica con un determinado servidor/servicio:

```
angular.module('F1FeederApp.services', []).
  factory('ergastAPIservice', function($http) {

    var ergastAPI = {};

    ergastAPI.getDrivers = function() {
      return $http({
        method: 'JSONP',
        url: 'http://ergast.com/api/f1/2013/driverStandings.json?callback=JSON_
↪CALLBACK'
      });
    }

    return ergastAPI;
  });
```

¿qué hemos hecho?:

- Hemos creado un modulo para los servicios "F1FeederApp.services"..
- Hemos creado el sevicio "ergastAPIservice" y lo hemos añadido al módulo. Pasamos como parámetro *$http* para declarar esa dependencia.

¿Cómo se usaría esta API desde un controlador?:

```
angular.module('F1FeederApp.controllers', []).
  controller('driversController', function($scope, ergastAPIservice) {
    $scope.nameFilter = null;
    $scope.driversList = [];

    ergastAPIservice.getDrivers().success(function (response) {
        //Dig into the responde to get the relevant data
        $scope.driversList = response.MRData.StandingsTable.StandingsLists[0].
↪DriverStandings;
    });
  });
```

Para la siguiente vista:

```
<!DOCTYPE html>
<html>
<head>
    <title>Ejemplo2</title>
    <script type="text/javascript"
        src="http://ajax.googleapis.com/ajax/libs/angularjs/1.0.7/angular.min.js" />
    <script type="text/javascript" src="controller.js" ></script>
    <script src="app.js" ></script>

</head>
```

```
<body ng-app="EjemploApp" ng-controller="mivistaController">
    <div ng-repeat="f in files">
       <p>{{ f.name }} {{f.extension}}</p><br/>
    </div>
</body>
</html>
```

notar que:

- ng-app: ahora asignamos un nombre de aplicación (puede haber múltiples aplicaciones y así acotamos su scope).
- ng-controller: identifica el controller que aplica. El contenido del controller es accesible dentro del scope.

Creamos el fichero *controller.js* de la siguiente forma:

```
angular.module('EjemploApp.controllers', []).
controller('mivistaController', function($scope) {
  $scope.files = [
    {
      name: 'foto_bonita',
      extension: 'jpg'
    },
    {
      name: 'imagen',
      extension: 'png'
    }
  ];
});
```

Para terminar, ya tenemos que definir la aplicación *app.js*:

```
angular.module('EjemploApp', [
  'EjemploApp.controllers'
]);
```

Services

AngularJS proporciona los siguientes servicios para leer datos de un servidor:

- $http: es una capa sobre XMLHttpRequest y JSONP. Devuelve una promesa.
- $resource: aporta mayor nivel de abstracción.

Creamos el fichero *services.js*. Encapsula las comunicaciones con el servidor:

```
angular.module('EjemploApp.services', []).
  factory('lyricsAPIservice', function($http) {

    var lyricsAPI = {};

    lyricsAPI.get = function() {
      return $http({
        method: 'JSONP',
        //url: 'http://api.chartlyrics.com/apiv1.asmx/SearchLyric?artist=Bjork&
↪song=Hunter'
        url: 'http://api.lyricsnmusic.com/songs?api_
↪key=d39c276574d9bcffa3804b0f97bde1&q=Bjork%20Hunter&callback=JSON_CALLBACK'
      });
```

```
    }

    return lyricsAPI;
  });
```

El controller es el que adapta dichos datos a cada vista:

```
angular.module('EjemploApp.controllers', []).
  controller('mivistaController', function($scope, lyricsAPIservice) {
  $scope.files = [];


  lyricsAPIservice.get().success(function (response) {
        //Dig into the responde to get the relevant data
        //$scope.files =[{'name':'hola','ext':'png'}];
        //var xml = response;
        //$scope.files = [{'name':'hola','ext':'png'}]; //.MRData.StandingsTable.
↪StandingsLists[0].DriverStandings;
        $scope.files = response.data;//[{'name':'hola','ext':'png'}];
    });
});
```

Tenemos que añadir el servicio a app.js:

```
angular.module('EjemploApp', [
  'EjemploApp.controllers',
  'EjemploApp.services'
]);
```

Y por último, la vista será:

```
<!DOCTYPE html>
<html>
   <head>
       <title>Ejemplo2</title>
       <script type="text/javascript"
           src="http://ajax.googleapis.com/ajax/libs/angularjs/1.0.7/angular.min.js">
↪</script>
       <script type="text/javascript"  src="app.js" ></script>
       <script type="text/javascript" src="services.js"></script>
       <script type="text/javascript" src="controller.js"></script>
   </head>
   <body  ng-app="EjemploApp" ng-controller="mivistaController">
       <div ng-repeat="f in files">
          <p> {{ f.artist.name }} {{ f.title }} {{ f.url }}</p><br/>
       </div>
   </body>
</html>
```

Ejemplo mínimo

Un ejemplo de Model sería:

```
//Model: Objetos JavaScript
$scope.files = ['foo','bar', 'baz'];
```

que guarda un array de nombres. Son los datos que vamos a presentar o malipular.

Un ejemplo de View sería:

```
<!-- View: HTML -->
<div ng-repeat="f in files"></div>
```

que es un trozo de HTML. Aparece *ng-repeat* que es una directiva.

Por último, un ejemplo de Controller:

```
//Controller: Código Javascript
function addFile(fileName){
  $scope.files.push(fileName);
}
```

Qué hay que saber

1. Implement the basic nuts-and-bolts of Angular: views, models, scope, two-way databinding, and controllers. Our hands-on how-to guide.
2. Routing – how to use Angular's page routing system to implement single-page-applications that are the new popular approach to modern Javascript/AJAX style development
3. Directives – how Angular allows us to create special active extensions to HTML
4. CSS integration and creating polished interfaces

Enlaces

- Empezando
- Primera aplicación
- Tutoriales

Primera Aplicación

Prerrequisitos

Necesitaremos **nodejs**:

- grunt
- angular-
- Ensayar:
    - jasmine:
    - karma:

Creamos la aplicación

Haremos:

```
$ mkdir ex01
$ yo angular
```

Comienzo

Comenzamos concentrándonos en el resultado: ¿qué es lo que queremos obtener?:

1. Queremos una tabla que muestre los aeropuertos de la Dafif.
2. Queremos que la tabla contenga campos con los que podamos filtrar la información.
3. Queremos que las cabeceras permitan ordenar la información.

Solución

Borrador

Lo primero es hacer un borrador en HTML que indique más o menos cómo se mostrará la información. Por ejemplo:

```
<!doctype html>
<html>
<head>
  <title>ARPT</title>
</head>
<body>
  <table>
    <thead>
      <tr><th colspan="4">Airports List</th></tr>
      <tr><th>ID</th><th>Name</th><th>ICAO</th><th>Lat</th><th>Long</th></tr>
      <tr>
         <th><input></input></th>
         <th><input></input></th>
         <th><input></input></th>
         <th><input></input></th>
         <th><input></input></th>
      </tr>
    </thead>
    <tbody>
      <tr>
        <td>1</td>
        <td>EMJD</td>
        <td>Barajas</td>
        <td>35.456</td>
        <td>-1.0</td>
      </tr>
      <tr> <!-- ng-repeat="driver in driversList" -->
        <td>2</td>
        <td>ELAA</td>
        <td>Getafe</td>
        <td>36.456</td>
        <td>-0.3</td>
      </tr>
    </tbody>
  </table>
```

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

Usando directivas de AngularJS (parecido a los templates), podemos:

- Se usan vistas, por lo que dejamos de momento la tabla en un * *.
- En el * * le damos un nombre a la aplicación y definimos su controlador.
- Referirnos a variables de *$scope* en *<input>*.
- Las filas de la tabla pasar a ser una iteración de la lista.

Nota: llamaremos directivas a los atributos que empiezan por **ng-**.

quedará algo como:

```
<div ng-app="ARPT_ReaderApp" ng-controller="ARPTs_Controller">
  <table>
    <thead>
      <tr><th colspan="4">Airports List</th></tr>
      <tr><th>ID</th><th>Name</th><th>ICAO</th><th>Lat</th><th>Long</th></tr>
      <tr>
         <th><input type="text" ng-model="id"></input></th>
         <th><input type="text" ng-model="icao"></input></th>
         <th><input type="text" ng-model="name"></input></th>
         <th><input type="text" ng-model="lat"></input></th>
         <th><input type="text" ng-model="lon"></input></th>
      </tr>
    </thead>
    <tbody>
      <tr ng-repeat="arpt in arptList">
        <td>{{$index+1}}</td>
        <!-- td>
          <img src="img/flags/{{driver.Driver.nationality}}.png" />
          {{driver.Driver.givenName}} {{driver.Driver.familyName}}
        </td -->
        <td>{{arpt.ICAO}}</td>

        <td>{{arpt.name}}</td>
        <td>{{arpt.lat}}</td>
        <td>{{arpt.lon}}</td>
      </tr>
    </tbody>
  </table>
</div>
```

HMI

Introducción

Podemos usar HTML puro y duro o podemos usar algún toolkit. Usaremos por ejemplo dojo.

Toolkits

Ensayos

Tenemos Karma, Jasmine, y para end-to-end ...