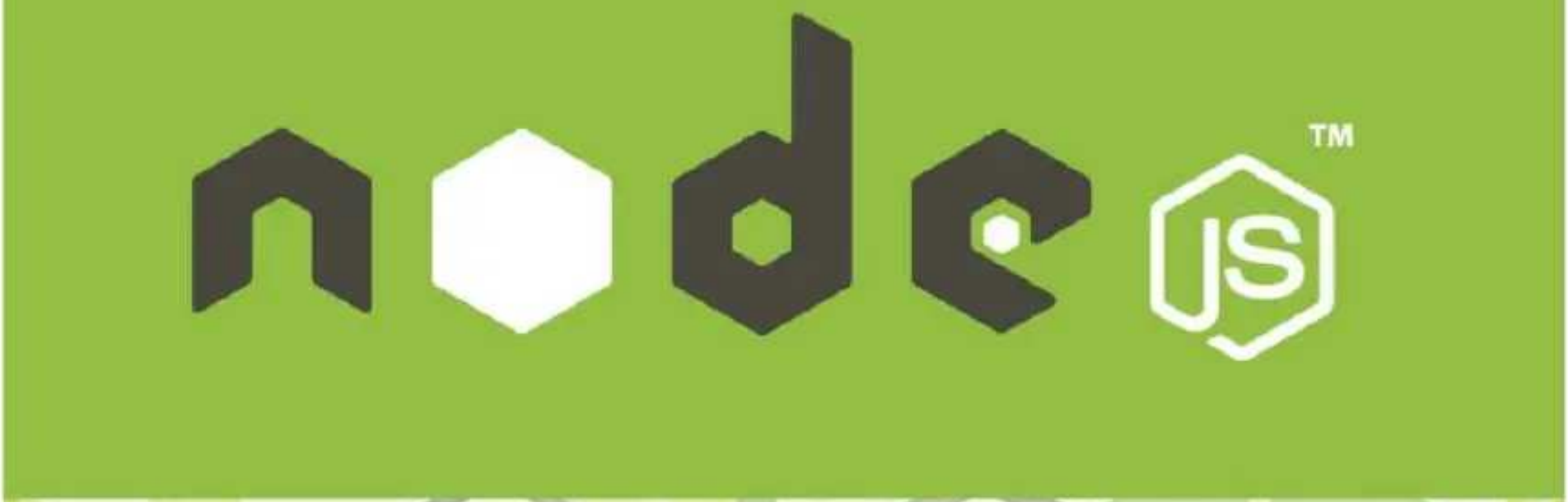
Node.js

JavaScript en el lado del servidor

Manual práctico avanzado

Ismael López Quintero



Alfaomega

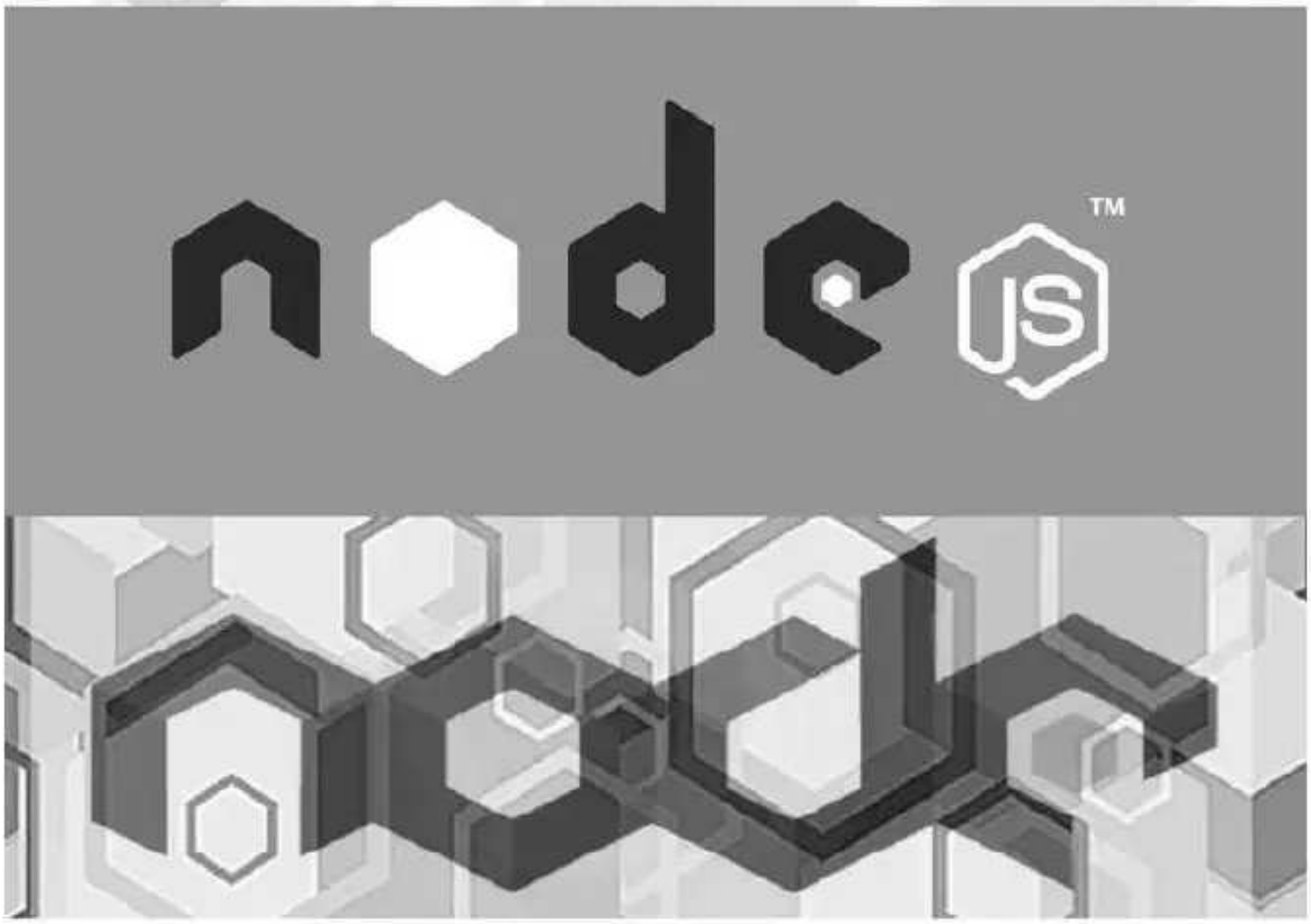
Altaria publicaciones

Node.js

JavaScript en el lado del servidor

Manual práctico avanzado

Ismael López Quintero



Alfaomega

Altaria publicaciones

Corrección y revisión:
Marta Giménez y Miriam Msaoury

Datos catalográficos

López, Ismael
NodeJs del lado del servidor
Primera Edición
Alfaomega Grupo Editor, S.A. de C.V., México

ISBN: 978-607-622-576-9

Formato: 17 x 23 cm Páginas: 536

NodeJs del lado del servidor
Ismael López Quintero
ISBN: 978-84-944009-3-1, edición en español publicada por Publicaciones Altaria S.L., Tarragona, España

Primera edición: Alfaomega Grupo Editor, México, enero 2016

Pitágoras 1139, Col. Del Valle, 03100, México D.F.

Miembro de la Cámara Nacional de la Industria Editorial Mexicana
Registro No. 2317

Pág. Web: **http://www.alfaomega.com.mx**
E-mail: **atencionalcliente@alfaomega.com.mx**

ISBN: 978-607-622-576-9

Empresas del grupo:

México: Alfaomega Grupo Editor, S.A. de C.V. – Pitágoras 1139, Col. Del Valle, México, D.F. – C.P. 03100.
Tel.: (52-55) 5575-5022 – Fax: (52-55) 5575-2420 / 2490. Sin costo: 01-800-020-4396
E-mail: atencionalcliente@alfaomega.com.mx

Colombia: Alfaomega Colombiana S.A. – Calle 62 No. 20-46, Barrio San Luis, Bogotá, Colombia,
Tels.: (57-1) 746 0102 / 210 0415 – E-mail: cliente@alfaomega.com.co

Chile: Alfaomega Grupo Editor, S.A. – Av. Providencia 1443. Oficina 24, Santiago, Chile
Tel.: (56-2) 2235-4248 – Fax: (56-2) 2235-5786 – E-mail: agechile@alfaomega.cl

Argentina: Alfaomega Grupo Editor Argentino, S.A. – Paraguay 1307 P.B. Of. 11, C.P. 1057, Buenos Aires, Argentina, – Tel./Fax: (54-11) 4811-0887 y 4811 7183 – E-mail: ventas@alfaomegaeditor.com.ar

"A mis padres".

Índice general

¿A quién va dirigido este libro?

Este libro va dirigido a desarrolladores web con cierta experiencia en el uso de JavaScript.

No se cubren los aspectos básicos del lenguaje ni su sintaxis. Es deseable que el lector tenga experiencia con AJAX, con las hojas de estilo CSS, e, incluso, con jQuery como framework de JavaScript en el lado del cliente.

No es necesaria experiencia previa con node.js. Es una tecnología reciente y con este manual se pretende dar las nociones suficientes para que el lector pueda desarrollar una aplicación web completa y adentrarse en el mundo de node.js.

Convenciones generales

El manual que tiene ante sí encamina todo su contenido hacia la capacitación para crear una aplicación web completa en node.js. Partiendo de las características más avanzadas de JavaScript (desde un nivel que presupone el conocimiento de los aspectos más básicos del lenguaje), el manual aborda en un tercer capítulo el estudio básico de node para pasar a estudiar en el siguiente capítulo la implementación del patrón arquitectónico Modelo-Vista-Controlador mediante Express. Del mismo modo se estudia una serie de paquetes que están a la orden del día en cualquier proyecto node. Para la persistencia de datos se le dedica un capítulo a MongoDB, solución NoSQL altamente eficiente para entornos con gran número de transacciones con la Base de Datos. En el capítulo final se muestra al lector la implementación de una pequeña red social en la que los usuarios pueden crear relaciones de amistad, escribir posts, hacer comentarios sobre estos posts, y establecer conversaciones de chat con sus amigos. Todo ello acompañado de ejercicios del mismo nivel que los ejemplos que se ilustran a lo largo del texto. Con la lectura de este manual y la implementación de sus ejercicios, el lector dará el paso definitivo a una nueva tendencia en el mundo del software, que se espera va a ocupar un lugar trascendente en los próximos años.

En el lenguaje habitual del programador, es normal escribir "los fuentes", refiriéndonos al código fuente. Por razones de estilo gramatical, a lo largo del libro se ha utilizado el género femenino, pero creemos conveniente señalarlo aquí porque el libro va dirigido fundamentalmente a programadores.

Introducción

1

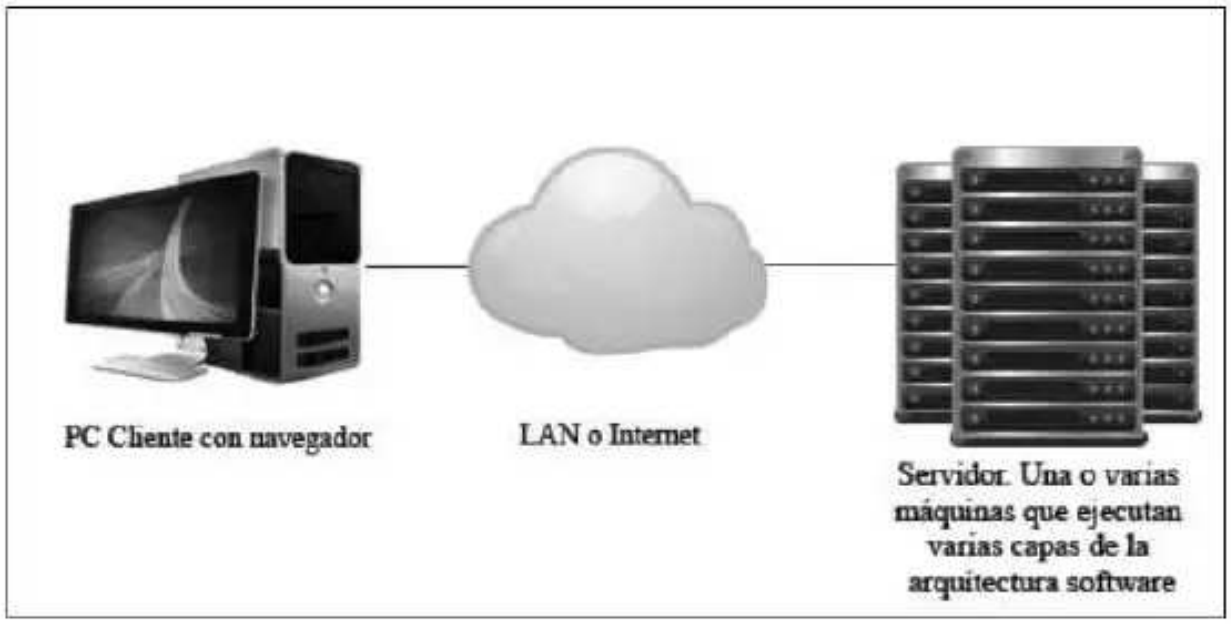
CAPÍTULO 1

INTRODUCCIÓN

1.1 Introducción

JavaScript en el lado del servidor. La primera vez que muchos programadores hemos oído esta opción de desarrollo nos hemos quedado sorprendidos. Tradicionalmente vinculado al navegador del cliente, al comienzo de JavaScript, y es una opinión personal, se concibió como un lenguaje "de juguete" destinado a crear efectos de animación y a hacer la web en el lado del cliente (el navegador), más amigable. Hacer que los controles de los formularios tengan un aspecto más elegante, juego de colores al hacer *rollover* con el ratón, hacer aparecer y desaparecer elementos HTML... convirtieron a JavaScript en la "solución dinámica en el cliente". No debemos de olvidar que existían y existen otras tecnologías destinadas a crear animaciones en la Web, tales como Flash o el propio Java mediante los ya tradicionales Applets, que pueden incluir objetos Canvas para el dibujo de gráficos. JavaScript puede ser más o menos potente para según qué tareas, pero posee una ventaja sobre las otras opciones que se han mencionado: trabaja directamente con el árbol de objetos HTML, también denominado árbol DOM (Document Object Model o Modelo de Objetos del Documento), que es la estructura de datos a modo de árbol que crea el navegador tras realizar el análisis sintáctico y semántico del fichero de entrada HTML. Al trabajar directamente con el árbol DOM, no se añade ningún elemento "pesado" a dicho árbol, como podría ser una animación Flash. Debido a esto los sitios web son más ligeros y rápidos de cargar. Otra ventaja es que, al no incluirse elementos que no sean única y exclusivamente HTML, los buscadores de Internet analizan con facilidad el contenido del documento y facilitan a los expertos en Marketing de Contenidos la labor de posicionamiento de la Web en Internet. Han existido y existen sitios web desarrollados en su totalidad en Flash. Esto hace que los buscadores no sean capaces de indexar adecuadamente el contenido, ya que un archivo de Flash es en realidad una película de diapositivas. Lo que los buscadores de Internet analizan son documentos HTML.

No se ha visto carente de problemas JavaScript en su evolución. El mayor problema, sin lugar a dudas, ha sido la falta de estandarización en los navegadores. En este campo ha sido muy importante la labor realizada por la W3C, que es la organización que se dedica a redactar recomendaciones relacionadas con la World Wide Web, el protocolo HTTP y los documentos HTML. Pero no siempre los navegadores han ido a la par y, a veces, han tratado de forma muy dispar características como los objetos relacionados con el navegador o la gestión de eventos. En este sentido y como ya se ha mencionado, es muy importante la labor realizada por la W3C, pero también ha sido muy importante la labor realizada por la comunidad de programadores con el desarrollo de librerías o frameworks tales como Prototype o jQuery, que hacen posible la implementación de soluciones cross-browser o multi-navegador, aislando al programador de este problema.



El proceso hasta la llegada de JavaScript al lado del servidor ha tenido un paso previo, conocido con los mismos cuatro caracteres con los que se conoce a un equipo de fútbol: AJAX. Siendo AJAX las siglas de Asynchronous JavaScript And Xml, vino a suplir la necesidad de acceder en tiempo real en el navegador a datos que no se poseen en él, sino en el servidor. Cuando trabajamos con una aplicación web, la información que en cada petición o request se envía al cliente suele ser la propia página HTML en texto plano, una vez procesada por el servidor; y algunos datos a poner en campos de formulario, como pueden ser selects, inputs, etc; que comúnmente se envían en el propio documento HTML. La idea de AJAX es hacer una petición al servidor y recibir una respuesta con los datos solicitados sin necesidad de recargar todo el documento HTML. Antes de AJAX, la experiencia del usuario en la web era pobre si la comparábamos con la experiencia de las aplicaciones de Escritorio, ya que con JavaScript y sin AJAX sólo podíamos implementar una simple lógica de la interfaz de usuario, como los ya mencionados efectos *rollover*, la aparición y desaparición de elementos, etc. En las aplicaciones de escritorio tradicionales suele ser común tener todos los datos en la misma máquina en la que se ejecuta la interfaz de usuario. Hablando del patrón de diseño Modelo-Vista-Controlador (MVC), podemos decir que en una aplicación de escritorio, por lo general, las tres capas residen en la máquina del usuario que ejecuta la aplicación. Pero no es así en las aplicaciones Web. La arquitectura *software* de las aplicaciones Web puede ser muy variada, teniendo claro que la interfaz de usuario se ejecuta en el navegador del cliente, y teniendo la lógica del modelo del dominio, el acceso a datos y el controlador de la aplicación, en una o varias máquinas, siempre distintas de la máquina cliente.

¿Qué datos pueden estar en el servidor y pueden necesitarse desde el cliente sin ser deseable el envío de todo el documento HTML? Normalmente suelen ser listados extraíbles de bases de datos, que pueden ayudar al usuario a tomar decisiones en tiempo real. El ejemplo por excelencia de implementación de AJAX es aquel en el que, conforme rellenamos un campo de entrada de texto de un formulario, enviamos la cadena introducida por el usuario al servidor, y éste nos devuelve, vía AJAX, el listado de elementos "texto", que comienzan por la cadena de texto introducida. Veamos un ejemplo de implementación de AJAX, en el que se nos muestra un listado con los municipios españoles dependiendo de la entrada de usuario:

Según introducimos caracteres en el campo, le enviamos una petición AJAX al servidor, y éste nos devuelve, en un documento XML (de ahí lo de Asynchronous JavaScript and **XML**), el listado de entradas en la base de datos que comienzan por el texto introducido por el usuario:

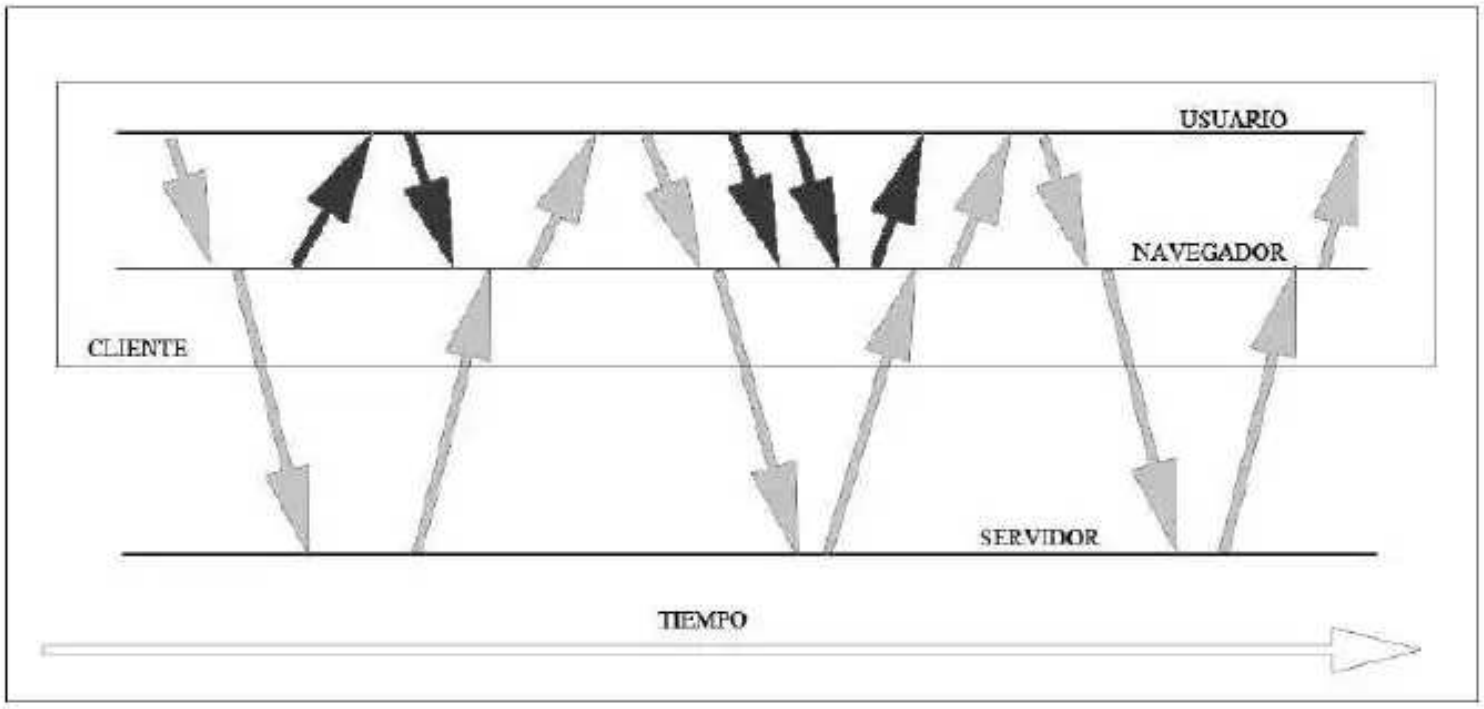
El listado se nos devuelve sin necesidad de pinchar en ningún botón ***submit*** de formulario, y sin necesidad de hacer click en ningún enlace, por lo que no hacemos ninguna *request* al servidor (al menos no por el método tradicional), no recargándose, por lo tanto, todo el documento HTML. Esto mejora considerablemente la experiencia del cliente en la Web, ya que aproxima mucho la experiencia de las aplicaciones Web a la de las aplicaciones de Escritorio.

AJAX surgió en 2005 de la mano de su creador, Jesse James Garrett. En realidad, AJAX no fue ninguna tecnología nueva, sino la unión de un conjunto de tecnologías existentes. JavaScript, HTML y su estructura de datos en el navegador (DOM), XML para el envío de datos... La única pieza que falta para completar el puzzle se comenzó a implementar en las comunicaciones http junto con Internet Explorer 5. Esta pieza se llama XMLHttpRequest, y es un objeto, con estado, que permite la comunicación asíncrona entre el cliente y el servidor.

En realidad, el servidor no sólo puede enviar al cliente un documento XML. Puede enviar una cadena de texto con una estructura XML definida y correcta. También puede enviar un objeto JSON. JSON son las siglas de JavaScript Object Notation o Notación de Objetos de JavaScript. Es una estructura de objetos más fácil de manejar en JavaScript que XML, ya que accedemos a cada objeto y sus campos de la misma manera en que accedemos a los atributos de cualquier objeto de JavaScript, mediante puntos. En el capítulo 2 se tratará más detenidamente JSON y su especificación.

También se puede enviar del servidor al cliente texto HTML. Una vez recibido en el cliente el texto HTML (que no es directamente recargado en el navegador), analizamos con JavaScript el HTML recibido y creamos un árbol DOM diferente al que está cargado en el navegador y visualiza el usuario. Buscamos en el "árbol recibido" la parte que queremos reemplazar. Cuando tenemos dicho fragmento, reemplazamos la parte del documento objeto de cambio mediante JavaScript, simplemente reemplazando el nodo del árbol DOM. A la alteración del árbol DOM sin necesidad de recargar completamente la página se le denomina DHTML o HTML Dinámico.

Vamos a profundizar un poco más en cómo trabaja AJAX, ya que es la base para entender el salto a node.js. En AJAX, el cliente hace una petición asíncrona al servidor, y continúa su funcionamiento. Es decir, el usuario puede seguir moviendo el ratón, puede seguir escribiendo texto, generando eventos... antes de que se obtenga la respuesta del servidor, que se procesa en paralelo, o de forma concurrente, con lo que esté haciendo el usuario. Visualmente, podemos decir que AJAX funciona así:



Las flechas que salen desde el usuario hacia el navegador son eventos generados por éste primero, y las flechas que vuelven del navegador al usuario, son las respuesta en interfaz de usuario por parte del navegador. Se ha señalado en color claro las operaciones AJAX que "siempre" comienzan por parte del usuario. A veces pueden ser generadas por eventos del sistema, pero prácticamente siempre, las operaciones AJAX comienzan por eventos creados por el usuario. Apreciamos que existe una asincronía porque las operaciones AJAX, en llamada desde el cliente al servidor, se ejecutan en paralelo. El cliente no queda esperando a que el servidor responda para continuar su ejecución. Pero la pregunta es: ¿tenemos una asincronía real? El cliente hace una petición AJAX, pero ¿y si el servidor no responde? El navegador responderá "en local", a

las entradas del usuario, pero nunca podrá mostrar la respuesta del servidor. En ese caso, tras un tiempo de espera, habría que indicar de alguna forma al cliente que se ha perdido la comunicación con el servidor, e intentar retomar la comunicación. Si la demora es debida a que el servidor tiene sobrecarga de trabajo podemos tener suerte, pero si se ha caído, la aplicación se quedará "colgada".

Evidentemente, si se cae el servidor en el que estamos ejecutando una aplicación, la aplicación no funcionará, sea cual sea la tecnología que estemos usando (AJAX haciendo peticiones a servidores síncronos o a servidores asíncronos node.js). Lo que se pretende con llevar JavaScript y su asincronía al lado del servidor es **tener una asincronía real en el lado del servidor**. Con ello lo que conseguimos es que en una máquina servidora se puedan hacer múltiples peticiones a otras máquinas servidoras, y esperar a que éstas respondan. Si alguna de ellas no responde, la aplicación en ningún caso se quedará “colgada”, a menos que se caiga la máquina principal que ejecuta JavaScript en el lado del servidor o node.js.

Los lenguajes de script en el lado del servidor tradicionales (PHP, Perl, Java...) no implementan de forma tan clara o fácil esta asincronía que estamos buscando. De ahí el porqué de node.js. También hay que indicar que el motor de node.js es JavaScript, lo que lo convierte en una tecnología bastante rápida. Y lo mejor de todo, no necesita de servidor Web, ni Apache, ni Tomcat, ni IIS, ni NGinx ni ningún otro.

1.2 A tener en cuenta antes de comenzar

Antes de afrontar la lectura de este libro, es bueno que el programador tenga alguna experiencia en el desarrollo web, lo que implica trabajo con JavaScript, con AJAX, que conozca la arquitectura *software* de tres capas MVC. Es bueno que conozca el lenguaje de hojas de estilos CSS. No es necesaria experiencia con node.js. Es una tecnología relativamente nueva, por lo tanto, el mismo autor que escribe este libro está en aprendizaje continuo de un lenguaje que comenzó a ser relativamente popular en 2011, y que está en constante evolución, debido al continuo avance y transformación de las tecnologías de la Web. Precisamente para eso se escribe este libro. Con la indicación "Manual Práctico Avanzado", no se pretende publicar un libro para aquellos gurús de node.js que quieran saber más y más; sino simplemente aprender a crear una aplicación Web completa con esta incipiente tecnología que se espera que va a ser una potente herramienta de desarrollo en los próximos años.

¿Qué es necesario saber de JavaScript?

2

CAPÍTULO 2

¿QUÉ ES NECESARIO SABER DE JAVASCRIPT?

2.1 Introducción

Aunque el presente texto está enfocado a programadores que tengan cierta experiencia con JavaScript, hay un conjunto de características que lo diferencian del resto de lenguajes de programación estructurados u orientados a objetos. Se da por hecho que el usuario de este libro conoce en profundidad estos dos paradigmas de programación: estructurado y orientado a objetos. El usuario debe conocer los tipos de datos elementales, las estructuras condicionales, los bucles, la recursividad, las estructuras de datos. Debe saber qué significa una clase, un atributo, un método, la herencia y el polimorfismo. Bajo estas premisas, y dando por hecho que se ha trabajado previamente con JavaScript, se pasa a detallar aquellas características del lenguaje que lo hacen algo diferente a otros lenguajes, características, dicho sea de paso, que hacen a JavaScript y a sus frameworks (tales como jQuery), algo difíciles, en parte, de entender. Si bien el estudio de frameworks no es el objetivo, la lectura de este capítulo ayudará a entender aspectos oscuros de dichos marcos de trabajo en JavaScript. Comenzaremos hablando de la notación JSON. El modelo del dominio y el acceso a datos va a estar expresado en notación JSON, en vez de en las clases que estaría en otros lenguajes como Java o PHP. Hablaremos del ámbito de las variables, que son tratadas de una forma un tanto diferente a como lo hace Java. Posteriormente, nuestra atención se centrará en los cierres, las lambdas, los callbacks, y otra serie de características antes de adentrarnos, en el siguiente capítulo, en el mundo de node.js.

2.2 Entorno de trabajo con JavaScript en el lado del cliente

JavaScript es uno de los lenguajes que menos despliegue necesita. De hecho, para echarlo a andar, sólo se necesita un editor de textos y un navegador. En este primer capítulo vamos a ver JavaScript de forma general, sin dar aún el salto al lado del servidor. Para poder trabajar, vamos a usar herramientas libres, que se pueden descargar desde sus respectivos sitios web. Como editor de textos usaremos el Bloc de Notas o Notepad++, y como navegador, Mozilla Firefox, con el complemento Firebug instalado. Son herramientas conocidas por todos los desarrolladores Web. Descargaremos las últimas versiones desde los siguientes sitios:

- http://notepad-plus-plus.org/.
- http://www.mozilla.org/.

Una forma de comprobar el correcto funcionamiento del código que escribimos son las pruebas unitarias. Existe una metodología de desarrollo de *software* que se denomina "Desarrollo Dirigido por Tests" o Test Driven Development (TDD). El TDD realiza las pruebas unitarias de nuestro código cada vez que implementamos una nueva función. En nuestro caso vamos a usar QUnit como framework que nos va a facilitar mucho la ejecución y comprobación de nuestro código. Podemos descargarlo desde su sitio web:

- http://qunitjs.com/.

En las pruebas unitarias que vamos a usar existen varios métodos:

- **assert.ok(parámetro,mensaje)**. Si el parámetro es igual a **true**, la prueba es correcta y se muestra el mensaje en el navegador.
- **assert.equal(valor1,valor2,mensaje)**. Si valor1 y valor2 coinciden, la prueba es correcta y se muestra el mensaje en el navegador.
- **funcion = assert.assync()**. Indica al motor de QUnit que vamos a lanzar pruebas asíncronas, por lo tanto, debe quedar esperando aserciones incluso una vez acabada la ejecución del código JavaScript que se cargue al inicio. Para indicar el fin de las pruebas asíncronas, simplemente hemos de ejecutar la función que se nos ha devuelto: funcion().
- **assert.expect(n)**. De manera opcional, podemos indicarle al motor de pruebas unitarias el número de aserciones que debe esperar.

Indicar que la igualdad esperada en equal sólo debe de ser de valor, no es necesario que sea de valor y tipo. Se podría comparar, el valor entero 0 con la cadena "0", y la aserción sería correcta.

Como se aprende mucho más con un ejemplo que con veinte mil palabras, vamos a hacer una prueba unitaria en JavaScript sobre la ejecución de la función factorial. De Matemáticas sabemos que el factorial de un número natural es una función recursiva que se define como el propio número por el factorial del número menos 1.

$$n! = n * (n-1)!$$

Y también sabemos, de Matemáticas, que la recursión termina en el 0, donde el factorial de 0 es 1.

$$0! = 1$$

Por ejemplo, el factorial de 6 por recursión sería:

$$6! = 6 * 5!$$

$$5! = 5 * 4!$$

$$4! = 4 * 3!$$

$$3! = 3 * 2!$$

$$2! = 2 * 1!$$

$$1! = 1 * 0!$$

$$0! = 1$$

Por lo tanto, el resultado es:

$$6! = 6 * 5 * 4 * 3 * 2 * 1 * 1 = 720.$$

Vamos a implementar, en el lado del cliente, una función en JavaScript que calcule el factorial de un número, y vamos a lanzar una prueba unitaria que compruebe que el factorial de 6 es 720. Así comprobaremos que el factorial está bien implementado.

Código HTML de nuestra aplicación.

```
<html>
<head>
<meta charset="utf-8">
<title>Ejemplo básico de JavaScript</title>
<link rel="stylesheet" href="./qunit-1.16.0.css">
</head>
<body>
<div id="qunit"></div>
<div id="qunit-fxture"></div>
<script src="./qunit-1.16.0.js"></script>
<script src="./codigo.js"></script>
</body>

</html>
```

Donde el fichero qunit-1.16.0.js y la hoja de estilos qunit-1.16.0.css son las descargadas desde el sitio oficial del framework QUnit. Damos por hecho que en este primer ejemplo se encuentran en la misma carpeta que el código html y que el script codigo.js.

- Ejemplo del factorial a implementar. Contenido del fichero codigo.js.

```
function factorial(n) {
 if (n % 1 == 0) {
  if (n>0) {
   return n*factorial(n-1);
  } else {
   return 1;
  }
 } else {
  return -1;
 }
}
QUnit.test( "Prueba función factorial", function( assert ) {
 assert.equal(factorial(6),720, "¡Correcto!" );
});
```

La salida que se muestra en el navegador es la que se muestra seguidamente.

¿Por qué usamos las pruebas unitarias y no escribimos las salidas en el navegador con document.write()? Simplemente, por comodidad. Nos olvidamos del código HTML y nos centramos única y exclusivamente en el código JavaScript que pondremos en codigo.js. Nuestra página HTML siempre será la misma, sólo estaremos centrados en el código JavaScript. Además, QUnit comprobará los valores por nosotros. No tenemos que centrarnos en estudiar las salidas una a una.

Vemos un poco más en detalle las pruebas unitarias. Realicemos varias comprobaciones.

5! = 120.

6! = 720.

7! = 5040.

8! = 40320.

En el caso del factorial del número 8, vamos a indicar un valor incorrecto para ver cómo reacciona QUnit a los fallos.

```
QUnit.test( "Prueba función factorial", function( assert ) {
 assert.equal(factorial(5),120, "Correcto el factorial de 5.");
 assert.equal(factorial(6),720, "Correcto el factorial de 6.");
 assert.equal(factorial(7),5040, "Correcto el factorial de 7.");
 assert.equal(factorial(8),40321, "Correcto el factorial de 8.");
});
```

Si se mira con detenimiento, podemos ver que el factorial de 8 lo hemos indicado mal a propósito, para ver la salida de las prueba unitarias cuando el resultado no es el esperado. Tras ejecutar estas pruebas en el navegador, la salida es la que se indica seguidamente.

2.3 Notación JSON

El desarrollo de JSON se ha debido a la característica de JavaScript de que sus arrays son asociativos. Si en JavaScript declaramos un array y le asignamos valores, los índices que le proporcionará el motor del lenguaje serán los números naturales comenzando por el cero.

```
var miarray = ["hola" , "estamos", "haciendo", "una", "prueba"];
QUnit.test( "Prueba Array", function( assert ) {
 assert.equal(miarray[0],"hola", "Correcto" );
 assert.equal(miarray[1],"estamos", "Correcto" );
 assert.equal(miarray[2],"haciendo", "Correcto" );
 assert.equal(miarray[3],"una", "Correcto" );
 assert.equal(miarray[4],"prueba", "Correcto" );
});
```

Todas las anteriores aserciones son correctas. En el código se aprecia perfectamente que el indexado se está realizando por números naturales. Pero JavaScript nos permite especificar otro tipo de índice. Estos índices son las cadenas de texto. Entonces, a cada cadena de texto, se le asocia un valor. Como ya se ha indicado, a los arrays que permiten este tipo de asociación, se les denominan arrays asociativos.

```
var miarray = ["hola" , "estamos", "haciendo", "una", "prueba"];
miarray["indice"]="cadena de texto";
QUnit.test( "Prueba Array", function( assert ) {
 assert.equal(miarray[0],"hola", "Correcto" );
 assert.equal(miarray[1],"estamos", "Correcto" );
```

```
 assert.equal(miarray[2],"haciendo", "Correcto" );
 assert.equal(miarray[3],"una", "Correcto" );
 assert.equal(miarray[4],"prueba", "Correcto" );
 assert.equal(miarray["indice"],"cadena de texto", "Correcto" );
});
```

Todas las aserciones anteriores vuelven a ser correctas. Como se puede ver, se permite que en un mismo array "convivan" el indexado por números e indexado por cadenas de texto.

Imaginemos que queremos crear un objeto del modelo del dominio llamado "usuario". De cada usuario, los atributos que podríamos guardar son los siguientes:

- Nick.
- Nombre completo.
- Email.
- Password.

Podríamos crear un array asociativo de la siguiente forma:

```
var usuario = new Array();
usuario["nick"] = "jvix";
usuario["nombreCompleto"] = "Javier Pérez Álvarez";
usuario["email"] = "jvix@jvix.com";
usuario["password"] = "jvix543";
QUnit.test( "Prueba Array", function( assert ) {
 assert.equal(usuario["nick"],"jvix", "Correcto" );
 assert.equal(usuario["nombreCompleto"],"Javier Pérez Álvarez",
 "Correcto" );
 assert.equal(usuario["email"],"jvix@jvix.com", "Correcto" );
 assert.equal(usuario["password"],"jvix543", "Correcto" );
 assert.equal(usuario.nick,"jvix", "Correcto" );
 assert.equal(usuario.nombreCompleto,"Javier Pérez Álvarez",
 "Correcto" );
 assert.equal(usuario.email,"jvix@jvix.com", "Correcto" );
 assert.equal(usuario.password,"jvix543", "Correcto" );
});
```

Todas las aserciones anteriores son correctas. Vemos que **es exactamente lo mismo** usar indexación por cadenas de texto que usar el nombre del array y el nombre del campo separados por un punto. Una vez hemos ejecutado y visto que este código funciona, nos formulamos la siguiente pregunta. ¿Hay alguna forma de crear el mismo objeto sin la necesidad de especificar el índice para cada elemento del array? La respuesta es sí. ¿Cómo? Usando la notación JSON. El siguiente código es equivalente al anterior.

```
var usuario = {
 nick : "jvix",
 nombreCompleto : "Javier Pérez Álvarez",
 email : "jvix@jvix.com",
 password : "jvix543"
};
QUnit.test( "Prueba Array", function( assert ) {
```

```
 assert.equal(usuario["nick"],"jvix", "Correcto" );
 assert.equal(usuario["nombreCompleto"],"Javier Pérez Álvarez",
 "Correcto" );
 assert.equal(usuario["email"],"jvix@jvix.com", "Correcto" );
 assert.equal(usuario["password"],"jvix543", "Correcto" );
 assert.equal(usuario.nick,"jvix", "Correcto" );
 assert.equal(usuario.nombreCompleto,"Javier Pérez Álvarez",
 "Correcto" );
 assert.equal(usuario.email,"jvix@jvix.com", "Correcto" );
 assert.equal(usuario.password,"jvix543", "Correcto" );
});
```

Para definir a un usuario, ahora estamos usando la notación JSON. En las comunicaciones de datos entre máquinas se está implantando el uso de JSON, incluso por delante de XML. Imaginemos un conjunto de libros. Para cada libro, almacenaremos:

- Título.
- Editorial.
- Autor.
- Fecha primera edición
- ISBN.

Y para cada autor, almacenaremos la siguiente información:

- Nombre completo.
- Fecha de nacimiento.
- Nacionalidad.

Podríamos definir en JSON una pequeña biblioteca de tres libros con la siguiente información:

```
var biblioteca = [ {
 titulo : "JavaScript en el Lado del Cliente",
 editorial : "Editorial Programación en la Red",
 autor : {
  nombreCompleto : "Javier Pérez Álvarez",
  fechaNacimiento : "01/01/1970",
  nacionalidad : "Española"
 },
 fechaPrimeraEdicion : "04/07/1983",
 isbn : "123456789"
} , {
 titulo : "JavaScript en el Lado del Servidor",
 editorial : "Editorial Programación en la Red",
 autor : {
  nombreCompleto : "Ismael López Quintero",
  fechaNacimiento : "04/07/1983",
  nacionalidad : "Española"
 },
 fechaPrimeraEdicion : "05/06/1998",
 isbn : "987654321"
} , {
```

```
 titulo : "Introducción a JavaScript",
 editorial : "Editorial Universitaria de Programación",
 autor : {
  nombreCompleto : "Agústín Méndez Castaño",
  fechaNacimiento : "03/05/1993",
  nacionalidad : "Española"
 },
 fechaPrimeraEdicion : "06/08/2005",
 isbn : "789123456"
}
];
QUnit.test( "Prueba Array", function( assert ) {
 assert.equal(biblioteca[1].autor.nombreCompleto,"Ismael López
 Quintero","El autor del segundo libro es correcto!");
});
```

Del anterior ejemplo hemos de destacar dos peculiaridades. La estructura biblioteca es un array indexado por números naturales, comenzando por el cero, ya que no se especifican cadenas de texto para la indexación. Cada posición del array es un objeto JSON, que se corresponde con la estructura de un libro que previamente hemos definido. La segunda peculiaridad es que, para cada libro, todos los campos son cadenas de texto, a excepción del campo autor, que se corresponde con otro objeto JSON. Dicho "subobjeto", contiene la información de un autor que también previamente hemos definido.

Podemos observar que en JSON se pueden anidar los objetos, de la misma manera que en XML podíamos definir etiquetas de objetos dentro de otras etiquetas. La facilidad que nos ofrece JSON en JavaScript es que podemos acceder a propiedades de objetos en el interior de objetos, simplemente secuenciando puntos en el uso de las variables, igual que anteriormente realizamos *biblioteca[1].autor.nombreCompleto.*

Vamos a ir un poco más allá. Vamos a definir en JavaScript la misma estructura de clases del modelo del dominio que acabamos de ver, no como un conjunto de datos prefijados en un fichero, sino como lo que son, un conjunto de clases.

Éste va a ser el primer ejemplo en el que tendremos varios archivos fuente de JavaScript. El código de la página HTML también va a cambiar, para incluir todos los ficheros de JavaScript. Tendremos la clase Libro (fichero libro.js), tendremos la clase Autor (fichero autor.js) y, también, tendremos el programa principal en nuestro fichero codigo.js, que será el que realiza las pruebas unitarias. Los ficheros QUnit se encuentran en el directorio padre de aquel en el que están alojados los ficheros.

- Fichero HTML:

```
<html>
<head>
<meta charset="utf-8">
<title>Prueba de JavaScript</title>
<link rel="stylesheet" href="../qunit-1.16.0.css">
</head>
<body>
<div id="qunit"></div>
<div id="qunit-fixture"></div>
<script src="../qunit-1.16.0.js"></script>
<script src="./autor.js"></script>
<script src="./libro.js"></script>
<script src="./codigo.js"></script>
</body>
</html>
```

- Fichero autor.js:

```
// Implementación de la clase Autor con getters y setters.
var Autor = function() {
 var sThis = this;
 this.datosAutor = {
  nombreCompleto : '',
  fechaNacimiento : '',
  nacionalidad : ''
};
var getNombreCompleto = function() {
 return sThis.datosAutor.nombreCompleto;
},
setNombreCompleto = function(nombreCompleto) {
 sThis.datosAutor.nombreCompleto=nombreCompleto;
},
getFechaNacimiento = function() {
return sThis.datosAutor.fechaNacimiento;
},
setFechaNacimiento = function(fechaNacimiento) {
 sThis.datosAutor.fechaNacimiento=fechaNacimiento;
},
getNacionalidad = function() {
 return sThis.datosAutor.nacionalidad;
},
 setNacionalidad = function(nacionalidad) {
 sThis.datosAutor.nacionalidad=nacionalidad;
};
return {
 getNombreCompleto : getNombreCompleto,
 setNombreCompleto : setNombreCompleto,
 getFechaNacimiento : getFechaNacimiento,
 setFechaNacimiento : setFechaNacimiento,
```

```
  getNacionalidad : getNacionalidad,
  setNacionalidad : setNacionalidad
 }
};
```

- Fichero libro.js:

```
// Implementación de la clase Libro con getters y setters.
var Libro = function() {
 var sThis = this;
 this.datosLibro = {
  titulo : '',
  editorial : '',
  autor : {},
  fechaPrimeraEdicion : '',
  isbn : ''
 };
 var getTitulo = function() {
  return sThis.datosLibro.titulo;
 },
 setTitulo = function(titulo) {
  sThis.datosLibro.titulo = titulo;
 },
 getEditorial = function() {
  return sThis.datosLibro.editorial;
 },
 setEditorial = function(editorial) {
  sThis.datosLibro.editorial = editorial;
 },
 getAutor = function() {
  return sThis.datosLibro.autor;
 },
 setAutor = function(autor) {
  sThis.datosLibro.autor = autor;
 },
 getFechaPrimeraEdicion = function() {
  return sThis.datosLibro.fechaPrimeraEdicion;
 },
 setFechaPrimeraEdicion = function(fechaPrimeraEdicion) {
  sThis.datosLibro.fechaPrimeraEdicion = fechaPrimeraEdicion;
 },
 getIsbn = function() {
  return sThis.datosLibro.isbn;
 },
 setIsbn = function(isbn) {
  sThis.datosLibro.isbn=isbn;
 };
 return {
  getTitulo : getTitulo,
  setTitulo : setTitulo,
  getEditorial : getEditorial,
```

```
  setEditorial : setEditorial,
  getAutor : getAutor,
  setAutor : setAutor,
  getFechaPrimeraEdicion : getFechaPrimeraEdicion,
  setFechaPrimeraEdicion : setFechaPrimeraEdicion,
  getIsbn : getIsbn,
  setIsbn : setIsbn
 }
};
```

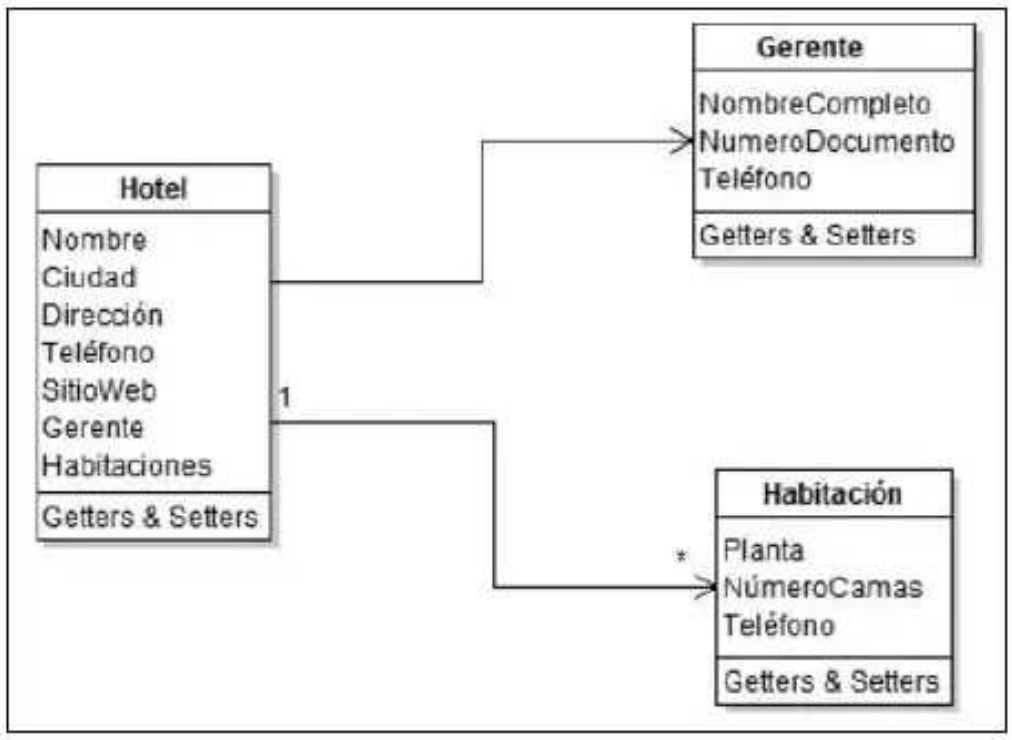
- Fichero principal codigo.js

```
QUnit.test( "Prueba", function( assert ) {
 var autor = new Autor();
 autor.setNombreCompleto('Ismael López Quintero');
 autor.setFechaNacimiento('04/07/1983');
 autor.setNacionalidad('española');
 var libro = new Libro();
 libro.setTitulo('Aprendiendo Notación JSON');
 libro.setEditorial('Publicaciones Universitarias SL');
 libro.setAutor(autor);
 libro.setFechaPrimeraEdicion('01/01/2012');

 libro.setIsbn('123456789');
 /*
 Aquí iría otro código, como visualización en la interfaz de usuario,
 acceso a datos... Vamos a recuperar los datos de la estructura
 creada y a hacer aserciones.
 */
 var tituloLibro = libro.getTitulo();
 var editorialLibro = libro.getEditorial();
 var autorLibro = libro.getAutor();
 var nombreCompletoAutor = autorLibro.getNombreCompleto();
 var fechaNacimientoAutor = autorLibro.getFechaNacimiento();
 var nacionalidadAutor = autorLibro.getNacionalidad();
 var fechaPrimeraEdicionLibro = libro.getFechaPrimeraEdicion();
 var isbnLibro = libro.getIsbn();

 assert.equal(tituloLibro,'Aprendiendo Notación JSON',
 'Título correcto');
 assert.equal(editorialLibro,'Publicaciones Universitarias
 SL','Editorial correcta');
 assert.equal(fechaPrimeraEdicionLibro,'01/01/2012','Fecha Primera
 Edición correcta');
 assert.equal(isbnLibro,'123456789','ISBN correcto');
 assert.equal(nombreCompletoAutor,'Ismael López Quintero','Nombre del
 autor correcto');
 assert.equal(fechaNacimientoAutor,'04/07/1983','Fecha de nacimiento
 del autor correcta');
 assert.equal(nacionalidadAutor,'española','Nacionalidad del autor
 correcta');
});
```

Para implementar las clases, usamos una técnica conocida en JavaScript como cierre o "closure", que se verá más adelante en este capítulo. También se ha tenido una aproximación al ámbito de las funciones, mediante el uso de la variable sThis.

2.3.1 Ejercicio 1

Como ejercicio, se propone escribir los ficheros HTML y JavaScript necesarios para implementar un hotel, que se corresponde con el siguiente diagrama de clases:



En el código del programa principal se deben hacer las aserciones adecuadas para comprobar todos los datos del hotel, los datos del gerente, así como los datos de tres habitaciones.

2.4 Ámbitos

Una de las peculiaridades de JavaScript a la que puede que muchos programadores no estén acostumbrados es la del ámbito de declaración de las variables. En otros lenguajes, tales como Java o PHP, el ámbito de declaración es el del bloque en el que esté definido. Veamos un ejemplo de código Java:

```
public class Programa {
 private static int i = 1;
 public static void main(String[] args) {
  if (i>0) {
   int i=3;
   System.out.println("El valor de i es "+i+".");
  }
  System.out.println("El valor de i es "+i+".");
 }
}
```

Si ejecutamos dicho programa, la salida por consola será la siguiente:

```
El valor de i es 3.
El valor de i es 1.
```

Se aprecia que el ámbito de declaración de la variable es el del bloque del código en el que está definida. En JavaScript, en cambio, no ocurre esto. La palabra reservada **var** usa el ámbito de función, en vez del ámbito de bloque. Veamos un código, aparentemente similar, en JavaScript. Este ejemplo lo ejecutaremos sin pruebas unitarias, para ver en el navegador el valor de las variables:

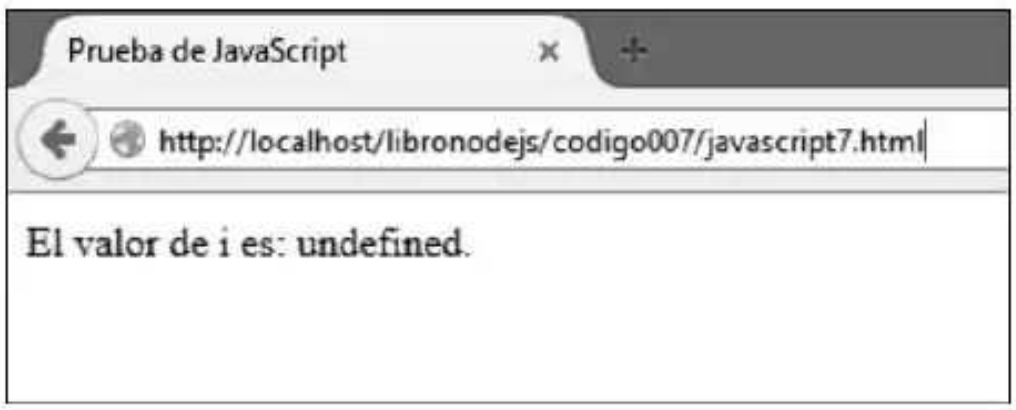
- Código HTML:

```
<html>
<head>
<meta charset="utf-8">
<title>Prueba de JavaScript</title>
</head>
<body>
<script src="./codigo.js"></script>
</body>

</html>
```

- Código JavaScript. Fichero codigo.js:

```
var i = 1;
(function main() {
 if (i>0) {
  var i = 3;
  document.writeln("El valor de i es: "+i+". <br />");
 }
 document.writeln("El valor de i es: "+i+". <br />");
})();
```

Probablemente muchos estemos esperando que el resultado de la ejecución sea exactamente el mismo que el resultado de la ejecución del código Java anterior. Sin embargo, el navegador nos muestra la siguiente salida:



¿Qué ocurre? ¿No se tiene dentro de la función acceso al valor de la variable definida en el ámbito superior? Puede que sí o puede que no. Vamos a probar a comentar la declaración de la variable dentro de la condicional:

```
var i = 1;
(function main() {
 if (i>0) {
  //var i = 3;
  document.writeln("El valor de i es: "+i+". <br />");
 }
 document.writeln("El valor de i es: "+i+". <br />");
})();
```

Ahora lo que nos muestra el navegador es lo siguiente:

Vemos que sí se tiene acceso a la variable exterior. Entonces, ¿qué es lo que está pasando? Están ocurriendo dos peculiaridades:

- El ámbito de declaración de las variables, como ya se ha comentado, es el de la función, pero, evidentemente, si en dicha función no se ha definido ninguna variable con el nombre de la variable de ámbito superior, la variable existe con el mismo identificador y valor que tenía en el ámbito superior.
- En JavaScript existe un procedimiento de "alzada" ó "hoisting" de las variables. El intérprete realiza una doble pasada por las funciones, en búsqueda de todas las variables que se definen en ella (la función). Entonces, de forma interna, traduce una declaración de variable a "no definida" al comienzo de dicha función. El código inicial, por lo tanto, y una vez hecha la primera pasada del intérprete, es similar a éste:

```
var i = 1;
(function main() {
 var i;
 if (i>0) {
  i = 3;
  document.writeln("El valor de i es: "+i+". <br />");
 }
 document.writeln("El valor de i es: "+i+". <br />");
})();
```

El procedimiento de alzada que se ha mencionado es el que hace que cuando se realiza la comprobación del valor, este valor sea "undefined", y, por lo tanto, no se pueda entrar en el bloque condicional. Tras el condicional tenemos el document.writeln() que se nos muestra en pantalla.

Otra peculiaridad de JavaScript que ha aparecido en el ejemplo anterior es que permite la implementación de funciones autoinvocadas. Para lanzar una función

autoinvocada, simplemente hay que rodearla de paréntesis y acto seguido, ponerle los paréntesis de los parámetros, que son inexistentes en este caso. El esquema es éste:

```
(function identificador(){
})();
```

Es más, este tipo de funciones, al ser inmediatamente autoinvocadas, pueden prescindir de identificador. O sea, las funciones autoinvocadas pueden ser funciones anónimas.

```
(function(){
})();
```

Para concluir con el ámbito de las variables, todos los programadores JavaScript nos hemos encontrado alguna vez con el problema de la variable **this**. Este problema suele ser muy común en la gestión de eventos de JavaScript y sus frameworks como jQuery. Pero como nos hemos topado ya con este problema en lo que llevamos de texto y estamos en la sección correcta, vamos a pasar a explicarlo.

Recordemos la declaración de la clase Autor que hemos visto en el apartado de Notación JSON.

```
var Autor = function() {
 var sThis = this;
 this.datosAutor = {
  nombreCompleto : '',
  fechaNacimiento : '',
  nacionalidad : ''
 };
 var getNombreCompleto = function() {
  return sThis.datosAutor.nombreCompleto;
 },
 setNombreCompleto = function(nombreCompleto) {
  sThis.datosAutor.nombreCompleto=nombreCompleto;
 },
  // ... resto de getters & setters.
return {
  getNombreCompleto : getNombreCompleto,
  setNombreCompleto : setNombreCompleto,
 // ... asignamos el resto de campos JSON a los getters & setters.
  }
};
```

Tras la declaración de la clase vemos la declaración **var sThis = this**. Con sThis simplemente se quiere indicar self This, o sea, una referencia a la propia clase, cuando el ámbito de ejecución cambie y el acceso a los datos no sea posible con **this** porque dicha palabra reservada referencie a otra clase u objeto. En nuestro caso concreto, cuando hagamos autor.getNombreCompleto(), **this** hará referencia al objeto JSON devuelto, por lo que **this** contendrá los campos getNombreCompleto, setNombreCompleto, getFechaNacimiento, setFechaNacimiento, getNacionalidad y setNacionalidad, que son los que se han incluido en el objeto JSON devuelto. Campos, dicho sea de paso, que son funciones. Pero el objeto **this** no tiene acceso a datosAutor ya que no se introdujo

en la instrucción **return**. Precisamente para salvaguardar este escollo se escribe la instrucción **var sThis = this**. De esta forma tenemos en el propio objeto autor una referencia permanente a la propia clase que me permite acceder a todos sus campos. Para más detalle, ver el apartado "cierre".

2.5 Lambdas

Una lambda en JavaScript no es ni más ni menos que una función que se comporta como si fueran datos. Y abundan mucho. Las funciones pueden ser pasadas como parámetros, pueden ser devueltas en instrucciones **return**, pueden ser usadas como parte de operaciones aritméticas o lógicas (esto último también es común en otros lenguajes). JavaScript es un lenguaje tan débilmente tipado que una variable puede contener prácticamente "cualquier cosa": tipos primitivos, arrays, strings, funciones, objetos JSON, todo ello sin especificar el tipo, simplemente asignando una variable a un tipo u otro.

Veamos un ejemplo con lambdas. Le pasaremos a una función, dos funciones como parámetros. La función que recibe las lambdas hará una comprobación, y, dependiendo del resultado, ejecutará una de las dos funciones recibidas como parámetro.

- Código HTML:

```
<html>
<head>
<meta charset="utf-8">
<title>Prueba de JavaScript</title>
<link rel="stylesheet" href="../qunit-1.16.0.css">
</head>
<body>
<div id="qunit"></div>
<div id="qunit-fxture"></div>
<script src="../qunit-1.16.0.js"></script>
<script src="./codigo.js"></script>
</body>
</html>
```

- Código JavaScript (codigo.js):

```
var saludo = function(idioma) {
 var cadena;
 switch(idioma) {
 case 'espanol':
  cadena = 'Hola, ¿qué tal?';
  break;
 case 'ingles':
  cadena = 'Hello, how are you?'
  break;
 case 'frances':
  cadena = 'salut! comment ça va?';
  break;
```

```
 }
 return cadena;
}
var despedida = function(idioma) {
 var cadena;
 switch(idioma) {
 case 'espanol':
  cadena = '¡Hasta luego!';
  break;
 case 'ingles':
  cadena = 'See you later!';
  break;
 case 'frances':
  cadena = 'à tout à l\'heure';
  break;
 }
 return cadena;
}
function funcionPrincipal(valor,umbral,funcion1,funcion2,idioma) {
 var cadena;
 if (valor>=umbral) {
  cadena = funcion1(idioma);
 } else {
  cadena = funcion2(idioma);
 }
 return cadena;
}
QUnit.test( "Prueba", function( assert ) {
 // Saludo en inglés.
 var cadena = funcionPrincipal(1,0,saludo,despedida,'ingles');
 assert.equal(cadena,'Hello, how are you?','Saludo en inglés
 correcto');
 // Despedida en francés.
 cadena = funcionPrincipal(0,1,saludo,despedida,'frances');
 assert.equal(cadena,'à tout à l\'heure','Despedida en francés
 correcta');
 // Despedida en español.
 cadena = funcionPrincipal(0,1,saludo,despedida,'espanol');
 assert.equal(cadena,'¡Hasta luego!','Despedida en español correcta');
});
```

Todas las aserciones anteriores son correctas.

Hemos visto el ejemplo de una función que toma otras dos como parámetro. Veamos ahora el ejemplo de una función que devuelve otra función como si fuera un dato.

El código HTML es el mismo que en el ejemplo anterior. El fichero codigo.js queda como sigue:

```
var factorial = function(n) {
 if (n % 1 == 0) {
  if (n>0) {
   return n*factorial(n-1);
```

```
  } else {
   return 1;
  }
 } else {
  return -1;
 }
}
var cubo = function(n) {
 return Math.pow(n,3);
}
function funcionPrincipal(valor,umbral) {
 if (valor>=umbral) {
  return factorial;
 } else {
  return cubo;
 }
}
QUnit.test( "Prueba", function( assert ) {
 var mifuncion = funcionPrincipal(1,0);
 assert.equal(mifuncion(5),120,"El factorial de 5 es 120");
 var mifuncion = funcionPrincipal(0,1);
 assert.equal(mifuncion(5),125,"El resultado de elevar 5 al cubo es
 125");
});
```

JavaScript es tan versátil que permite usar funciones anónimas como lambdas. Veamos los mismos ejemplos anteriores con funciones anónimas.

- **Primer ejemplo**: funciones anónimas como parámetros.

```
function funcionPrincipal(valor,umbral,funcion1,funcion2,idioma) {
 var cadena;
 if (valor>=umbral) {
  cadena = funcion1(idioma);
 } else {
  cadena = funcion2(idioma);
 }
 return cadena;
}
QUnit.test("Prueba", function( assert ) {
 // Saludo en inglés.
 var cadena = funcionPrincipal(1,0,function(idioma){
  var cadena;
  switch(idioma) {
  case 'espanol':
   cadena = 'Hola, ¿qué tal?';
   break;
  case 'ingles':
   cadena = 'Hello, how are you?'
   break;
  case 'frances':
   cadena = 'salut! comment ça va?';
   break;
```

```
  }
  return cadena;
 },function(idioma){
  var cadena;
  switch(idioma) {
  case 'espanol':
   cadena = '¡Hasta luego!';
   break;
  case 'ingles':
   cadena = 'See you later!'
   break;
  case 'frances':
   cadena = 'à tout à l\'heure';
   break;
  }
  return cadena;
 },'ingles');
 assert.equal(cadena,'Hello, how are you?','Saludo en inglés
 correcto');
});
```

- **Segundo ejemplo**: funciones anónimas en instrucciones **return**.

```
function funcionPrincipal(valor,umbral) {
 if (valor>=umbral) {
  return function factorial(n) {
     if (n % 1 == 0) {
      if (n>0) {
       return n*factorial(n-1);
      } else {
       return 1;
      }
     } else {
      return -1;
     }
  };
 } else {
  return function(n) {
   return Math.pow(n,3);
  };
 }
}
QUnit.test( "Prueba", function( assert ) {
 var mifuncion = funcionPrincipal(1,0);
 assert.equal(mifuncion(5),120,"El factorial de 5 es 120");
 var mifuncion = funcionPrincipal(0,1);
 assert.equal(mifuncion(5),125,"El resultado de elevar 5 al cubo es
 125");
});
```

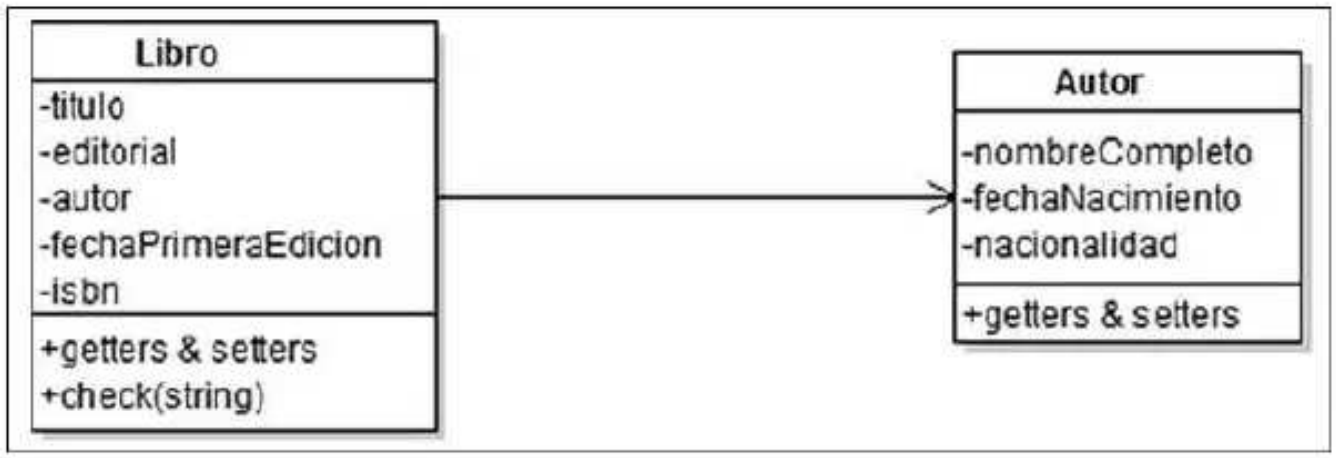
La función factorial necesita un identificador para poder llamarla mediante recursión. Si usamos un identificador para la función del tipo **var** a = **function**(), ya no

será una función anónima. No ocurre esto con la función cubo, que es "anónima pura", debido a que no necesita recursión.

Vamos a retomar nuestro ejemplo de la biblioteca visto en el apartado Notación JSON y vamos a hacer que las comprobaciones de los campos de cada libro los realice cada clase concreta. Hablando de la arquitectura de *software*, existen comprobaciones que debe realizar el controlador. En muchas aplicaciones, los objetos del modelo del dominio están pensados para ser "objetos tontos", destinados a tener únicamente los datos y los getters & setters. Pero ¿debemos sobrecargar de responsabilidades al controlador de la aplicación encargándole que conozca (que tenga el código) de comprobación de todos los campos de todas las clases? ¿No será mejor que dicha comprobación la realice el controlador (como esperamos), pero que dicho código lo tenga la clase concreta en este caso? Tenemos dos soluciones a este problema. Vamos a trasladar la cuestión a nuestra biblioteca, en particular a la clase Libro:

- Insertar en la clase Libro un método que se llame check*Campo*(), que devuelva **true** o **false**. Este método será ejecutado por el controlador. Tendremos un método check por cada campo que queramos testear.
- Insertar en la clase Libro un método check(string). El string que le pasemos será el identificador del campo que queremos testear. La clase nos devolverá la función (una lambda) que contendrá el código cuya lógica testea el campo concreto.

Hagamos un refactoring al diagrama de clases de la biblioteca, para introducir el método check.



Vamos a implementar el caso en el que el único campo que necesita comprobación va a ser el ISBN. Trabajaremos, por defecto, con el ISBN de 13 dígitos, dispuestos de la siguiente forma: DDD-DD-DDDDDD-D-D. Donde las "D" son dígitos del 0 al 9. Vamos a considerar que los demás campos del modelo del dominio: título, editorial, fecha de primera edición; y los atributos de la clase autor: nombre completo, fecha de nacimiento y nacionalidad, son correctos, contengan lo que contengan; aunque evidentemente, podríamos hacer que la fecha tuviera una sintaxis concreta. Para no alargar demasiado el texto y los ejemplos, hagamos la comprobación del ISBN únicamente.

- Veamos cómo queda el código de la clase Libro (fichero libro.js):

```
// Implementación de la clase Libro con getters y setters.
var Libro = function() {
 var sThis = this;
```

```
this.datosLibro = {
 titulo : '',
 editorial : '',
 autor : {},
 fechaPrimeraEdicion : '',
 isbn : ''
};
var getTitulo = function() {
 return sThis.datosLibro.titulo;
},
setTitulo = function(titulo) {
 sThis.datosLibro.titulo = titulo;
},
getEditorial = function() {
 return sThis.datosLibro.editorial;
},
setEditorial = function(editorial) {
 sThis.datosLibro.editorial = editorial;
},
getAutor = function() {
 return sThis.datosLibro.autor;
},
setAutor = function(autor) {
 sThis.datosLibro.autor = autor;
},
getFechaPrimeraEdicion = function() {
 return sThis.datosLibro.fechaPrimeraEdicion;
},
setFechaPrimeraEdicion = function(fechaPrimeraEdicion) {
 sThis.datosLibro.fechaPrimeraEdicion = fechaPrimeraEdicion;
},
getIsbn = function() {
 return sThis.datosLibro.isbn;
},
setIsbn = function(isbn) {
 sThis.datosLibro.isbn=isbn;
},
check = function(campo) {
 if ((campo) && (campo!=='')) {
  if (campo!=='isbn') {
   return function() {
    return true;
   }
  } else {
   return function() {
    var isbn = sThis.datosLibro.isbn;
    var partesIsbn = isbn.split('-');
    var nPartes = partesIsbn.length;
    if (nPartes!==5) {
     return false;
    } else {
```

```
      var valido = true;
      for(var i=0;i<nPartes;i++) {
       var estaParte = partesIsbn[i];
       // Usamos expresión regular.
       if (!/^([0-9])*$/.test(estaParte)) {
        valido = false;
        break;
       }
      }
      return valido;
     }
    }
   }
  } else {
   return function() {
    return true;
   }
  }
};
return {
 getTitulo : getTitulo,
 setTitulo : setTitulo,
 getEditorial : getEditorial,
 setEditorial : setEditorial,
 getAutor : getAutor,
 setAutor : setAutor,
 getFechaPrimeraEdicion : getFechaPrimeraEdicion,
 setFechaPrimeraEdicion : setFechaPrimeraEdicion,
 getIsbn : getIsbn,
 setIsbn : setIsbn,
 check : check
 }
};
```

Apreciamos claramente que lo que la función check devuelve no son valores, sino funciones, que bien pueden ser anónimas o no. En esta implementación en concreto son anónimas. Precisamente debido al uso de funciones anónimas tenemos la sensación de tener el antipatrón "espagueti". Veamos cómo queda la aplicación (codigo.js):

```
QUnit.test( "Prueba", function( assert ) {
 var autor = new Autor();
 autor.setNombreCompleto('Ismael López Quintero');
 autor.setFechaNacimiento('04/07/1983');
 autor.setNacionalidad('española');
 var libro = new Libro();
 libro.setTitulo('Aprendiendo Notación JSON');
 libro.setEditorial('Publicaciones Universitarias SL');
 libro.setAutor(autor);
 libro.setFechaPrimeraEdicion('01/01/2012');
 // Vamos a jugar con los getters y los setters y a probar ISBN´s.
 // Primero con un ISBN incorrecto.
 libro.setIsbn('123456789');
```

```
var isbnLibro = libro.getIsbn(); // Este get no es necesario.
var comprobacion = libro.check('isbn');
var isbnCorrecto = comprobacion();
// Esta primera aserción debe fallar porque no se ajusta al formato.
assert.ok(isbnCorrecto,'El ISBN del libro es correcto');
// Vamos a introuducir un ISBN que sí se ajusta al formato.
libro.setIsbn('978-15-678213-8-0');
// Extraemos la función comprobación.
comprobacion = libro.check('isbn');
isbnCorrecto = comprobacion();
// Esta segunda aserción debe de ser correcta.
assert.ok(isbnCorrecto,'El ISBN del libro es correcto');
// Realizamos la comprobación de cualquier otro campo.
// Devuelve true porque no lo hemos implementado.
comprobacion = libro.check('fechaPrimeraEdicion');
var fechaPrimeraEdicionCorrecta = comprobacion();
// Esta aserción debe ser correcta porque no la hemos implementado.
assert.ok(fechaPrimeraEdicionCorrecta,'La fecha de la primera edición
del libro es correcta');
});
```

2.5.1 Ejercicio 2

Se plantea al lector modificar el ejercicio 1 (del hotel) para realizar la comprobación de los siguientes campos:

- El teléfono del hotel, del gerente y de cada habitación deben de ajustarse al siguiente formato:

+DD. DDDDDDDDD

Donde D son dígitos del 0 al 9.

- El sitio web del hotel debe tener la siguiente nomenclatura:

$$\text{http://www.C}^{+}\text{.C}^{2-3}$$

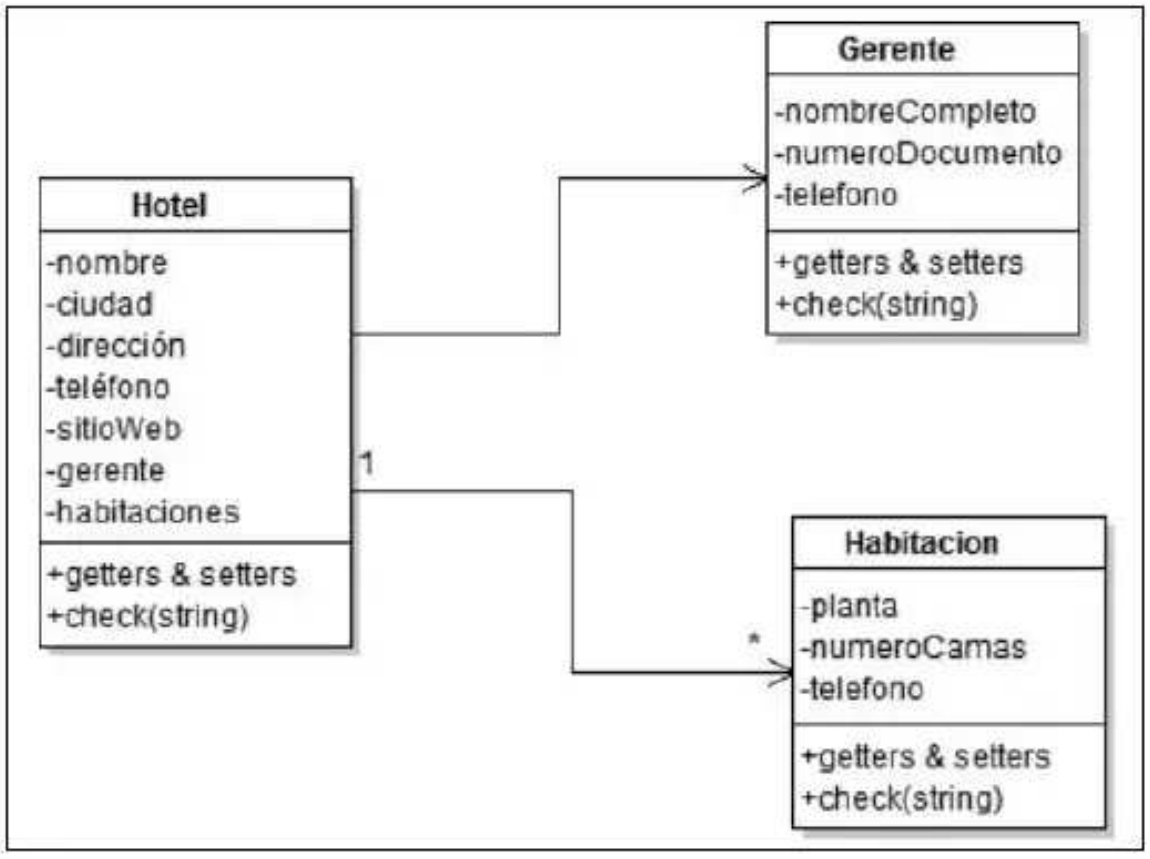
Donde C es cualquier carácter del alfabeto inglés (caracteres del intervalo a-z) excluidos los números. Debemos tener al menos un carácter. La extensión del identificador de dominio debe tener dos o tres caracteres.

- El número de documento se corresponde a la siguiente nomenclatura:

DDDDDDDD-C

Donde D son dígitos del 0 al 9 y C es un carácter en el intervalo A-Z. Para comprobar que la letra del documento es correcta, usar el siguiente algoritmo:

```
function letraDocumento(documento) {
 var caracterCorrecto = 'TRWAGMYFPDXBNJZSQVHLCKE';
     return caracterCorrecto.charAt(documento % 23);
}
```



2.6 Cierres

También denominados "closures", se usan para encapsular el estado de un objeto o función (recordemos que en JavaScript son lo mismo), incluso tras la ejecución de la función. Ya nos hemos encontrado con este problema en lo que llevamos de texto, pero vamos a pasar a ver aquí varios ejemplos, para conocer mejor el problema y saber cuándo tenemos que aplicar un cierre.

Imaginemos la clase "Persona":

Tenemos dos formas de implementar dicha clase: añadiendo métodos al prototipo de la función o mediante cierres.

Veamos la implementación añadiendo métodos al prototipo:

```
// Implementar la clase Persona usando el prototype de la función.
var Persona = function() {
 this.nombreCompleto = '';
 this.domicilio = '';
 this.fechaNacimiento = '';
};
```

```
Persona.prototype.getNombreCompleto = function() {
 return this.nombreCompleto;
}
Persona.prototype.setNombreCompleto = function(nombreCompleto) {
 this.nombreCompleto = nombreCompleto;
}
Persona.prototype.getDomicilio = function() {
 return this.domicilio;
}
Persona.prototype.setDomicilio = function(domicilio) {
 this.domicilio = domicilio;
}
Persona.prototype.getFechaNacimiento = function() {
 return this.fechaNacimiento;
}
Persona.prototype.setFechaNacimiento = function(fechaNacimiento) {
 this.fechaNacimiento = fechaNacimiento;
}
```

Vemos que en cada método podemos usar **this**, ya que el objeto devuelto es la propia función Persona. El fichero codigo.js es el siguiente:

```
QUnit.test( "Prueba", function( assert ) {
 var p = new Persona();
 p.setNombreCompleto('Javier Pérez Contreras');
 p.setDomicilio('C/ Calle N.1 CP: 21001. Huelva.');
 p.setFechaNacimiento('01/01/1970');

 var nombreCompleto = p.getNombreCompleto();
 var domicilio = p.getDomicilio();
 var fechaNacimiento = p.getFechaNacimiento();

 assert.equal(nombreCompleto,'Javier Pérez Contreras','El nombre es
 correcto');
 assert.equal(domicilio,'C/ Calle N.1 CP: 21001. Huelva.','El domici-
 lio es correcto');
 assert.equal(fechaNacimiento,'01/01/1970','La fecha de nacimiento es
 correcta');
});
```

Las anteriores aserciones son correctas. Sin embargo, la anterior ejecución presenta dos problemas:

- Conlleva mucha repetición. Para cada nuevo método que queramos añadir hay que poner previamente **Persona.prototype**.
- El acceso a los atributos **no es privado**. En el siguiente código podemos ver que podemos acceder sin problemas a los atributos de la clase sin necesidad de los getters (modificación del fichero codigo.js).

```
QUnit.test( "Prueba", function( assert ) {
 var p = new Persona();
 p.setNombreCompleto('Javier Pérez Contreras');
 p.setDomicilio('C/ Calle N.1 CP: 21001. Huelva.');
 p.setFechaNacimiento('01/01/1970');
 var nombreCompleto = p.nombreCompleto;
 var domicilio = p.domicilio;
 var fechaNacimiento = p.fechaNacimiento;
 assert.equal(nombreCompleto,'Javier Pérez Contreras','El nombre es
 correcto');
 assert.equal(domicilio,'C/ Calle N.1 CP: 21001. Huelva.','El domici-
 lio es correcto');
 assert.equal(fechaNacimiento,'01/01/1970','La fecha de nacimiento es
 correcta');
});
```

Las anteriores aserciones son correctas y hemos accedido, desde fuera de la clase, a los atributos privados. Desde el punto de vista de la POO, esto no es correcto, ya que estamos rompiendo el principio de encapsulamiento.

Cuando hemos hecho **new** Persona(), al no tener la función Persona ninguna instrucción **return**, el objeto que estamos obteniendo es la propia función con los 3 atributos. Si hiciéramos un **return** tal y como ya hicimos en el apartado de lambdas, el objeto que obtenemos no es el objeto Persona, sino el objeto JSON que devuelve la instrucción **return**, que hará visible lo que queramos (lo hará público). Lo más importante de todo esto, es que los métodos siguen teniendo acceso a las variables de orden superior, definidas dentro de la clase. En este caso, tienen acceso a la variable sThis.

```
// Implementación de la clase Persona usando un cierre o closure.
var Persona = function() {
 var sThis = this;
 this.datosPersona = {
  nombreCompleto : '',
  domicilio : '',
  fechaNacimiento : ''
 };
 var getNombreCompleto = function() {
  return sThis.datosPersona.nombreCompleto;
 },
 setNombreCompleto = function(nombreCompleto) {
  sThis.datosPersona.nombreCompleto=nombreCompleto;
 },
 getDomicilio = function() {
  return sThis.datosPersona.domicilio;
 },
 setDomicilio = function(domicilio) {
  sThis.datosPersona.domicilio=domicilio;
 },
 getFechaNacimiento = function() {
  return sThis.datosPersona.fechaNacimiento;
```

```
 },
 setFechaNacimiento = function(fechaNacimiento) {
  sThis.datosPersona.fechaNacimiento = fechaNacimiento;
 };
 return {
  getNombreCompleto : getNombreCompleto,
  setNombreCompleto : setNombreCompleto,
  getDomicilio : getDomicilio,
  setDomicilio : setDomicilio,
  getFechaNacimiento : getFechaNacimiento,
  setFechaNacimiento : setFechaNacimiento
 }
};
```

- Fichero codigo.js:

```
QUnit.test( "Prueba", function( assert ) {
 var p = new Persona();
 p.setNombreCompleto('Javier Pérez Contreras');
 p.setDomicilio('C/ Calle N.1 CP: 21001. Huelva.');
 p.setFechaNacimiento('01/01/1970');
 var nombreCompleto = p.getNombreCompleto();
 var domicilio = p.getDomicilio();
 var fechaNacimiento = p.getFechaNacimiento();
 assert.equal(nombreCompleto,'Javier Pérez Contreras','El nombre es
 correcto');
 assert.equal(domicilio,'C/ Calle N.1 CP: 21001. Huelva.','El
 domicilio es correcto');
 assert.equal(fechaNacimiento,'01/01/1970','La fecha de nacimiento
 es correcta');
});
```

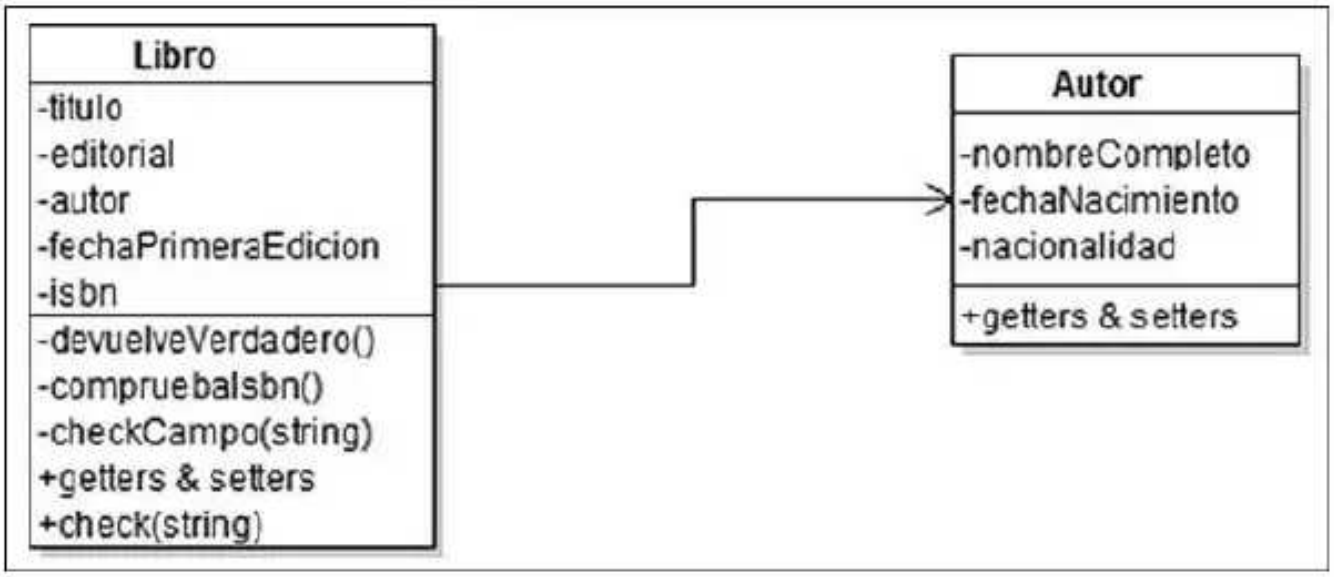
Vamos a intentar acceder desde fuera de la clase a los atributos de la clase Persona. Vamos a rodear el acceso de try-catch porque el programa va a fallar en tiempo de ejecución.

```
QUnit.test( "Prueba", function( assert ) {
 var p = new Persona();
 p.setNombreCompleto('Javier Pérez Contreras');
 p.setDomicilio('C/ Calle N.1 CP: 21001. Huelva.');
 p.setFechaNacimiento('01/01/1970');
 try {
  var nombreCompleto = p.datosPersona.nombreCompleto;
  var domicilio = p.datosPersona.domicilio;
  var fechaNacimiento = p.datosPersona.fechaNacimiento;
  assert.equal(nombreCompleto,'Javier Pérez Contreras','El nombre
  es correcto');
  assert.equal(domicilio,'C/ Calle N.1 CP: 21001. Huelva.','El
  domicilio es correcto');
  assert.equal(fechaNacimiento,'01/01/1970','La fecha de nacimiento es
  correcta');

 } catch(error) {
```

```
  assert.ok(true,'Se ha producido un error al acceder a los datos
  privados de la clase.')
 }
});
```

A parte del acceso a atributos privados que ya se ha visto, existe una diferencia entre implementar métodos mediante prototipo y hacerlo mediante variables: el rendimiento en memoria. Cuando asignamos un método al prototipo de una función, no estamos reservando memoria por cada instancia de esa función que se crea, sino que se hace uso de la estructura de la función (no debemos olvidar nunca que en JavaScript las clases se implementan mediante funciones). Si lo hacemos mediante variables, por cada instancia de esa función (o sea, por cada instancia de esa clase u objeto), tendremos una reserva de memoria en código. En el caso de que vayamos a tener miles de objetos con métodos de muchas líneas de código, sería interesante replantearse el usar el prototipo de las funciones.

Con los closures, evidentemente, también podemos implementar métodos privados. Vamos a hacer un refactoring de la clase Libro vista anteriormente para que el método check sea un método público, pero para poder funcionar llame a un método privado llamado; por ejemplo, checkCampo. Para hacer la clase más legible, vamos a hacer que las funciones no sean anónimas. El refactoring del diagrama de clases es el siguiente:



Como podemos ver, los métodos públicos serán tanto los getters & setters como el método check. Estos serán los métodos que devolvamos en el objeto JSON de la instrucción **return**. Privadamente tendremos un conjunto de métodos que nos facilitarán la programación y harán el código más legible. Veamos cómo queda.

- Fichero libro.js:

```
// Implementación de la clase Libro con getters y setters.
var Libro = function() {
 var sThis = this;
 this.datosLibro = {
  titulo : '',
  editorial : '',
  autor : {},
```

```
 fechaPrimeraEdicion : '',
 isbn : ''
};
this.devuelveVerdadero = function() {
 return true;
};
this.compruebaIsbn = function() {
 var isbn = sThis.datosLibro.isbn;
 // Esta función se ejecutará desde otro ámbito.
 // Por ello usamos sThis.
 var partesIsbn = isbn.split('-');
 var nPartes = partesIsbn.length;
 if (nPartes!==5) {
  return false;
 } else {
  var valido = true;
  for(var i=0;i<nPartes;i++) {
   var estaParte = partesIsbn[i];
   // Usamos expresión regular.
   if (!/^([0-9])*$/.test(estaParte)) {
    valido = false;
    break;
   }
  }
  return valido;
 }
};
this.checkCampo = function(nombreCampo) {
 if ((nombreCampo) && (nombreCampo!==')) {
  if (nombreCampo==='isbn') {
   // No poner return this.compruebaIsbn();
   // porque si lo hacemos ejecutaremos la función
   // y devolverá el valor.
   // Lo que queremos es que devuelva la función,
   // por lo que hay que quitar los paréntesis de
   // los parámetros.
   return this.compruebaIsbn;
  } else {
   return this.devuelveVerdadero; // no poner ().
  }
 } else {
  return this.devuelveVerdadero; // no poner ().
 }
};
var getTitulo = function() {
 return sThis.datosLibro.titulo;
},
setTitulo = function(titulo) {
 sThis.datosLibro.titulo = titulo;
},
```

```
getEditorial = function() {
 return sThis.datosLibro.editorial;
},
setEditorial = function(editorial) {
 sThis.datosLibro.editorial = editorial;
},
getAutor = function() {
 return sThis.datosLibro.autor;
},
setAutor = function(autor) {
 sThis.datosLibro.autor = autor;
},
getFechaPrimeraEdicion = function() {
 return sThis.datosLibro.fechaPrimeraEdicion;
},
setFechaPrimeraEdicion = function(fechaPrimeraEdicion) {
 sThis.datosLibro.fechaPrimeraEdicion = fechaPrimeraEdicion;
},
getIsbn = function() {
 return sThis.datosLibro.isbn;
},
setIsbn = function(isbn) {
 sThis.datosLibro.isbn=isbn;
 },
 check = function(campo) {
  return sThis.checkCampo(campo);
 };
 return {
  getTitulo : getTitulo,
  setTitulo : setTitulo,
  getEditorial : getEditorial,
  setEditorial : setEditorial,
  getAutor : getAutor,
  setAutor : setAutor,
  getFechaPrimeraEdicion : getFechaPrimeraEdicion,
  setFechaPrimeraEdicion : setFechaPrimeraEdicion,
  getIsbn : getIsbn,
  setIsbn : setIsbn,
  check : check
 }
};
```

Vemos que el código es más fácilmente legible que el anterior. Las funciones anónimas son poderosas en según qué situaciones, pero hacen que el código sea más difícil de entender. Lo que tiene que quedar claro es que en este cierre estamos usando lambdas. Para ser más precisos en este ejemplo:

- La función/clase Libro: es un cierre o closure, ya que conservará el estado una vez hayamos ejecutado la función.

- Las funciones devuelveVerdadero() y compruebaIsbn() son lambdas. Son funciones que tratamos como si fueran datos. En este caso, como resultado de una función (instrucción **return**).

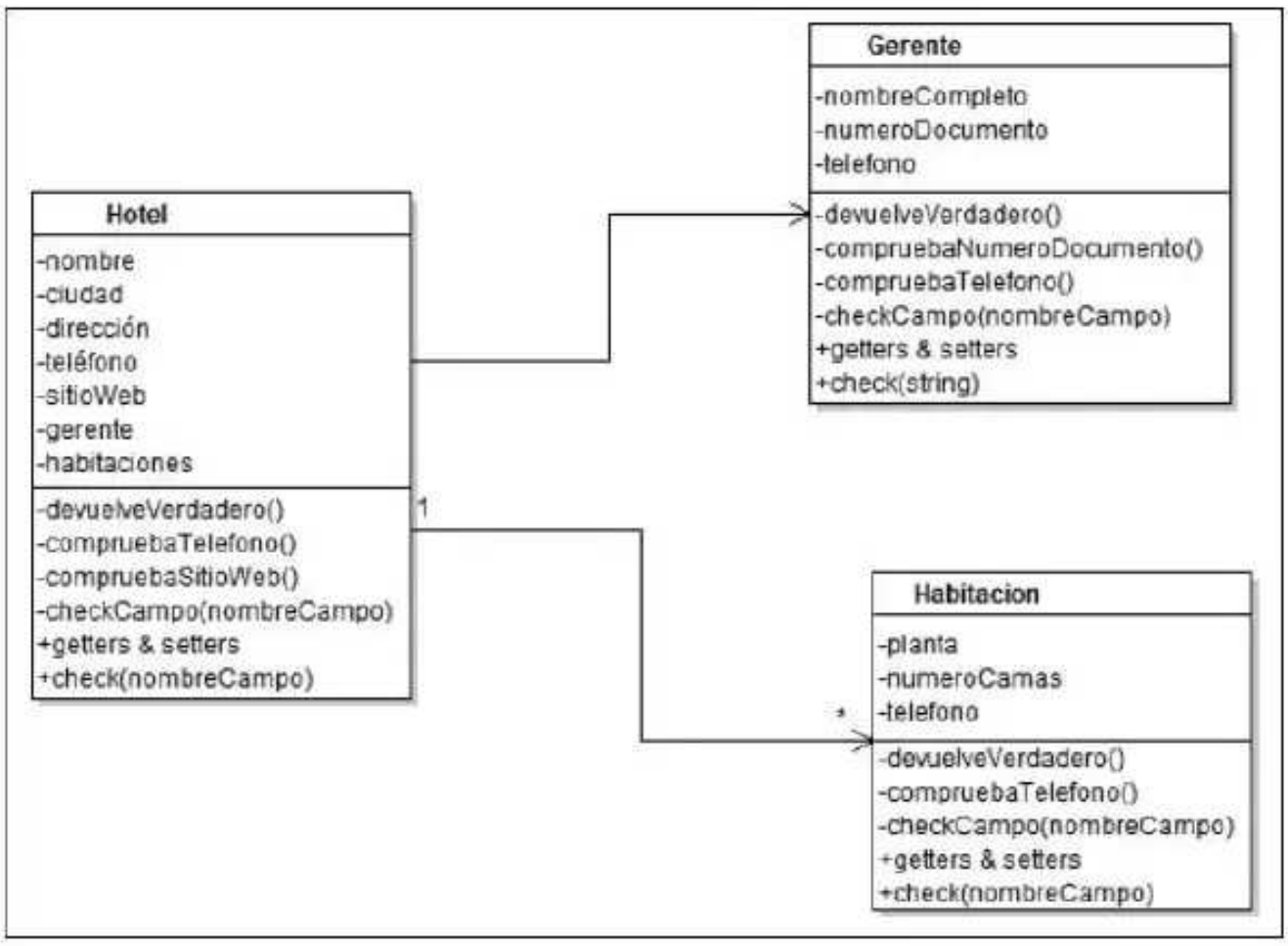
Volvamos a escribir el código que prueba la clase Libro:

```
QUnit.test( "Prueba", function( assert ) {
 var autor = new Autor();
 autor.setNombreCompleto('Ismael López Quintero');
 autor.setFechaNacimiento('04/07/1983');
 autor.setNacionalidad('española');
 var libro = new Libro();
 libro.setTitulo('Aprendiendo Notación JSON');
 libro.setEditorial('Publicaciones Universitarias SL');
 libro.setAutor(autor);
 libro.setFechaPrimeraEdicion('01/01/2012');
 // Vamos a jugar con los getters y los setters y a probar ISBN´s.
 // Primero con un ISBN incorrecto.
 libro.setIsbn('123456789');
 var isbnLibro = libro.getIsbn(); // Este get no es necesario.
 var comprobacion = libro.check('isbn');
 var isbnCorrecto = comprobacion();
 // Esta primera aserción debe fallar porque no se ajusta al formato.
 assert.ok(isbnCorrecto,'El ISBN del libro es correcto');
 // Vamos a introuducir un ISBN que sí se ajusta al formato.
 libro.setIsbn('978-15-678213-8-0');
 // Extraemos la función comprobación.
 comprobacion = libro.check('isbn');
 isbnCorrecto = comprobacion();
 // Esta segunda aserción debe de ser correcta.
 assert.ok(isbnCorrecto,'El ISBN del libro es correcto');
 // Realizamos la comprobación de cualquier otro campo.
 // Devuelve true porque no lo hemos implementado.
 comprobacion = libro.check('fechaPrimeraEdicion');
 var fechaPrimeraEdicionCorrecta = comprobacion();

 // Esta aserción es correcta porque no la hemos implementado.
 assert.ok(fechaPrimeraEdicionCorrecta,'La fecha de la primera
 edición del libro es correcta');
});
```

Y vemos que la salida es exactamente la misma que esperamos (la misma de antes).

2.6.1 Ejercicio 3

Hacer un refactoring del ejemplo del hotel para incluir métodos privados a las tres clases del diagrama, en particular, los métodos que se muestran en el diagrama de clases:



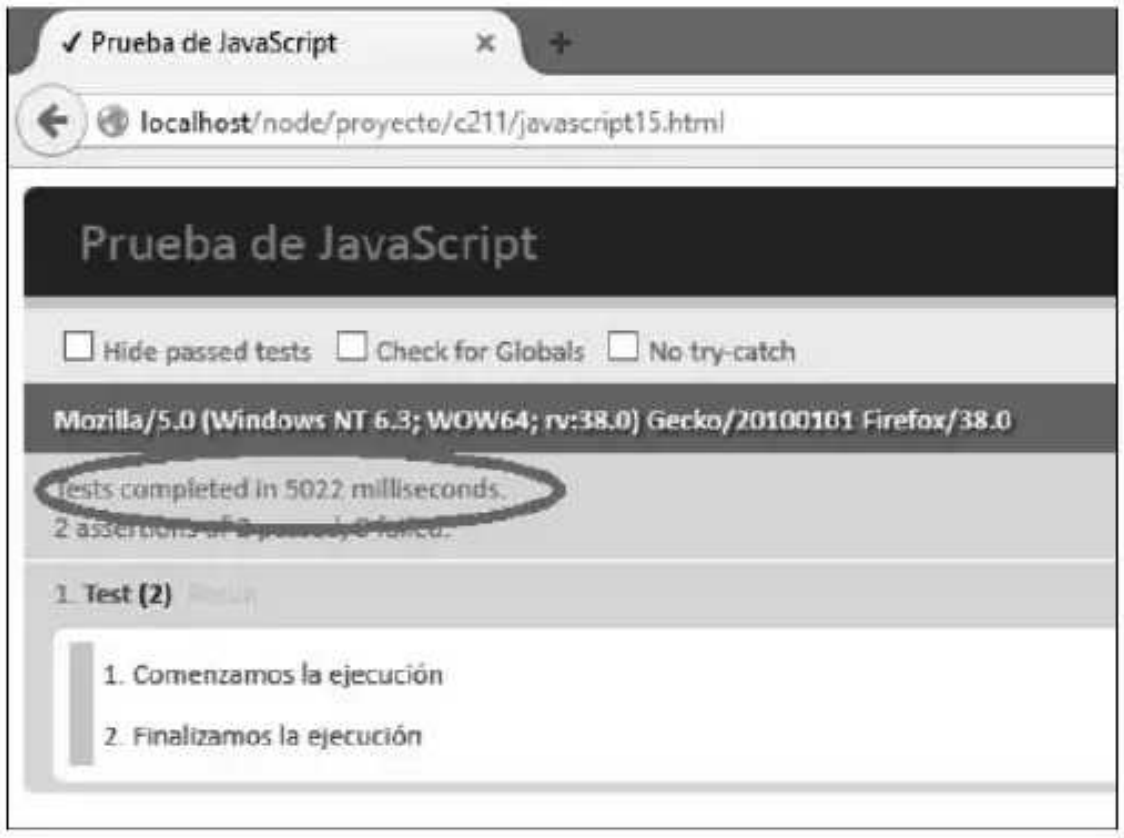
Del mismo modo, implementar las pruebas unitarias necesarias para comprobar el correcto funcionamiento de la implementación.

2.7 Callbacks

Un callback no es ni más ni menos que una lambda (una función que se ha pasado como parámetro), y que se ejecutará una vez que la función de orden superior (la que recibe la lambda) se haya ejecutado. Veamos un ejemplo de callback, que se usa mucho en JavaScript en las operaciones relacionadas con tiempo:

```
QUnit.test("Test",function(assert){
 var done = assert.async();
 setTimeout(function(){
  assert.ok(true,'Finalizamos la ejecución');
  done();
 },5000);

 assert.ok(true,'Comenzamos la ejecución');
});
```

Transcurridos los cinco segundos, la salida de este programa es la siguiente:



En este conciso ejemplo, la función setTimeout es la función de orden superior que recibe la lambda. La llamada a la lambda, una vez que ha transcurrido el tiempo de espera, es un callback. Dicho de otra forma, es una función recibida por la función de orden superior y que será llamada por la función de orden superior una vez que ésta finalice. No tenemos acceso al código de la función setTimeout de JavaScript, pero seguro, al final realizará una llamada callback del siguiente estilo:

```
function setTimeout(callback,esperaMiliSegundos) {
 // Implementa la espera en milisegundos.
 callback();
}
```

¿Qué ventaja nos ofrece el uso del callback? Con QUnit hemos simulado la asincronía que vamos a tener en node.js. En un esquema tradicional síncrono, la espera se gestionaría de la siguiente forma:

```
// Ejemplo en pseudocódigo.
// Ejemplo síncrono.
write('Comenzamos la ejecución');
wait(5000);
write('Finalizamos la ejecución');
```

Con el esquema síncrono tradicional, las instrucciones se ejecutan una tras otra, y sabemos que una instrucción se ejecuta una vez que haya finalizado su ejecución la instrucción precedente. Con el esquema asíncrono no ocurre esto. Sabemos que las instrucciones se lanzarán en el orden secuencial definido, pero no sabemos en qué orden terminarán de ejecutarse. Normalmente, siendo instrucciones aritméticas, lógicas, o llamadas a otras funciones que ejecuten también instrucciones aritméticas o lógicas, el orden de finalización será el mismo que el definido (secuencial). Pero si lo que hacemos son llamadas a servidores de acceso a datos o servicios, entonces sí que no sabremos en qué orden va a acabar cada instrucción ni qué datos o servicios

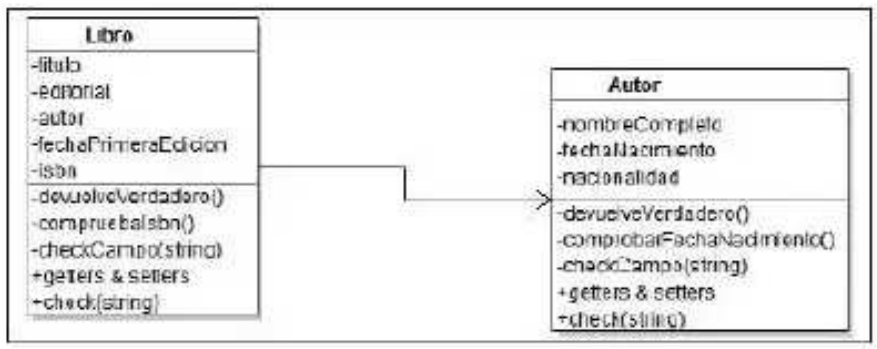
se nos proporcionarán antes o después. Pero, siempre hay instrucciones que deben de ejecutarse, por ejemplo, tras una llamada a un servidor. ¿Cómo garantizamos que dicha ejecución se realice tras la finalización de la llamada, si el método tradicional secuencial no nos sirve? La respuesta es mediante callbacks. Sirven para asegurar sincronía dentro de la asincronía.

Como aún no hemos dado el salto al lado del servidor, cuesta explicar en su contexto un callback. Por lo tanto, vamos a implementar un ejemplo en el que vamos a guardar vía callback los datos recibidos en un formulario, en una estructura de clases. Vamos a implementar un pequeño MVC con nuestra biblioteca de ejemplo en el que la vista será un formulario de recogida de los datos de un autor, el modelo del dominio será la clase Autor, y el controlador será el que procese el volcado de datos desde el formulario hacia el objeto. Lo más importante: usaremos callbacks en el controlador.

Llegados a este punto, es bueno tener acceso a las características de jQuery como framework de JavaScript, ya que nos va a facilitar mucho el acceso al DOM y el procesamiento de eventos. Vamos a instalar jQuery poniéndolo en la raíz de nuestro proyecto. Para ello, accedemos a su sitio:

- http://jquery.com/.

Procedemos primero a hacer un refactoring del modelo del dominio:



Nuestra estructura de carpetas es la siguiente:

Apreciamos que hemos importado el framework de jQuery. Veamos los ficheros.

- Fichero vista.html:

```
<html>
<head>
<meta charset="utf-8">

<title>Prueba de JavaScript</title>
<link rel="stylesheet" href="../qunit-1.16.0.css">
<script src="../qunit-1.16.0.js"></script>
<script src="../jquery-2.1.3.min.js"></script>
<script src="./autor.js"></script>
<script src="./libro.js"></script>
<script src="./vista.js"></script>
<script src="./controlador.js"></script>
</head>
<body>
<div id="qunit"></div>
<div id="qunit-fxture"></div>
<div id="formulario">
 <form>
  Nombre del Autor: <input type="text" id="nombre" name="nombre" />
  <br /> <br />
  Fecha de Nacimiento del Autor: <input type="text" id="fnacimiento"
  name="fnacimiento" /> <br /> <br />
  Nacionalidad del Autor: <input type="text" id="nacionalidad"
  name="nacionalidad" /> <br /> <br />
  <input id="procesar" type="submit" value="Procesar" />
 </form>
</div>
</body>
</html>
```

- Fichero vista.js:

```
$( document ).ready(function() {
 var $sThis = $(this);
 var $procesar = $sThis.find('input#procesar');
 $procesar.click(function(event){
  event.preventDefault();
  controlador($sThis);
 });
});
```

- Modelo del dominio (sólo vamos a trabajar con la clase Autor). Fichero autor.js:

```
// Implementación de la clase Autor con métodos.
var Autor = function() {
 var sThis = this;
 this.datosAutor = {
  nombreCompleto : '',
  fechaNacimiento : '',
  nacionalidad : ''
 };
 this.devuelveVerdadero = function() {
  return true;
 };
 this.comprobarFechaNacimiento = function() {
  var NPARTESCORRECTA=3;
  var fechaNacimiento = sThis.datosAutor.fechaNacimiento;
  var partesFecha = fechaNacimiento.split("/");
  var nPartes = partesFecha.length;
  if (nPartes!==NPARTESCORRECTA) {
   return false;
  }
  var i;
  var valido = true;
  for (i=0;i<nPartes;i++) {
   var estaParte = partesFecha[i];
   if (!/^([0-9])*$/.test(estaParte)) {
    valido = false;
    break;
   }
  }
  if(!valido) {
   return false;
  }
  var dias = partesFecha[0];
  var meses = partesFecha[1];
  var anos = partesFecha[2];
  if ((dias.length===2) && (meses.length===2) && (anos.length===4)) {
   return true;
  } else {
   return false;
  }
```

```
};
this.checkCampo = function(nombreCampo) {
 if ((nombreCampo) && (nombreCampo!==')) {
  if (nombreCampo==='fechaNacimiento') {
   return this.comprobarFechaNacimiento;
  } else {
   return this.devuelveVerdadero;
  }
 } else {
  return this.devuelveVerdadero;
 }
};
var getNombreCompleto = function() {
 return sThis.datosAutor.nombreCompleto;
},
setNombreCompleto = function(nombreCompleto) {
 sThis.datosAutor.nombreCompleto=nombreCompleto;
},
getFechaNacimiento = function() {
 return sThis.datosAutor.fechaNacimiento;
},
setFechaNacimiento = function(fechaNacimiento) {
 sThis.datosAutor.fechaNacimiento=fechaNacimiento;
 },
 getNacionalidad = function() {
  return sThis.datosAutor.nacionalidad;
 },
 setNacionalidad = function(nacionalidad) {
  sThis.datosAutor.nacionalidad=nacionalidad;
 },
 check = function(campo) {
  return sThis.checkCampo(campo);
 }
 return {
  getNombreCompleto : getNombreCompleto,
  setNombreCompleto : setNombreCompleto,
  getFechaNacimiento : getFechaNacimiento,
  setFechaNacimiento : setFechaNacimiento,
  getNacionalidad : getNacionalidad,
  setNacionalidad : setNacionalidad,
  check : check
 }
};
```

- Fichero controlador.js:

```
var controlador = function controlador($document) {
 var $nombre = $document.find('input#nombre');
 var $fNacimiento = $document.find('input#fnacimiento');
 var $nacionalidad = $document.find('input#nacionalidad');
 var nombre = $nombre.val();
 var fNacimiento = $fNacimiento.val();
 var nacionalidad = $nacionalidad.val();
```

```
 var campos = {
  nombre : nombre,
  fNacimiento : fNacimiento,
  nacionalidad : nacionalidad
 };
 // Vamos a hacer la secuencialidad vía callbacks.
 var autor = comprobaciones(campos,crearAutor);
 if(autor) {
  QUnit.test('Probando los datos introducidos',function(assert){
   assert.equal(nombre,autor.getNombreCompleto(),'El nombre es
   correcto');
   assert.equal(fNacimiento,autor.getFechaNacimiento(),'La fecha
   de nacimiento es correcta');
   assert.equal(nacionalidad,autor.getNacionalidad(),'La
   nacionalidad es correcta');
  });
 } else {
  alert('Hay errores en los datos');
 }
}
var comprobaciones = function comprobaciones(campos,callback){
 var nombre = campos.nombre;
 var fNacimiento = campos.fNacimiento;
 var nacionalidad = campos.nacionalidad;
 if ((nombre && nombre!='') && (fNacimiento && fNacimiento!='') &&
 (nacionalidad && nacionalidad!='')) {
  return callback(true,campos);
 } else {
  return callback(false);
 }
}
var crearAutor = function crearAutor(valido,campos) {
 if(valido) {
  var nombre = campos.nombre;
  var fNacimiento = campos.fNacimiento;
  var nacionalidad = campos.nacionalidad;
  var autor = new Autor();
  autor.setNombreCompleto(nombre);
  autor.setFechaNacimiento(fNacimiento);
  autor.setNacionalidad(nacionalidad);
  var comprobacion = autor.check('fechaNacimiento');
  var correcto = comprobacion();
  if(correcto) {
   return autor;
  } else {
   return null;
  }
 } else {
  return null
 }          ;
}
```

Para comprender mejor los callbacks, hemos implementado este ejemplo mediante ellos. Evidentemente esto no es una aplicación web completa. Le falta algo tan importante como la persistencia de datos (acceso a base de datos). Tampoco hemos definido diagramas de secuencia o casos de uso. Simplemente ha sido un ejemplo en el que hemos hecho uso de callbacks, para entender bien qué significan y cómo se implementan.

Cuando se produce el evento de hacer click en el botón "Procesar" del formulario, lo capturamos con jQuery y le pasamos el control al controlador. La secuencia lógica sería:

- Comprobar que los datos no estén vacíos.
- Comprobar que los datos sean correctos.
- Crear el objeto autor y asignar a sesión o llamar a la capa de servicio para guardarlo en Base de Datos.

Pero no estamos siguiendo esta estructura clásica. Estamos usando callbacks para simular sincronía en un entorno asíncrono. Como hemos asignado la lógica de comprobación de clases a cada objeto concreto del dominio (aunque la comprobación la realice el navegador), la secuencia que estamos siguiendo es la siguiente:

- Comprueba que los datos no estén vacíos. Al final de esta llamada concatenamos con la llamada a la creación del objeto.
- Creamos el objeto. Es necesario tener el objeto creado para hacer la comprobación mediante la lambda que nos devuelve check(string). Una vez hecho esto devolvemos el objeto si todo ha ido bien, o devolvemos **null** si ha habido algún error.

2.7.1 Ejercicio 4

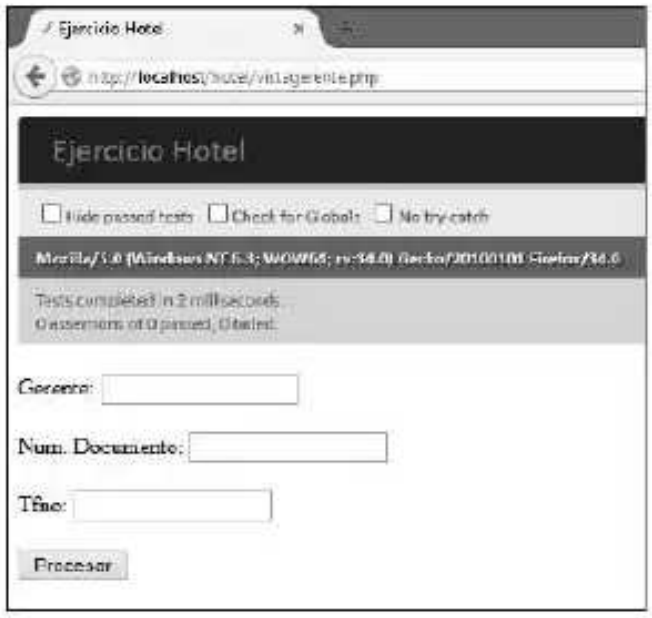
Se propone al lector implementar el análogo del ejemplo, correspondiente al hotel, creando las siguientes vistas:

Cuando se pinche en procesar, se ejecutará el controlador de la aplicación, que comprobará los campos y creará un objeto hotel, que será testeado mediante pruebas unitarias.

El actual ejemplo incluye dos enlaces, independientes del formulario, que nos llevan a dos vistas diferentes.

- Insertar gerente.
- Insertar habitación.

Si se pincha en insertar gerente, accederemos a la siguiente pantalla:



Pulsando en procesar haremos el mismo procedimiento, realizando las pruebas unitarias como finalización del proceso.

Si en la pantalla del hotel pinchamos en insertar habitación, accedemos a la siguiente pantalla:

Pinchando en procesar, realizaremos el proceso establecido, finalizando con pruebas unitarias.

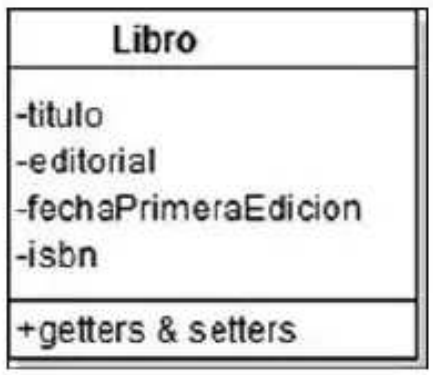
2.8 Objetos ligeros

Existe un problema relacionado con el rendimiento en memoria. Por ahora, y gracias a los closures, estamos consiguiendo encapsulamiento de la información, pero cada uno de los objetos que creamos para una respectiva clase, reserva memoria tanto para los atributos como para los métodos. ¿Hace falta reservar memoria para las funciones? La respuesta es no. Las funciones deben de estar definidas en el prototipo de la clase, o estructura que define "como es la clase". Antes de explicar cómo se definen los objetos ligeros, vamos a ver un par de instrucciones:

- **Object.create(prototipo)**. Pasando un prototipo de objeto, devuelve una copia de dicho objeto.
- **extend**. Deriva un objeto con ciertas características. No es nativo de JavaScript, por lo tanto, debemos de hacer uso de dicha función con alguna librería, o implementarlo nosotros. jQuery lo implementa. Por lo tanto, usaremos la implementación de jQuery.

La solución que vamos a proponer soluciona el problema del peso del objeto con sus correspondientes funciones, pero deja abierto el problema del encapsulamiento. Para según qué aplicaciones debemos estudiar si nos merece la pena una solución u otra. O buscar una solución que cree objetos ligeros y encapsulados.

Como vamos a entender mejor lo que estamos explicando con un ejemplo, centrémonos en nuestra clase Libro:



Para no repetir código, hemos cogido una pequeña versión de la clase Libro ya escrita (incluso hemos eliminado el campo autor). Por el método visto hasta ahora, la clase quedaría así:

```
// Implementación de la clase Libro con getters y setters.
var Libro = function() {
 var sThis = this;
 this.datosLibro = {
  titulo : '',
  editorial : '',
  fechaPrimeraEdicion : '',
  isbn : ''
 };
```

```
var getTitulo = function() {
 return sThis.datosLibro.titulo;
};
var setTitulo = function(titulo) {
 sThis.datosLibro.titulo = titulo;
};
var getEditorial = function() {
 return sThis.datosLibro.editorial;
};
var setEditorial = function(editorial) {
 sThis.datosLibro.editorial = editorial;
};
var getFechaPrimeraEdicion = function() {
 return sThis.datosLibro.fechaPrimeraEdicion;
};
var setFechaPrimeraEdicion = function(fechaPrimeraEdicion) {
 sThis.datosLibro.fechaPrimeraEdicion = fechaPrimeraEdicion;
};
var getIsbn = function() {
 return sThis.datosLibro.isbn;
};
var setIsbn = function(isbn) {
 sThis.datosLibro.isbn=isbn;
 };
 return {
  getTitulo : getTitulo,
  setTitulo : setTitulo,
  getEditorial : getEditorial,
  setEditorial : setEditorial,
  getFechaPrimeraEdicion : getFechaPrimeraEdicion,
  setFechaPrimeraEdicion : setFechaPrimeraEdicion,
  getIsbn : getIsbn,
  setIsbn : setIsbn
 }
};
```

Cada método que definimos: getTitulo, setTitulo, getEditorial... en realidad es una variable. Cada vez que hacemos **new** Libro(), estamos reservando memoria para los datos del libro y para cada uno de los métodos. Sólo deberíamos hacer la reserva de memoria para los datos. Veamos cómo solucionar este problema de rendimiento. La solución reside en usar Object.create.

- Object.create(prototipo);

Partimos con la ventaja de que Object.create no reserva memoria para las funciones de los prototipos, sólo reserva memoria para los datos.