Capítulo 1: Entendiendo JavaScript

La empresa La Buena Espina es una cadena de restaurantes de comida peruana que logró crecer gracias al boom gastronómico local. De tener un local familiar ahora tienen varios restaurantes en todo el país, por lo que decidieron lanzar un sitio web donde muestren información sobre su carta y sus locales.

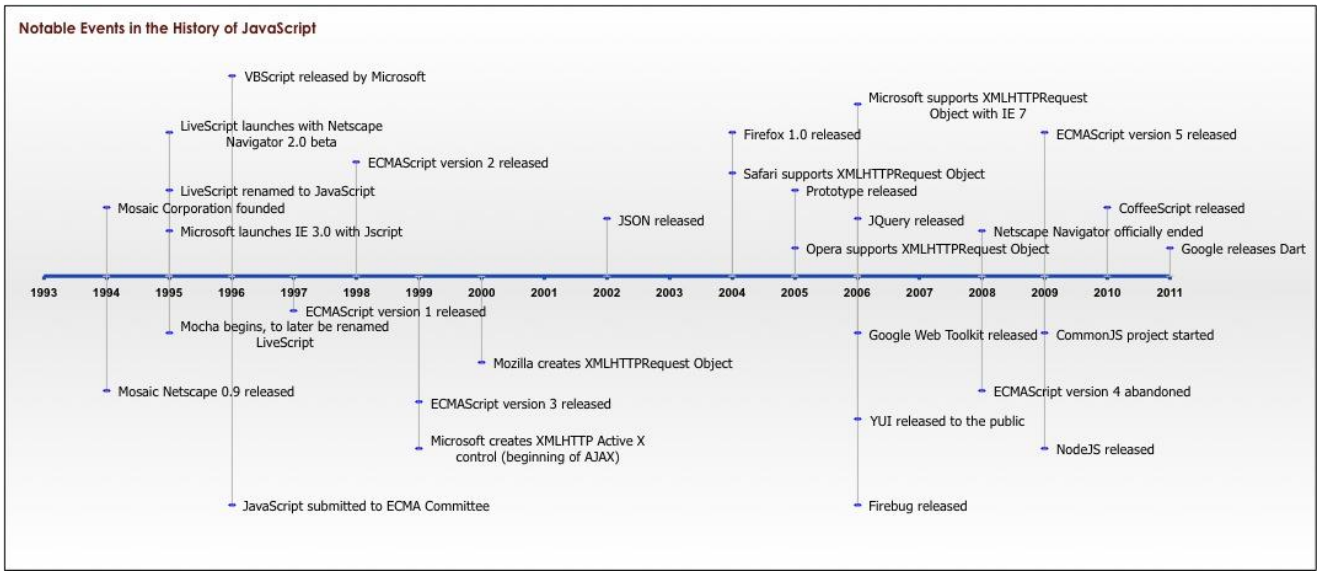
El dueño de La Buena Espina te ha pedido personalmente realizar el sitio y quiere que visitarla sea una experiencia tan buena como su comida, así que es tu deber como desarrollador crear una aplicación con contenido fácilmente mantenible y de un aspecto visual impactante.

Entendiendo JavaScript

JavaScript es un lenguaje de programación dinámico orientado a objetos creado en 1995 por Brendan Eich. El uso del nombre *Java* fue una decisión comercial debido al auge que tenía dicho lenguaje en aquel entonces, pero no están relacionados más allá de algunas similitudes en la sintaxis.

Al momento de su creación existieron diferentes implementaciones del mismo, haciendo caótico su uso. Esto, sumado al nombre, que ocasionaba confusiones con respecto a su funcionamiento, y algunos errores de diseño, hizo que se volviera un lenguaje subestimado y mal usado por mucho tiempo.

Para 1997 la ECMA, una organización creada para desarrollar estándares de comunicación e información, realizó una especificación estándar llamada ECMAScript, la cual debe ser implementada por todos los navegadores. JavaScript en sí no es sinónimo de ECMAScript, si no una implementación de esta, al igual que ActionScript o JScript. ECMAScript actualmente está en su versión 5.1, existiendo ya avances de la versión 6.



JavaScript Timeline. http://tom-barker.com/

JavaScript es un lenguaje que está influenciado por muchos otros lenguajes. Tiene similitudes con Java (y por lo tanto, algo de C/C++), y un poco de Self y Scheme, logrando hacer de él un lenguaje imperativo (se le dice al computador qué hacer y cómo, como C/C++ y Java), pero con conceptos de programación funcional (los programas son escritos en forma de funciones aritméticas, gracias a Self y Scheme). Además, tiene sus propias características:

- **Es un lenguaje de tipado dinámico**, esto quiere decir que una variable puede ser tanto un número como una cadena de caracteres o un objeto sin necesidad de una conversión especial.
- **Es orientada a objetos**, pero con la particularidad que no tiene clases. Las clases son reemplazadas por funciones y los prototipos permiten manejar herencia simple.
- **Las funciones también son objetos**, por lo que tienen atributos y métodos, además de poder ser asignados a variables y ser devueltos por otras funciones.
- **Permite evaluar sentencias en tiempo de ejecución**, así que se pueden crear y ejecutar sentencias (y funciones) a partir de datos ingresados en el programa en ejecución.

Si bien su propósito inicial fue el de ser un lenguaje de scripting para web, actualmente es usado en muchos otros entornos, desde realizar scripts para Adobe Photoshop y manejar bases de datos como CouchDB hasta servir como interfaz para manejar hardware o levantar aplicaciones del lado del servidor con Node.js.

Sintaxis básica

Tipos

Números

JavaScript no tiene tipos de datos específicos para números enteros y flotantes (como `short`, `long`, `float` o `double`), solo tiene números, los cuales pueden tener o no punto flotante.

Las operaciones aritméticas básicas están soportadas, así como el operador módulo (`%`)

```
1 + 10;
// 11

0.5 + 12.2;
// 12.7

11 - 9;
// 2

19.8-0.5;
// 19.3
```

```
10 / 2;
// 5

310/15.5;
// 20

3 * 4;
// 12

21.2 * 7;
// 148.4

6 % 4;
// 2

12 % 7.5;
// 4.5
```

Así mismo, JavaScript tiene un objeto `Math`, el cual contiene operaciones matemáticas adicionales, así como constantes. Se debe tener en cuenta que las operaciones trigonométricas (`sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `atan2`) trabajan con ángulos en radianes.

```
Math.E;
// 2.718281828459045

Math.PI;
// 3.141592653589793

Math.pow(6, 2); // potencia de 6 elevado a 2
//36

Math.tan(45 * (Math.PI / 180)); // Convertimos 45 grados a radianes
// 0.9999999999999999
```

En el último ejemplo se puede ver que el resultado da `0.9999999999999999` cuando debería ser `1`. Esto sucede porque internamente el lenguaje guarda los números en formato binario con un número limitado de dígitos.

En formato binario, una fracción tiene su representación finita solo si el denominador tiene al 2 como único factor primo. Es por esto que, por ejemplo, expresar 1/10 (0.1) en base 2 ($0.00011001100110011..._2$) no tiene una representación finita, ya que el denominador (10) está compuesto por dos factores, siendo ambos números primos: 2 y 5. Una explicación más detallada puede ser encontrada en [What Every JavaScript Developer Should Know About Floating Points].

Cadenas

Las cadenas en JavaScript son valores que pueden ser escritos tanto con comillas simples como dobles:

```
"Bienvenidos a La Buena Espina".length;
// 29

// o...

'Bienvenidos a La Buena Espina'.length;
// 29
```

Las cadenas también son objetos, por lo que tienen propiedades (como `length`) y métodos:

```
'Bienvenidos a La Buena Espina'.toUpperCase();
// "BIENVENIDOS A LA BUENA ESPINA"

'La Buena Espina'.split('');
// ["L", "a", " ", "B", "u", "e", "n", "a", " ", "E", "s", "p", "i",
"n", "a"]
```

La propiedad `length` en una cadena es de solo lectura.

Booleanos

El tipo de dato lógico o booleano solo puede tener dos valores: `true` (verdadero) y `false` (falso). JavaScript tiene la particularidad de convertir implícitamente valores que son interpretados como verdaderos o falsos: `false`, `0`, `""` (o `''`), `NaN` (*Not A Number*), `null` y `undefined` son interpretados como `false` en una condicional, mientras que el resto de valores posibles son interpretados como `true`.

Variables

Las variables en JavaScript permiten guardar objetos para su posterior uso. Al ser JavaScript un lenguaje de tipado dinámico, una misma variable puede guardar diferentes tipos u objetos a lo largo de su ciclo de vida. Una variable es declarada utilizando la palabra reservada `var` y puede o no tener un valor inicial:

```
var siteTitle = 'Bienvenidos a La Buena Espina';
var currentUser;

siteTitle;
// "Bienvenidos a La Buena Espina"
currentUser;
// undefined
```

Arrays

Los arreglos (*arrays*) son un tipo especial de objetos y representan colecciones que pueden guardar cualquier tipo de dato en JavaScript y sus elementos pueden o no ser del mismo tipo.

Los arreglos tienen una propiedad llamada `length` y utilizan los corchetes (`[]`) para acceder a un elemento del arreglo a través de su índice. En JavaScript, el índice de los arreglos empieza en 0 y el valor de `length` es igual al último índice del arreglo más uno.

Al ser objetos, los arreglos tienen una serie de métodos que sirven para manipularlos:

join

Concatena los elementos de un arreglo en una cadena usando el parámetro que recibe este método como separador.

```
var dateParts = [19, 8, 1990];
dateParts.length;
// 3

dateParts.join('/');
// "19/8/1990"

var menuCategories = ['Entradas', 'Segundos', 'Postres'];
menuCategories.join(', ');
// "Entradas, Segundos, Postres"
```

pop

Quita el último elemento del arreglo y retorna su valor.

push

Agrega un elemento al final del arreglo y retorna el nuevo tamaño.

```
var dateParts = [19, 8, 1990];
dateParts.pop();
// 1990

dateParts;
// [19, 8]
dateParts.push(1989);
// 3

dateParts;
// [19, 8, 1989]
```

indexOf

Busca en el arreglo el primer elemento que sea igual al parámetro que se le pasa y devuelve su índice.

```
var dateParts = [19, 8, 1990];
dateParts.indexOf(8);
// 1
```

`indexOf` está definido en Internet Explorer 9 y superiores, y en Firefox 1.5 y superiores.

reverse

Modifica el arreglo invirtiendo sus elementos y retorna el arreglo invertido.

```
var dateParts = [19, 8, 1990];
dateParts.reverse();
// [1990, 8, 19]

dateParts;
// [1990, 8, 19]
```

concat

Agrega elementos a una copia del arreglo original y devuelve la copia con los nuevos elementos agregados.

```
var dateParts = [1990, 8, 19];
dateParts.concat(9, 30, 0);
// [1990, 8, 19, 9, 30, 0]

dateParts;
// [1990, 8, 19]
```

slice

`slice` crea una copia del arreglo de acuerdo a los parámetros que esta función recibe: el primer parámetro es el índice del elemento en el que inicia la copia y el segundo argumento es el índice siguiente al elemento final de la copia. El segundo parámetro es opcional, y si se obvia, la copia se realizará desde el índice inicial hasta el final del arreglo original.

Esta función devuelve la copia del arreglo y deja el arreglo original intacto.

```
var array = [8, 20, 12, 9, 1];
array.slice(2, 4);
//[12, 9]

array;
```

```
// [8, 20, 12, 9, 1]
```

splice

`splice` modifica el arreglo original, de acuerdo a los parámetros que recibe. El primer parámetro es el índice del elemento donde se empezará a cortar, mientras que el segundo parámetro es el número de elementos que se cortarán del arreglo original, incluyendo el elemento inicial. Puede recibir uno o más parámetros opcionales, los cuales se insertan en la posición definida en el primer parámetro.

Esta función devuelve el nuevo arreglo cortado y modifica el arreglo original.

```
var array = [8, 20, 12, 9, 1];
array.splice(2, 1);
// [12]

array;
// [8, 20, 9, 1]

array.splice(1, 0, 15);
// []

array;
// [8, 15, 20, 9, 1]
```

Objetos

Los objetos literales son colecciones de pares nombre-valor donde la primera parte (el *nombre*) es único dentro del objeto y por lo general es una cadena, mientras que la segunda parte (el *valor*) puede ser de cualquier tipo, incluyendo otros objetos.

Se puede crear un objeto de tres formas:

A. De forma literal:

```
var object = {};

object;
// {}
```

B. Creando una instancia de `Object`:

```
var object = new Object({});

object;
// {}
```

La tercera forma utiliza `Object.create`, por lo que para entenderla tenemos que ver un poco más a detalle cómo se comportan las propiedades de un objeto literal.

La forma B crea un objeto del mismo constructor del parámetro que se le pase:

```
new Object({});
// Object {}
new Object(1);
// Number {}
new Object("");
// String {}
new Object(true);
// Boolean {}
new Object([]);
// []
```

Existen dos formas para asignar y acceder al valor de una propiedad en un objeto:

```
var dish = {
  name: 'Ceviche simple',
  ingredients: [
    '1 kilo de pescado',
    '2 cebollas',
    '1 taza de jugo de limón',
    '1 ají limo',
    'sal'
  ],
  garnishes: [
    'lechuga (2 hojas por plato)',
    'maíz cancha',
    '4 porciones de yuca',
    '4 choclos sancochados',
    'camote sancochado en rodajas (2 por plato)'
  ],
  diners: 4
};
```

A. Con punto:

```
dish.name;
// "Ceviche simple"

dish.name = 'Ceviche simple (estilo trujillano)';
// "Ceviche simple (estilo trujillano)"
```

B. Con corchetes:

```
dish['name'];
// "Ceviche simple"

dish['name'] = 'Ceviche simple (estilo trujillano)';
// "Ceviche simple (estilo trujillano)"
```

La forma B tiene una ventaja importante con respecto a la forma A, debido a que se accede a la propiedad con el nombre de esta propiedad en forma de cadena, permitiendo utilizar una variable cuyo valor sea definido dinámicamente:

```
var dinersPropertyName = 'name';

dish[dinersPropertyName];
// "Ceviche simple"

dish[dinersPropertyName] = 'Ceviche simple (estilo trujillano)';
// "Ceviche simple (estilo trujillano)"
```

Cada propiedad definida en un objeto tiene por defecto las siguientes características:

- Es enumerable: Saldrá listada si se recorre el objeto en un estructura repetitiva o utilizando el método `Object.keys`.
- Es configurable: Se podrá eliminar dicha propiedad. Así mismo, podrán cambiarse el resto de configuraciones (incluidas las mencionadas en esta lista) de la propiedad utilizando `Object.defineProperty`.
- Es grabable: Se podrá cambiar el valor de la propiedad.

El constructor `Object` tiene una serie de métodos que permiten crear y manipular objetos de forma más avanzada:

`Object.create`

Esta es una tercera forma de crear un objeto, permitiendo además definir sus propiedades y cuál será su *prototype*. El *prototype* de un objeto es otro objeto, el cual guardará las propiedades y métodos que compartirá con el objeto a crear.

```
var dish = Object.create(Object.prototype, {
  name: {
    value: '',
    writable: true,
    configurable: false
  },
  ingredients: {
    value: [],
    writable: true,
    configurable: false
  },
  garnishes: {
    value: [],
    writable: true,
    configurable: true
  },
  diners: {
    value: 1,
    writable: true,
    configurable: false
  }
});

dish.name = 'Ceviche simple';
dish.name;
```

```
// "Ceviche simple"

dish.ingredients.push('1 kilo de pescado');
// 1
```

Este método toma dos parámetros. El primero es el *prototype* del objeto, mientras que el segundo es un objeto plano que enumera las propiedades que tendrá el nuevo objeto. Este segundo parámetro tiene la misma forma que el segundo parámetro de `Object.defineProperties`.

`Object.create` es útil para definir prototipos, sobre todo si tienen propiedades especiales (por ejemplo, dinámicos o de solo lectura).

`Object.defineProperty`

Crea o modifica una propiedad en un objeto. A diferencia de la manipulación de propiedades mediante asignación con punto o corchetes, este método permite tener un control más avanzado de cómo se podrá manipular la propiedad.

Cuando se define una propiedad con `Object.defineProperty`, por defecto no es ni enumerable ni configurable ni grabable(excepto en Chrome, donde por defecto sí es grabable).

```
var siteTitle = {};

Object.defineProperty(siteTitle, 'internalValue', {
  writable: true,      // internalValue podrá cambiar de valor
  configurable: false, // internalValue podrá cambiar de valor pero no
ser eliminado del objeto
  enumerable: true     // internalValue aparecerá en Object.keys o
usando for..in
});
```

Este método recibe 3 parámetros: el objeto a modificar, la propiedad a definir y el descriptor de la propiedad. Un descriptor de propiedad puede ser de dos tipos:

- Descriptor de datos: para propiedades que tienen un valor
  - `value`: El valor asignado por defecto a la propiedad. Este valor puede ser de cualquier tipo.
  - `writable`: Valor booleano que define si la propiedad es grabable o no.
- Descriptor de acceso: para definir métodos de acceso (`get` y `set`).
  - `get`: Si está definida, esta función se ejecutará al intentar acceder a la propiedad.
  - `set`: Si está definida, esta función se ejecutará al intentar asignar un valor a la propiedad.

Ambos tipos de descriptores comparten dos atributos con valores booleanos: `configurable` y `enumerable`.

Object.defineProperties

Crea o modifica propiedades para un objeto. En este caso el segundo parámetro es un objeto plano donde cada par nombre : valor corresponde al nombre de la propiedad y el descriptor de la misma.

```
var siteTitle = {};

Object.defineProperties(siteTitle, {
  internalValue: {
    writable: true,       // internalValue podrá cambiar de valor
    configurable: true,   // internalValue podrá cambiar de valor y
ser eliminado del objeto
    enumerable: true      // internalValue aparecerá en Object.keys o
usando for..in
  },
  toTitle: {
    writable: false,      // el método toTitle no podrá cambiar de
valor
    configurable: false,  // el método toTitle no podrá ser eliminado
ni cambiado de valor,
    enumerable: false,    // el método toTitle no aparecerá en
Object.keys o usando for..in
    value: function() {
      return this.internalValue.toUpperCase().split('').join(' ');
    }
  }
});
```

Object.preventExtensions

Bloquea la capacidad del objeto de tener nuevas propiedades.

```
var object = {};

Object.preventExtensions(object);
```

Object.freeze

Bloquea futuras modificaciones en un objeto.

```
var object = {};

Object.freeze(object);
```

Object.seal

Sella un objeto, negando la capacidad del objeto de tener nuevas propiedades y de tener propiedades configurables.

```
var object = {};
```

```
Object.seal(object);
```

Object.getOwnPropertyNames

Lista todas las propiedades (incluyendo métodos) de un objeto, sean estos enumerables o no.

```
var siteTitle = Object.create(String.prototype, {
  internalValue: {
    enumerable: false,
    writable: true,
    configurable: false
  }
});

Object.getOwnPropertyNames(superString);
// ["internalValue"]
```

Object.getPrototypeOf

Devuelve el *prototype* de un objeto.

```
var siteTitle = Object.create(String.prototype, {
  internalValue: {
    enumerable: false,
    writable: true,
    configurable: false
  }
});

Object.getPrototypeOf(siteTitle);
// String {}

siteTitle.internalValue = 'La Buena Espina';
// "La Buena Espina"
Object.getPrototypeOf(siteTitle.internalValue);
// TypeError: Object.getPrototypeOf called on non-object
```

Como se ve en el segundo ejemplo, `Object.getPrototypeOf` solo funciona para objetos mas no para valores primitivos.

Object.isExtensible

Verifica si el objeto permite agregar nuevas propiedades.

```
var object = {};

Object.preventExtensions(object);
Object.isExtensible(object);
// false
```

Este comportamiento es definitivo. Es decir, una vez que un objeto deja de ser extensible, no puede volver a su estado anterior.

`Object.isFrozen`

Verifica si un objeto está *congelado* (Ver `Object.freeze`)

```
var object = {};

Object.freeze(object);
Object.isFrozen(object);
// true
```

`Object.isSealed`

Verifica si el objeto está sellado. Un objeto está sellado si no es extensible (`Object.isExtensible` devolviendo `false`) y si ninguna de sus propiedades son configurables.

```
var object = {};

Object.seal(object);
Object.isSealed(object);
// true
```

`Object.keys`

Lista todas las propiedades de un objeto que sean enumerables.

```
var siteTitle = Object.create(String.prototype, {
  internalValue: {
    enumerable: false,
    writable: true,
    configurable: true
  }
});

Object.keys(siteTitle);
// []

Object.defineProperty(siteTitle, 'internalValue', {
  enumerable: true
});

Object.keys(superString);
// ["internalValue"]
```

En general, estos métodos no están disponibles para versiones anteriores a Internet Explorer 9 o las primeras versiones de Chrome y Firefox. En el caso de Opera, estos métodos están disponibles a partir de la versión 12.

Fechas

A diferencia de los arreglos y los objetos, que pueden crearse de forma literal, las fechas en JavaScript son instanciadas utilizando el constructor `Date`. Este constructor tiene diferentes modos, ya que puede tomar como argumentos una cadena, un número que represente una marca de tiempo en milisegundos, o valores separados por comas empezando por el año, mes y día, hasta llegar al milisegundo.

```
new Date();
// Mon Feb 03 2014 21:22:52 GMT-0500 (PET)
new Date('2014-02-04T02:23:16.198Z');
// Mon Feb 03 2014 21:23:16 GMT-0500 (PET)
new Date(1391480663373);
// Mon Feb 03 2014 21:24:23 GMT-0500 (PET)
new Date(2014, 2, 3, 21, 25, 12, 0);
// Mon Mar 03 2014 21:25:12 GMT-0500 (PET)
```

En el último ejemplo se puede ver que aún cuando el segundo parámetro, que corresponde al mes, tiene como valor `2`, genera una fecha con el mes de Marzo. Esto es porque en JavaScript, los valores numéricos de los meses empiezan en 0. Algo similar pasa con el parámetro que representa a los años, pues los valores del 0 al 99 son equivalentes respectivamente a los años del 1900 a 1999.

Por defecto, si ningún parámetro es pasado al constructor, `Date` devuelve la fecha y hora actual de acuerdo a la zona horaria definida en el sistema.

Así mismo, el constructor `Date` tiene 3 métodos:

`Date.now`

Devuelve el valor correspondiente al número de milisegundos pasados desde el primero de Enero de 1970 a las 00:00:00 hasta la fecha actual en la zona horaria definida en el sistema.

```
var now = Date.now();
now;
// 1391576500965

new Date(now);
// Wed Feb 05 2014 00:01:40 GMT-0500 (PET)
```

`Date.parse`

Analiza una fecha en forma de cadena y devuelve el valor correspondiente al número de milisegundos pasados desde el primero de Enero de 1970 a las 00:00:00 en tiempo local. La fecha debe tener formato RFC2822, o ISO 8601 (disponible a partir de versiones superiores a Internet Explorer 9, Firefox 3, Safari 3.2 y Opera 10).

```
var date = Date.parse('Aug 19, 1990 09:30:00');
date;
// 651076200000

new Date(date);
// Sun Aug 19 1990 09:30:00 GMT-0500 (PET)
```

Date.UTC

Similar a la última forma de usar el constructor de `Date`, devuelve el valor correspondiente al número de milisegundos pasados desde el primero de Enero de 1970 a las 00:00:00 hasta la fecha actual en UTC.

```
var date = Date.UTC(2014, 2, 3, 21, 25, 12, 0);
date;
// 1393881912000
new Date(date);
// Mon Mar 03 2014 16:25:12 GMT-0500 (PET)
```

Se debe tener en cuenta que los valores numéricos devueltos por `Date.now`, `Date.parse` y `Date.UTC` no contienen información referente a la zona horaria, por lo que si se utilizan estos números en otros métodos de `Date` se tomará en cuenta la zona horaria local, a menos que la referencia indique lo contrario.

Las instancias de `Date` tienen métodos que permiten obtener información detallada de sus atributos:

getFullYear

Devuelve el año de una fecha (valor de 4 dígitos).
`setFullYear`: Asigna el año a una fecha.

Método equivalente en UTC: `getUTCFullYear` / `setUTCFullYear`.

getMonth

Devuelve el mes de una fecha (valor entre 0 y 11).
`setMonth`: signa el mes a una fecha.

Método equivalente en UTC: `getUTCMonth` / `setUTCMonth`.

getDate

Devuelve el día del mes (valor entre el 1 al 31) de una fecha.
`setDate`: Asigna el día del mes a una fecha.

Método equivalente en UTC: `getUTCDate` / `setUTCDate`.

`getDay`

Devuelve el día de semana (valor del 0 al 6) de una fecha. El número `0` en este caso representa al día domingo, el `1` al lunes y así sucesivamente.

Método equivalente en UTC: `getUTCDay`.

`getHours`

Devuelve la hora en formato de 24 horas (valor entre el 0 al 23) de una fecha.
`setHours`: Asigna las horas en formato de 24 horas a una fecha.

Método equivalente en UTC: `getUTCHours` / `setUTCHours`.

`getMinutes`

Devuelve los minutos (valor entre el 0 al 59) de una fecha.
`setMinutes`: Asigna los minutos a una fecha.

Método equivalente en UTC: `getUTCMinutes` / `setUTCMinutes`.

`getSeconds`

Devuelve los segundos (valor entre el 0 al 59) de una fecha.
`setSeconds`: Asigna los segundos a una fecha.

Método equivalente en UTC: `getUTCSeconds` / `setUTCSeconds`.

`getMilliseconds`

Devuelve los milisegundos (valor entre el 0 al 999) de una fecha.
`setMilliseconds`: Asigna los milisegundos a una fecha.

Método equivalente en UTC: `getUTCMilliseconds` / `setUTCMilliseconds`.

`getTime`

Devuelve el número de milisegundos transcuridos desde el primero de Enero de 1970 a las 00:00:00 UTC hasta una fecha determinada.
`setTime`: Reemplaza una fecha por la fecha correspondiente a un número de milisegundos transcuridos desde el primero de Enero de 1970 a las 00:00:00 UTC.

`getTimezoneOffset`

Devuelve la diferencia en minutos entre la zona horaria del sistema y el UTC. Si la zona horaria es negativa (por ejemplo, UTC-05:00 para el caso de Perú), el valor devuelto por `getTimezoneOffset` será positivo (es decir, 5 * 60).

Así mismo, tienen métodos que permiten convertir las fechas a cadenas:

`toDateString`

Devuelve la porción de fecha (ignorando hora, minutos, segundos, milisegundos y zona horaria).

`toTimeString`

Devuelve la porción de hora (hora, minutos, segundos, milisegundos y zona horaria).

`toISOString`

Devuelve la fecha completa en formato ISO 8601.

`toJSON`

Similar a `toISOString`, devuelve la fecha completa en formato ISO 8601.

`toUTCString`

Similar a `toDateString`, devuelve la fecha completa en UTC.

`toLocaleDateString`

Devuelve la porción de fecha (ignorando hora, minutos, segundos, milisegundos y zona horaria) en un formato local.

`toLocaleTimeString`

Devuelve la porción de hora (hora, minutos, segundos, milisegundos y zona horaria) en un formato local.

`toLocaleString`

Devuelve la fecha completa (fecha y hora) en un formato local.

Funciones

Las funciones en JavaScript permiten crear operaciones reutilizables.

El número de parámetros pasados a una función no es estricta. Si una función está definida para aceptar 3 argumentos y se le pasan 2 parámetros, el tercer parámetro será asignado a `undefined`, mientras que si a la misma función se le

pasan 4 parámetros, el cuarto parámetro será ignorado. Es decir, si se tiene una función `sum`:

```
function sum(a, b, c) {
  var result = a + b;

  if (c) {
    result += c;  // similar a: result = result + c;
  }

  return result;
};

sum(1, 2);
// 3
sum(1, 2, 3, 4);
// 6
```

Tipos vs Objetos

En JavaScript, como en muchos otros lenguajes, existen los llamados **tipos**, los cuales corresponden a los valores más básicos que tiene un lenguaje. Los tipos en JavaScript son:

- Undefined
- Null
- Boolean
- String
- Number
- Object

Podemos identificar el tipo de un valor mediante el operador `typeof`:

```
typeof undefined
// "undefined"
typeof null
// "object"
typeof true
// "boolean"
typeof false
// "boolean"
typeof "hola"
// "string"
typeof 10
// "number"
typeof 1.6
// "number"
typeof {}
// "object"
```

Debido a un error de diseño, el tipo de `null` es `object`. Este error está explicado a profundidad en The history of "typeof null".

Los objetos, por otro lado, son valores que pueden ser creados de dos formas: utilizando constructores propios del lenguaje o utilizando nuevos constructores creados por el desarrollador o por terceros. Los constructores propios del lenguaje son:

- Boolean
- String
- Number
- Object
- Array
- Date
- RegExp
- Function
- Error

Si bien no existe un operador que devuelva el constructor del objeto, se puede comparar el constructor del objeto con otros constructores y saber si el objeto es una **instancia** del mismo:

```
var object = {};
var array = [];
var date = new Date();
var regexp = /(.*)/;

function func() {};

var string = "";
var number = 10;
var bool = true;

object instanceof Object;
// true
array instanceof Array;
// true
date instanceof Date;
// true
regexp instanceof RegExp;
// true
func instanceof Function;
// true

string instanceof String;
// false
number instanceof Number;
// false
bool instanceof Boolean;
// false
```

Valor primitivos

Como se puede ver en los 3 últimos ejemplos, `instanceof` devuelve false aún cuando sabemos que `string` es una cadena, `number` es un número y `bool` contiene un valor lógico. Esto sucede porque `instanceof` no trabaja bien con los denominados valores primitivos. Cada valor primitivo tiene un constructor asociado:

Valor primitivo	Constructor
string	String
number	Number
boolean	Boolean

Los valores primitivos pueden ser confundidos con objetos debido a que tienen acceso a propiedades y métodos (como `"hola".length`). Internamente, el intérprete crea una instancia del constructor asociado al valor primitivo y le da el valor de este, para luego acceder a la propiedad o método que se requiera.

Estructuras condicionales

Las estructuras condicionales en JavaScript son muy similares a las de otros lenguajes parecidos a C/C++. Es de este lenguaje que toma prestada la sintaxis de llaves, usada para delimitar los bloques en JavaScript.

`if..else`

`if` ejecutará las sentencias ubicadas en el primer bloque si la condición de `if` es `true`. Si se definen la sentencias `else` o `else if`, se ejecutarán sus respectivos bloques de sentencias en caso cumplan su condición lógica.

```
var randomNumber = 10;

if (randomNumber < 10) {
  'Menor a 10';
}
else {
  'Igual o mayor a 10';
}

// "Igual o mayor a 10"
```

```
if (randomNumber < 10) {
  'Menor a 10';
}
else if (randomNumber == 10) {
  'Igual a 10';
}
else {
  'Mayor a 10';
}

// "igual a 10"
```

switch

`switch` utiliza el operador `===` internamente para comparar el valor de `switch` con todos los casos definidos en él. Si el caso `default` está definido y ninguna de las comparaciones con los otros casos da `true`, se ejecutará el bloque definido para `default`.

```
var obj = '20';

switch(obj) {
  case '20':
    'obj es una cadena';
    break;
  case 20:
    'obj es un número';
    break;
  default:
    'obj tiene otro tipo de dato';
}

// "obj es una cadena"
```

En las estructuras condicionales se utilizan los operadores de comparación, como `==`, `!=`, `===` y `!==`. La diferencia entre `==` y `===` es que el primero solo compara valores, mientras que el segundo compara valores y tipos de datos. De igual forma pasa con `!=` y `!==`.

Si solo se evalua un valor en un operador condicional, este valor se convertirá implícitamente en valores booleanos `true` o `false`:

```
if (0) {
  '0 se convierte a true';
}
else {
  '0 se convierte a false';
}
// "0 se convierte a false"

var obj = {};
```

```
if (obj) {
  'obj se convierte a true';
}
else {
  'obj se convierte a false';
}
```

Estructuras repetitivas

Las estructuras repetitivas en JavaScript permiten recorrer arreglos u objetos, así como ejecutar operaciones mientras se cumpla una condición.

`for`

La sentencia `for` es similar a la usada en C o Java. Tiene 3 partes: una expresión inicial que define el inicio del bucle o repetición, la condición que debe darse para que el bloque de la sentencia se ejecute, y una expresión que incremente el valor utilizado en la condición de la segunda parte.

```
var counter;

for (counter = 0; counter < 5; counter++) { // inicio del bucle; condición; expresión incremental
  console.log(counter);
}

// 0
// 1
// 2
// 3
// 4
```

`for..in`

Permite recorrer objetos a través de sus propiedades enumerables y es similar a `for`, excepto que en este caso tiene dos expresiones, separadas por la palabra reservada `in`. La primera expresión es una variable auxiliar que tendrá asignado el nombre de la propiedad que está leyéndose en cada iteración, mientras que la segunda expresión es el objeto que se va a recorrer.

```
var obj = {
  string: 'hola',
  number: 24,
  date: new Date()
};

var prop;

for (prop in obj) {
  console.log(prop + ' : ' +  obj[prop]);
```

```
}

// string : "hola"
// number : 24
// date : Wed Feb 05 2014 00:01:40 GMT-0500 (PET)
```

Si el objeto evaluado en una sentencia `for..in` tiene propiedades heredadas de otro objeto, estas también se iterarán.

while

`while` ejecuta las sentencias de su bloque mientras la condición pasada sea verdadera.

```
var counter = 0;

while (counter < 5) {
  console.log(counter);
  counter++;
}

// 0
// 1
// 2
// 3
// 4
```

do..while

`do..while` ejecuta las sentencias de su bloque hasta que la condición pasada sea falsa. A diferencia de `while`, `do..while` ejecuta al menos una las sentencias de su bloque.

```
var counter = 0;

do {
  console.log(counter);
} while (counter != 0);

// 0
```

En este ejemplo, se ve que `i != 0` dará `false`. Sin embargo, se ejecuta la sentencia `console.log(i)` una vez con el valor inicial.

Todas las estructuras repetitivas aceptan una sentencia llamada `break`, la cual interrumpe las iteraciones de la estructura. Así mismo, existe la sentencia `continue`, que salta a la siguiente iteración.

Capítulo 2: Funciones

Con lo aprendido en el capítulo anterior has iniciado con buen pie la tarea de desarrollar el sitio web de La Buena Espina, pero aún tienes camino por recorrer si deseas crear un sitio web fácilmente mantenible en el futuro.

Para lograr una buena estructura en cualquier tipo de proyecto es necesario hacer uso de algunos patrones de diseño, que son soluciones probadas a situaciones comunes en todo tipo de sitio o aplicación web. Estos patrones utilizan funciones a un nivel más avanzado de lo visto en el capítulo anterior, que es justo de lo que trata esta parte.

Funciones

Las funciones en JavaScript también son objetos, por lo que tienen propiedades y métodos. Además de ser objetos, son llamadas *ciudadanos de primera clase* (*first-class citizen*), el tipo de estructura más importante en un lenguaje, así que pueden ser pasadas como parámetros (*callbacks*), ser asignadas a una variable (constructores y funciones anónimas) o ser retornadas por otra función (*closures*).

En JavaScript se pueden crear funciones de 3 formas:

A. Declarando una función, con la sentencia `function`:

```
function sum(a, b) {
  return a + b;
};

sum(1, 2);
```

B. Expresando una función, con el operador `function`:

```
var sum = function sum(a, b) {
  return a + b;
};

sum(1, 2);
```

C. Creando una instancia del constructor `Function`:

```
var sum = new Function('a', 'b', 'return a + b');

sum(1, 2);
```

La forma A y B son similares en sintaxis. Sin embargo, la diferencia principal se da en cómo el navegador carga las funciones. En el primer caso, el navegador cargará todas las funciones declaradas y luego ejecutará el código en el orden en el que fue

escrito, mientras que en el segundo caso, la función se cargará según la posición donde esté definida.

La forma C tiene la ventaja de permitir *evaluar sentencias en tiempo de ejecución*. Esto quiere decir que se pueden crear y ejecutar funciones a partir de datos que ingrese un usuario, como en el caso de una consola de JavaScript.

Scope y context

Una de las más grandes diferencias en JavaScript con respecto a los lenguajes de los cuales está influido es en el ámbito (*scope*) y en el contexto de función.

El ámbito de una variable es el lugar dentro de un programa en el cual dicha variable vive y por lo tanto, donde puede ser usada. El scope de una variable es a nivel de funciones, lo que significa que una variable definida dentro de una función (con la palabra reservada `var`) va a poder ser usada dentro de esa función, pero no fuera de la misma.

Por otro lado, no es recomendable declarar variables sin `var` ya que, si se omite esta palabra reservada, el programa buscará la variable en los ámbitos (o *scopes*) superiores hasta llegar al ámbito global. Si la variable existe, reemplaza su valor, y si no existe la crea en el ámbito global:

```
function globalFunction() {
  function innerFunction() {
    function deeperFunction() {
      globalVar = 'globalVar'; // sin `var`, `globalVar` se vuelve
global
    };

    deeperFunction();
    console.log('deeperFunction', globalVar);
  };

  innerFunction();
  console.log('innerFunction', globalVar);
};

globalFunction();
console.log('globalFunction', globalVar);

// deeperFunction globalVar
// innerFunction globalVar
// globalFunction globalVar
```

El contexto dentro de una función puede cambiar de valor, de acuerdo a la forma cómo está definida la función y cómo se la ejecuta. El contexto es el "dueño" del ámbito de la función que se está ejecutando y se puede acceder a él mediante la palabra reservada `this`.

El contexto cambia de valor según los siguientes casos:

A. Cuando se define una función como método de un objeto, el contexto de dicha función es el objeto:

```
var obj = {
  property: 'value'
};

obj.getValue = function() {
  console.log('context: ', this);
  return this.property;
};

obj.getValue();
// context:  Object {property: "value", getValue: function}
// "value"
```

B. Cuando la función es una función constructora, el contexto de dicha función es el objeto instanciado usando dicha función:

```
var Constructor = function Constructor(newValue) {
  this.property = newValue;

  console.log('context: ', this);
};

var obj = new Constructor('value');
// context:  Constructor {property: "value"}
obj;
// Constructor {property: "value"}
obj.property = 'value';
// "value"
```

C. Cuando la función solo es una función (creada de las 3 formas explicadas anteriormente), el contexto es el contexto global, el cual en navegadores es `window`.

```
function globalContext1() {
  console.log('context1: ', this);
};

var globalContext2 = function() {
  console.log('context2: ', this);
};

var globalContext3 = new Function("console.log('context3: ', this);");

globalContext1();
// context1: Window {top: Window, window: Window, location: Location,
external: Object, chrome: Object…}
globalContext2();
```

```
// context2: Window {top: Window, window: Window, location: Location,
external: Object, chrome: Object…}
globalContext3();
// context3: Window {top: Window, window: Window, location: Location,
external: Object, chrome: Object…}
```

En JavaScript se puede ejecutar una función y cambiar el contexto utilizando los métodos `call` y `apply`. Ambos métodos, que también son funciones, son similares en propósito, pero difieren en el número y forma de sus parámetros.

```
function buildSiteTitle(part1, part2) {
  return part1 + ' – ' + part2;
};

buildSiteTitle('La Buena Espina', 'Carta');
// "La Buena Espina – Ceviches"
buildSiteTitle.call(null, 'La Buena Espina', 'Locales');
// "La Buena Espina – Locales"
buildSiteTitle.apply(null, ['La Buena Espina', 'Historia']);
// "La Buena Espina – Historia"
```

`call` y `apply` tienen como primer argumento el nuevo contexto de la función, el cual en este caso es `null` debido a que no es necesario tener un contexto definido para este ejemplo. Para el caso de `call` el resto de argumentos deben ser los mismos de la función al ser ejecutada, mientras que para el caso de `apply` solo toma un segundo argumento, un arreglo, el cual contiene todos los argumentos de la función a ejecutar.

`apply` tiene una ventaja con respecto a `call`, que es permitir pasar los argumentos de forma dinámica. En el caso de `call`, cada parámetro debe ser pasado dentro del método, como un parámetro más; en el caso de `apply`, solo basta agregar un elemento en el segundo parámetro, que es un arreglo.

```
var titleParts = [];

titleParts.push('La Buena Espina');
buildSiteTitle.apply(null, titleParts);
// "La Buena Espina – undefined"

titleParts.push('Historia');
buildSiteTitle.apply(null, titleParts);
// "La Buena Espina – Historia"
```

Esta flexibilidad al momento de pasar los argumentos en una función se ve limitada hasta este punto. Es aquí donde se empieza a utilizar la palabra reservada `arguments`, el cual es un objeto que representa a los argumentos de la función que se está ejecutando en ese instante.

```
function buildSiteTitle() {     // Ya no es necesario definir los
parámetros
  var separator = ' – ';
```

```
  var title = '';

  if (arguments.length > 2) {   // arguments tiene una propiedad
llamada length
    separator = ' › ';
  }

  for (var i = 0; i < arguments.length; i++) {
    if (i == 0) {
      title += arguments[i];    // La primera parte del título no debe
tener separador
    }
    else {
      title += separator + arguments[i];
    }
  }

  return title;
};

buildSiteTitle.apply(null, ['La Buena Espina', 'Historia']);
// "La Buena Espina — Historia"

buildSiteTitle.apply(null, ['La Buena Espina', 'Carta', 'Ceviches']);
// "La Buena Espina › Carta › Ceviches"
```

Cabe notar que aunque `arguments` tiene una propiedad llamada `length` y puede ser iterada mediante una estructura `for`, no es un arreglo, si no un objeto cuyas propiedades son los argumentos de la función, y donde el nombre de cada propiedad es un índice que empieza en 0 y termina en un número igual a `length` menos uno.

Funciones anónimas

Una función anónima es una función expresada con el operador `function` (forma B para crear funciones) y que no tiene nombre.

```
var nammedFunction = function funcionConNombre() {
  return 'función con nombre';
};

var anonymousFunction = function () {
  return 'función anónima';
};

nammedFunction();
// "función con nombre"
anonymousFunction();
// "función anónima"
```

Este tipo de funciones suelen ser utilizadas como funciones inmediatamente invocadas y callbacks, ya que al ser una función sin nombre, se espera que sea de un solo uso.

Funciones inmediatamente invocadas

Una función inmediatamente invocada (*Immediately-Invoked Function Expression - IIFE*) es una expresión que permite ejecutar una función anónima inmediatamente después de ser definida, lo cual hace que el valor devuelto por la expresión no sea la función en sí, si no el valor de su ejecución.

```
var sum = (function(a, b) {
  return a + b;
})(10, 15);

sum;
// 25
```

La función anónima de este ejemplo está encerrada por paréntesis, lo que permite tratarla como un objeto más, de igual forma a que si esa misma función anónima esté asignada a una variable:

```
var sumFn = function(a, b) {
  return a + b;
};

sumFn(10, 15);
// 25
```

En el caso de una función inmediatamente invocada, la función anónima es ejecutada una sola vez, por lo que no hay motivo para ser guardada en una variable.

Funciones constructoras

Las funciones constructoras permiten definir una especie de “clase” en JavaScript, con la cual luego se pueden instanciar objetos que tengan propiedades y métodos en común.

```
function Dish(options) {
  this.name = options.name;
  this.ingredients = options.ingredients;
  this.garnishes = options.garnishes;
  this.diners = options.diners;
};

var cevicheSimple = new Dish({
  name: 'Ceviche simple',
  ingredients: [
```

```
    '1 kilo de pescado',
    '2 cebollas',
    '1 taza de jugo de limón',
    '1 ají limo',
    'sal'
  ],
  garnishes: [
    'lechuga (2 hojas por plato)',
    'maíz cancha',
    '4 porciones de yuca',
    '4 choclos sancochados',
    'camote sancochado en rodajas (2 por plato)'
  ],
  diners: 4
});

cevicheSimple;
// Dish {name: "Ceviche simple", ingredients: Array[5], garnishes:
Array[5], diners: 4}

cevicheSimple instanceof Dish;
// true
```

Objeto prototype

La orientación a objetos en JavaScript no se maneja mediante clases, si no utilizando funciones constructoras y *prototypes*. Mientras las primeras fungen de clases, las segundas permiten aplicar herencia simple.

Todas los objetos que son instancias de una función constructora comparten las propiedades y métodos definidos en la propiedad `prototype` de dicha función. De igual forma, el *prototype* de un solo objeto puede definirse con el método `Object.create`, visto en el capítulo anterior.

Esta propiedad puede ser extendida (agregar o quitar elementos), así como ser reemplazada por otro objeto, que viene a ser el nuevo *prototype*, o incluso negarle la posibilidad de tener uno asignándole `null` a la propiedad `prototype`.

Al extender el *prototype* de una función, todos los objetos que comparten dicha propiedad actualizan automáticamente su valor:

```
function Dish(options) {
  this.name = options.name;
  this.ingredients = options.ingredients;
  this.garnishes = options.garnishes;
  this.diners = options.diners;
};

var cevicheSimple = new Dish({
  name: 'Ceviche simple',
  diners: 4
});
```

```
cevicheSimple.setIngredients([]);
// TypeError: Object #<Dish> has no method 'setIngredients'

Dish.prototype.setIngredients = function(ingredients) {
  return this.ingredients = ingredients;
};

cevicheSimple.setIngredients(['1 kilo de pescado']);
// ["1 kilo de pescado"]

var sudadoPescado = new Dish({
  name: 'Sudado de pescado',
  diners: 6
});

sudadoPescado.setIngredients(['6 filetes de 160 g. de pescado
blanco']);
// ["6 filetes de 160 g. de pescado blanco"]
```

Extendiendo objetos nativos

Extender el *prototype* de una función no está limitado a las funciones constructoras propias, ya que también se pueden extender los *prototypes* de funciones nativas, como `String`, `Number`, `Date` o `Array`, entre otros.

Esta posibilidad permite *mejorar*, en cierto sentido, el lenguaje y dotar a los objetos de métodos utilitarios. Un ejemplo de esto se da en la biblioteca Sugar.js, la cual extiende los objetos nativos de JavaScript para simplificar y automatizar algunas operaciones comunes como son operaciones entre arreglos, manejar cadenas, números o fechas.

Sin embargo, también existe la posibilidad de extender el *prototype* de objetos que están definidos en el entorno en el cual el programa está ejecutándose, como los que se utilizan en el Document Object Model (la interfaz en JavaScript para manipular HTML). Estos objetos son denominados *host objects*, debido a que su implementación depende del entorno en el que se ejecuta JavaScript.

Mientras que los objetos nativos (`String`, `Number`, `Date` o `Array`) están explícitamente especificados por ECMA, con ECMAScript, los *host objects* difieren entre implementaciones ya que sus especificaciones no son tan explícitas y interpretadas a libertad por quien decida ejecutar JavaScript en su propio entorno.

Es precisamente por la falta de explicitud en la especificación de los *host objects* que extender sus *prototypes* no solamente es recomendable, si no que se evita a toda costa, ya que su comportamiento varía entre implementaciones (es decir, entre navegadores). Mayores detalles se pueden encontrar en "What's wrong with extending the DOM".

Otra de los situaciones que sucede al extender objetos nativos es el hecho que algun nuevo método a implementar pueda ser implementado nativamente en una siguiente versión del lenguaje. El caso más llamativo es el de los métodos para manipular arreglos, como `forEach` o `map`, los cuales fueron agregados en las últimas versiones de navegadores como Chrome o Firefox. Esto es fácilmente solucionable verificando que el método no exista antes de implementarlo en el *prototype* del constructor.

¿Por qué no se debería extender `Object`?

Como se detalló en el capítulo anterior, cada nueva propiedad de un objeto es **enumerable**. Esto quiere decir que si se agrega un nuevo método al *prototype* de `Object`, esta aparecerá cuando se iteren las propiedades de un objeto con `for..in`. (un método en JavaScript no es más que una propiedad cuyo valor es una función):

```
Object.prototype.superMethod = function() {
  return 'instance of Object';
};

var obj = {};       // un objeto plano es una instancia de Object

for (var property in obj) {
  console.log(property);
}
// superMethod

var string = "";    // una cadena también es una instancia de Object

for (var property in string) {
  console.log(property);
}
// superMethod
```

De igual forma, si luego de haber extendido el *prototype* de `Object` se agrega una propiedad a un objeto plano con el mismo nombre de la nueva propiedad o método, el valor que devolverá será el de la propiedad del objeto plano. A este comportamiento se le denomina *property shadowing*:

```
Object.prototype.superMethod = function() {
  return 'instance of Object';
};

var obj = {
  superMethod: 150
};

obj.superMethod();
// TypeError: Property 'superMethod' of object #<Object> is not a
function
```

Patrones de diseño

Debido al auge que ha tenido JavaScript en los últimos años se hizo necesario crear y aplicar técnicas probadas que permitan escribir mejor código y solucionar problemas comunes. Estas técnicas son llamadas patrones de diseño y representan uno de los pilares en cuanto al desarrollo tanto de JavaScript como lenguaje como del uso que se le da al momento de crear aplicaciones web del lado frontend y backend.

Closure

Un closure en JavaScript es una función definida dentro de otra función, teniendo esta función (la función interna) acceso al ámbito (*scope*) de la función que la contiene (la función externa). En JavaScript este comportamiento no sucede a la inversa; es decir, una función externa no tiene acceso al ámbito de la función interna.

```
function buildTitle(parts) {
  var baseTitle = 'La Buena Espina';

  function getSeparator() {
    if (parts.length == 1) {
      return ' — ';
    }
    else {
      return ' › ';
    }
  };

  var separator = getSeparator();
  parts.unshift(baseTitle);

  return parts.join(separator);
}

buildTitle(['Carta']);
// "La Buena Espina — Carta"

buildTitle(['Carta', 'Postres']);
// "La Buena Espina › Carta › Postres"
```

Este ejemplo tiene una función interna llamada `getSeparator`, la cual tiene acceso al ámbito de la función externa (`buildTitle`) y a sus argumentos (`parts`). En este caso el uso de `getSeparator` se limita a crear el título del sitio, por lo que no es necesario hacer que `buildTitle` la retorne (Las funciones, al ser *ciudadanos de primera clase*, pueden retornar otras funciones).

Sin embargo, de ser necesario, el closure puede ser devuelto por la función externa, y aún tener acceso al ámbito de esa función (incluso después de haber ejecutado dicha función).

```
function titleBuilder() {
  var baseTitle = 'La Buena Espina';
  var parts = [baseTitle];

  function getSeparator() {
    if (parts.length == 2) {
      return ' — ';
    }
    else {
      return ' › ';
    }
  };

  function addPart(part) {
    parts.push(part);

    return parts.join(getSeparator());
  };

  return addPart;
};

var builder = titleBuilder();
builder('Carta');
// "La Buena Espina — Carta"

builder('Pescados');
// "La Buena Espina › Carta › Pescados"

builder('Ceviches');
// "La Buena Espina › Menú › Pescados › Ceviches"
```

En este caso, la función `titleBuilder` tiene definidos dos closures: `getSeparator` y `addPart`. A diferencia del ejemplo anterior, `titleBuilder` devuelve el closure `addPart`, por lo que, al guardar el valor devuelto en la variable `builder`, este se vuelve una referencia del closure `addPart`. Como los closures guardan acceso del ámbito de su función externa, aún después de haber sido ejecutadas, puede recrear el título del sitio utilizando la variable `parts`, que a su vez ha sido modificada por el closure.

Module

Un módulo utiliza las funciones inmediatamente invocadas y los closures para encapsular el comportamiento de una función y hacer públicos solo la funciones que se consideren necesarios, mientras que el resto de operaciones y variables quedan inaccesibles.

```
var titleBuilder = (function() {
  var baseTitle = 'La Buena Espina';
  var parts = [baseTitle];

  function getSeparator() {
    if (parts.length == 2) {
      return ' – ';
    }
    else {
      return ' › ';
    }
  };

  return {
    reset: function() {
      parts = [baseTitle];
    },
    addPart: function(part) {
      parts.push(part);
    },
    toString: function() {
      return parts.join(getSeparator());
    }
  };
})();

titleBuilder;
// Object {reset: function, addPart: function, toString: function}

titleBuilder.toString();
// "La Buena Espina"

titleBuilder.addPart('Carta');
titleBuilder.addPart('Pescados');
titleBuilder.addPart('Ceviches');

titleBuilder.toString();
// "La Buena Espina › Carta › Pescados › Ceviches"

titleBuilder.reset();

titleBuilder.toString();
// "La Buena Espina"
```

Un módulo es un función inmediatamente invocada, la cual devuelve un objeto. Este objeto, a su vez, contiene closures con acceso al ámbito de la función inmediatamente invocada y a sus variables internas. Sin embargo, quedan variables, como `parts`, que no pueden manipularse fuera del módulo (excepto al usar el closure `addPart`), así como funciones, como `getSeparator` que no pueden ser utilizadas fuera del módulo.

Callbacks

Un callback es una función pasada como parámetro en otra función, la cual ejecuta el callback luego de haber realizado sus propias operaciones. Usualmente los callbacks son funciones anónimas.

Este extiende el *prototype* de `Array` para crear el método `each`:

```
// como es un objeto nativo, se verifica que el método no exista antes de crearlo
if (!Array.prototype.each) {
  Array.prototype.each = function (callback) {
    // luego, se verifica que el callback sea una función
    if (callback instanceof Function) {
      var i;

      for (i = 0; i < this.length; i++) {
        // el callback de Array.prototype.each puede recibir 2 argumentos: elemento e índice
        callback.call(this[i], this[i], i);
      }
    }
  };
}

[0, 1, 2, 3, 4, 5].each(function(item, index) {
  console.log(item.toString(2), index);
});

// "0"    0
// "1"    1
// "10"   2
// "11"   3
// "100"  4
// "101"  5
```

Publish / Subscribe

El patrón publish / subscribe permite la comunicación entre objetos de forma asíncrona y define dos tipos de objetos: aquel que se suscribe a un canal (*subscriber*) y aquel que envía el mensaje (*publisher*). Cada suscripción permite definir un callback que se ejecutará cuando el objeto *publisher* envíe un mensaje en el canal el objeto *subscriber* está suscrito.

```
var SiteNotifier = (function() {                          // Este patrón utiliza el patrón módulo...
  var channels = {};

  return {
    subscribe: function(channelName, callback) {          // ...con 2 closures: subscribe y publish
      if (channels[channelName] === undefined) {          // en subscribe verifica si el canal existe
```

```
        channels[channelName] = [];                    // y si no, lo
inicializa como un array vacío
      }

      if (callback instanceof Function) {
        channels[channelName].push(callback);          // al ser un
closure tiene acceso a la variable channels del ámbito externo
      }
    },
    publish: function(channelName, message) {
      if (channels[channelName] instanceof Array) {    // solo se
ejecutará si el valor del canal es un array
        var subscribers = channels[channelName];

        for (var i = 0; i < subscribers.length; i++) {  // itera a
través de todos los suscriptores (callbacks)...
          var callback = subscribers[i];

          callback.call(null, message);                // ...y las
ejecuta sin contexto, pasándole el mensaje como parámetro
        }
      }
    }
  };
})();

SiteNotifier.subscribe('site_title:changed', function(message) {
  console.log(message.oldTitle, ' → ', message.newTitle);
});

// La ventaja de utilizar este patrón radica en que se pueden asignar
más de un callback a una acción (o canal)

SiteNotifier.subscribe('site_title:changed', function(message) {
  document.title = message.newTitle;
});

document.title;
""                  // En el caso de una ventana o pestaña nueva, esta
no tiene título

var message = { oldTitle: 'La Buena Espina', newTitle: 'Bienvenidos a
La Buena Espina' };
SiteNotifier.publish('site_title:changed', message);
// "La Buena Espina  →  Bienvenidos a La Buena Espina"

document.title;
// "Bienvenidos a La Buena Espina"
```

Este patrón se utiliza en casos donde se requiera condicionar la ejecución de una o más de una acción a la ejecución de otra previamente. Por ejemplo, si quisiera automatizar el cambio de título de la ventana al pasar de una sección a otra, podría utilizar `SiteNotifier` dentro del módulo `titleBuilder`:

```
var titleBuilder = (function() {
  var baseTitle = 'La Buena Espina';
  var parts = [baseTitle];

  function getSeparator() {
    if (parts.length == 2) {
      return ' – ';
    }
    else {
      return ' › ';
    }
  };

  return {
    reset: function() {
      var message = {
        oldTitle: this.toString()
      };

      parts = [baseTitle];
      message.newTitle = this.toString();

      SiteNotifier.publish('site_title:changed', message);
    },
    addPart: function(part) {
      var message = {
        oldTitle: this.toString()
      };

      parts.push(part);
      message.newTitle = this.toString();

      SiteNotifier.publish('site_title:changed', message);
    },
    toString: function() {
      return parts.join(getSeparator());
    }
  };
})();

SiteNotifier.subscribe('site_title:changed', function(message) {
  // en vez de mostrar en la consola se puede cambiar el título de la
pestaña
  console.log(message.oldTitle, ' → ', message.newTitle);
});

titleBuilder.addPart('Carta');
// "La Buena Espina  →  La Buena Espina – Carta"
titleBuilder.addPart('Pescados');
// "La Buena Espina – Carta  →  La Buena Espina › Carta › Pescados"
titleBuilder.addPart('Ceviches');
// "La Buena Espina › Carta › Pescados  →  La Buena Espina › Carta ›
Pescados › Ceviches"
```

```
titleBuilder.reset();
// "La Buena Espina › Carta › Pescados › Ceviches  →  La Buena Espina"
```

De esta forma, cada vez que agregue una parte al título (con `addPart`), o la devuelva a su estado original (con `reset`), se ejecutarán los callbacks suscritos al canal `site_title:changed`.

Mixins

En la sección sobre *prototypes* creamos la función constructora `Dish`, la cual permite recrear los platillos que ofrece La Buena Espina:

```
function Dish(options) {
  this.name = options.name;
  this.ingredients = options.ingredients;
  this.garnishes = options.garnishes;
  this.diners = options.diners;
};
```

Así mismo, creamos una nueva función llamada `Beverage`, que servirá para modelar las distintas bebidas que ofrece el restaurante:

```
function Beverage(options) {
  this.name = options.name;
  this.quantity = options.quantity;
};
```

Pero el dueño de La Buena Espina quiere tener una calculadora en el sitio web, que permita saber cuánto gastará un posible cliente según lo que vaya a pedir, y para esto necesitamos que todos los items de la carta (en este caso, `Dish`y `Beverage`) tengan un método que agregue el precio a una calculadora.

Podríamos tener un *prototype* en común para ambos pero suena un poco forzado. ¿Cómo es que `Dish` y `Beverage` podrían tener un objeto *padre* en común? Ambos necesitan el mismo comportamiento, pero son muy diferentes para compartir un *prototype*. Es aquí donde podemos usar un *mixin*.

Un *mixin* es una colección de métodos que pueden ser agregados a un objeto (generalmente al *prototype* de una función constructora) y así extender las funcionalidades que tiene dicho objeto. De esta forma podemos simular la herencia múltiple que el lenguaje no da por sí mismo (en JavaScript se maneja herencia simple al extender o reemplazar el *prototype* de una función):

```
var CalculatorItems = [];

var CalculatorMixin = {
  addToCalculator: function(price, quantity) {
    CalculatorItems.push({
      name: this.name,
      price: price,
```

```
      subtotal: price * quantity
    });
  }
};

for (var mixinMethodName in CalculatorMixin) {
  Dish.prototype[mixinMethodName] = CalculatorMixin[mixinMethodName];
  Beverage.prototype[mixinMethodName] =
CalculatorMixin[mixinMethodName];
}
```

La forma de usar un *mixin* es iterando en él (para eso utilizamos un `for..in`) y añadiendo cada método del *mixin* en el *prototype* destino. De esta forma, podemos agregar un platillo a la calculadora:

```
var cevicheSimple = new Dish({
  name: 'Ceviche simple',
  diners: 4
});

cevicheSimple.addToCalculator(20, 1);

var limonadaFrozen = new Beverage({
  name: 'Limonada frozen',
  quantity: '1 vaso'
});

limonadaFrozen.addToCalculator(7, 2);
```

Y calculando:

```
var total = 0;

for (var i = 0; i < CalculatorItems.length; i++) {
  total = total + CalculatorItems[i].subtotal;
}

// toFixed convierte un número a una cadena con determinado número de
decimales
console.log('Total:', 'S/.', total.toFixed(2));
// Total: S/. 34.00
```

Capítulo 3: DOM y CSSOM

Luego de haber visto las bases del lenguaje, es momento de conocer más del navegador y aprender a crear y modificar la interfaz de usuario. Existen dos APIs en el navegador que permiten manipular la estructura, contenido y presentación visual de lo que se muestra dentro de un navegador: el Document Object Model, o DOM, y el Cascade Style Sheet Object Model, o CSSOM.

DOM

El Document Object Model, o DOM, es una API para documentos HTML que representa cada elemento de una página web en forma de objetos, permitiendo su manipulación para cambiar tanto la estructura como presentación visual. De igual forma, permite manejar eventos del usuario dentro del navegador.

HTML

HyperText Markup Language, o HTML, es un lenguaje de marcado que permite definir la estructura y contenido de un documento mediante el uso de etiquetas. El proceso de convertir un documento en HTML en una estructura visual es denominado **renderizar**, y es el **motor de renderizado** el encargado de realizar esta acción.

Es este motor de renderizado el que, a su vez, se encarga de utilizar las hojas de estilo en cascada (CSS) para darle la presentación adecuada al documento HTML que se está renderizando. Actualmente, los motores de renderizado más populares son:

- Webkit, utilizado en Safari y Chrome hasta su versión 27.
- Gecko, utilizado por Firefox y los productos de la Fundación Mozilla.
- Blink (*fork* de Webkit), utilizado actualmente por Chrome a partir de su versión 28.
- Presto, utilizado por Opera, que luego pasó a utilizar Blink.
- Trident, utilizado principalmente por Internet Explorer y otros productos de Microsoft.

Nodos y Elementos

La abstracción que el DOM realiza de un documento HTML utiliza el concepto de **árbol de nodos** para representar la estructura de elementos anidados que tiene el documento. Esto quiere decir que un elemento (o nodo en el árbol) puede tener elementos anidados dentro del mismo (denominados *nodos hijos*); al tener nodos hijos, este elemento automáticamente se convierte en un nodo padre. El primer nodo de un árbol, es decir, aquel que tenga nodos hijos pero no es hijo de ningún otro nodo, es llamado **nodo raíz**.

Según la tabla de valores de la propiedad `nodeType` (ver Apéndice A), se pueden ver diferentes tipos de nodos. Al ser el DOM una abstracción basada en árboles de nodos, cada dato dentro de un documento HTML debe pertenecer a un tipo de nodo. Por ejemplo:

```
<!DOCTYPE html>
<html>
<head>
  <title></title>
</head>
<body>
  <div class="empty-div"></div>
  <!-- Comentario dentro de un documento HTML -->
  Contenido de texto
</body>
</html>
```

`nodeType`	**Tipo de dato en HTML**
`ELEMENT_NODE`	Elemento (`<div class="empty-div"></div>`)
`ATTRIBUTE_NODE`	Atributo (`class="empty-div"`)
`TEXT_NODE`	Texto (`Contenido de texto`)
`COMMENT_NODE`	Comentario (`<!-- Comentario dentro de un documento HTML -`
`DOCUMENT_NODE`	Documento (`window.document`)
`DOCUMENT_TYPE_NODE`	Doctype (`<!DOCTYPE html>`)

Uno de estos tipos de nodos es el elemento (`ELEMENT_NODE`), el cual es la representación para toda etiqueta en un documento HTML. Esto quiere decir que todos los elementos son nodos, pero no todos los nodos son elementos.

`window` y `document`

El contexto global de una aplicación web recae en `window`, el cual contiene referencias a diferentes APIs y objetos del navegador, como `screen`, `navigator`, `history`, `location` y `document`. Cada iframe tiene su propio objeto `window`, y pueden acceder al `window` que lo contiene mediante la propiedad `parent`.

`document` es el objeto que representa al nodo raíz de un documento HTML, y tiene acceso a los nodos que representan a las etiquetas `<head>` y `<body>`. Los iframes tienen su propio documento HTML, por lo que, si un documento tiene iframes, cada elemento `iframe` puede acceder a su propio documento mediante la propiedad `contentDocument`.

Las propiedades y métodos de `document` están definidas por dos interfaces: `HTMLDocument` y `Document`. Adicionalmente, `Document` hereda de `Node`. Mientras `HTMLDocument` y `Document` le dan a `document` la capacidad de representar al nodo raíz de un documento, `Node` añade propiedades y métodos relacionados al manejo de nodos. Tanto `window` y `document` heredan de la interfaz `EventTarget`, por lo que tienen la capacidad de manejar eventos.

El DOM en su forma más abstracta es una interfaz que permite leer y manipular documentos en XML y HTML (incluyendo XHTML) y representarlos como un árbol de nodos; sin embargo, el tipo documento estándar mostrado en un navegador es HTML, el cual tiene propiedades propias y distintas a un documento XML cualquiera. Es por eso que, en el navegador, el DOM tiene dos interfaces para representar a un documento: `Document` para un documento genérico, y `HTMLDocument` para un documento HTML.

`HTMLDocument` tiene propiedades propias de un documento HTML, como: `domain`, `title`, `body`, `forms`, `anchors`, `links` e `images`. Cabe aclarar que `anchors` y `links` devuelven listas de elementos de la misma etiqueta (`<a>`). Esto sucede porque esta etiqueta es usada tanto como ancla dentro del documento como para enlazar el documento actual a otros documentos, imágenes, archivos, etc.

`Document` tiene 3 propiedades: `doctype`, que devuelve el tipo de documento (DTD); `documentElement`, que representa a la etiqueta `html`; e `implementation`, que permite crear documentos (HTML o no), así como hojas de estilos. Así mismo, tiene una serie de métodos útiles para la manipulación de elementos dentro del documento, como crear nodos de tipo elemento, comentario, texto y atributo.

Agregando y eliminando nodos

Agregar un nodo en el árbol DOM consta de 2 pasos: Crear el nodo, y añadirlo. El primer paso utiliza un método de `document` que varía según el tipo de nodo que se desee agregar:

- Para agregar un nodo elemento se utiliza `document.createElement`.
- Para agregar un nodo atributo se utiliza `document.createAttribute`.
- Para agregar un nodo texto se utiliza `document.createTextNode`.
- Para agregar un nodo comentario se utiliza `document.createComment`.

Para el segundo paso, el futuro nodo padre debe ejecutar el método `appendChild`.

Para eliminar un nodo solo es necesario que el nodo padre ejecute el método `removeChild`.

Simplificando el manejo del DOM con `dom.js`

Como vimos en el punto anterior, tener que realizar 2 pasos para agregar un nodo al DOM puede llegar a ser tedioso (sobre todo si tenemos que hacerlo varias veces). Adicionalmente, pronto notarás que la API del DOM es verbosa, por lo que sería una buena idea crear una biblioteca que permita reducir el número de palabras escritas y simplifique los pasos para manejar el DOM; así que realizaremos una biblioteca llamada `dom.js`, la cual usaremos dentro de La Buena Espina.

Empecemos por crear un constructor llamado `Dom`:

```
function Dom(selector) {
  this.selector = selector;
  this.elements = document.querySelectorAll(selector);
};
```

Dentro de esta función utilizamos `document.querySelectorAll` para poder obtener una lista de elementos a través de un selector CSS. En este caso preferimos utilizar `document.querySelectorAll` y no `document.querySelector` para darnos la flexibilidad de trabajar con múltiples nodos.

Para poder crear un elemento también necesitamos un método propio, que también nos permita definir sus atributos:

```
Dom.createElement = function(options) {
  var element = document.createElement(options.tag),
      attributes = Object.keys(options.attributes || {}),
      i = 0;

  for(i; i < attributes.length; i++) {
    element.setAttribute(attributes[i],
options.attributes[attributes[i]]);
  }

  return element;
};
```

Luego de haber creado un elemento, debemos agregarlo a un elemento padre:

```
Dom.prototype.append = function(newChildElement) {
  this.elements[0].appendChild(newChildElement);

  return this;
};
```

En `append` no iteramos por todos los elementos de la instancia, dado que un nodo (`newChildElement`) solo puede ser agregado a un elemento, y tener un solo nodo padre.

En el caso anterior, `newChildElement` debería ser un elemento; pero también debería poder aceptar un objeto similar al pasado en `Dom.createElement`:

```
Dom.prototype.append = function(newChildElement) {
  if (!(newChildElement instanceof Element)) {
    if (newChildElement.hasOwnProperty('tag')) {
      newChildElement = Dom.createElement(newChildElement);
    }
  }

  this.elements[0].appendChild(newChildElement);

  return this;
};
```

De esta forma, verificamos si el parámetro pasado a `Dom.prototype.append` es un objeto plano o una instancia de `Element`.

```
var nav = new Dom('header nav');

nav.append({
  tag: 'a',
  content: 'Reservaciones',
  attributes: {
    href: '#reservaciones'
  }
});
```

Recorriendo nodos y elementos

Tanto las interfaces `Node` como `Element` tienen propiedades que permiten obtener los nodos (y elementos) hijos de otro nodo, así como obtener los nodos *hermanos* de un nodo en específico (un nodo *hermano* es aquel nodo que está al mismo nivel que otro y comparten el mismo nodo padre).

Cada interfaz tiene sus propias propiedades para obtener nodos hijos y nodos hermanos; así, `childNodes` devuelve todos los nodos hijos, incluyendo nodos textos, comentarios o elementos; mientras que `children` devuelve todos los nodos hijos que son elementos. La diferencia es más notoria con las propiedades `firstChild` y `firstElementChild`, o `nextSibling` y `nextElementSibling`.

Tanto `childNodes` como `children` devuelven una lista *viva* (también llamada colección *viva*).

Lista *viva*

Algunas propiedades y métodos del DOM devuelven listas "vivas". Una lista *viva* es una lista de elementos que automáticamente actualiza su contenido cuando estos cambian en otra parte del programa. Es decir, tanto si se agrega un elemento que concuerde con la lista o si se elimina un elemento que se encuentre en la lista, esta se actualizará con los elementos nuevos o quitando los eliminados posteriormente.

Vamos a extender `dom.js` para que permita obtener la lista de elementos hijos, que es con la que usualmente se trabaja.

```
Dom.prototype.children = function() {
  return this.elements[0].children;
};
```

Este método no es tan útil, ya que nos devuelve una lista nativa que no tendrá los métodos de `Dom`, por lo que deberíamos *envolver* esta lista en una instancia de `Dom`. Para esto vamos a actualizar el constructor:

```
function Dom(selectorOrElements) {
  if (typeof selectorOrElements === 'string') {
    this.selector = selectorOrElements;
    this.elements = document.querySelectorAll(selectorOrElements);
  }
  else {
    if (selectorOrElements instanceof Node) {   // aprovechamos para
verificar si se envolverá un solo elemento o una lista de elementos
      this.elements = [selectorOrElements];
    }
    else {
      this.elements = selectorOrElements;
    }
  }
};
```

De esta forma, `Dom.prototype.children` quedará de la siguiente forma:

```
Dom.prototype.children = function() {
  if (this.elements[0] !== undefined) {
    return new Dom(this.elements[0].children);
  }
  else {
    return Dom.empty([]);
  }
};
```

Adicionalmente, agregamos una validación simple para saber si existen elementos dentro de la instancia de `Dom` que permitan leer nodos hijos, de no existir elementos en `this.elements`, debería devolver una lista vacía. Si bien una

instancia de `NodeList` no es un arreglo, ambas tienen una propiedad llamada `length`, por lo que, para efectos prácticos, sirve para representar una lista vacía.

Atributos

En HTML, las etiquetas pueden guardar información sobre sus propiedades mediante atributos. Los atributos más comunes son `id` y `class` y, en el caso de elementos de formulario, los atributos más importantes son `type` y `name`.

La interfaz `Element` tiene métodos para leer, definir, eliminar y verificar si un atributo está
definido: `getAttribute`, `setAttribute`, `removeAttribute` y `hasAttribute`, respectivamente. Estos métodos reciben un parámetro en forma de cadena para el nombre (y otro para el valor, en el caso de `setAttribute`).

Dentro de `dom.js` crearemos los métodos `get`, `set`, `unset` y `has`:

```
Dom.prototype.get = function(attributeName) {
  var i = 0,
      attributeValues = [];

  for (i; i < this.elements.length; i++) {

attributeValues.push(this.elements[i].getAttribute(attributeName));
  }

  return attributeValues;
};

Dom.prototype.set = function(attributeName, attributeValue) {
  var i = 0;

  for (i; i < this.elements.length; i++) {
    this.elements[i].setAttribute(attributeName, attributeValue);
  }
};

Dom.prototype.unset = function(attributeName) {
  var i = 0;

  for (i; i < this.elements.length; i++) {
    this.elements[i].removeAttribute(attributeName);
  }
};

Dom.prototype.has = function(attributeName) {
  var i = 0,
      hasAttributeValues = [];

  for (i; i < this.elements.length; i++) {
```

```
hasAttributeValues.push(this.elements[i].hasAttribute(attributeName));
  }

  return hasAttributeValues;
};
```

Como los atributos también son nodos dentro del DOM, no solo se pueden manipular atributos con los métodos `getAttribute`, `setAttribute` y `removeAttribute`. También es posible utilizar nodos de tipo atributo en vez de cadenas como parámetros con los métodos `getAttributeNode`, `setAttributeNode` y `removeAttributeNode`.

Eventos

Los eventos permiten comunicar acciones realizadas tanto por el navegador como por el usuario, y ayudan a mejorar la interacción entre una persona y un sitio o aplicación web. Como ejemplo: cuando un usuario hace clic en un enlace, se puede capturar el evento *click* de ese elemento y lanzar una acción diferente a la habitual (la cual es enviar al usuario al documento enlazado). Otro ejemplo es validar formularios antes de ser enviados, capturando el evento *submit* de el elemento `<form>`.

Todos los elementos del DOM, además de `window`, heredan de la interfaz `EventTarget`, el cual permite enlazar eventos a callbacks definidos dentro de la aplicación. La interfaz `EventTarget` tiene 3 métodos: `addEventListener`, `removeEventListener` y `dispatchEvent`.

addEventListener

Para enlazar un evento a un callback se utiliza `addEventListener`:

```
window.addEventListener('load', function(e) {
  console.log('window:load', e);
});
```

El ejemplo anterior agrega un *listener* al evento `load` de `window`, donde el callback pasado como segundo parámetro es la función que se ejecutará cuando el evento se dispare (que es cuando el navegador termina de cargar el documento).

Todos los callbacks enlazados a eventos toman un solo parámetro (en este caso, `e`). Este parámetro puede ser instancia de `FocusEvent`, `MouseEvent`, `KeyboardEvent`, `UIEvent` o `WheelEvent`, dependiendo del evento que sea lanzado. Todos los eventos heredan de la interfaz `Event`.

removeEventListener

Para eliminar un *listener* de un elemento se utiliza el método `removeEventListener`, que toma los mismos valores de `addEventListener`. Esto quiere decir que, para eliminar un *listener* de un elemento, es obligatorio mandar como parámetro el mismo callback utilizado en `addEventListener`.

En este ejemplo el evento no se eliminará, puesto que los callbacks son diferentes:

```
window.addEventListener('load', function(e) {
  console.log('window:load', e);
});

window.removeEventListener('load', function(e) {
  console.log('window:load', e);
});
```

Por eso, es necesario guardar el callback utilizado en `addEventListener` en una variable para utlizarla luego en `removeEventListener`:

```
var windowOnLoad = function(e) {
  console.log('window:load', e);
};

window.addEventListener('load', windowOnLoad);

window.removeEventListener('load', windowOnLoad);
```

Ahora que ya vimos como agregar y eliminar *listeners* a un elemento, vamos a añadir esta funcionalidad a `dom.js`:

```
Dom.prototype.on = function (eventName, callback) {
  var i = 0,
      eventIdentifier = this.selector + ':' + eventName;

  if (this.events === undefined) {
    this.events = {};
  }

  if (this.events[eventIdentifier] === undefined) {
    this.events[eventIdentifier] = [];
  }

  this.events[eventIdentifier].push(callback);

  for (i; i < this.elements.length; i++) {
    this.elements[i].addEventListener(eventName, callback, true);
  }
};

Dom.prototype.off = function(eventName) {
  var i = 0,
      e = 0,
```

```
      eventIdentifier = this.selector + ':' + eventName;

  if (this.events === undefined) {
    this.events = {};
  }

  if (this.events[eventIdentifier] != undefined) {
    for (e; e < this.events[eventIdentifier].length; e++) {
      var callback = this.events[eventIdentifier][e];

      for (i; i < this.elements.length; i++) {
        this.elements[i].removeEventListener(eventName, callback,
true);
      }
    }

    this.events[eventIdentifier] = [];
  }
};
```

Como debemos tener constancia de los callbacks que están siendo utilizados en `addEventListener`, los guardamos en la propiedad `this.events`. Luego, si queremos eliminarlos, iteramos dentro de esa propiedad y eliminamos los *listeners* con `removeEventListener`.

Así, utilizamos nuestros métodos de la siguiente forma:

```
var win = new Dom(window);

win.on('load', function(e) {
  console.log('window:load');
});
```

Para que nuestro `dom.js` funcione con `window` debemos cambiar el constructor una vez más, y debemos verificar si el argumento pasado al constructor es instancia de `EventTarget` (de todas formas, `Node` hereda de `EventTarget`):

```
function Dom(selectorOrElements) {
  if (typeof selectorOrElements === 'string') {
    this.selector = selectorOrElements;
    this.elements = document.querySelectorAll(selectorOrElements);
  }
  else {
    if (selectorOrElements instanceof EventTarget) {
      this.elements = [selectorOrElements];
    }
    else {
      this.elements = selectorOrElements;
    }
  }
};
```

Eventos propios

Adicionalmente a los eventos nativos del navegador (como `load`, `click`, y otros), también se pueden crear eventos propios. Estos eventos propios son utilizados para propósitos propios de la aplicación. Por ejemplo, y siguiendo con el caso de La Buena Espina, como desarrollador es vital saber cuándo un usuario ha cambiado de sección. Podemos saber esto mediante el uso de eventos propios.

Existen dos formas de crear eventos personalizados:

La primera es utilizando el método `document.createEvent`. Este método funciona en todos los navegadores, incluyendo Internet Explorer 9 y superiores:

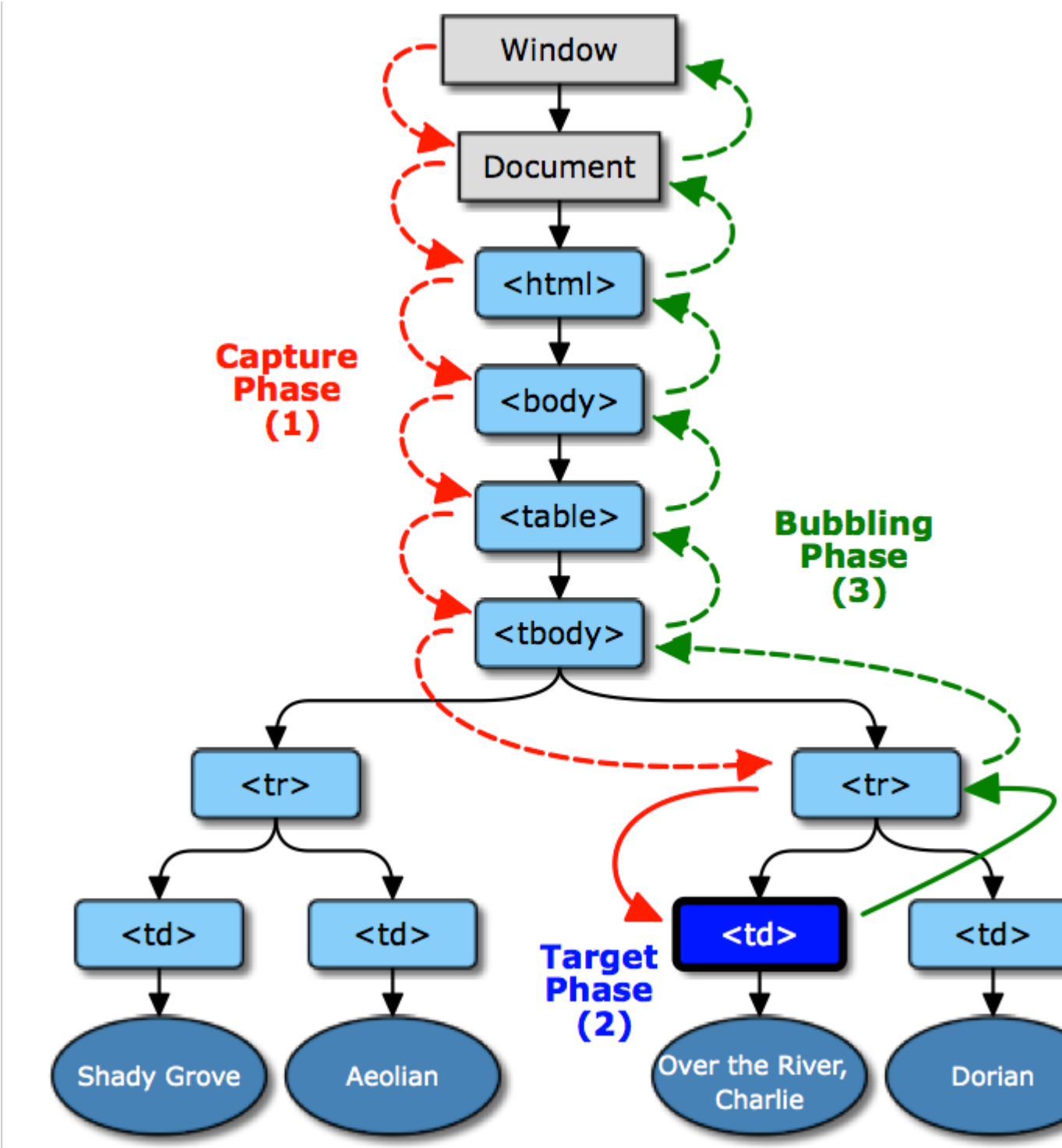
```
var sectionChangedEvent = document.createEvent('CustomEvent');
sectionChangedEvent.initCustomEvent('sectionchanged', true, false, {
previousSection: 'carta', nextSection: 'locales' });

document.addEventListener('sectionchanged', function(e) {
  console.log(e.detail.previousSection + ' → ' +
e.detail.nextSection);
});

document.dispatchEvent(sectionChangedEvent);
// "carta → locales"
```

La segunda forma es utilizando el constructor de `CustomEvent`, el cual funciona para todos los navegadores, excepto Internet Explorer:

```
var sectionChangedEvent = new CustomEvent('sectionchanged', {
  bubbles: true,
  cancelable: false,
  detail: {
    previousSection: 'carta',
    nextSection: 'locales'
  }
});

document.addEventListener('sectionchanged', function(e) {
  console.log(e.detail.previousSection + ' → ' +
e.detail.nextSection);
});

document.dispatchEvent(sectionChangedEvent);
// "carta → locales"
```

Event flow

Cuando un evento es lanzado, este pasa por 3 fases, en el siguiente orden: *Capture phase*, *Target phase* y *Bubbling phase*. El hecho de pasar por las 3 fases es denominado *event flow*.



Document Object Model (DOM) Level 3 Events Specification. http://www.w3.org/TR/DOM-Level-3-Events/

En el *event flow*, cada evento lanzado en el DOM empieza en el contexto global (es decir, `window`), pasa por el nodo raíz del documento (`document`) y sigue un camino a través de una serie de nodos hijos (*Capture phase*) que le permita llegar al elemento que lanza dicho evento (*Target phase*). En la *target phase*, el evento es lanzado. Luego, empieza la *bubbling phase*, siguiendo el mismo camino de la *capture phase*, pero en sentido inverso, hasta llegar al contexto global (`window`).

Cuando se registra un *listener*, se puede definir para que sea ejecutado en la *capture phase* o en la *bubbling phase*. El orden en que un *listener* es ejecutado depende de la *fase* en la que está agregado:

```
window.addEventListener('click', function() {
  console.log('Bubbling click event');
}, false);                                    // Este listener se
ejecutará segundo

window.addEventListener('click', function() {
  console.log('Capturing click event');
}, true);                                    // Este listener se
ejecutará primero
```

Para definir la fase en la que se ejecutará un *listener* se pasa un tercer parámetro a `addEventListener`, el cual debe tener un valor booleano: si el parámetro es `true`, el *listener* se ejecutará en la *capture phase*, y si es `false` el *listener* se ejecutará en la *bubbling phase*. Por defecto, el valor de este parámetro es `false`. Cabe señalar que también debe ser pasado a `removeEventListener` si existen dos *listeners*, uno para cada fase, que apunten al mismo evento y elemento.

En `dom.js` definimos el tercer parámetro como `true`, tanto en `Dom.prototype.on` como en `Dom.prototype.off`, haciendo que todos los *listeners* sean ejecutados en la *capture phase*. De esta forma, el orden en el que agregamos los *listeners* será el mismo en el que son lanzados.

Rendimiento

Trabajar con el DOM puede traer consecuencias inesperadas en temas de rendimiento si no se toman en cuenta algunas características propias de los navegadores. Por ejemplo, cuando se manipula el árbol DOM, el navegador recalcula posiciones y re-renderiza la pantalla (ver *Reflow y repaint*).

Reflow y repaint

Cuando se renderiza un documento HTML en un navegador ocurren dos acciones: *reflow* (o *layout*) y *repaint*. Al realizar el *reflow*, el navegador calcula las dimensiones y posiciones de cada elemento visible y los coloca en la posición previamente calculada dentro de la zona visible del navegador (o *viewport*). Cuando se realiza el *repaint*, el navegador obtiene la información de las hojas de estilo del documento, así como de los estilos del sistema y del navegador, y muestra los elementos de la forma como fue ideada (bordes, fondos, colores, imágenes, etc). Cuando se realiza un *reflow*, también se realiza un *repaint*, pero no es así de forma inversa (puede cambiarse el fondo de un elemento y el navegador no tendrá que recalcular posiciones de elementos).

Algunas de las acciones que obligan al navegador a realizar *reflow* (y su respectivo *repaint*) está relacionadas al uso de CSS; y otras a la manipulación del árbol DOM con JavaScript, como:

- Agregar o eliminar un elemento al documento.
- Cambiar el contenido de un elemento con `innerText` e `innerHTML`.
- Cambiar la visibilidad de un elemento con la propiedad `display` del CSS (manipulando el atributo `style` de un elemento).
- Cambiar la clase CSS o los estilos de un elemento (atributos `className`, `classList` o `style`).
- Redimensionar la ventana o el *viewport*.
- Utilizar el método `getComputedStyle`.
- Leer las propiedades de `MouseEvent`: `layerX`, `layerY`, `offsetX` y `offsetY`.
- Realizar scroll con los métodos `scrollIntoView`, `scrollIntoViewIfNeeded`, `scrollByLines` o `scrollByPages`.
- Leer algunas propiedades de elementos: `clientLeft/Top/Width/Height`, `scrollLeft/Top/Width/Height`, `offsetLeft/Top/Width/Height`, entre otras.

`document.createDocumentFragment`

Este método permite crear una versión más ligera y limitada de `document`, y sirve para mejorar el rendimiento de operaciones donde se necesiten agregar muchos nodos.

Cuando se agregan nodos dentro de un bucle (la operación más común cuando se quieren agregar un indeterminado número de elementos), se utiliza el método `appendChild`, el cual hace que el navegador recalcule posiciones y renderice la pantalla tantas veces como iteraciones tuvo el bucle.

La ventaja de utilizar un fragmento (una instancia de `DocumentFragment`) es que, al estar separado del árbol DOM del documento y es guardado en memoria, evita que el navegador tenga que renderizar de nuevo cada vez que se agregue un nodo al fragmento.

Para probar este método implementaremos un método llamado `html`, el cual permitirá agregar elementos a un nodo pero utilizando cadenas. Inicialmente usaremos la propiedad `innerHTML`:

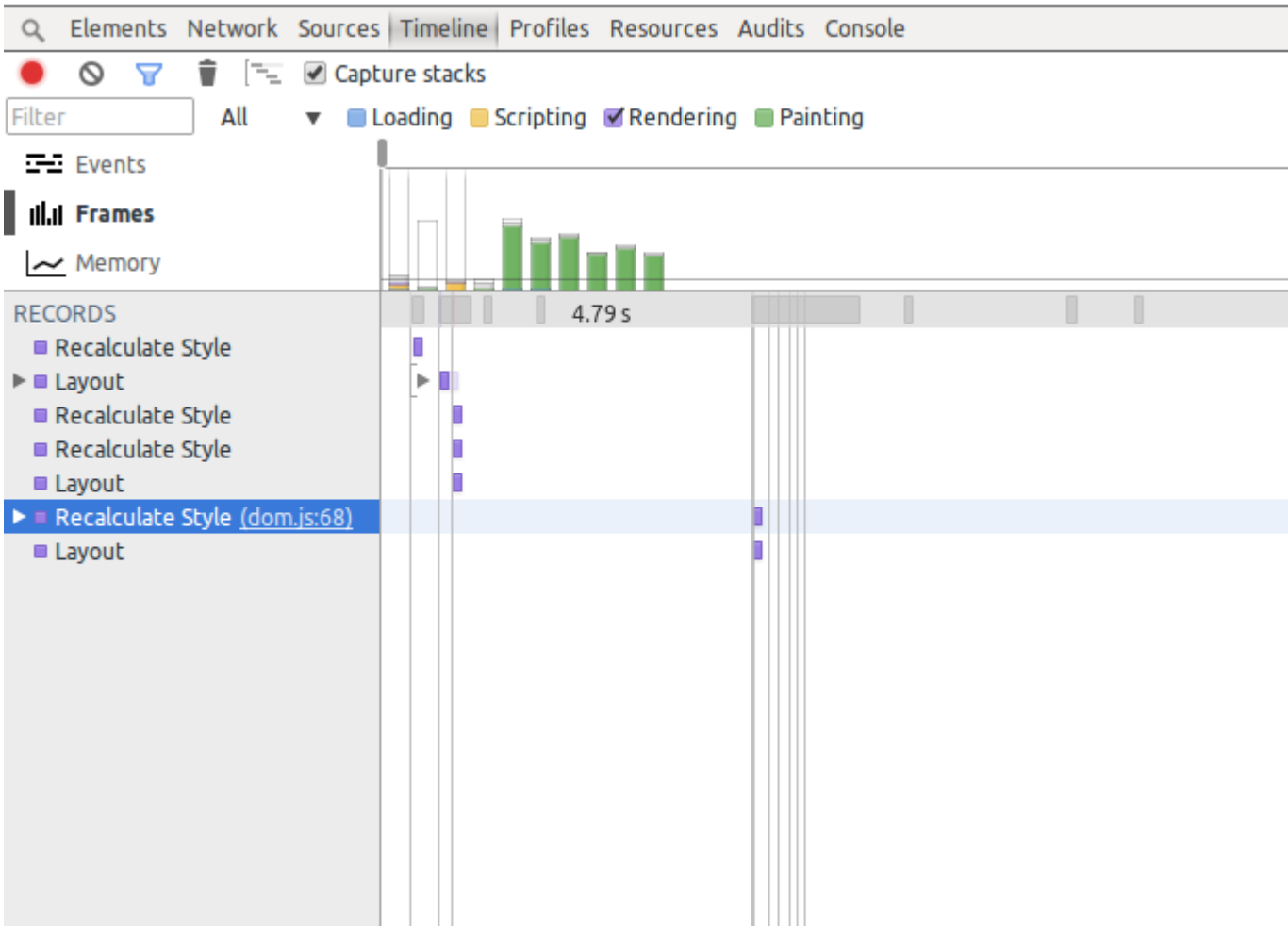
```
Dom.prototype.html = function(htmlString) {
  var i = 0;

  // Eliminamos el contenido de todos los elementos
  for (i; i < this.elements.length; i++) {
    this.elements[i].textContent = '';
  }

  // Agregamos el nuevo contenido a todos los elementos
  for (i = 0; i < this.elements.length; i++) {
```

```
    this.elements[i].innerHTML = htmlString;
  }
};
```

Vamos a probar en un nodo vacío ejecutando la siguiente instrucción:

```
new Dom('#background').html(
  '<div class="slide" id="slide-1" title="Créditos:
http://www.flickr.com/photos/saucesupreme/6774616862/"></div>' +
  '<div class="slide" id="slide-2" title="Créditos:
http://www.flickr.com/photos/c32/4775267221/"></div>' +
  '<div class="slide" id="slide-3" title="Créditos:
http://www.flickr.com/photos/renzovallejo/7998183161/"></div>' +
  '<div class="slide" id="slide-4" title="Créditos:
http://www.flickr.com/photos/renzovallejo/7998217897/"></div>' +
  '<div class="slide" id="slide-5" title="Créditos:
http://www.flickr.com/photos/renzovallejo/7998185723/"></div>' +
  '<div class="slide" id="slide-6" title="Créditos:
http://www.flickr.com/photos/renzovallejo/7998141711/"></div>'
);
```

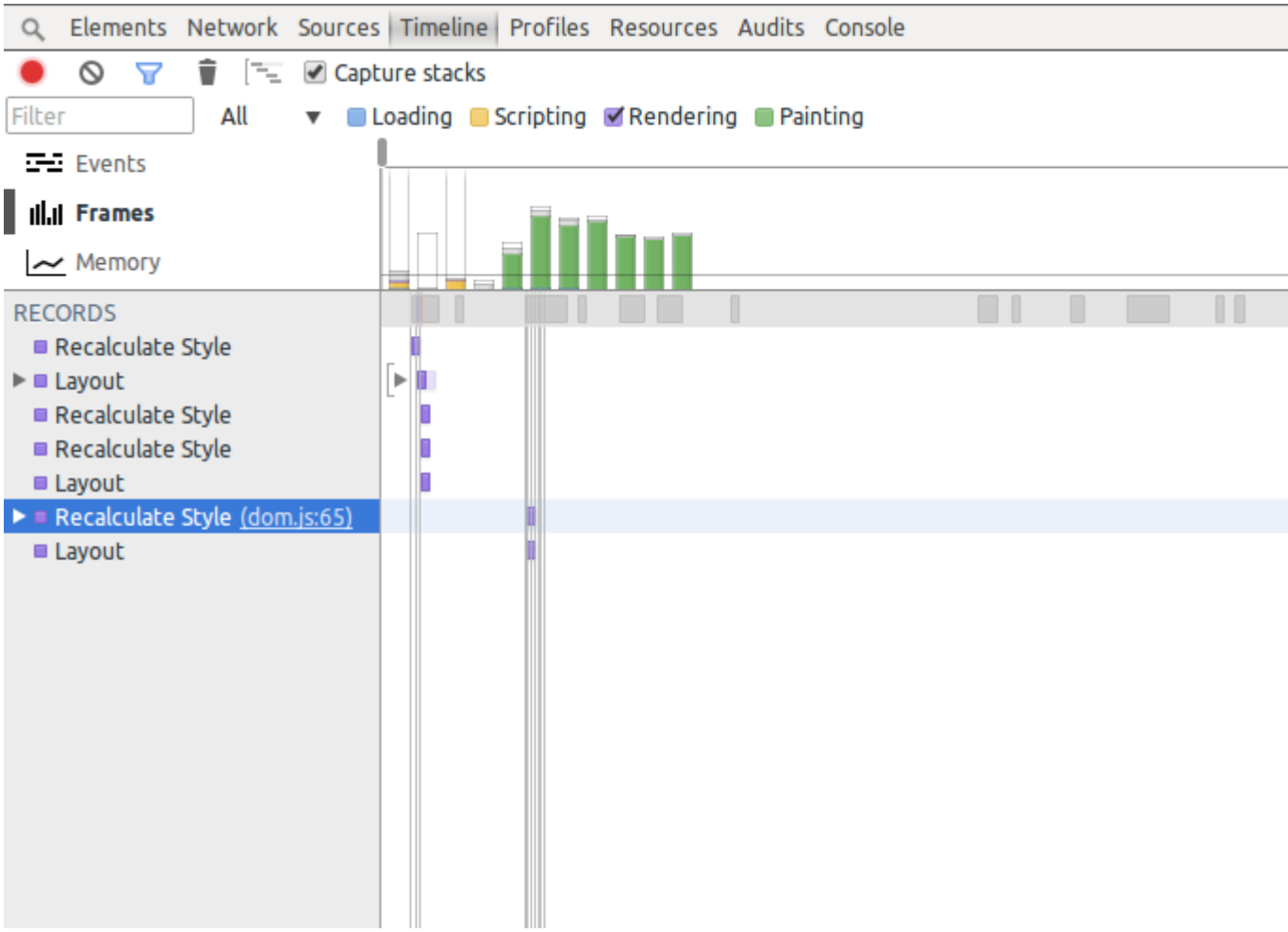
Este método nos da el siguiente gráfico dentro de la pestaña *Timeline* de Chrome:



Utilizando innerHTML

Ahora vamos a utilizar un fragmento:

```
Dom.prototype.html = function(htmlString) {
  var i = 0,
      f = 0;

  var fragment = document.createDocumentFragment(),
      root = Dom.createElement({
        tag: 'div',
        attributes: {
          id: 'root'
        }
      });

  root.innerHTML = htmlString;

  for (f; f < root.childNodes.length; f++) {
    fragment.appendChild(root.childNodes[f].cloneNode(true));
  }

  // Eliminamos el contenido de todos los elementos
  for (i; i < this.elements.length; i++) {
    this.elements[i].textContent = '';
  }

  root = null;

  // Agregamos el fragmento a todos los elementos
  for (i = 0; i < this.elements.length; i++) {
    this.elements[i].appendChild(fragment.cloneNode(true));
  }
};
```



Utilizando fragmento

El tiempo utilizado por el navegador para recalcular estilos luego de añadir los elementos al DOM, segun la técnica:

Técnica	**Tiem**
`Element.prototype.innerHTML`	4.047
`document.createDocumentFragment`	2.576

Enlazando eventos a múltiples elementos

Uno de los casos más comunes de uso de eventos es el de enlazar eventos a diferentes elementos que son similares (por ejemplo, un cliente de correo tiene el mismo enlace "marcar como importante" para cada correo en la bandeja). Con el DOM, se agrega un *listener* de un evento a un elemento utilizando `addEventListener`, pero no se puede agregar un *listener* a una lista de elementos.

Agregar un *listener* por cada elemento puede ser una solución pero, a medida que existan más elementos del mismo tipo, se necesitarán agregar más *listeners*, aumentando la cantidad de memoria utilizada por el navegador.

Event delegation

Event delegation es una técnica que permite disminuir la cantidad de *listeners* creados, y que tiene como ventaja adicional el permitir que un elemento recién creado pueda *escuchar* un evento, sin necesidad de tener su propio *listener*.

Esta técnica utiliza el *event flow* para agregar un *listener* al elemento padre de todos los elementos que compartirán la misma funcionalidad. Debido a la naturaleza del *event flow*, el elemento padre lanzará el evento si tiene un *listener* registrado.

Existen dos formas de obtener el elemento que lanza el evento: el contexto del mismo (`this`) o la propiedad `target` del evento (el parámetro del callback): Cuando se usa `addEventListener`, `this` y `target` referencian al mismo elemento, mientras que en *event delegation*, `this` será el elemento que ejecute `addEventListener` (es decir, el elemento padre), mientras que `target` será el elemento que lance el evento (es aquí donde ocurre la *target phase*).

Dentro del callback del *listener*, se verifica que el elemento referenciado en `target` sea el que se desea utilizar (usualmente comparando las clases o el id de `target`).

Vamos a implementar el *event delegation* en el método `delegate`:

```
Dom.prototype.delegate = function(eventName, selector, callbak) {
  var i = 0,
      eventIdentifier = selector + ':' + eventName;

  if (this.events === undefined) {
    this.events = {};
  }

  if (this.events[eventIdentifier] === undefined) {
    this.events[eventIdentifier] = [];
  }

  this.events[eventIdentifier].push(callback);

  for (i; i < this.elements.length; i++) {
    this.elements[i].addEventListener(eventName, function(e) {
      if (e.target.webkitMatchesSelector(selector)) {
        callback(e);
      }
    }, true);
```

```
  }
};
```

En esta primera implementación utilizamos el método `webkitMatchesSelector`, el cual verifica que un elemento concuerde con un selector CSS dado. Si lo dejamos de esta forma, `Dom.prototype.delegate` solo funcionará en navegadores basados en Webkit y Blink, así que crearemos un método auxiliar:

```
Dom.match = function(element, selector) {
  var matchesSelector = element.matchesSelector ||
element.webkitMatchesSelector || element.mozMatchesSelector ||
element.oMatchesSelector || element.msMatchesSelector;

  return matchesSelector.call(element, selector);
};
```

Así, cambiamos `e.target.webkitMatchesSelector(selector)` por `Dom.match(e.target, selector)`:

```
Dom.prototype.delegate = function(eventName, selector, callbak) {
  var i = 0,
      eventIdentifier = selector + ':' + eventName;

  if (this.events === undefined) {
    this.events = {};
  }

  if (this.events[eventIdentifier] === undefined) {
    this.events[eventIdentifier] = [];
  }

  this.events[eventIdentifier].push(callback);

  for (i; i < this.elements.length; i++) {
    this.elements[i].addEventListener(eventName, function(e) {
      if (Dom.match(e.target, selector)) {
        callback(e);
      }
    }, true);
  }
};
```

Y lo utilizamos de la siguiente manera (podemos probar en el archivo http://cevichejs.com/files/3-dom-cssom/index.html):

```
var doc = new Dom(document);

doc.delegate('click', 'nav a', function(e) {
  console.log(e.target.getAttribute('href'));
});
```

```
// Si empezamos a hacer clic a los enlaces de la barra superior nos
saldrán los siguientes mensajes:
// "#carta"
// "#locales"
// "#historia"
```

Si agregamos un enlace más a la barra superior y hacemos clic en él, el evento `click` también se lanzará:

```
var nav = new Dom('nav');
nav.append({
  tag: 'a',
  attributes: {
    href: '#reservaciones'
  },
  content: 'Reservaciones'
});

// Hacemos clic en el enlace recientemente creado y saldrá el
siguiente mensaje:
// "#reservaciones"
```

Esta es una de las ventajas de utilizar *event delegation*: con un solo *listener* hemos capturado eventos de 4 enlaces.

Mejorando `dom.js`

Existen algunos métodos útiles que podemos agregar a `dom.js` y que nos servirán en La Buena Espina.

Por ejemplo, necesitaremos agregar y eliminar clases a elementos dentro del DOM:

```
Dom.prototype.addClass = function(className) {
  var i = 0;

  for (i; i < this.elements.length; i++) {
    this.elements[i].classList.add(className);
  }
};

Dom.prototype.removeClass = function(className) {
  var i = 0;

  for (i; i < this.elements.length; i++) {
    this.elements[i].classList.remove(className);
  }
};

Dom.prototype.hasClass = function(className) {
```

```
  var i = 0,
      hasClass = [];

  for (i; i < this.elements.length; i++) {
    hasClass.push(this.elements[i].classList.contains(className));
  }
};
```

Así como saber cuál es el primer y último elemento hijo de un contenedor:

```
Dom.prototype.firstChild = function() {
  return new Dom(this.elements[0].firstElementChild);
};

Dom.prototype.lastChild = function() {
  return new Dom(this.elements[0].lastElementChild);
};
```

O el lugar que ocupa un elemento entre sus elementos hermanos:

```
Dom.prototype.index = function() {
  return
Array.prototype.indexOf.call(this.elements[0].parentNode.children,
this.elements[0]);
};
```

En este método utilizamos el método `indexOf` de `Array`. Las propiedades en el DOM como `children` no devuelven arreglos, si no listas instancias de `NodeList` (o `HTMLCollection` dependiendo del navegador); y si bien estas listas no son arreglos, es posible utilizar los métodos propios de instancias de `Array` juntando dos conceptos vistos en el capítulo anterior: *prototypes* y el método `call`.

A grandes rasgos, `Array.prototype.indexOf` es una función que itera a partir de los elementos del *arreglo* que lo ejecuta (es decir, su *contexto*) mientras busca por el elemento que es pasado como parámetro, cuando lo encuentra devuelve el número de la iteración en la que ha sido encontrado, el cual es el índice en el que se encuentra dicho elemento. En nuestro caso, `children` no es un arreglo, pero todas sus propiedades pueden ser accesibles mediante índices que empiezan en 0, y tiene una propiedad `length` que contiene el número de elementos dentro de la lista. Este tipo de objetos son llamados *array-like objects* y se pueden ver en muchas partes del DOM y del lenguaje en sí (por ejemplo, la palabra reservada `arguments` o las instancias de `CSSStyleDeclaration` y `NamedNodeMap` son *array-like objects*).

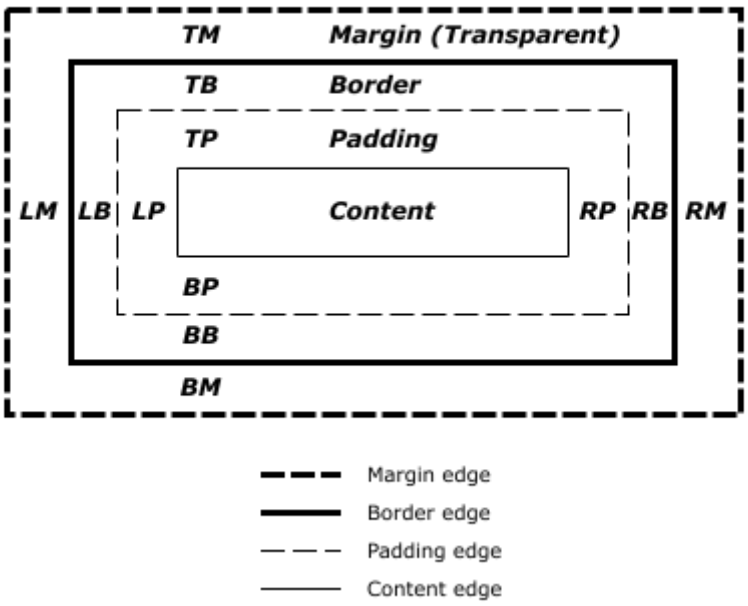
CSSOM

El Cascade Style Sheet Object Model, o CSSOM, toma los conceptos de DOM y los lleva a las hojas de estilo en cascada que componen un documento HTML. Esto permite tener un control más profundo de las reglas y propiedades que se aplican tanto a un elemento como a un documento HTML, utilizando JavaScript.

Box model

En el navegador, cada elemento mostrado en pantalla tiene forma de *caja* (o rectángulo), por lo que tiene un alto y un ancho, el cual puede ser definido por el navegador mismo o por el usuario mediante CSS.

La especificación de CSS define el denominado *box model*, el cual empieza definido como un rectángulo que limite el contenido del elemento (*content area*). Este rectángulo *content area* está rodeado por otro rectángulo más grande, denominado *padding*; el *padding*, a su vez, está rodeado por un rectángulo externo denominado *border*. Por último, el rectángulo *border* está rodeado por el rectángulo *margin*:



Box model. *http://www.w3.org/TR/CSS2/box.html*

Los rectángulos *padding*, *border* y *margin* pueden ser personalizados mediante CSS. Incluso, los 4 lados de cada rectángulo pueden tener valores diferentes.

Reglas y propiedades

Cada elemento de una hoja de estilos es una *regla*, y cada regla está definida por dos partes: el *selector* (es decir, el elemento o elementos a los cuales aplicará la regla), y una lista de *propiedades*.

```
body {
  font-family: 'Arial';
  background: green;
  color: white;
}
```

En este caso, existe una regla cuyo selector es `body`, y sus propiedades son las definidas por `font-family`, `background` y `color`.

Agregando reglas

Cada hoja de estilos en un documento es accesible mediante la propiedad `styleSheets` de `document`. Esta devuelve una lista viva, instancia de `StyleSheetList` (similar a `NodeList`), conteniendo objetos instancias de `CSSStyleSheet`. Por cada hoja de estilos, enlazada mediante la etiqueta `<link>` o definida dentro del documento con `<style>`, existe una instancia de `CSSStyleSheet` dentro de `document.styleSheets`.

Cada instancia de `CSSStyleSheet` tiene una propiedad `cssRules`, una lista *viva* instancia de `CSSRuleList` (similar a `StyleSheetList` y `NodeList`). Esta lista contiene objetos de diferentes interfaces, dependiendo del tipo de regla que referencia en la hoja de estilos:

Tipo de regla	**Interfaz**
Estilos	`CSSStyleRule`
Definir un charset (`@charset`)	`CSSCharsetRule`
Importar una hoja de estilos (`@import`)	`CSSImportRule`
Media query (`@media`)	`CSSMediaRule`
Font face (`@font-face`)	`CSSFontFaceRu`
Definir estilos para impresión (`@page`)	`CSSPageRule`
Una lista de key frames (`@keyframes`)	`CSSKeyframesRu`
Un elemento de una lista de key frames	`CSSKeyframeRu`

Solo en Webkit y Blink, los objetos `CSSStyleSheet` que referencian a hojas de estilos enlazadas desde un dominio diferente al del documento (como un CDN) no permiten leer sus reglas, mientras que en el resto de casos y navegadores sí es posible.

Para poder agregar una regla a una hoja de estilos se utiliza el método `addRule` o `insertRule`, disponible en cada instancia de `CSSStyleSheet`. Mientras `addRule` acepta 3 parámetros (selector, propiedades, índice donde agregar la regla), `insertRule` acepta solo 2 (cadena con el contenido completo de la regla, índice donde agregar la regla).

`window.getComputedStyle`

Este método devuelve un objeto instancia de `CSSStyleDeclaration` con todos los estilos de un elemento pasado como parámetro. Estos estilos son calculados por el navegador a partir de estilos propios del sistema operativo, el navegador y hojas de estilos incluidas en el documento que contengan reglas aplicables a dicho elemento. Estos estilos son de solo lectura, a diferencia de los obtenidos por la propiedad `style` de cada instancia de `CSSStyleRule`.

En `dom.js` crearemos un método `style`:

```
Dom.prototype.style = function() {
  var i = 0;

  var styles = [];

  for (i; i < this.elements.length; i++) {
    styles.push(window.getComputedStyle(this.elements[i]));
  }

  return styles;
};
```

Sin embargo, `window.getComputedStyle` es un método pesado que impacta en el rendimiento de la aplicación, por lo que hay que tener cuidado en qué momentos se va a utilizar. Además, `window.getComputedStyle` devuelve una lista *viva*, así que cada vez que cambiemos algún estilo dentro de un elemento, se verá reflejado en cualquier variable o propiedad que guarde un valor previo de `window.getComputedStyle`:

```
var bodyStyle = window.getComputedStyle(document.body);

bodyStyle.backgroundColor;
// "rgb(255, 255, 255)"

document.body.style.background = 'rgba(10, 10, 10, 0.2)';

bodyStyle.backgroundColor;
// "rgba(10, 10, 10, 0.2)"
```

Media queries

Los *media queries* permiten utilizar CSS para actualizar la presentación de un documento en cuanto cumpla algunas condiciones, como el ancho y alto del *viewport*, orientación del dispositivo, entre otras. Si bien muchas de las condiciones se pueden verificar mediante JavaScript y el DOM, este tipo de operaciones son lentas y suelen perjudicar notablemente el rendimiento de una aplicación web (este tipo de perjuicio se nota aún más en dispositivos móviles), por lo que, de ser posible, es recomendable utilizar *media queries*.

`window.matchMedia`

Con este método se puede saber si una o más condiciones corresponden al estado actual del navegador, y devuelve una instancia de `MediaQueryList` que permite agregar *listeners* con el método `addListener`. A diferencia del DOM, este método solo acepta un parámetro, el cual es un callback que se ejecutará cada vez que cambie alguna de las condiciones que se están evaluando. Este manejo de eventos es útil para precargar estilos, *scripts*, o imágenes específicas según resoluciones diferentes.

Para manejar mejor este método crearemos un archivo `cssom.js`, el cual contendrá un objeto `CSSom`:

```
CSSom = {
  mediaQueries: {}
};
```

Como las instancias de `MediaQueryList` permiten agregar *listeners*, crearemos dos métodos para manejarlos de forma más simple:

```
CSSom.on = function(mediaQueryString, callback) {
  var mediaQueryList;

  if (this.mediaQueries[mediaQueryString] === undefined) {
    this.mediaQueries[mediaQueryString] = [];
  }

  mediaQueryList = window.matchMedia(mediaQueryString);
  mediaQueryList.addListener(callback);

  this.mediaQueries[mediaQueryString].push({
    mediaQueryList: mediaQueryList,
    callback: callback
  });
};

CSSom.off = function(mediaQueryString) {
  var i = 0,
      mediaQueryResult;
```

```
  if (this.mediaQueries[mediaQueryString] !== undefined) {
    for (i; i < this.mediaQueries[mediaQueryString]; i++) {
      mediaQueryResult = this.mediaQueries[mediaQueryString];

mediaQueryResult.mediaQueryList.removeListener(mediaQueryResult.callba
ck);
    }

    this.mediaQueries[mediaQueryString] = [];
  }
};
```

De esta forma, podemos definir eventos para determinados media queries:

```
CSSom.on('(orientation: portrait)', function(mq) {
  if (mq.matches) {
    document.body.className = 'portrait';
  }
  else {
    document.body.className = 'landscape';
  }
});
```

Transiciones y Animaciones

Las transiciones y animaciones son nuevos estilos en CSS que permiten animar, valga la redundancia, elementos dentro de un documento, interpolando los valores de algunas de sus propiedades, como el alto, ancho y posición (aunque muchas otras propiedades pueden ser animadas).

Por ejemplo, **La Buena Espina** se vería bien si le ponemos un efecto simple para cambiar la imagen de fondo cada cierto tiempo. Inicialmente necesitaremos tener una serie de elementos `div` con clase `slide`, y definimos que todos esos elementos por defecto no deben ser visibles, dándole un valor de 0 a la propiedad `opacity`, mientras que el *slide* que quiera mostrarse debe tener el valor de 1 en la misma propiedad:

```
.slide {
  opacity: 0;
}

.slide.current {
  opacity: 1;
}
```

El siguiente paso es definir una transición para la propiedad `opacity`, la cual durará 5 segundos. El navegador es el encargado de calcular los valores que tendrá `opacity` durante los 5 segundos que dure la transición:

```
.slide {
  opacity: 0;
```

```
  transition-property: opacity;
  transition-duration: 5.0s;
}

.slide.current {
  opacity: 1;
}
```

Las animaciones en CSS son similares a las transiciones, con la principal diferencia que las animaciones dan más control al usuario con respecto a los valores que puede tener una propiedad dentro del ciclo de vida de una animación (mientras que, en una transición, es el navegador el que calcula dichos valores).

Para definir una animación en CSS, necesitamos definir una regla del tipo *keyframes*, la cual contendrá una lista de puntos en donde una propiedad cambiará de valor. De esta forma, el navegador se encargará de tomar dos *keyframes* (uno de partida y uno de fin) y calculará los valores intermedios para el tiempo que debe transcurrir entre estos dos *keyframes*. Si bien en esto las animaciones son similares a las transiciones, en las animaciones pueden haber más de dos *keyframes*, lo que le da más control al usuario al momento de animar un elemento.

Basados en el ejemplo anterior, podemos utilizar animaciones en vez de transiciones:

```
.slide {
  opacity: 0;
}

.slide.current {
  animation-name: slide;
  animation-duration: 5s;
  animation-fill-mode: forwards;
}

@keyframes slide {
  0% {
    opacity: 0;
  }

  50% {
    opacity: 0.3;
  }

  80% {
    opacity: 0.7;
  }

  100% {
    opacity: 1;
  }
}
```

En este caso, con las animaciones tenemos el poder de definir qué valores tendrá `opacity` al inicio (0%), fin (100%), a los 2 segundos y medio (50%), y a los 4 segundos (80%) de transcurrida la animación.

El hecho de tener más control en las animaciones también se ve reflejado en el CSSOM. Mientras que solo existe el evento `transitionend` para las transiciones, las animaciones tienen 3 eventos: `animationstart`, `animationiteration`, `animationend`.

Por ejemplo, si queremos agregar un *listener* al evento `transitionend` que muestre en la consola el tiempo transcurido en la transición:

```
dom('.slide').on('transitionend', function(e) {
  console.log('transitionend', e.elapsedTime);
});
```

Mientras que, si utilizamos animaciones, podemos agregar un *listener* al evento `animationstart` para saber cuándo empezó una animación:

```
dom('.slide').on('animationstart', function(e) {
  console.log('animationstart', e);
});
```

Y agregar un *listener* al evento `animationend` para saber en qué momento terminó una animación:

```
dom('.slide').on('animationend', function(e) {
  console.log('animationend', e);
});
```

El evento `animationiteration` es lanzado cada vez que empieza una iteración de la animación. Una animación puede ser definida para ser ejecutada un número determinado de veces (y cada vez es una **iteración**), mediante la propiedad `animation-iteration-count`.

Sabiendo un poco más sobre transiciones y animaciones crearemos un *script* simple para crear el efecto para cambiar la imagen de fondo. Primero, debemos tener un poco de HTML base:

```
<div id="background">
  <div class="slide current" id="slide-1" title="Créditos:
http://www.flickr.com/photos/saucesupreme/6774616862/"></div>
  <div class="slide" id="slide-2" title="Créditos:
http://www.flickr.com/photos/c32/4775267221/"></div>
  <div class="slide" id="slide-3" title="Créditos:
http://www.flickr.com/photos/renzovallejo/7998183161/"></div>
</div>
```

Notemos que el primer *slide* tiene la clase `current`, de esta forma nos aseguramos de mostrar una imagen al cargar el sitio.

Ahora, agregamos el CSS respectivo:

```
#background {
  display: block;
  width: 100%;
  height: 100%;
  position: relative;
}

#background .slide {
  display: block;
  width: 100%;
  height: 100%;
  position: absolute;
  opacity: 0;
  transition-property: opacity;
  transition-duration: 3.5s;
}

#background .slide.current {
  opacity: 1;
}

#slide-1 {
  background: url('../images/slides/slide-1.jpg') no-repeat center
center;
  background-size: 100% auto;
}

#slide-2 {
  background: url('../images/slides/slide-2.jpg') no-repeat center
center;
  background-size: 100% auto;
}

#slide-3 {
  background: url('../images/slides/slide-3.jpg') no-repeat center
center;
  background-size: 100% auto;
}
```

Hemos definido una transición para la propiedad `opacity` que dure 3 segundos y medio. De esta forma, si le agregamos la clase `current` a un elemento con la clase `slide`, se *disparará* la transición que cambie el valor de `opacity` de 0 a 1.

Hasta este momento solo se mostrará la primera imagen de fondo y no cambiará. Para empezar con la secuencia de imágenes, usaremos el evento `load` de `window` para agregarle la clase `current` al siguiente elemento:

```
dom(window).on('load', function() {
  var current = dom('.slide.current'),
      next = current.next();

  current.removeClass('current');
  next.addClass('current');
});
```

Este código tiene dos particularidades que no hemos visto en este capítulo:

- La función `dom`: En realidad solo devuelve una instancia de `Dom`, pero es útil ya que evita tener que crear una instancia de `Dom` cada vez que queramos trabajar con el DOM.
- El método `next`: Toma el elemento actual (en este caso, el *slide* que tenga la clase `current`) y devuelve su siguiente elemento hermano, con `nextElementSibling`.

Con este código, ya podremos ver que cambia de la primera imagen a la segunda mediante una transición de opacidad (propiedad `opacity`), pero al terminar esta transición no cambia a la tercera. Para lograrlo usaremos el evento `transitionend`:

```
dom('#background').delegate('transitionend', '.slide.current',
function(e) {
  var current = dom(e.target),
      next = current.next();

  current.removeClass('current');
  next.addClass('current');
});
```

Estamos utilizando el *event delegation* para definir un solo evento `transitionend`, en vez de tener uno por cada elemento con clase `slide`. Así mismo, definimos el evento solo para el *slide* visible en el momento, ya que el evento `transitionend` se disparará tanto para cuando termina la transición del valor de `opacity` de 0 a 1 (invisible a visible) como de 1 a 0 (visible a invisible).

Sin embargo, cuando ya se haya mostrado el último elemento (aquel que tiene id `slide-3`) y se lance el evento `transitionend`, el callback tratará de encontrar el siguiente elemento con `current.next()`. Dado que `current` guarda una referencia al último *slide* de la lista, `next()` no encontrará ningún valor, por lo que, internamente, la propiedad `elements` tendrá valor `null` y dará error al intentar ejecutar el método `addClass`.

Para corregir este pequeño *bug*, verificamos si se llegó al último elemento mediante el método `isLastSibling`. Si es el último elemento de la lista (en otras palabras, el último de sus elementos hermanos), `next` guarda una referencia al primer elemento de la lista con el método `firstSibling`.

```
dom('#background').delegate('transitionend', '.slide.current',
function(e) {
  var current = dom(e.target),
      next = current.next();

  current.removeClass('current');

  if (current.isLastSibling()) {
    next = current.firstSibling();
  }

  next.addClass('current');
});
```

Capítulo 4: APIs del navegador

Ya conocemos el DOM, con el que podemos modificar elementos y manejar eventos. De paso, vimos algo del CSSOM, y logramos saber cuándo se ejecutan algunas animaciones. Pero el navegador no se limita a ello y ofrece más APIs para crear aplicaciones interactivas y complejas.

Aplicaciones web y HTML5

Una aplicación web es una herramienta similar a una aplicación de escritorio, pero que es utilizada dentro del navegador, y tiene dos ventajas importantes:

- Es ubicuo: Una aplicación web está disponible en casi cualquier equipo que tenga un navegador web incorporado. Debido a que no existe necesidad de instalar una aplicación, la información del usuario está disponible sin importar el equipo desde donde se acceda a la aplicación.
- Es auto-actualizable: Una aplicación web no reside en el equipo, si no en un servidor web. Esto tiene como ventaja que puede ser actualizada sin necesidad de la interacción del usuario.

Mientras que, como desventajas:

- Su disponibilidad depende de otros factores: De una conexión a Internet, del servidor de la aplicación (tanto para aplicaciones de Internet como intranets), y en situaciones menos comunes, del navegador usado.
- Está limitado al navegador: El navegador por definición está limitado en cuanto a lo que puede acceder del equipo, lo que en términos técnicos se conoce como *sandboxing*. Este tipo de limitaciones, por consiguiente, limitan a las aplicaciones web que se ejecutan dentro de él.

Las aplicaciones web no son de ahora y no implican utilizar solo JavaScript. Existían aplicaciones web antes del llamado *Web 2.0*, que utilizan JavaScript, Java, Flash, Flex, Silverlight, e incluso algunas solo utilizan HTML y CSS. Sin embargo, el

avance que ha tenido JavaScript, desde la *Web 2.0* hasta el HTML5, ha logrado superar en alguna forma las desventajas que tenían las aplicaciones web.

Ahora se pueden realizar peticiones al servidor sin recargar toda la página, tener una comunicación interactiva con el servidor, realizar algunas operaciones sin necesidad de tener una conexión, leer y escribir archivos en el equipo, entre otros.

Algunas de las siguientes APIs del navegador servirán para **La Buena Espina**, pero utilizarlas no significa que estemos creando una aplicación web, ya que una aplicación web no está determinada por las tecnologías que usa.

Web Storage

Empecemos con una API simple de usar pero que soluciona un problema común al trabajar con una aplicación web: El dueño de **La Buena Espina** quiere un formulario de contacto para que los comensales puedan dar sus impresiones sobre el servicio y la comida. Pero, ¿qué pasaría si luego de enviar el formulario se pierde la conexión, el usuario cierra su navegador o el servidor no responde? Los comentarios no llegarán al dueño y se pueden perder buenas críticas con respecto al restaurante.

Las APIs de Web Storage solucionan este problema, al menos en parte, ya que permiten guardar información en el navegador. Esta información es guardada en formato *nombre - valor*, similar a un solo objeto plano en JavaScript, y puede existir en dos formas:

- *Local Storage*: Existiendo uno por cada *origen* (el valor devuelto por `location.origin`). Estará disponible luego de haber cerrado el navegador.
- *Session Storage*: Similar al *Local Storage*, solo está disponible mientras el navegador esté abierto.

Estos valores sobreviven a pestañas cerradas y recargadas, similar a las *cookies*, excepto que estas tienen un tamaño máximo, y otras limitaciones. Sin embargo, tanto el *Local Storage* como el *Session Storage* tienen un tamaño máximo de 5 megabytes por *origen*.

Un punto importante a resaltar es que ambos *Storages* se comportan como objetos planos globales, y guardan tanto los nombres como los valores en forma de cadenas. Adicionalmente, tienen dos métodos para acceder y asignar valores: `getItem` y `setItem`.

Para el caso descrito en el primer párrafo, podríamos utilizar el *Local Storage* junto a `dom.js`:

```
dom('#contact-form').on('submit', function() {
  window.localStorage.setItem('contact_content', dom('#contact-
comment').value());
});
```

De esta forma, cuando enviemos el formulario, se guardará el contenido del elemento `#contact-comment` en el *Local Storage* bajo el nombre `contact_content`.

Soporte para Web Storage

Geolocation

El dueño de **La Buena Espina** quiere que el sitio web de su restaurante invite al usuario a ir a sus locales, y una forma de lograr eso es indicarle a sus posibles clientes la ubicación exacta de sus locales. Pero con eso no basta, porque también podría indicarle al potencial cliente **cómo llegar** a alguno de sus locales, dependiendo de un dato que ahora es más fácil de conseguir: la *geolocalización*.

La *Geolocation API* utiliza diferentes formas para conocer la ubicación de un usuario (o, mejor dicho, del equipo que está utilizando un usuario), diiriendo cada forma en la precisión de la ubicación; y, cuando esta API logra encontrar la ubicación del equipo, devuelve un objeto con 2 valores básicos: la latitud y la longitud. Estos valores numéricos permiten ubicar un lugar en la Tierra a partir de un sistema de coordenadas único para todo el mundo, así que podemos tener la certeza que el valor que devuelva esta API será (relativamente) exacto.

Para trabajar con la *Geolocation API* tenemos que acceder a un objeto dentro de `navigator` llamado `geolocation`, el cual contiene 3 métodos:

- `getCurrentPosition`: Trata de obtener la ubicación del equipo y toma 3 parámetros: Un *callback* que se ejecutará si se logra obtener la ubicación del equipo, un segundo *callback* que se ejecutará si no se logra obtener la ubicación (indicando el motivo del error), y un tercer objeto con configuración de la petición.
- `watchPosition`: Toma los mismos parámetros de `getCurrentPosition` y realiza un monitoreo de la ubicación del equipo, ejecutándose cada vez que el navegador detecte que la ubicación del equipo ha cambiado. Este método devuelve un id, el cual es utilizado por `clearWatch`.
- `clearWatch`: Detiene el monitoreo creado por el método `watchPosition`.

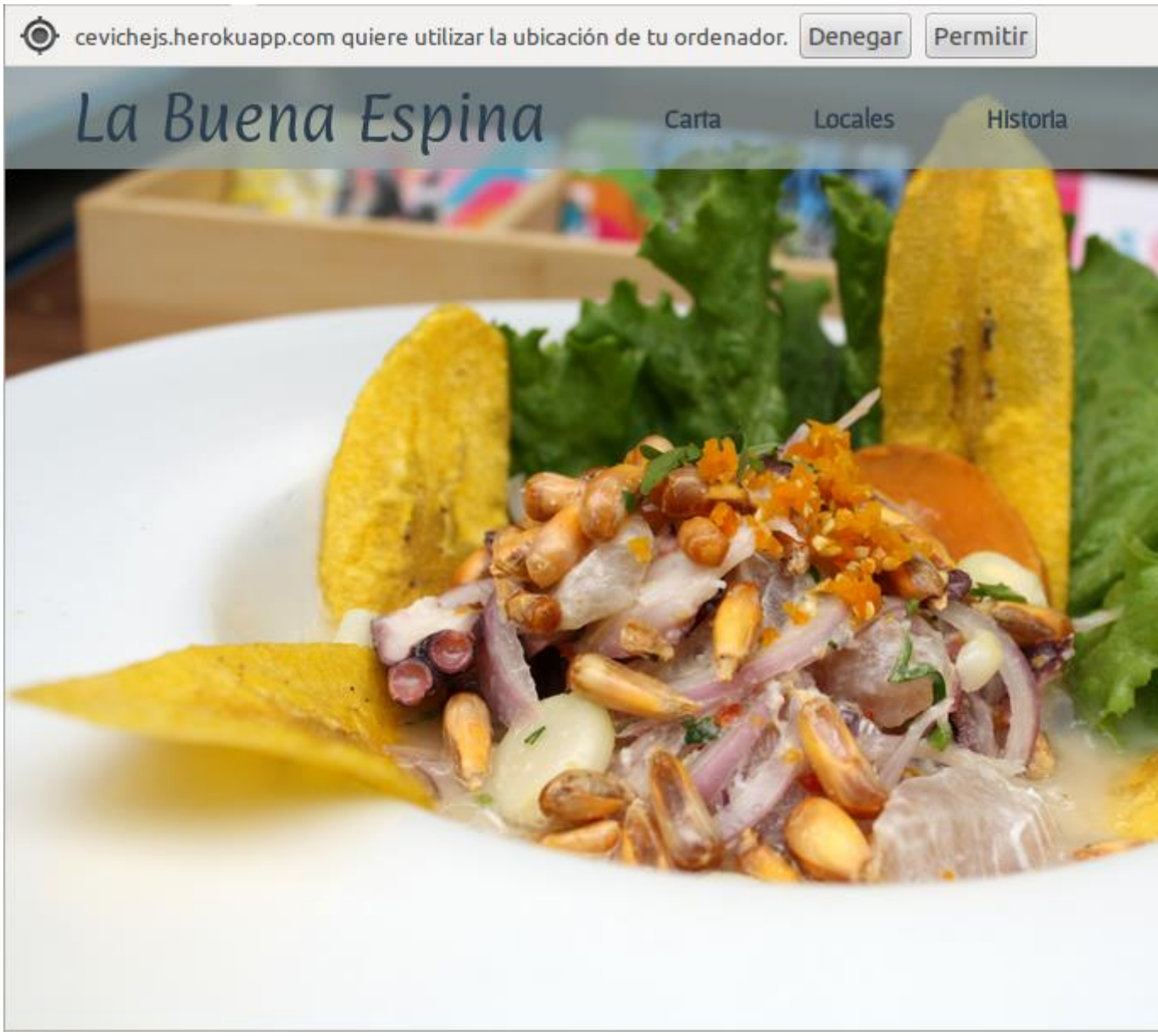
El último parámetro de `getCurrentPosition` y `watchPosition` puede tener los siguientes valores:

- `enableHighAccuracy`: Define un valor booleano que indica si la API tratará de obtener el valor más exacto para la ubicación.
- `timeout`: Indica el tiempo máximo (en milisegundos) que la API esperará por obtener un resultado, o, en caso contrario, lanzar el *callback* de error (segundo parámetro).
- `maximumAge`: Indica el tiempo máximo (en milisegundos) que el navegador guardará en memoria el valor devuelto por la API.

Sabiendo esto, podemos empezar a trabajar con la *Geolocation API*:

```
navigator.geolocation.getCurrentPosition(function(position) {
  console.log(position.coords);
}, function(error) {
  console.log(error.message, error.code);
}, {
  enableHighAccuracy: true,
  timeout: 2500,
  maximumAge: 0
});
```

Cuando se ejecute este código se mostrará una ventana o un mensaje (dependiendo del navegador) pidiendo permiso al usuario para poder realizar la geolocalización. Es importante resaltar este punto ya que no es posible obtener la ubicación de un equipo sin previo permiso del usuario.



Permisos para geolocalización

Si denegamos el permiso de geolocalización al navegador, la consola nos mostrará este mensaje:

```
"User denied Geolocation"  1
```

Mientras que el primer valor devuelve un mensaje entendible para el usuario, el segundo valor es un código de error devuelto por la API, el cual puede tener 3 valores:

Valor	Descripción
1	El usuario no dio permiso al navegador
2	No se pudo encontrar la ubicación
3	Pasó más del tiempo permitido en el *timeout* definido por el tercer parámetro

Y si le damos el permiso, nos devolverá el siguiente objeto (cuyos valores pueden cambiar de acuerdo al equipo y al tipo de conexión):

```
{
  accuracy: 75,
  altitude: null,
  altitudeAccuracy: null,
  heading: null,
  latitude: -12.1042457,
  longitude: -76.9628362,
  speed: null
}
```

Cuando ya tenemos estos valores podemos utilizar algún servicio de mapas, como Google Maps, Mapbox o Leaflet para mostrar la ubicación de forma visual en un mapa.

Soporte para Geolocation

Application Cache

Para el caso de aplicaciones web suele ser de vital importancia el poder acceder a ellas de manera *offline*, sobre todo si la conexión a Internet solo se hace necesaria para respaldar información en un servidor externo. En este tipo de aplicaciones donde se debería poder acceder a los archivos "estáticos" de la aplicación

independientemente del estado de conexión que tenga el equipo, y es aquí donde el *Application Cache* entra en acción.

Esta API permite definir un archivo *manifiesto* que indicará cuáles son los archivos que se desean descargar cuando el navegador se conecta a la aplicación cuando está *online*, y que luego utilizará cuando no exista una conexión a Internet disponible.

Un manifiesto básico sigue el siguiente formato:

```
CACHE MANIFEST

/
/images/logo.png
/images/sprites.png
/styles/layout.css
/javascript/libraries/dom.js
/javascript/app.js
```

En este caso, el manifiesto le indica al navegador que debe descargar y guardar en caché todos los archivos ubicados en esas rutas. Sin embargo, las capacidades de este manifiesto no se reducen a indicar la lista de archivos a guardar en caché, si no que permite indicar cuáles deben ser obtenidos siempre desde el servidor, así como indicar archivos que se utilizarán cuando la conexión falle.

```
CACHE MANIFEST

CACHE:
/
images/logo.png
images/sprites.png
styles/layout.css
scripts/libraries/dom.js
scripts/app.js

NETWORK:
*

FALLBACK:
/ /offline.html
```

Este nuevo manifiesto indica explícitamente cuáles son los archivos que deben ser guardados en caché (similar al primer manifiesto), así como los archivos que deben obtenerse del servidor (por defecto, todos los que no están definidos debajo de la línea `CACHE`), y define el archivo que el navegador debe usar en caso algún archivo no pueda ser obtenido.

Para que el navegador sepa dónde encontrar este archivo, debe ser incluido como atributo dentro de la etiqueta `<html>`:

```
<html manifest="manifest.appcache">
```

```
  ...
</html>
```

Soporte para Application Cache

File

Con esta API podemos leer archivos que cargamos desde el navegador, mediante la etiqueta `<input type="file">`, así como al realizar operaciones *drag and drop* de manera nativa. De esta forma, podemos previsualizar imágenes antes de subirlas a un servidor o realizar operaciones con los archivos aunque la aplicación esté *offline*.

Cuando trabajamos con elementos `<input type="file">` podemos acceder a los archivos que han sido elegidos mediante la propiedad `files`, la cual es una lista instancia de `FileList`. Cada elemento de esta lista es un objeto instancia de `File` y tiene algunas propiedades:

Propiedad	Descripción
`name`	Nombre del archivo
`size`	Tamaño en bytes del archivo
`type`	*MIME type* del archivo
`lastModifiedDate`	Última fecha de modificación del archivo

Sabiendo las propiedades que tienen estos objetos de *File API*, podemos crear un demo simple, empezando con el código HTML básico:

```
<input type="file" name="files" id="files" multiple>

<h4>Imágenes elegidas:</h4>
<div id="preview"></div>
```

Y luego utilizamos la API propiamente dicha:

```
var input = document.getElementById('files'),
    preview = document.getElementById('preview');

input.addEventListener('change', function(e) {
  var files = e.target.files;

  for (var i = 0; i < files.length; i++) {
```

```
    (function(file) {
      var reader = new FileReader(),
          img = document.createElement('img');

      img.width = 300;
      preview.appendChild(img);

      reader.addEventListener('load', function(e) {
        img.src = e.target.result;
      });

      reader.readAsDataURL(file);
    })(files[i]);
  }
});
```

Dentro de este código de ejemplo utilizamos la función constructora `FileReader`, la cual permite leer las instancias de `File` y convertirlo a una cadena de tipo **Data URI** para, de esta forma, poder cargarlo en un elemento `<img>`.

En el bucle que lee cada imagen obtenida por el input `files` utilizamos una función inmediatamente invocada debido a la naturaleza asíncrona de `FileReader`. Con este tipo de funciones, se obliga al navegador a ejecutar todo el código dentro de la función antes de pasar a la siguiente iteración, lo que nos asegura que se lean los valores correctos para cada iteración.

Soporte para File

File System

Esta API simula un sistema de archivos en el navegador, permitiendo crear, modificar y leer archivos mediante JavaScript. Este sistema de archivos simulado no es el sistema de archivos del sistema operativo, si no que está separado en un entorno controlado (a este tipo de entornos se le llama *sandbox*). Actualmente esta API está en fase experimental y está disponible en Chrome y Opera, por lo que tiene un uso potencial en aplicaciones para Chrome OS o aplicaciones web que funcionan con Chromium.

Al ser un entorno controlado, *File System API* tiene ciertas restricciones:

- **Cada origen tiene su sistema de archivos**: Un origen está formado por el protocolo, dominio y puerto de un documento. Similar a las APIs de Storage, cada origen tiene su propio sistema de archivos y no se pueden acceder entre sí.
- **No se pueden crear o renombrar archivos ejecutables**: Por seguridad, no se pueden crear archivos ejecutables, ya que estos pueden ser aplicaciones maliciosas (virus, troyanos, etc).
- **No se puede salir del *sandbox***: Igualmente, por seguridad, una aplicación no puede usar la API para acceder a archivos que estén en el sistema de archivos del sistema operativo.

- **No se puede ejecutar desde el protocolo `file://`**: También por seguridad, si se tratar de utilizar esta API en un archivo desde `file://`, el navegador lanzará una excepción y fallará.

Adicionalmente, esta API tiene soporte para trabajar de forma síncrona y asíncrona, recomendando utilizar WebWorkers para el primer caso:

Interfaz	Descripción
`LocalFileSystem`	Permite acceder al sistema de archivos controlado
`FileSystem`	Representa un sistema de archivos
`Entry`	Representa una entrada en el sistema de archivos, el cual puede ser un archivo o un directorio
`DirectoryEntry`	Representa un directorio en el sistema de archivos
`DirectoryReader`	Permite leer un directorio en el sistema de archivos
`FileEntry`	Representa un archivo en el sistema de archivos
`FileError`	Error lanzado cuando falla el acceso al sistema de archivos

Para empezar a trabajar con esta API debemos pedirle al navegador que nos de un sistema de archivos para el origen en el cual estamos trabajando:

```
var requestFileSystem = window.requestFileSystem ||
window.webkitRequestFileSystem;

requestFileSystem(window.TEMPORARY, 1024 * 1024 * 5,
function(fileSystem) {
  console.log(fileSystem);
}, function(error) {
  console.log(error);
});
```

Donde `requestFileSystem` es una variable que guardará una referencia a `window.requestFileSystem` (de existir), o de `window.webkitRequestFileSystem` (en caso `window.requestFileSystem` no exista). Este tipo de asignaciones son comunes cuando se trabaja con APIs que aún no son estándares, ya que primero se busca la implementación *estándar*, y luego la implementación propia del

navegador (la cual va acompañada de un prefijo, que puede ser `webkit`, `moz`, `ms` u `o`).

`requestFileSystem` es una función que permite obtener un sistema de archivos dentro del navegador, y tiene 4 parámetros:

- **Tipo de almacenamiento**: el cual puede ser `window.TEMPORARY` (el navegador puede borrar los archivos si necesita espacio) o `window.PERSISTENT` (solo el usuario puede borrar los archivos).
- **Tamaño en bytes**: El tamaño que se quiere asignar al sistema de archivos, el cual puede requerir un permiso explícito del usuario si el tamaño pedido es muy grande.
- **Callback de éxito**: Este callback toma un parámetro, el cual es una instancia de `DOMFileSystem` y tiene dos propiedades: `name` y `root` (instancia de `DirectoryEntry`)
- **Callback de error**: Este callback también toma un solo parámetro, el cual es una instancia de `FileError`y contiene 3 propiedades: el código del error, el nombre del error y un mensaje descriptivo.

Otro punto importante es ver cómo una variable guarda una referencia a una función. Recordemos que las funciones son ciudadanos de primera clase en JavaScript, por lo que es posible guardarlas en una variable, o pasarlas como parámetros (como en los dos últimos valores de `requestFileSystem`).

Si se desea crear un sistema de archivos *persistente* se debe pedir una cuota de espacio al navegador:

```
window.webkitStorageInfo.requestQuota(window.PERSISTENT, 1024 * 1024 *
5, function(bytes) {
  window.webkitRequestFileSystem(window.PERSISTENT, bytes,
function(fileSystem) {
    console.log(fileSystem);
  }, function(error) {
    console.log('Error en requestFileSystem', error);
  });
}, function(error) {
  console.log('Error en requestQuota', error);
});
```

Chrome recomienda utilizar `navigator.webkitTemporaryStorage` o `navigator.webkitPersistentStorage` en vez de `window.webkitStorageInfo` para obtener la cuota de espacio. Ambos objetos siguen teniendo el método `requestQuota`.

Escribiendo archivos

Luego de haber obtenido el sistema de archivos, podemos crear un archivo de la siguiente forma:

```
function successCallback(fileSystem) {
  fileSystem.root.getFile('demo.txt', { create : true },
function(fileEntry) {
    fileEntry.createWriter(function(writer) {
      writer.onwriteend = function(e) {
        console.log('Archivo creado.');
      };

      var blob = new Blob(['Texto', ' de ', 'prueba']);
      writer.write(blob);
    });
```

```
  });
}
```

Para crear un archivo tenemos que seguir dos pasos: obtener una referencia al archivo que queremos crear (con `getFile`), y crear una instancia de `FileWriter` (con `createWriter`).

`getFile` acepta 4 parámetros: El nombre del archivo, un objeto de opciones y dos callbacks, uno de éxito y otro de error. Es en el objeto de opciones donde se indica si el archivo se creará o editará (en ambos casos se utiliza `create : true`, pero si solo se quiere crear un archivo y evitar reescribir uno existente, se añade `exclusive : true`).

La instancia de `FileWriter` tiene diferentes *handlers* para manejar los eventos relacionados a la escritura del archivo, pero también tiene a `EventTarget` en su **cadena de prototypes**. Esto quiere decir que podemos utilizar los métodos `addEventListener` y `removeEventListener` para manejar los eventos de esta instancia.

Por último, para poder realizar la escritura del archivo, propiamente dicha, debemos crear una instancia de `Blob`, el cual tiene como primer parámetro un arreglo, el cual contiene las partes del contenido del archivo. Estas partes pueden ser cadenas, u otras instancias de `Blob`. Luego, se debe utilizar el método `write` de la instancia de `FileWriter` para escribir el *blob*.

Leyendo archivos

Para leer archivos también necesitamos obtener una referencia del archivo que deseamos leer; y para esto usamos el método `getFile`, solo que en este caso el segundo parámetro no tendrá ningúna propiedad.

```
function successCallback(fileSystem) {
  fileSystem.root.getFile('demo.txt', {}, function(fileEntry) {
    fileEntry.file(function(file) {
      var reader = new FileReader();

      reader.onloadend = function(e) {
        console.log(this.result);
      };

      reader.readAsText(file);
    });
  });
}
```

Luego de obtener la referencia del archivo debemos obtener el archivo en sí, mediante el método `file`, para luego crear una instancia de `FileReader`. Esta función, que ya ha sido utilizada por la *File API*, permite leer un archivo como texto plano, utilizando el método `readAsText`.

Actualizando archivos

Para actualizar un archivo debemos seguir los mismos que se utilizaron para crear y escribir un archivo nuevo, excepto por un par de cambios:

- El valor de `create` debe ser `false`.
- Mover la posición del cursor de la instancia de `File Writer` al final del archivo, utilizando el método `seek`.

```
function successCallback(fileSystem) {
  fileSystem.root.getFile('demo.txt', { create : false },
function(fileEntry) {
    fileEntry.createWriter(function(writer) {
      writer.seek(writer.length);

      writer.onwriteend = function(e) {
        console.log('Archivo actualizado.');
      };

      var blob = new Blob(['\n', 'Texto', ' de ', 'prueba']);
      writer.write(blob);
    });
  });
}
```

En este caso, cualquier *blob* que se escriba en el archivo va a sobreescribir el contenido que pueda existir en la posición que el cursor se encuentre (por defecto está en la posición `0`, al inicio del archivo). Es por eso que, en este caso, se pone el cursor al final del archivo.

Creando carpetas

Para crear una carpeta es necesario utilizar el método `getDirectory`, el cual es similar a `getFile` en cuanto a parámetros:

```
function successCallback(fileSystem) {
  fileSystem.root.getDirectory('examples', { create : true },
function(directoryEntry) {
    // directoryEntry
  });
}
```

De esta forma ya tenemos la carpeta creada. `directoryEntry` es una instancia de `DirectoryEntry` (recordemos: `fileSystem.root` también es una instancia de `DirectoryEntry`, por lo que se pueden realizar las mismas operaciones que hemos visto anteriormente).

Obtener el contenido de una carpeta

Las instancias de `DirectoryEntry` tienen un método llamado `createReader`, el cual crea una instancia de `DirectoryReader`. Las instancias de `DirectoryReader` tienen un método llamado `readEntries`, el cual ejecuta dos callbacks, según si la operación ha sido exitosa o no, y devuelve la lista de *entradas* de una carpeta (una *entrada* puede ser un archivo o una carpeta).

```
function successCallback(fileSystem) {
  var directoryReader = fileSystem.root.createReader();
  directoryReader.readEntries(function(entries) {
    for (var i = 0; i < entries.length; i++) {
      console.log(entries[i]);
    }
  });
}
```

En este caso, es `fileSystem.root` quien crea una instancia de `DirectoryReader`, ya que queremos ver cuáles son los archivos y carpetas que se encuentran dentro de la carpeta raíz. Cabe notar que los elementos de *entries* puede ser instancias de `FileEntry` o de `DirectoryEntry`, dependiendo del tipo de entrada (archivo o carpeta, respectivamente).

Eliminando carpetas

Para eliminar una carpeta tenemos dos métodos de `DirectoryEntry`: `remove` y `removeRecursively`. El primer método solo podrá eliminar una carpeta si esta está vacía, mientras que el segundo método eliminará todo el contenido de la carpeta antes de eliminar la carpeta en sí.

```
function successCallback(fileSystem) {
  fileSystem.root.getDirectory('examples', {},
function(directoryEntry) {
    directoryEntry.remove(function() {
      console.log('Carpeta eliminada');
    }, function() {
      console.log('La carpeta no pudo ser elimninada');
    });
  });
}
```

El método `removeRecursively` funciona exactamente igual:

```
function successCallback(fileSystem) {
  fileSystem.root.getDirectory('examples', {},
function(directoryEntry) {
    directoryEntry.removeRecursively(function() {
      console.log('Carpeta eliminada');
    });
  });
}
```

Cabe destacar que todos los métodos usados dentro de la *FileSystem API* toman dos callbacks: uno de éxito y otro de error, donde este último siempre recibirá un único parámetro con las causas del error. De esta forma, es posible crear una sola función que sirva como callback de error para todos los métodos de la *FileSystem API*, como en este código de ejemplo.

Soporte para File System

History

Con la *History API* podemos simular entradas en el historial del navegador sin necesidad de realizar peticiones al servidor donde la aplicación está alojada (una entrada en el historial es cada página visitada en una pestaña de navegador).

Tradicionalmente, cuando un usuario ingresa a una dirección desde el navegador, o haciendo clic en un enlace, pasa lo siguiente:

1. El navegador realiza una petición al servidor al que apunta la dirección ingresada.
2. El servidor recibe la petición y la procesa, devolviendo una respuesta hacia el navegador.
3. El navegador muestra dicha respuesta al usuario final, lo cual también implica cambiar la dirección de la barra de direcciones del navegador mismo.
4. Se crea una entrada en el historial del navegador para la ventana actual. De esta forma el usuario sabe que está en una nueva página y que tiene la opciónde regresar a la anterior.

History API hace que estos pasos ya no sean obligatoriamente seguidos, ya que es posible cambiar la dirección de la barra de direcciones del navegador sin necesidad de hacer que el navegador envíe una petición al servidor, de tal forma que del flujo tradicional solo se ejecute el paso 4. Así mismo, ofrece un evento completamente nuevo, el cual se dispara cuando navegamos por las entradas del historial.

Agregando una entrada con `pushState`

El objeto `history` es el encargado de manejar el historial del navegador, y tiene algunos métodos como `back`, `forward` o `go` para navegar a través del historial, y una propiedad `length` que indica el número de entradas en el historial. A su vez, `history` tiene un método llamado `pushState`, el cual agrega una entrada al historial del navegador y cambia la dirección en la barra de direcciones del navegador, pero no realiza ninguna petición al servidor de la nueva dirección.

Suponiendo que los visitantes de **La Buena Espina** utilizan navegadores que soportan la *History API*, podemos hacer que los enlaces de la barra superior

utilicen `pushState`, y de esta forma mostrar las secciones manipulando el DOM (para esto, todas las secciones deben estar previamente cargadas en la página):

```
var state = {
  prevURL: '/carta',
  actualURL: '/locales'
};

history.length;
// 1

history.pushState(state, 'Locales', '/locales');

history.length;
// 2
```

El método `pushState` toma 3 parámetros:
1. Un objeto representado el *state* o estado de la nueva entrada en el historial. Sirve para guardar información relacionada a la URL que se está agregando.
2. El nuevo título que tendrá la pestaña del navegador. Este parámetro es ignorado por algunos navegador, por lo que podría no ser útil de momento.
3. La URL que se agregará al historial. Este parámetro reemplazará todo lo que venga después del origen (un origen está conformado por el protocolo, el dominio y el puerto de una dirección).

Cabe notar aquí que ni `pushState` ni `replaceState` pueden poner una URL cuyo origen sea diferente al actual, esto es por un tema de seguridad: Por ejemplo, se podría tener un enlace que modifique la URL actual por la URL de un banco de confianza, pero sin la necesidad de cargar la web de dicho banco.

Así mismo, por seguridad, la *History API* no está disponible en archivos en local (es decir, aquellos que se ejecuten desde el protocolo `file://`).

Reemplazando una entrada con `replaceState`

Si con `pushState` podemos agregar una entrada al historial, con `replaceState` podemos reemplazar la entrada actual (es decir, donde estemos navegando actualmente).

Siguiendo con el ejemplo anterior, tenemos que la dirección actualmente es `/locales`, y queremos que al buscar locales por distrito, cambie la dirección pero no agregue una entrada más en el historial:

```
var state = {
  prevURL: '/carta',
  actualURL: '/locales?buscar_en=Lince',
};

history.length;
// 2
```

```
history.replaceState(state, 'Locales', '/locales?buscar_en=Lince');

history.length;
// 2
```

El método `replaceState` toma los mismos parámetros que `pushState` y sirve, principalmente, para actualizar la entrada actual con algunos valores propios de la interacción del usuario con el sitio web.

Evento `popstate`

El evento `popstate` es lanzado cada vez que se *viaja* a través del historial, ya sea con los botones del navegador, o con los métodos `back`, `forward` o `go` de `history`.

Por ejemplo, para conocer el *state* de la entrada del historial a la cual se navegó, se puede utilizar el siguiente código:

```
window.addEventListener('popstate', function(e) {
  console.log(e.state);
});
```

Dado que este evento corresponde a la pestaña (o ventana) actual, es `window` el encargado de *escuchar* el evento.

Soporte para History

WebSocket

Los websockets permiten una comunicación bi-direccional entre el navegador y el servidor, de tal forma que este puede enviarnos datos sin necesidad de hacerle una petición (como ocurre en un modelo tradicional). Así, no solo podemos enviarle información al servidor, si no que podemos estar a la espera de *escuchar* los datos que el servidor pueda mandar por su cuenta.

Para poder utilizar websockets necesitamos tener un servidor de websockets, al cual se accede mediante el protocolo `ws://`, o `wss://` en caso de querer una conexión segura. Existen bibliotecas para crear servidores de websockets en varios lenguajes.

A grandes rasgos, el navegador abre conexiones HTTP de petición/respuesta: iniciará enviando una petición hacia el servidor y este enviará una respuesta; y cuando la respuesta ha sido enviada por el servidor, la conexión se cierra. Pero puede existir el caso donde se necesiten enviar peticiones o recibir respuestas sucesivamente (como tener notificaciones en una aplicación web o un chat en tiempo real), y abrir y cerrar conexiones puede demorar mucho. es aquí donde aparece WS: En WS se crea una

conexión y se deja abierta hasta que alguna de las dos partes cierre la conexión, no importa cuantos mensajes se manden entre sí.

En el navegador solo necesitamos crear una instancia de `WebSocket`, pasándole la url del servidor de websockets:

```
var connection = new WebSocket('ws://html5rocks.websocket.org/echo');

connection.onopen = function(e) {
  console.log('Connected');
};

connection.onclose = function(e) {
  console.log('Disconnected');
};

connection.onerror = function(e) {
  console.log('An error ocurred');
};

connection.onmessage = function(e) {
  console.log('Message received: ', e.data);
};
```

Estas 4 funciones definen 4 *handlers* para 4 eventos diferentes:

- `open`: Este evento es lanzado cuando se logra abrir una conexión con el servidor de websocket (en este caso con `ws://html5rocks.websocket.org/echo`)
- `close`: Es lanzado cuando se cierra una conexión, ya sea del lado del navegador o del servidor.
- `error`: Este evento es lanzado cuando existe un error en la conexión o en alguno de los lados de la comunicación.
- `message`: Es lanzado cuando llega un mensaje desde el otro lado de la comunicación (en este caso, del servidor). El parámetro que recibe el *handler* tiene una propiedad llamada `data`, el cual contiene el mensaje que el servidor envió.

`WebSocket` tiene a `EventTarget` en su **cadena de prototypes** (que es como se define la *herencia* en JavaScript), por lo que podemos usar `addEventListener` en la variable `connection`:

```
var connection = new WebSocket('ws://html5rocks.websocket.org/echo');

connection.addEventListener('open', function(e) {
  console.log('Connected');
});

connection.addEventListener('close', function(e) {
  console.log('Disconnected');
});
```

```
connection.addEventListener('error', function(e) {
  console.log('An error ocurred');
});

connection.addEventListener('message', function(e) {
  console.log('Message received: ', e.data);
});
```

Con este código tenemos lo básico para poder escuchar los datos que el servidor mande, pero si queremos enviarle información al servidor, debemos utilizar el método `send`:

```
connection.send('Hi from La Buena Espina');
```

Hay que tener algunas consideraciones al momento de utilizar un servidor de websockets. Por ejemplo, si se usan websockets en una web que usa HTTPS, las conexiones al servidor de websockets deben ser con WSS.

A diferencia de HTTP(S), los protocolos de websockets no cambian automáticamente (un protocolo cambia automáticamente cuando tenemos una imagen cuya url es `//labuenaespina.pe/logo.png` y según la página actual puede cargar `http://labuenaespina.pe/logo.png` o `https://labuenaespina.pe/logo.png`); por lo que se debe cambiar manualmente:

```
var websocketURL = '//html5rocks.websocket.org/echo';

if (location.protocol === 'http:') {
  websocketURL = 'ws:' + websocketURL;
}
else if (location.protocol === 'https:') {
  websocketURL = 'wss:' + websocketURL;
}

var connection = new WebSocket(websocketURL);
```

Opcionalmente, en el servidor se puede restringir si se aceptan o no conexiones de diferentes orígenes de websockets (mediante el Content Security Policy, específicamente la directiva `connect-src`).

Soporte para Websocket

Server-Sent Event

La *Server-Sent Event API* es una alternativa para los websockets, ya que permite que el navegador esté *escuchando* los datos que un servidor pueda mandar. En este caso, la API utiliza el protocolo HTTP(S), en comparación al protocolo WS que es utilizado por los websockets. Sin embargo, solo se pueden *escuchar* datos, mas no

enviar datos al servidor, por lo que puede ser utilizado en casos donde no es necesario o se quiere evitar que el navegador envíe datos al servidor.

Para abrir una conexión al servidor debemos crear una instancia de `EventSource`:

```
var sseConnection = new EventSource('/sse_stream');

sseConnection.addEventListener('open', function(e) {
  console.log('Connected');
});

sseConnection.addEventListener('close', function(e) {
  console.log('Disconnected');
});

sseConnection.addEventListener('error', function(e) {
  console.log('An error ocurred');
});

sseConnection.addEventListener('message', function(e) {
  console.log('Message received: ', e.data);
});
```

Una de las ventajas que tiene esta API es que podemos *escuchar* eventos propios, los cuales deben ser generados por el servidor mismo. Por ejemplo, podríamos tener un *feed* de eventos en La Buena Espina que indique cuando una mesa ha sido reservada:

```
sseConnection.addEventListener('booked_table', function(e) {
  var tableId = e.data;
  console.log('La mesa ' + tableId + ' ha sido reservada');
});
```

Hay que tener en cuenta que tanto para websockets como para server-sent events la información que recibimos del servidor (o que enviamos, en el caso de websockets) es una cadena, por lo que, de ser el caso, se deben hacer las conversiones necesarias, o utilizar JSON si se trabaja con arreglos u objetos.

Soporte para Server-Sent Event

Capítulo 5: Pruebas

Ahora que conocemos un poco más a fondo JavaScript y cómo manejar el DOM, así como algunas APIs del navegador, necesitamos estar completamente seguros de que nuestro código funcione, por lo que es importante realizar pruebas en él.

Pruebas automatizadas

Cada vez que copiamos un código y refrescamos el navegador para saber si funciona o no, estamos probando. Es un proceso rápido, pero en ocasiones puede ser tedioso y aburrido, así que lo ideal sería dejar que la computadora pruebe por nosotros. Es aquí donde aparecen las pruebas automatizadas.

En términos simples, una prueba automatizada verifica el correcto funcionamiento de un código mediante valores `true` o `false`. Por ejemplo, si deseamos probar una suma, podemos hacer lo siguiente:

```
var resultado = 10 + 15;

console.assert(resultado === 25, 'La suma de 10 y 15 debe ser 25');
```

Usamos `console.assert` para verificar una condición, que debe dar `true`, o, en caso contrario, mostrar un mensaje de error, el cual contiene la descripción de la validación. `console.assert` es un método de la consola disponible en todos los navegadores modernos (desde IE8 en adelante, Firefox, Chrome, Safari y Opera).

Idealmente, nuestro código más importante debe tener pruebas automatizadas que validen su correcto funcionamiento.

Pruebas unitarias

Existen diferentes tipos de pruebas, de acuerdo a la forma cómo se desea probar el código:

- Pruebas unitarias: Buscan validar una parte del código a la vez, sin importar para qué se utilice dicho código.
- Pruebas funcionales: Buscan validar toda acción que un usuario normalmente haría en el sitio o la aplicación web en la que se trabaja.

Las pruebas unitarias pueden ser sencillas de realizar, si se identifican correctamente las partes de la aplicación que deben probarse. En el capítulo 2 creamos un módulo llamado `titleBuilder`, que permite crear un título para la web de La Buena Espina según la sección que estemos visitando:

```
var titleBuilder = (function() {
  var baseTitle = 'La Buena Espina';
  var parts = [baseTitle];
```

```
  function getSeparator() {
    if (parts.length == 2) {
      return ' – ';
    }
    else {
      return ' › ';
    }
  };

  return {
    reset: function() {
      parts = [baseTitle];
    },
    addPart: function(part) {
      parts.push(part);
    },
    toString: function() {
      return parts.join(getSeparator());
    }
  };
})();
```

Ahora, crearemos algunas validaciones con `console.assert`:

```
console.assert(titleBuilder.toString() === 'La Buena Espina', 'El
título por defecto debe ser "La Buena Espina"');
```

De esta forma validamos que el título sea "La Buena Espina" si no hemos navegado por ninguna sección del sitio. ¿Qué pasaría si una validación falla? Tenemos un mensaje de error en la consola de la siguiente forma:

```
console.assert(titleBuilder.toString() === ' - La Buena Espina - ',
'El título por defecto debe ser "La Buena Espina"');
// Assertion failed: El título por defecto debe ser "La Buena Espina"
```

El primer parámetro es una condición que debe evaluarse como verdadero, mientras que el segundo parámetro es la descripción de la validación.

Podemos seguir haciendo más validaciones, de la siguiente forma:

```
titleBuilder.addPart('Carta');
titleBuilder.addPart('Pescados');
titleBuilder.addPart('Ceviches');

console.assert(titleBuilder.toString() === 'La Buena Espina › Carta ›
Pescados › Ceviches', 'El título ahora debe ser "La Buena Espina ›
Carta › Pescados › Ceviches"');

titleBuilder.reset();
titleBuilder.addPart('Locales');
```

```
console.assert(titleBuilder.toString() === 'La Buena Espina — Locales', 'El título ahora debe ser "La Buena Espina — Locales"');
```

Si las validaciones con `console.assert` pasan correctamente, la descripción de la validación no se mostrará. Este comportamiento puede no ser tan útil: **¿Cómo sabemos cuántas validaciones han pasado correctamente, y cuántas no?** Además, **¿de qué nos sirve tener los mensajes de error en la consola?**. Es aquí donde entran en escena diversos frameworks para pruebas, una de las cuales es QUnit.

QUnit

QUnit es un framework para pruebas unitarias creado por jQuery, donde, en lugar de utilizar la consola para mostrar los resultados, crea un reporte en HTML con los resultados de las pruebas realizadas. En QUnit, cada comparación que hacemos se llama *assert*, mientras que el conjunto de *asserts* es llamado *test*.

Para poder utilizar QUnit debemos descargar dos archivos desde su web (o utilizar los archivos vía su CDN):

```
<!DOCTYPE html>
<html>
<head>
  <title>Pruebas unitarias</title>
  <link rel="stylesheet" type="text/css"
href="http://code.jquery.com/qunit/qunit-1.14.0.css">
</head>
<body>
  <div id="qunit"></div>
  <div id="qunit-fixture"></div>
  <script type="text/javascript"
src="http://code.jquery.com/qunit/qunit-1.14.0.js"></script>
</body>
</html>
```

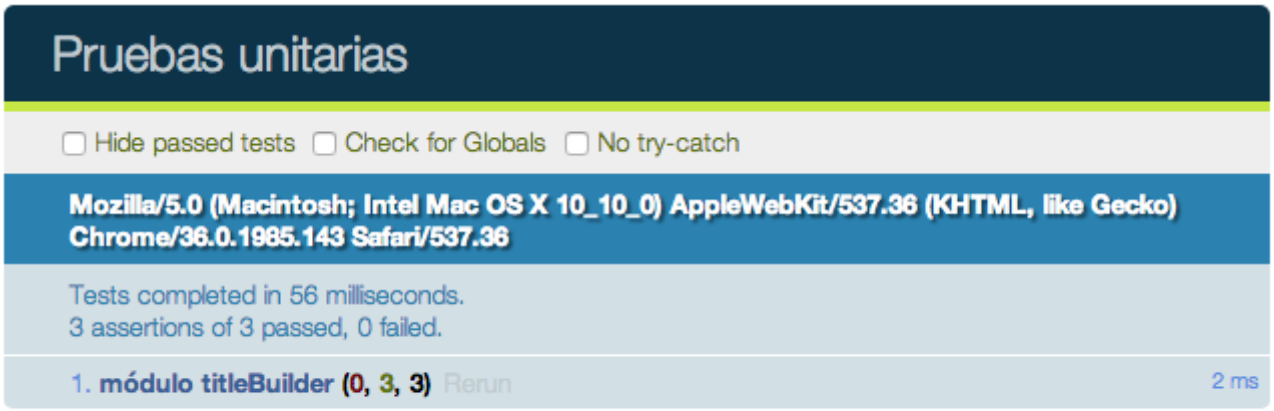
Pruebas con QUnit

Con este código tenemos la base necesaria para poder realizar pruebas unitarias con QUnit. Lo siguiente será pasar las validaciones que hicimos con `console.assert` a una prueba unitaria con QUnit:

```
QUnit.test('módulo titleBuilder', function(assert) {
  assert.ok(titleBuilder.toString() === 'La Buena Espina', 'El título
por defecto debe ser "La Buena Espina"');

  titleBuilder.addPart('Carta');
  titleBuilder.addPart('Pescados');
  titleBuilder.addPart('Ceviches');
```

```
  assert.ok(titleBuilder.toString() === 'La Buena Espina › Carta ›
Pescados › Ceviches', 'El título ahora debe ser "La Buena Espina ›
Carta › Pescados › Ceviches"');

  titleBuilder.reset();
  titleBuilder.addPart('Locales');

  assert.ok(titleBuilder.toString() === 'La Buena Espina – Locales',
'El título ahora debe ser "La Buena Espina – Locales"');
});
```

En este caso `assert.ok` toma los mismos valores que `console.assert` (una condición que debe evaluarse como verdadera y la descripción de la validación). Las 3 validaciones o *asserts* son agrupadas en una *prueba* o *test*, definida por el método `QUnit.test`. Al final debe quedar así:



Jasmine

QUnit nos permite realizar pruebas unitarias utilizando un lenguaje un tanto *técnico*, lo que nos permite crear pruebas para un módulo, una función constructora o algún caso similar. Pero, ¿cómo podríamos crear pruebas que sean más legibles? Es aquí donde entra el concepto de Behavior-Driven Development, o BDD.

Behavior-Driven Development es un modo de realizar pruebas donde estas se enfocan en función al *comportamiento* de lo que se va a probar (por ejemplo, qué debería hacer un módulo o una función), y no a, verificar que el código probado devuelva un valor en específico.

Jasmine es una biblioteca que permite realizar pruebas unitarias utilizando BDD, lo que nos da la opción de crear pruebas más interesantes que solo hacer *El título por defecto debe ser "La Buena Espina"*. Así mismo, nos da métodos para realizar validaciones en un lenguaje más natural y no tan técnico, como verificar si un número es mayor o menor que otro, si una cadena es parte de otra cadena, si algún

valor puede ser **considerado** como `true` o `false` (denominados *truthy* o *falsy*, respectivamente), entre otros.

Type coercion y valores *truthy* y *falsy*

Cuando comparamos un valor dentro de una condicional pueden pasar dos cosas: O el valor que se compara es un booleano (es `true` o `false`), o no lo es. Si es booleano, la condicional se ejecuta directamente:

```
if (10 + 15 === 25) {
  console.log('10 + 15 es igual a 25');
}

// 10 + 15 es igual a 25
```

Pero si el valor que se compara no es un booleano, ocurre un *type coercion*, o conversión implícita. JavaScript es un lenguaje con tipado dinámico, lo que significa que una variable, o propiedad, pueden tener cualquier tipo de valor, sin necesidad de hacer una conversión explícita, o *casting*. Esto puede ocurrir en dos casos:

1. Al usar `==`, o `!=`:

```
if (10 + 15 == '25') {
  console.log('10 + 15 es igual a 25, aunque sea una cadena');
}

// 10 + 15 es igual a 25, aunque sea una cadena
```

En este caso, `==` (y `!=`) compara valores y no tipos de datos (es decir: `10 + 15` es igual a 25`, y` '25' `tiene el mismo valor que` 25`).

1. O al pasar un valor a una condicional:

```
if (10) {
  console.log('10 es convertido implícitamente a true');
}

// 10 es convertido implícitamente a true
```

En JavaScript, un número diferente a `0` es *igual* a `true`, mientras que `0` es igual a `false`. De igual forma, una cadena vacía es *igual* a `false`, mientras que, si tuviera algún caracter (incluyendo espacios), sería *igual* a `true`.

Cada vez que nos referimos a que un valor es *igual* a `true` (así, en cursiva), decimos que ese valor es *truthy*. Por otro lado, si decimos que un valor es *igual* a `false` (de nuevo, en cursiva), decimos que ese valor es *falsy*.

Algunos ejemplos más sobre *type coercion* se pueden encontrar en este link.

Para poder usar Jasmine debemos descargarlo desde la cuenta del proyecto en GitHub. En este caso, trabajaremos con la versión 2.0.2. El zip descargado contiene una estructura de archivos y carpetas que utiliza Jasmine:

- Carpeta `lib`: Aquí se encuentran todos los archivos que componen Jasmine, incluyendo hojas de estilo y la biblioteca en sí.
- Carpeta `spec`: En esta carpeta deben estar todas las pruebas que haremos a nuestro código.
- Archivo `SpecRunner.html`: Este archivo servirá de reporte para las pruebas que haremos. Similar a la página que armamos para QUnit.
- Carpeta `src`: Aquí debería ir el código que queremos probar.

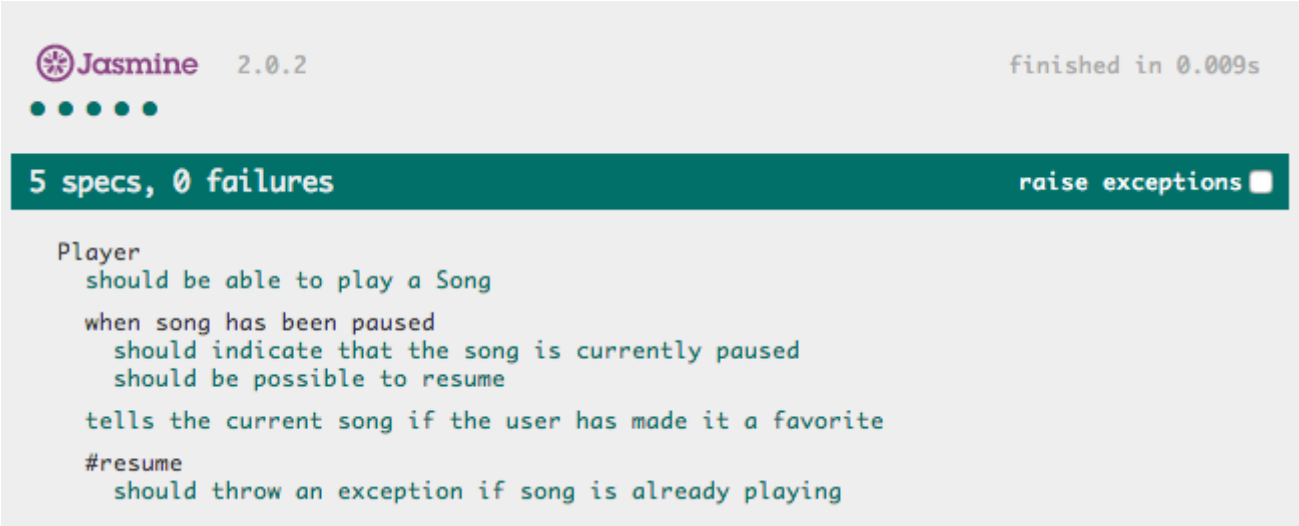
Por defecto, Jasmine viene con código de ejemplo dentro de las carpetas `spec` y `src`, e indicando el orden en el que debe ir nuestro código en `SpecRunner.html` (en este caso, usando el ejemplo que el mismo Jasmine nos da):

```
<!DOCTYPE HTML>
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  <title>Jasmine Spec Runner v2.0.2</title>

  <link rel="shortcut icon" type="image/png" href="lib/jasmine-
2.0.2/jasmine_favicon.png">
  <link rel="stylesheet" type="text/css" href="lib/jasmine-
2.0.2/jasmine.css">

  <script type="text/javascript" src="lib/jasmine-
2.0.2/jasmine.js"></script>
  <script type="text/javascript" src="lib/jasmine-2.0.2/jasmine-
html.js"></script>
  <script type="text/javascript" src="lib/jasmine-
2.0.2/boot.js"></script>

  <!-- include source files here... -->
  <script type="text/javascript" src="src/Player.js"></script>
  <script type="text/javascript" src="src/Song.js"></script>

  <!-- include spec files here... -->
  <script type="text/javascript" src="spec/SpecHelper.js"></script>
  <script type="text/javascript" src="spec/PlayerSpec.js"></script>

</head>

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

El *Spec Runner* de Jasmine carga los siguientes archivos:

- `jasmine.js`: La biblioteca que contiene el código de Jasmine
- `jasmine-html.js`: Contiene el código necesario para mostrar los resultados en forma de HTML, dentro del *Spec Runner*.
- `boot.js`: Este archivo se encargar de cargar todo el entorno de pruebas de Jasmine y activar el reporte en HTML (`jasmine-html.js`)

Luego de esto, se debe cargar el código que deseamos probar (los que se encuentran en la carpeta `src`), y luego las pruebas en sí (carpeta `spec`). Al final debe quedar así:



Pruebas con Jasmine

Jasmine tiene una forma de organizar las pruebas, según el enfoque de BDD. De acuerdo a Jasmine, el código debe ser legible como si fuera un texto *en inglés*. Esto se logra utilizando ciertos métodos como `describe` e `it`:

```
describe("Player", function() {
  it("should be able to play a Song", function() {});

  describe("when song has been paused", function() {
    it("should indicate that the song is currently paused", function()
{});

    it("should be possible to resume", function() {});
  });

  it("tells the current song if the user has made it a favorite",
function() {});

  describe("#resume", function() {
    it("should throw an exception if song is already playing",
function() {});
  });
```

```
});
```

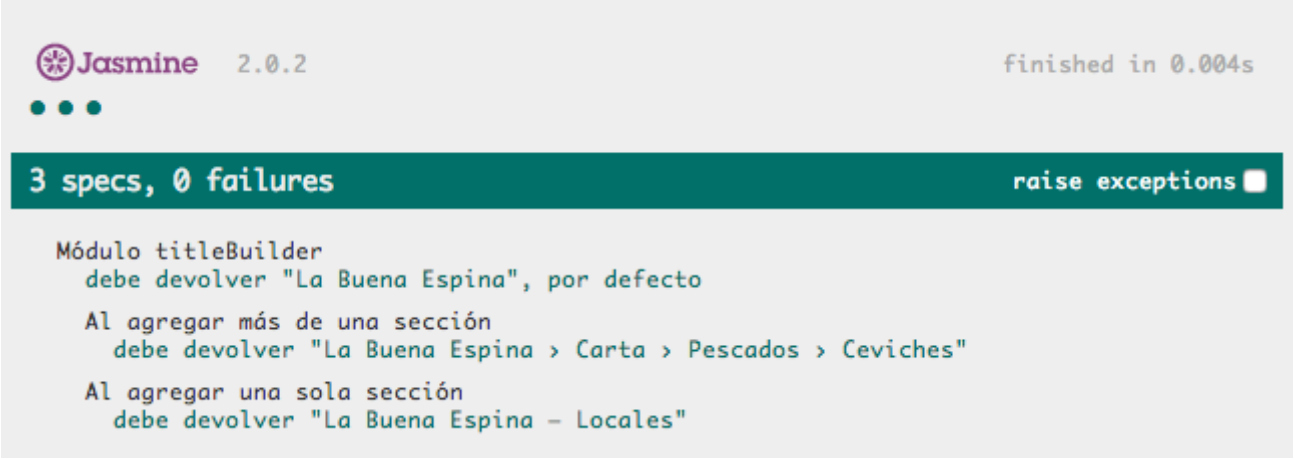
En Jasmine, cada método `describe` crea una suite de pruebas (una suite de pruebas es un conjunto de pruebas), y cada método `it` permite definir una prueba (aquí son llamados *specs*). Según la imagen de arriba, el código anterior se debería leer así:

```
Player should be able to play a Song

Player, when song has been paused, should indicate that the song is
currently paused
Player, when song has been paused, should be possible to resume

Player tells the current song if the user has made it a favorite

Player#resume should throw an exception if song is already playing
```

Entonces, nuestras pruebas deben ser escritas de manera similar:

```
describe('Módulo titleBuilder', function() {
  beforeEach(function() {
    titleBuilder.reset();
  });

  it('debe devolver "La Buena Espina", por defecto', function() {
    expect(titleBuilder.toString()).toEqual('La Buena Espina');
  });

  describe('Al agregar más de una sección', function() {
    it('debe devolver "La Buena Espina › Carta › Pescados ›
Ceviches"', function() {
      titleBuilder.addPart('Carta');
      titleBuilder.addPart('Pescados');
      titleBuilder.addPart('Ceviches');

      expect(titleBuilder.toString()).toEqual('La Buena Espina › Carta
› Pescados › Ceviches');
    });
  });

  describe('Al agregar una sola sección', function() {
    it('debe devolver "La Buena Espina – Locales"', function() {
      titleBuilder.addPart('Locales');

      expect(titleBuilder.toString()).toEqual('La Buena Espina –
Locales');
    });
  });
});
```

Y el resultado sería el siguiente:



En Jasmine, cada prueba (definida con el método `it`) puede tener una o más validaciones o *asserts*, que, en este caso, son definidas con el método `expect`.

El método `expect` permite utilizar lo que en Jasmine se llaman *matchers*. Estos *matchers* permiten validar a un nivel más complejo que simplemente comparar dos valores con `===` o `!==`. Los *matchers* que vienen por defecto son:

- `toBe`: Igual a utilizar `===`.
- `toEqual`: Similar a `toBe` pero permite comparar objetos literales.
- `toMatch`: Compara cadenas con expresiones regulares o con otras cadenas.
- `toBeDefined`: Valida si una propiedad está definida (que no sea `undefined`)
- `toBeUndefined`: Lo opuesto a `toBeDefined`
- `toBeNull`: Valida si una variable o propiedad tiene valor `null`.
- `toBeTruthy`: Permite saber si un valor es *truthy*.
- `toBeFalsy`: Permite saber si un valor es *falsy*.
- `toContain`: Valida si un elemento está dentro de un array.
- `toBeLessThan`: Valida si un número es menor a otro.
- `toBeGreaterThan`: Valida si un número es mayor a otro.
- `toBeCloseTo`: Valida si un número decimal es cercano a un número entero.
- `toThrow`: Valida si una función lanzará una excepción al ser ejecutada.

Así mismo, el valor devuelto por `expect` tiene una propiedad llamada `not`, el cual permite invertir el valor de cada matcher (recordemos que, a fin de cuentas, una validación debe devolver `true` o `false`).

Capítulo 6: Peticiones asíncronas

Junto al DOM, las peticiones asíncronas son las características más utilizadas en un sitio o aplicación web, y permiten disminuir la carga que contiene una llamada al servidor, dando la impresión de tener un sitio mucho más rápido.

Una petición asíncrona es una operación que, mientras esté siendo procesada, deja libre al navegador para que pueda hacer otras operaciones. Llamaremos peticiones asíncronas a las operaciones que tengan que ver con realizadas llamadas a servidores; sin embargo, existen muchas más operaciones asíncronas en JavaScript, como las que se realizan para leer y escribir en archivos, obtener la geolocalización de un navegador, o manejar base de datos.

Las peticiones asíncronas en el navegador se realizan con la función `XMLHttpRequest`, la cual permite realizar peticiones de tipo `GET` (obtener información), `POST` (enviar información), y otros más.

XMLHttpRequest

Para poder enviar una petición asíncrona a un servidor se debe crear una instancia de la función `XMLHttpRequest`, de la siguiente manera:

```
var xhr = new XMLHttpRequest();
```

Luego, debemos definir la dirección a donde se enviará la petición, e indicar el tipo de petición (`GET`, `POST`, etc). El último parámetro es importante: es el que define si la petición será asíncrona o no. Si la petición es síncrona, se corre el riesgo de congelar el navegador, ya que este dejará de hacer cualquier operación y se dedicará a realizar la petición síncrona.

```
xhr.open('GET', url, true);
```

Por último, enviamos la petición al servidor con el método `send`. En este momento el navegador continúa ejecutando el código que está después de esta línea, mientras que, por interno, la petición es esperada.

```
xhr.send();
```

Hasta este punto, el proceso está incompleto: Enviamos la petición pero no sabemos en qué momento ha terminado de procesarse, ni cuál es la información que el servidor ha devuelto.

Felizmente, `XMLHttpRequest` *hereda* de `EventTarget`. Recordemos qué hacía `EventTarget`:

Todos los elementos del DOM, además de `window`, heredan de la interfaz `EventTarget`, el cual permite enlazar eventos a callbacks definidos dentro

de la aplicación. La interfaz `EventTarget` *tiene 3 métodos:* `addEventListener`, `removeEventListener` *y* `dispatchEvent`.

Las peticiones asíncronas tienen sus propios eventos:

- `abort`: Lanzado cuando la petición ha sido cancelada, vía el método `abort()`.
- `error`: Lanzado cuando la petición ha fallado.
- `load`: Lanzado cuando la petición ha sido completada satisfactoriamente.
- `loadend`: Lanzado cuando la petición ha sido completada, ya sea con éxito o con error.
- `loadstart`: Lanzado cuando la petición ha sido iniciada.
- `progress`: Lanzado cuando la petición esté enviando o recibiendo información.
- `readystatechange`: Lanzado cuando el atributo `readyState` cambie de valor.
- `timeout`: Lanzado cuando la petición ha sobrepasado el tiempo de espera límite (definido por la propiedad `timeout`).

Así que debemos escuchar al menos un evento para saber si la petición devuelve algún tipo de información. En este caso escucharemos 2 eventos importantes: `error` para saber si hubo un error en la petición, y `readystatechange` para saber los distintos estados de la petición:

```
xhr.addEventListener('error', function(e) {
  console.log('Un error ocurrió', e);
});

xhr.addEventListener('readystatechange', function() {
  console.log('xhr.readyState:', xhr.readyState);
});
```

Juntando cada parte, tenemos el siguiente código, el cual obtiene los últimos *tweets* que contengan la palabra *ceviche*:

```
var xhr = new XMLHttpRequest();

var url = 'http://coffeemaker.herokuapp.com/twitter.json?q=ceviche';

xhr.open('GET', url, true);

xhr.addEventListener('error', function(e) {
  console.log('Un error ocurrió', e);
});

xhr.addEventListener('readystatechange', function() {
  console.log('xhr.readyState:', xhr.readyState);
});

xhr.send();
```

La ejecución de este código daría el siguiente resultado en la consola:

```
// xhr.readyState: 2
// xhr.readyState: 3
// xhr.readyState: 4
```

La propiedad `readyState` indica el estado de la petición y tiene los siguientes valores:

- `0`: El valor inicial.
- `1`: Luego de haber ejecutado el método `open()`.
- `2`: El navegador envió la petición (método `send()`) pero aún no recibe una respuesta.
- `3`: El navegador está esperando por la respuesta a la petición.
- `4`: La petición obtiene información de respuesta.

Ahora ya sabemos los estados por los que pasa una petición, pero aún no sabemos cuál es la respuesta. Para obtenerla utilizamos la propiedad `responseText`.

```
xhr.addEventListener('readystatechange', function() {
  if (xhr.readyState === 4) {
    console.log(xhr.responseText);
  }
});
```

En este caso, verificamos que el `readyState` sea 4, dado que la petición solo tendrá una respuesta cuando tenga dicho estado.

Peticiones POST

Las peticiones asíncronas son, generalmente, peticiones GET, por lo que si se envían valores en la petición, estos estarán expuestos fácilmente en la URL de la misma petición, creando un potencial problema de seguridad. Así mismo, las URLs tienen un límite de caracteres, por lo que no se podrá enviar toda la información que uno desee. Estos dos puntos son cruciales al momento de realizar peticiones, asíncronas o no. Es aquí donde aparecen las peticiones POST: peticiones que pueden enviar gran cantidad de información, la cual no es accesible de forma fácil.

En el caso de `XMLHttpRequest`, crear una petición POST es sencillo y agrega dos pasos a lo descrito anteriormente: indicar el tipo de petición y agregar los valores que se deseen ingresar.

Para indicar el tipo de petición simplemente cambiamos el primer parámetro del método `open()`:

```
xhr.open('POST', url, true);
```

Cabe notar que la url debe aceptar peticiones POST, lo cual es definido en el servidor.

Para agregar los valores que se desean enviar se utiliza una instancia de `FormData`, donde se agregan los valores utilizando el método `append()`:

```
var data = new FormData();

data.append('nombre', 'valor');
```

Luego, el nuevo objeto `FormData` debe ser pasado como parámetro en el método `send()` de la instancia de `XMLHttpRequest`:

```
xhr.send(data);
```

Así, el código final quedaría de esta forma:

```
var xhr = new XMLHttpRequest();

var url = 'http://coffeemaker.herokuapp.com/form';

xhr.open('POST', url, true);

xhr.addEventListener('error', function(e) {
  console.log('Un error ocurrió', e);
});

xhr.addEventListener('readystatechange', function() {
  if (xhr.readyState === 4) {
    console.log(xhr.responseText);
  }
});

var data = new FormData();

data.append('nombre', 'valor');

xhr.send(data);
```

Y el resultado de la petición sería:

```
// {"nombre":"valor"}
```

Carga asíncrona de archivos

Una de las características de `FormData` es que no solo permite adjuntar texto, si no también archivos. Esto se logra agregando instancias de `File` con el método `append`. Recordemos que cada campo de formulario de tipo archivo (`<input type="file">`) tiene una propiedad llamada `files`, el cual contiene una lista de instancias `File`. De esta forma, podemos subir archivos a un servidor de manera **asíncrona**.

La ventaja de `FormData` es que, al crear una instancia, podemos pasarle como parámetro un elemento formulario (`<form>`), por lo que automáticamente toma todos los campos del formulario, siempre y cuando tengan un nombre (atributo `name`), incluyendo los campos de tipo archivo.

```
var form = document.querySelector('#formulario_comentario');
var data = new FormData(form);
```

JSON

En los ejemplos donde se utiliza `coffeemaker.herokuapp.com` vemos que las respuestas vienen en forma de texto, pero con un formato que nos recuerda a objetos u arreglos en JavaScript. Este formato se llama JSON (JavaScript Object Notation), y permite enviar y recibir información de una manera simple y liviana.

Para poder leer este formato utilizamos el método `JSON.parse`, el cual es nativo en todos los navegadores, y en Internet Explorer 9 y superiores:

```
JSON.parse('{"nombre":"valor"}');
// Object {nombre: "valor"}
```

En el caso opuesto, si deseamos convertir un objeto a una cadena en formato JSON (por ejemplo, si deseamos guardarlo en `localStorage`), utilizamos el método `JSON.stringify`, el cual convertirá un objeto a su contraparte en JSON.

```
JSON.stringify({nombre: 'valor'});
// "{"nombre":"valor"}"
```

Este método no funciona en casos donde un objeto o un arreglo contiene una referencia a sí mismo:

```
var a = [];

a.push(a);

JSON.stringify(a);
// Uncaught TypeError: Converting circular structure to JSON

JSON.stringify(window);
// Uncaught TypeError: Converting circular structure to JSON
```

En el caso de `window`, este tiene propiedades como `top`, `parent` o `self` que son referencias a sí mismos. `JSON.stringify` recorre todo el arreglo u objeto que se desea convertir a formato JSON y, de encontrar una referencia al mismo objeto, falla al tratar de convertir una estructura que se referencia a sí misma en algún punto.

Simplificando las peticiones asíncronas con `xhr.js`

Manejar peticiones asíncronas puede ser un tanto tedioso. Por ejemplo, para realizar una petición GET sencilla se debe escribir el siguiente código:

```
var xhr = new XMLHttpRequest();

var url = 'http://coffeemaker.herokuapp.com/twitter.json?q=ceviche';

xhr.open('GET', url, true);

xhr.addEventListener('error', function(e) {
  console.log('Un error ocurrió', e);
});

xhr.addEventListener('readystatechange', function() {
  console.log('xhr.readyState:', xhr.responseText);
});

xhr.send();
```

Y si queremos realizar una petición POST:

```
var xhr = new XMLHttpRequest();

var url = 'http://coffeemaker.herokuapp.com/form';

xhr.open('POST', url, true);

xhr.addEventListener('error', function(e) {
  console.log('Un error ocurrió', e);
});

xhr.addEventListener('readystatechange', function() {
  console.log('xhr.readyState:', xhr.responseText);
});

var data = new FormData();
data.append('nombre', 'valor');

xhr.send(data);
```

Queremos evitar tener que escribir tanto código, así que crearemos una biblioteca, similar a `dom.js` (ver Capítulo 3), que permita manejar peticiones asíncronas en menos líneas.

Empecemos por lo básico, creando una función llamada `xhr`:

```
function xhr(options) {
  var xhRequest = new XMLHttpRequest();
```

```
  var url = options.url;

  xhRequest.open(options.method, url, true);

  xhRequest.send();

  return xhRequest;
}
```

Hasta ahora, lo único que hicimos fue encapsular el cuerpo de una petición asíncrona en una función, que será llamada de la siguiente forma:

```
var request = xhr({
  url: 'http://coffeemaker.herokuapp.com/twitter.json?q=ceviche',
  method: 'GET'
});
```

Hasta aquí no tenemos control de los eventos que pueda lanzar la variable `request`, así que necesitamos agregar soporte para ello:

```
function xhr(options) {
  var xhRequest = new XMLHttpRequest();

  var url = options.url;

  xhRequest.open(options.method, url, true);

  xhRequest.addEventListener('error', options.onError);
  xhRequest.addEventListener('readystatechange',
options.onReadyStateChange);

  xhRequest.send();

  return xhRequest;
}
```

Y lo usamos de la siguiente forma:

```
var request = xhr({
  url: 'http://coffeemaker.herokuapp.com/twitter.json?q=ceviche',
  method: 'GET',
  onError: function(e) {
    console.log('Un error ocurrió', e);
  },
  onReadyStateChange: function() {
    console.log('xhr.readyState:', this.readyState);
  }
});
```

Ya tenemos una biblioteca que cumple con su trabajo, pero se puede mejorar. Por ejemplo, ¿qué pasaría si el método `onReadyStateChange` crece más?

Recordemos que la meta de toda aplicación es mantenerla simple. Una de las formas de convertir algo complejo en simple es dividirlo en pequeñas partes (como vimos en el ejemplo de módulos); y en este caso necesitamos dividir la función `xhr` en dos partes: la petición por un lado, y los eventos que maneja por otro.

Y al separar la petición de los eventos nos ataca otra duda: ¿Y si necesitamos más de un método `onReadyStateChange`? Si el método `onReadyStateChange` crece, deberíamos poder dividirlo en pequeños métodos `onReadyStateChange`. ¿Cómo solucionamos estos dos problemas?

Promises

Una *promesa*, o *promise*, es un objeto con el que se puede trabajar sin necesidad de saber su valor, ya que este se sabrá en *el futuro* (de ahí el nombre). ¿Y cómo funciona? En términos simples, guarda callbacks que van a trabajar con el valor a futuro, los cuales se ejecutarán, en el orden en el que fueron agregados, inmediatamente después de que la promesa obtenga un valor.

Debemos tener en cuenta que los *callbacks* pueden ser ejecutadas tanto si la promesa ha sido cumplida o rechazada. Una promesa es un contenedor de una operación que devuelve un valor a futuro, como las **peticiones asíncronas**. Si la petición asíncrona falla, la promesa es **rechazada**; pero, por el contrario, si la petición asíncrona ha devuelto un valor, la promesa es **cumplida**. Para ambos casos se pueden definir *callbacks* diferentes.

Para crear una promesa, debemos usar el constructor `Promise`:

```
var promise = new Promise(function(resolve, reject) {
  //
});
```

El constructor `Promise` toma como único parámetro una función, la cual a su vez toma dos parámetros, que también son funciones:

```
var promise = new Promise(function(resolve, reject) {
  if (1 == '1') {
    resolve(1);
  }
  else {
    reject('Esta promesa nunca será rechazada');
  }
});
```

Por su parte, cada instancia de `Promise` tiene dos métodos: `then` y `catch`. Ambos métodos permiten guardar los *callbacks* que se ejecutarán cuando la promesa devuelva un valor: mientras que `then` toma dos valores (un *callback* para la promesa cumplida y otro para la promesa rechazada), `catch` solo permite guardar *callbacks* que se ejecutarán si la promesa es rechazada:

```
promise.then(function(value) {
  console.log('Promesa cumplida.', value);
}, function(error) {
  console.log('Promesa rechazada.', error);
});

promise.catch(function(error) {
  console.log('Esto tampoco se ejecutará')
});

// Promesa cumplida. 1
```

En este caso la promesa se evaluará de inmediato, ya que no existe una operación asíncrona. Para ver su funcionamiento real, crearemos una petición asíncrona dentro de la promesa:

```
var url = 'http://coffeemaker.herokuapp.com/twitter.json?q=ceviche';

var method = 'GET';

var xhRequest = new XMLHttpRequest();

xhRequest.open(method, url, true);

var promise = new Promise(function(resolve, reject) {
  xhRequest.addEventListener('readystatechange', function() {
    if (xhRequest.readyState === 4) {
      resolve(xhRequest);
    }
  });

  xhRequest.addEventListener('error', function() {
    reject(xhRequest);
  });
});

xhRequest.send();
```

Y se usa de la siguiente manera:

```
promise.then(function(request) {
  console.log('Promesa cumplida.', request);
}, function(request) {
  console.log('Promesa rechazada.', request);
});

// Promesa cumplida. XMLHttpRequest {...}
```

Al final, la función `xhr` quedaría así:

```
function xhr(options) {
  var xhRequest = new XMLHttpRequest();
```

```
  var url = options.url;

  xhRequest.open(options.method, url, true);

  var promise = new Promise(function(resolve, reject) {
    xhRequest.addEventListener('readystatechange', function() {
      if (xhRequest.readyState === 4) {
        resolve(xhRequest);
      }
    });

    xhRequest.addEventListener('error', function() {
      reject(xhRequest);
    });
  });

  xhRequest.send();

  return promise;
}
```

Y se usaría de la siguiente forma:

```
var request = xhr({
  url: 'http://coffeemaker.herokuapp.com/twitter.json?q=ceviche',
  method: 'GET'
});

request.then(function(xhRequest) {
  console.log('Estado: ', xhRequest.status);
});

request.then(function(xhRequest) {
  console.log('Resultado: ', JSON.parse(xhRequest.responseText));
});

// Estado:  200
// Resultado:  [Object, Object, Object, Object, Object, Object,
Object, Object, Object, Object, Object, Object, Object, Object,
Object]
```

Como se ve en el código, se pueden añadir varios callbacks con el método `then`, y estos se ejecutan en el orden en el que fueron agregados. Otra de las características de las promesas es que, tanto `then` como `catch`, devuelven una promesa nueva, lo que permite **encadenar promesas**: Cada método `then` toma el valor de la promesa anterior:

```
var request = xhr({
  url: 'http://coffeemaker.herokuapp.com/twitter.json?q=ceviche',
  method: 'GET'
});

request.then(function(xhRequest) {
```

```
  var newPromiseValue = JSON.parse(xhRequest.responseText);

  console.log(newPromiseValue.length + ' elementos');

  return newPromiseValue;
}).then(function(value) {
  // Aquí value es un arreglo
  var newPromiseValue = value[0];

  console.log('Primer elemento: ', newPromiseValue);

  return newPromiseValue;
}).then(function(value) {
  // Y aquí value es un objeto
  var newPromiseValue = value.id;
  console.log('ID del primer elemento: ', newPromiseValue);
});

// 15 elementos
// Primer elemento:  Object {...}
// ID del primer elemento:  530216282797264900
```

Por otro lado, es posible que en alguna promesa de la cadena ocurra un error, por lo que es importante manejar callbacks de error, ya sea como segundo parámetro de `then`, o utilizando el mètodo `catch`:

```
var request = xhr({
  url: 'http://coffeemaker.herokuapp.com/twitter.json?q=ceviche',
  method: 'GET'
});

request.then(function(xhRequest) {
  var newPromiseValue = JSON.parse(xhRequest.responseText);

  conosle.log(newPromiseValue.length + ' elementos'); // ojo aquí

  return newPromiseValue;
}).then(function(value) {
  // Aquí value es un arreglo
  var newPromiseValue = value[0];

  console.log('Primer elemento: ', newPromiseValue);

  return newPromiseValue;
}).then(function(value) {
  // Y aquí value es un objeto
  var newPromiseValue = value.id;
  console.log('ID del primer elemento: ', newPromiseValue);
}).catch(function(error) {
  console.log('Error', error);
});
```

```
// Error ReferenceError: conosle is not defined {stack: (...),
message: "conosle is not defined"}
```

De esta forma, podemos manejar las peticiones asíncronas de una manera más flexible, separando la petición de las funciones que trabajan con su resultado, así como manejar errores de una forma mucho más simple.

El constructor `Promise` es soportado por todos los navegadores actuales, excepto por Internet Explorer.

Capítulo 7: jQuery

Si bien ya sabemos cómo manejar el DOM, necesitamos que nuestro sitio web funcione bien en diferentes navegadores por igual. jQuery está diseñado desde sus inicios para dar soporte al manejo del DOM en todos los navegadores conocidos, simplificando drásticamente el desarrollo de un sitio web, resolviendo uno de los más grandes problemas en el desarrollo web: El código *cross-browser*. Hace muchos años, se tenía que crear dos versiones del mismo código: una para Netscape y otra para Internet Explorer. Cuando Netscape desapareció y apareció Firefox, se dio el mismo caso, una vez más con Internet Explorer del otro lado. Si a eso le sumamos otros navegadores, como Opera o Safari (para Mac OS), el código crece rápidamente.

jQuery ofrece una serie de métodos para manipular el DOM, manejar eventos y realizar llamadas asíncronas, de tal forma que todo funcione de la misma manera en todos los navegadores.

Para utilizar jQuery en un sitio web debemos ir a la sección *Download* y elegir una de las versiones que ofrece jQuery. Cabe resaltar que jQuery está dando soporte a dos versiones: la 1.x y la 2.x. La diferencia entre ambas es que la 2.x ya no tiene soporte para Internet Explorer 6, 7 y 8 (haciendo que la biblioteca pese bastante menos); así que elegir entre una y otra versión depende del soporte que quieras para tu sitio o aplicación.

En este caso, elegimos la versión 1.11.1 en su versión para desarrollo:

```
<!DOCTYPE html>
<html>
<head>
  <title></title>
</head>
<body>
  <script src="http://code.jquery.com/jquery-1.11.1.js"></script>
</body>
</html>
```

Para cargar un archivo JavaScript utilizamos la etiqueta `<script>`, poniendo la dirección del archivo en el atributo `src`. En algunos casos la etiqueta `<script>` estará dentro de la etiqueta `<head>`, pero en otros casos puede estar al final de la etiqueta `<body>`. ¿Por qué pasa esto?

Los navegadores leen un documento HTML y lo muestran de forma visual en un proceso que es llamado **renderizado**. En este proceso, que puede variar un poco entre navegadores, el navegador lee el documento HTML y lo va interpretando progresivamente, lo cual quiere decir que, por cada parte que lee, verifica si existe algún recurso que debe ser cargado (puede ser una imagen, un iframe, una hoja de estilos o un archivo JavaScript).

Este proceso va de inicio a fin, por lo que, si las etiquetas `<script>` se ponen dentro de la etiqueta `<head>`, el navegador va a esperar a que terminen de cargar los archivos JavaScript para seguir leyendo el resto del documento. Esto puede ser contraproducente en la mayoría de casos, por lo que se recomienda poner las etiquetas `<script>` al final de la etiqueta `<body>`, de esta forma todo el documento cargará y se mostrará en la pantalla de una forma más rápida.

---

jQuery tiene una función del mismo nombre, pero se utiliza comúnmente un alias: `$`. Esta función acepta diferentes parámetros:

- Un selector (por ejemplo: `body`, o `#elemento_1`). Puede aceptar como segundo parámetro un nodo elemento de *contexto*, para limitar la búsqueda del selector.
- Una cadena conteniendo HTML (por ejemplo: `<p></p>`), para crear nodos elementos de una manera más rápida. Puede aceptar un segundo parámetro, el cual servirá como nodo documento, que es donde se agregará el o los elementos a crear.
- Un elemento, un arreglo o una lista de nodos elementos (por ejemplo: `document.querySelectorAll('a')`).

Usando la función `$` con cualquiera de los 3 parámetros se devuelve un objeto instancia de jQuery. Esta instancia es parecida a un arreglo, y tiene diferentes métodos para manejar sus elementos.

Adicionalmente, `$` puede aceptar una función, la cual se ejecutará cuando todo el documento ha terminado de cargar.

Selectores

jQuery permite obtener los elementos del DOM mediante selectores, de la misma forma como lo hace el método `querySelectorAll`, con la diferencia que también acepta selectores propios:

Atributos

- `[name!="value"]`: Devuelve todos los elementos cuyo atributo de nombre `name` **no** tiene el valor `value`

Básico

- `:animated`: Devuelve los elementos que están siendo animados en ese instante.
- `:eq(index)`: Devuelve el elemento que se encuentra en el índice seleccionado, dentro de un conjunto de elementos.
- `:even`: Devuelve los elementos cuyos índices sean pares, teniendo en cuenta que el índice empieza en `0`, por lo que selecciona los elementos en los índices `0`, `2`, `4` y sucesivos.

- `:first`: Devuelve el primer elemento de un conjunto de elementos.
- `:gt(index)`: Devuelve los elementos cuyos índices sean mayores al índice seleccionado.
- `:header`: Devuelve todos los elementos que sean `h1`, `h2`, `h3` y similares.
- `:last`: Devuelve el último elemento de un conjunto de elementos.
- `:lt(index)`: Devuelve los elementos cuyos índices sean menores al índice seleccionado.
- `:odd`: Devuelve los elementos cuyos índices sean impares, teniendo en cuenta que el índice empieza en `0`, por lo que selecciona los elementos en los índices `1`, `3`, `5` y sucesivos.

Contenido

- `:has(selector)`: Devuelve todos los elementos que contienen los elementos definidos en el segundo selector.
- `:parent`: Devuelve todos los elementos que tienen al menos un nodo hijo (ya sea elemento o no).

Formularios

- `:button`: Devuelve los elementos que sean botones, ya sean elementos `<button>` o `<input type="button">`
- `:checkbox`: Devuelve todos los elementos que son `<input type="checkbox">`
- `:file`: Devuelve todos los elementos que son `<input type="file">`
- `:image`: Devuelve todos los elementos que son `<input type="image">`
- `:input`: Devuelve todos los elementos que son `<input>`, `<textarea>`, `<select>` y `<button>`
- `:password`: Devuelve todos los elementos que son `<input type="password">`
- `:radio`: Devuelve todos los elementos que son `<input type="radio">`
- `:reset`: Devuelve todos los elementos que son `<input type="reset">`
- `:selected`: Devuelve el elemento `<option>` seleccionado para un elemento `<select>`
- `:submit`: Devuelve todos los elementos que son `<input type="submit">`
- `:text`: Devuelve todos los elementos que son `<input type="text">`

Visibilidad

- `:hidden`: Devuelve todos los elementos ocultos, los cuales pueden ser: por tener `display: none` en sus estilos, ser elementos `<input type="hidden">`, tener `width` y `height` en 0, o si tiene algún elemento ancestro oculto.
- `:visible`: Devuelve todos los elementos que son visibles. En jQuery, un elemento es considerado visible si ocupa espacio en la pantalla, por lo que elementos con `visibility: hidden` u `opacity: 0` en sus estilos son considerados elementos visibles.

Si vemos el ejemplo usado en el capítulo 3, podremos cambiar el siguiente código:

```
var container = dom('#background');
```

```
container.delegate('transitionend', '.slide.current', function(e) {
  var current = dom(e.target),
      next = current.next();

  current.removeClass('current');

  if (current.isLastSibling()) {
    next = current.firstSibling();
  }

  next.addClass('current');
});
```

por:

```
var container = $('#background');
container.on('transitionend', '.slide.current', function(e) {
  var current = $(e.target),
      next = current.next();

  current.removeClass('current');

  if (current.is(':last-child')) {
    next = current.siblings().first();
  }

  next.addClass('current');
});
```

Cuando diseñamos `dom.js` tuvimos en mente algunos métodos que maneja jQuery (como `next`, `addClass` y `removeClass`), por lo que el código es bastante similar. Sin embargo, en jQuery no tenemos `isLastSibling` ni `firstSibling`.

En el primer caso, cambiamos `isLastSibling()` por `is(':last-child')`. `is` es un método que permite comparar entre el *set* de elementos seleccionado y un selector (el cual puede ser de CSS o uno de los descrito al inicio del capítulo). En el segundo caso, reemplazamos `firstSibling()` por `siblings().first()`, donde `siblings` es un método que devuelve todos los nodos *hermanos* del nodo seleccionado (pero **no incluye al nodo seleccionado en el resultado**), y `first`, que devuelve el primer elemento de un *set* de nodos en jQuery.

Eventos

jQuery permite manejar eventos, tanto del navegador como propios, utilizando los métodos `on` y `off` (para agregar y eliminar *listeners*, respectivamente). Estos métodos funcionan de la misma manera para eventos del navegador y propios, e incluso se pueden lanzar (o *disparar*) manualmente utilizando el método `trigger`.

Cabe recordar que jQuery agrega *listeners* a los eventos en la *bubbling phase*, y no en la *capture phase*. Esto es importante a tener en cuenta, dada la diferencia que existe entre agregar un *listener* en cualquiera de las dos fases.

Por otro lado, jQuery utiliza *event delegation*, el cual permite definir eventos en elementos que aún no han sido creados, así como definir el mismo evento a un conjunto de elementos, sin la necesidad de crear un *listener* por cada elemento.

Volvamos al ejemplo de la sección anterior:

```
var container = $('#background');
container.on('transitionend', '.slide.current', function(e) {
  var current = $(e.target),
      next = current.next();

  current.removeClass('current');

  if (current.is(':last-child')) {
    next = current.siblings().first();
  }

  next.addClass('current');
});
```

En este caso, seguimos usando *event delegation*. Sabemos que `$('#background')` devolverá un elemento, y que este, a su vez, tiene elementos hijo cuyas clases son `slide` y también serán `current`, y necesitamos agregar lanzar un evento `transitionend` para cada elemento `.slide.current` (es decir, el elemento `.slide` visible). En `dom.js` se llamaba `delegate`, pero en jQuery toma el nombre de `on`.

Ya sabiendo cómo usar jQuery, podemos terminar el sitio web de **La Buena Espina**. Agreguemos un evento `hashchange` a `window`:

```
$(window).on('hashchange', function(e) {
  $('.panel.current').removeClass('current');

  if (location.hash !== '') {
    $('.panel' + location.hash).addClass('current');
  }
});
```

De esta forma, cada vez que naveguemos por la barra de navegación, aparecerá el contenido correcto.

Sin embargo, ocurre un *bug* si recargamos la página y tenemos el hash `#historia` en la dirección: **El contenido de Historia no aparece**. Recordemos que al usar el evento `hashchange`, la función del *listener* solo se

ejecutará cuando el *hash* cambie, así que necesitamos agregar un evento más a `window`. jQuery permite agregar un listener a más de un evento utilizando `on` una sola vez:

```
$(window).on('hashchange load', function(e) {
  $('.panel.current').removeClass('current');

  if (location.hash !== '') {
    $('.panel' + location.hash).addClass('current');
  }
});
```

De esta forma, nuestro código se ejecutará tanto al cambiar el *hash* en la barra de direcciones, como al cargar la ventana. Podemos ver el código funcionando en http://cevichejs.com/files/7-jquery/index.html

---

Ajax

Además de manejar operaciones en el DOM, jQuery es capaz de manejar operaciones asíncronas. jQuery utiliza `XMLHttpRequest` o `ActiveXObject`, según sea el caso (por ejemplo, en versiones de Internet Explorer donde existe `ActiveXObject`, se utiliza este).

Para poder realizar operaciones asíncronas, jQuery ofrece una serie de métodos, los cuales van desde el básico `$.ajax` hasta `$.get` o `$.post`.

En el capítulo anterior vimos cómo realizar llamadas asíncronas a un servidor. Utilizamos `http://coffeemaker.herokuapp.com` para probar con el siguiente código:

```
var xhr = new XMLHttpRequest();

var url = 'http://coffeemaker.herokuapp.com/twitter.json?q=ceviche';

xhr.open('GET', url, true);

xhr.addEventListener('error', function(e) {
  console.log('Un error ocurrió', e);
});

xhr.addEventListener('readystatechange', function() {
  console.log('xhr.readyState:', xhr.readyState);
});

xhr.send();
```

Luego, creamos `xhr.js`, que simplificaba todo el código anterior a:

```
var request = xhr({
```

```
  url: 'http://coffeemaker.herokuapp.com/twitter.json?q=ceviche',
  method: 'GET'
});
```

Una de las ventajas de jQuery es que permite tomar el código anterior y convertirlo a una sola línea:

```
var request =
$.get('http://coffeemaker.herokuapp.com/twitter.json?q=ceviche');
```

Actualmente, jQuery tiene soporte para promesas, por lo que podemos usarlo de la siguiente forma:

```
var request =
$.get('http://coffeemaker.herokuapp.com/twitter.json?q=ceviche');

request.then(function(data) {
  console.log(data.length + ' elementos');

  return data;
}).then(function(data) {
  var newPromiseValue = data[0];

  console.log('Primer elemento: ', newPromiseValue);

  return newPromiseValue;
}).then(function(data) {
  var newPromiseValue = data.id;
  console.log('ID del primer elemento: ', newPromiseValue);
});
```

Por otro lado, una buena práctica sería separar la dirección de la cadena de búsqueda, y pasar los parámetros de búsqueda como un objeto:

```
var request = $.get('http://coffeemaker.herokuapp.com/twitter.json', {
q: 'ceviche' });
```

jQuery se encargará de generar la URL antes de enviar la petición, pero ganamos flexibilidad si deseamos cambiar la dirección de la petición.

En la primera parte vimos cómo mostrar las diferentes secciones del sitio web de **La Buena Espina**, excepto una: el formulario de contacto.

Para poder hacer funcionar el formulario de contacto usaremos `$.post`:

```
var contactForm = $('#contact-form'),
    contactName = $('#contacto_nombre'),
    contactMessage = $('#contacto_mensaje');

contactForm.on('submit', function(e) {
```

```
  e.preventDefault();

  window.localStorage.setItem('contact-form', contactMessage.val());

  var xhr = $.post('http://coffeemaker.herokuapp.com/form',
contactForm);

  xhr.then(function() {
    alert('¡Gracias por contactarnos!');
  });

  xhr.then(function() {
    contactMessage.val('');
    contactName.val('');
    window.localStorage.removeItem('contact-form');
  });
});
```

Una de las ventajas de `$.post` es que podemos pasarle un objeto jQuery, y este se **serializará** automáticamente, para obtener todos los valores de los elementos de formulario dentro del mismo objeto que tengan atributo `name`.

También utilizamos la API de *Web Storage* en este código. Recordemos el primer párrafo de esta API del navegador:

*Empecemos con una API simple de usar pero que soluciona un problema común al trabajar con una aplicación web: El dueño de **La Buena Espina** quiere un formulario de contacto para que los comensales puedan dar sus impresiones sobre el servicio y la comida. Pero, ¿qué pasaría si luego de enviar el formulario se pierde la conexión, el usuario cierra su navegador o el servidor no responde? Los comentarios no llegarán al dueño y se pueden perder buenas críticas con respecto al restaurante.*

De esta forma, nos aseguramos que el mensaje del usuario no se pierda si es que existe un error al momento de enviar el mensaje.

Plugins

Una de las ventajas de jQuery es la comunidad que tiene detrás, creada en buena parte gracias a los *plugins* que permite crear. Un *plugin* en jQuery es, básicamente, un método agregado al *prototype* de la función `jQuery` al cual se puede acceder mediante la propiedad `jQuery.fn` (o `$.fn`).

De nuevo, en el ejemplo de las dos primeras secciones, tenemos:

```
var container = $('#background');
container.on('transitionend', '.slide.current', function(e) {
  var current = $(e.target),
      next = current.next();
```

```
  current.removeClass('current');

  if (current.is(':last-child')) {
    next = current.siblings().first();
  }

  next.addClass('current');
});
```

En `dom.js` teníamos `isLastSibling` y `firstSibling`, pero en jQuery no. Sin embargo, podemos extender el *prototype* de jQuery y agregar estos métodos:

```
$.fn.isLastSibling = function() {
  return $(this).is(':last-child');
}

$.fn.firstSibling = function() {
  return $(this).siblings().first();
}
```

Y, de esta forma, tendríamos el siguiente código, más entendible:

```
var container = $('#background');
container.on('transitionend', '.slide.current', function(e) {
  var current = $(e.target),
      next = current.next();

  current.removeClass('current');

  if (current.isLastSibling()) {
    next = current.firstSibling();
  }

  next.addClass('current');
});
```

Capítulo 8: Mejorando el flujo de trabajo

Trabajar como desarrollador web frontend no solo implica saber HTML, CSS y JavaScript. También debemos conocer herramientas que aligeran el flujo de trabajo y reducen tiempo en tareas que llegan a ser repetitivas.

Grunt

En un flujo de trabajo común vamos a verificar que el código no tenga errores de sintáxis, realizar pruebas unitarias automatizadas, y minificar el código para reducir espacio, entre otras acciones. Realizar cada una de estas tareas puede tomar tiempo, y las vamos a realizar siempre cada cierto tiempo, sobre todo después de realizar un cambio fuerte en el código, así que es vital tener una herramienta que le delegue a la computadora este trabajo tedioso y aburrido. Grunt es un *task runner*, una herramienta que permite definir y realizar este tipo de tareas automatizadas.

Para utilizar Grunt necesitamos Node.js, una plataforma que permite ejecutar JavaScript fuera del navegador, el cual también instalará la herramienta de comandos `npm`. Luego de esto, es necesario instalar la herramienta `grunt-cli` utilizando el comando `npm`:

```
npm install -g grunt-cli
```

Con esto ya tenemos instalado el comando `grunt` en nuestra consola, el cual es necesario para ejecutar las tareas.

El siguiente paso es crear un archivo `package.json` en la raíz de la carpeta del proyecto. Este archivo es utilizado para definir los paquetes de NPM a utilizar en el proyecto, pero en este caso lo usaremos para trabajar con Grunt:

```
{
  "name": "buena-espina",
  "version": "0.1.0",
  "devDependencies": {
    "grunt": "~0.4.5"
  }
}
```

Un paquete de NPM es solo una biblioteca de JavaScript que, por lo general, funciona dentro de Node.js, y pueden ser encontradas en el sitio web de NPM (cualquiera puede subir sus bibliotecas a NPM, haciéndolas disponibles al público). NPM se encarga de manejar cada paquete y las dependencias de cada paquete, y las descarga de ser necesarias.

El archivo `package.json` contiene 3 propiedades principales, pudiendo tener más: el nombre del proyecto, la versión y las dependencias del proyecto (en este caso, los plugins para Grunt).

Lo siguiente que tenemos que hacer es instalar el paquete `grunt` desde NPM (lo que instalamos líneas arriba solo era el comando para consola, pero también necesitamos la biblioteca que permita utilizar los plugins de Grunt):

```
npm install
```

`npm install` es un comando que descargará cualquier paquete definida como parámetro, o los paquetes definidos en un archivo `package.json`, de existir uno, y los guardará en una carpeta llamada `node_modules` (el cual se creará si no existe). Vamos a utilizar este comando cada vez que querramos instalar una biblioteca definida dentro del archivo `package.json`. Por ejemplo, para agregar un plugin de Grunt al proyecto podemos utilizar el siguiente comando en la consola:

```
npm install grunt-contrib-jshint --save-dev
```

`grunt-contrib-jshint` es un plugin para Grunt que permite utilizar JSHint, una herramienta que analiza el código y lanza advertencias sobre su calidad y posibles errores que pueda tener.

Utilizando la propiedad `--save-dev` en el comando de la consola, NPM agregará una nueva propiedad (`grunt-contrib-jshint`) dentro de la propiedad `devDependencies`. Esta propiedad (`nombre : valor`) tendrá por nombre el nombre del paquete de NPM, y el valor será la versión que se desea instalar:

```
{
  "name": "buena-espina",
  "version": "0.1.0",
  "devDependencies": {
    "grunt": "~0.4.5",
    "grunt-contrib-jshint": "~0.10.0"
  }
}
```

Luego de agregar todos los módulos, debemos crear un segundo archivo, llamado `Gruntfile.js`, el cual también debe estar en la raíz del proyecto.

Un archivo `Gruntfile.js` es un módulo de Node.js; esto es, una función que es guardada en `module.exports`:

```
module.exports = function(grunt) {};
```

Dentro de esta función, debemos realizar 3 pasos:

1. Definir la configuración de cada plugin.

```
module.exports = function(grunt) {
  grunt.initConfig({
    pkg: grunt.file.readJSON('package.json'),
    jshint: {
      all: ['scripts/index.js'],  // definir los archivos que se
analizarán
      options: {
        curly: true,    // usar siempre llaves en bloques como if,
while, for
        eqeqeq: true,   // usar === en vez de ==
        browser: true,  // evita lanzar advertencias sobre variables
globales relacionadas al navegador
        globals: {      // evita lanzar advertencias sobre variables
globales específicas
          jQuery: true
        }
      }
    }
  });
};
```

1. Cargar los módulos de Node.js, definidos en el archivo `package.json`.

```
module.exports = function(grunt) {
  grunt.initConfig({
    // ...
  });

  grunt.loadNpmTasks('grunt-contrib-jshint');
};
```

1. Registrar las tareas de cada módulo.

```
module.exports = function(grunt) {
  grunt.initConfig({
    // ...
  });

  grunt.loadNpmTasks('grunt-contrib-jshint');

  grunt.registerTask('default', ['jshint']);
};
```

Al final, el archivo `Gruntfile.js` quedaría así:

```
module.exports = function(grunt) {
  grunt.initConfig({
    pkg: grunt.file.readJSON('package.json'),
    jshint: {
      all: ['scripts/index.js'],
      options: {
        curly: true,
        eqeqeq: true,
```

```
        browser: true,
        globals: {
          jQuery: true
        }
      }
    }
  });

  grunt.loadNpmTasks('grunt-contrib-jshint');

  grunt.registerTask('default', ['jshint']);
};
```

Por último, para correr las tareas, solo basta con ejecutar el siguiente comando en la consola:

```
grunt
```

Bower

Cuando trabajamos con proyectos medianos o grandes, tendremos que utilizar varias bibliotecas, las cuales pueden depender, a su vez, de otras bibliotecas. Si bien hemos visto el manejo de dependencias con RequireJS, esta maneja dependencias a nivel **lógico**, pero no a nivel de archivos. Bower permite manejar este tipo de dependencias (de archivos), descargando las bibliotecas que necesitemos, así como sus dependencias.

Para instalar Bower también necesitamos Node.js. En este caso el comando es el siguiente:

```
npm install -g bower
```

La propiedad `-g` instala el paquete a instalar (en este caso, `bower`), a nivel global, para que pueda ser utilizado por cualquier proyecto en la computadora. Este paquete instalará un comando en la consola, llamado `bower`. Este comando permitirá instalar bibliotecas simplemente con pasarle un nombre.

Por ejemplo, si deseamos instalar jQuery desde bower, solo debemos ejecutar el siguiente comando en la consola:

```
bower install jquery
```

Bower buscará, en su propio repositorio, una biblioteca con ese nombre. En este punto funciona bastante parecido a NPM, ya que cada biblioteca (o paquete de NPM) tiene un nombre único. Estas bibliotecas también manejan versiones, por lo que puedo instalar una versión específica de jQuery:

```
bower install jquery#1.11.1
```

Después de descargar la biblioteca a instalar (jQuery en este caso), creará una carpeta llamada `bower_components`, donde guardará la biblioteca.

RequireJS

Cuando se trabajan en aplicaciones, es necesario separar el código de acuerdo a sus responsabilidades, es decir, lo que realiza cada parte del código, y una buena forma de hacerlo es mediante el patrón Module. De esta forma, separamos el código por responsabilidades, y este se vuelve código reusable.

Sin embargo, si los módulos que creamos dependen de otros módulos (como seguramente será), vamos a tener problemas. En un documento HTML, deberíamos definir primero el módulo que no depende de nadie (llamado módulo `a.js`), luego definir el módulo que depende de `a.js` (el cual será llamado `b.js`), para luego definir al módulo que depende de `b.js`, si existiera, y así sucesivamente. El código quedaría así:

```
<script src="a.js"></script>
<script src="b.js"></script>
<script src="c.js"></script>
```

Pero esto no es óptimo. Si en algún momento `b.js` ya no depende de `a.js`, o `a.js` empieza a depender de un módulo nuevo, las cosas se complican más: No solo vamos a tener que cambiar el código dentro de cada archivo, si no el orden de las etiquetas `<script>`, para que cargue correctamente. Incluso, podría darse el caso en el que `a.js` empieza a depender de `c.js`, y este sigue dependiendo de `b.js` (lo cual pasa, pero debería hacerse lo posible para que no suceda). Es aquí donde aparece RequireJS.

RequireJS permite definir módulos, con sus respectivas dependencias, y solo necesita una etiqueta `<script>`. Para utilizar RequireJS es necesario descargarlo, y luego llamar a la biblioteca agregando la siguiente etiqueta:

```
<script data-main="main" src="require.js"></script>
```

La biblioteca es cargada en una etiqueta `<script>` a la que se define un atributo llamado `data-main`. RequireJS necesita un archivo principal desde donde empezar a cargar la aplicación, usualmente llamado `main.js`. El atributo `data-main` indica la ruta de ese archivo **en relación** al archivo HTML.

Adicionalmente a ello, se puede definir cierta configuración para RequireJS en una etiqueta `<script>` aparte:

```
<script >
  requirejs.config({
    urlArgs: 'timestamp=' + Date.now()
  });
</script>
```

Por ejemplo, utilizando este código tendremos que cada archivo cargado por RequireJS tendrá una dirección parecida a `archivo.js?timestamp=1418873637178`. De esta forma, el navegador evitará guardar en caché a estos archivos (útil cuando probamos un código muy seguido y necesitamos que el navegador siempre utilice el archivo real).

En nuestro caso, solo tenemos dos archivos, `jquery.js` e `index.js`, que será suficiente para hacer un ejemplo básico de RequireJS.

Como ya hemos instalado Bower, lo usaremos para instalar RequireJS. Si bien puede descargarse directamente, en este caso usaremos el comando `bower install`:

```
bower install requirejs
```

Luego de haber instalado RequireJS, vamos a modificar el archivo `index.html`, donde tenemos las siguientes etiquetas:

```
<script src="bower_components/jquery/dist/jquery.js"></script>
<script src="scripts/index.js"></script>
```

Lo primero que debemos hacer es dejar una sola etiqueta `<script>`, de la siguiente forma:

```
<script data-main="scripts/index"
src="bower_components/requirejs/require.js"></script>
```

Nuestro archivo `scripts/index.js` necesita convertirse en un módulo de RequireJS. Para esto, definimos un módulo con la función `require`, el cual toma dos parámetros:

- Un arreglo con las rutas de las dependencias del módulo que se está creando (que deben ser otros módulos de RequireJS)
- Una función que contendrá el código del módulo.

Esta función, a su vez, debe tener definidos tantos argumentos como elementos tenga el arreglo, asumiendo que cada elemento es una ruta a un módulo.

Así, el código que estaba dentro de `index.js` debería ir aquí (notemos que la función no tiene ningún argumento):

```
require(['../bower_components/jquery/dist/jquery'], function() {
  // ...
});
```

`../bower_components/jquery/dist/jquery` es la ruta de la biblioteca jQuery descargada desde Bower, y esta ruta es relativa a `scripts/index.js`. Si usamos más bibliotecas desde Bower, o armamos una estructura mucho más compleja de carpetas y archivos, tendremos muchas rutas que escribir, y posiblemente en más de un solo lugar.

RequireJS tiene una propiedad de configuración llamada `paths`, donde se pueden definir los nombres de cada módulo y sus respectivas rutas. De esta forma, les damos un alias a cada módulo:

```
<script type="text/javascript">
  requirejs.config({
    paths: {
      jquery: '../bower_components/jquery/dist/jquery'
    }
  });
</script>
```

Así, solo necesitamos escribir el alias dentro de `require`.

```
require(['jquery'], function($) {
  // ...
});
```

---

jQuery es una biblioteca que trata de funcionar en todos los casos posibles, ya sea llamándolo desde una etiqueta `<script>` o utilizando RequireJS.

Para el primer caso, la biblioteca agrega una variable global llamada `jQuery` (y su alias `$`) a `window`, mientras que en el segundo caso, crea un módulo del tipo AMD (que es el tipo de módulo que usa RequireJS). Puedes leer más sobre AMD en la misma web de RequireJS.

Estas dos porciones de código están dentro de `jquery.js`, donde el primero crea las variables globales `jQuery`y `$`:

```
if ( typeof noGlobal === strundefined ) {
  window.jQuery = window.$ = jQuery;
}
```

Mientras que el segundo crea un módulo AMD:

```
if ( typeof define === "function" && define.amd ) {
  define( "jquery", [], function() {
    return jQuery;
  });
}
```

Aquí podemos notar dos cosas importantes: se usa la función `define`, y esta toma 3 parámetros. La función `define` permite, como su nombre indica, definir un módulo, y puede tener un nombre *propio* (que es el primer parámetro). Los otros dos parámetros son similares a los usados en la función `require`: un arreglo de dependencias y una función que englobe el código del módulo. Cabe resaltar que `require` solo debe usarse en el archivo principal (definido en el atributo `data-main`), ya que no solo define un módulo, si no que lo **ejecuta**

inmediatamente, mientras que `define` solo define un módulo que será utilizado luego.

En el caso de jQuery, el módulo es llamado `jquery`, y es por eso que debemos usarlo en la configuración de RequireJS, dentro de la propiedad `paths`.

Para terminar, volvamos por un momento a la primera implementación de nuestro código con RequireJS:

```
require(['../bower_components/jquery/dist/jquery'], function() {
  // ...
});
```

Aquí vemos que la función que englobará nuestro código no tiene ningún argumento, pero el código funciona sin problemas. Esto significa que jQuery está usando como una variable global (utilizando `window.jQuery` o `window.$`).

Sin embargo, en la última implementación, vemos que la función sí tiene un argumento:

```
require(['jquery'], function($) {
  // ...
});
```

Esto sucede porque, al usar el nombre del módulo de jQuery (definido por `jquery.js`) en la propiedad `paths`, ya estamos usando el módulo de tipo AMD.