Servicios web con JavaScript

Objetivos

- Saber usar las funciones anónimas en Javascript
- Conocer node.js y saber sus conceptos fundamentales.
- Aprender los conceptos básicos de los servicios web basados en REST, la representación de datos usada y cómo implementarlos en node.js
- Realizar prototipos rápidos de cliente y servidor de servicio web usando node.js

- *Página de la asignatura Arquitectura de Altas Prestaciones, de donde procede parte de este material*
- *Página principal del curso Web 3.0*
- Página del profesor

1 Introducción: servicios web con JavaScript

Javascript ha pasado de ser cuando se introdujo hace unos 15 años un lenguaje para hacer animaciones chorras a convertirse en un lenguaje *adulto* con el que se pueden programar aplicaciones completas, desde el navegador (en lo que se han denominado RIA, Rich Internet Applications), hasta un cliente genérico, hasta el servidor. En este tutorial lo usaremos para hablar de servicios web, consumirlos y hacer aplicaciones que permitan trabajar con ellos.

Es discutible si JS es el lenguaje ideal para esto; incluso si hay un sólo lenguaje. Lo cierto es que es el único lenguaje que permite crear la aplicación completa (cliente/middleware/servidor). Ningún otro lenguaje está incrustado el en navegador y permite usar todas las capacidades del mismo, y a la vez se puede usar en el servidor. Por eso, con sus defectos (principalmente la falta de madurez y la falta de estandarización de algunos aspectos) hoy por hoy es el lenguaje de programación con más futuro; por eso lo usaremos en este curso para hablar de servicios web.

Un Servicio web es un protocolo para intercambiar datos entre aplicaciones a través de internet. Esta interoperabilididad requiere que se usen estándares abiertos, y por eso los servicios web suelen seguir estándares establecidos por organizaciones tales como el consorcio WWW o la OASIS (Organization for the Advancement of Structured Information Standards).

Pasemos a ver los servicios web en sí. Uns servicio web es una aplicación que se coloca entre la aplicación servidor y la aplicación cliente, formando una capa de lo que se ha dado en llamar *middleware*; generalmente, no están destinados al consumo por parte del usuario, aunque puede haber clientes particulares que los interpreten y los sirvan; muchas aplicaciones de móviles, plugins del navegador o aplicaciones de escritorio son en realidad aplicaciones cliente que consumen servicios web. Por lo mismo, los servicios web independizan la aplicación del lenguaje; siendo una especificación estándar, no importa ni el lenguaje ni la plataforma en la que estén escritos el cliente ni el servidor; un cliente en Perl puede tirar de un servidor en Java, y viceversa; por eso conviene conocer varios lenguajes y usar en cada momento el más apropiado para una aplicación en particular.

¿Cómo funciona entonces un servicio web? Esencialmente, los servicios web son servicios de mensajería; un servicio debe ser capaz de emitir y recibir mensajes usando una combinación de protocolos estándar de Internet. Y contiene dos partes: el *service listener*, que *escucha* directamente un protocolo de internet tal como SMTP, HTTP o Jabber, y un *service proxy*, o proxy de servicio, que es el que decodifica la petición e invoca el código de la aplicación. Pueden ser programas independientes: por ejemplo, el *service listener* puede ser el servidor Apache y el *service proxy* un módulo, *servlet* o CGI que se ejecute desde el servidor.

Aunque más arriba hemos hablado en general de *clientes* y *servidores*, en realidad los servicios Web pueden usar cualquier estilo de interacción, desde un simple cliente/servidor hasta P2P o combinaciones en n fases (*n-tier*) más complicadas. En general, y en los ejemplos que vamos a usar, sin embargo, usaremos un modelo cliente-servidor.

Otro de los objetivos de los servicios web es lograr integración dinámica entre aplicaciones; es decir, que se puedan añadir nuevos servicios o nuevas funcionalidades a una aplicación tras su despliegue inicial. De hecho, debería ser posible no saber de antemano quién va a proporcionar una funcionalidad determinada, sino usar un servicio de *descubrimiento* para encontrarlo y adaptarlo a la aplicación particular.

Se habla de SOA (o Service Oriented Architecture) cuando se integran diferentes servicios débilmente acoplados y fuertemente interoperables. En general, SOA usa *SOAP* (Simple Object Access Protocol) y *WSDL* (Web Services Description Languages) para la descripción de servicios; pero no necesariamente tiene por qué usarlos. La idea es que los servicios SOAP se integran dinámicamente en un aplicación a través de sus descriptores WSDL.

Todos estos servicios integran la llamada pila de protocolos de los servicios web, que tiene 4 capas diferentes:

- Capa de *transporte*: son los protocolos de bajo nivel que se usan para llevar información entre aplicaciones. Se puede usar, en principio, cualquier protocolo: FTP, UDP, TCP, XMPP (el de Jabber), e incluso una capa de transporte extensible denominada BEEP. En general, sin embargo, la mayoría usan HTTP. Únicamente se han usado protocolos como XMPP en experimentos como el Google Wave (ahora Apache Wave). En todo caso, los servicios web no fuerzan a usar una capa de transporte determinada.
- Capa de *mensajería* (en algunos sitios se le llama de *empaquetamiento*), que generalmente se lleva a cabo en XML, pero también puede incluir algún otro tipo de protocolo: YAML o JSON, por ejemplo; este último es, al menos, tan popular como XML útlimamente. O incluso texto bien formateado; en este protocolo se basan servicios como SOAP, XML-RPC o JSON-RPC. Esta capa se encarga de serializar peticiones y estructuras de datos y deserializarlas, en lo que se suele denominar a veces *marshalling*.
- Capa de *descripción*, necesaria para la integración dinámica de nuevas funcionalidades. Aquí parte la pana *WSDL*, que describe los servicios web.
- Capa de *descubrimiento*, que sirve para buscar qué servicios son los que tienen una funcionalidad determinada. Aunqeu inicialmente se propuso un protocolo denominado UDDI (Universal Description and Discovery INterface), prácticamente murió en el año 2006. Ahora se puede usar el conocido (y vetusto) protocolo DNS, o simplemente no usar ningún tipo de registro, sino eliminar esta capa y usar sólo servicios *conocidos*.

Aparte de estos servicios, hay toda una sopa de letras: WSFL, BPEL... es cuestión de ir desenmarañando la maraña poco a poco. En todo caso, la tendencia del cambio de una web de documentos a una web de servicios y el acceso a la misma mediante múltiples dispositivos y aplicaciones ha hecho que la programación de servicios web sea ubicua, y la arquitectura más común casi para cualquier aplicación de cierto tamaño (o sin él) que se haga hoy en día.

Dentro de la sopa de letras se pueden incluir los procedimientos de autenticación; no todo el mundo puede usar todos los servicios. Hay varios tipos; los más simples incluyen en las peticiones el nombre de usuario y clave, pero los más complejos, basados en OAuth , requieren varios niveles de autenticación, a nivel de aplicación (con claves por aplicación) y a nivel de usuario (cuando un usuario autoriza a una aplicación a hacer algo en su nombre). A lo largo de este tutorial usaremos autenticaciones, pero no nos meteremos en profundidad.

Los servicios web se usan sobre todo en aplicaciones a nivel de grandes empresas. La mayor parte de lo que se ha dado en llamar web 2.0 (y, por extensión, la 3.0) se basan en interfaces más simples de programar, los denominados REST. Lo veremos a continuación, tras ver la principal forma de representar estructuras de datos en servicios web, el estándar JSON.

2 Objetos para el camino: el formato JSON

A pesar de la estandarización del XML, resulta *pesado* y algo difícil de manejar, sobre todo en clientes ligeros. Por eso se está estandarizando otra forma de representar la información, que procede del JavaScript y se llama JSON (JavaScript Object Notation, pronúnciese Jota-son o Yeison).

Lo interesante de los objetos en JS es que hay una forma muy fácil de *serializarlos* (es decir, convertirlos en texto u otro formato de forma que se puedan intercambiar fácilmente con otros programas a través de la red); este formato es JSON. Y como cualquier estructura de datos es un objeto en JS, se puede usar esta notación para asignar valores prácticamente a cualquier cosa. Vamos a usar una vez más el intérprete en modo interactivo (se puede hacer desde el navegador usando la consola Javascript, por ejemplo, o usando rhino o cualquier otro intérprete de línea de órdenes como node) para ver un ejemplo:

```
js> var objeto = { Madrid : 25, Atleti: 33, Ponferradina: 44 };
js> for (i in objeto) { print( i + " : "+ objeto[i] )};
Madrid : 25
Atleti : 33
Ponferradina : 44
```

La declaración `var` devuelve undefined en la consola; más abajo se verá la razón. Más fácil no puede ser. Se le asigna valor a un objeto con el formato clave : valor (con coma al final), de la misma forma que se haría a un array asociativo. Además, se pueden crear objetos sobre la marcha y asignárselos a una variable cuyo valor se cree también sobre la marcha:

```
js> eval("var objeto2 = { Madrid : 25, Atleti: 33, Ponferradina: 44 }");
js> for (i in objeto2) { print( i + " : "+ objeto[i] )};
Madrid : 25
Atleti : 33
Ponferradina : 44
```

donde usamos `eval`, que interpreta una expresión en Javascript como si del propio intérprete se tratara. Las expresiones de asignación de valores a objetos se pueden anidar, para dar lugar a objetos más complejos

```
js> eval("var objeto2 = { Madrid : 25, Atleti: 33, Ponferradina: { casa: 33, fuera: 44} }");
js> for (i in objeto2) { print( i + " : "+ objeto2[i] )};
Madrid : 25
Atleti : 33
Ponferradina : [object Object]
```

Que parece más raro de la cuenta, pero que, con un poco de código, se podría también imprimir. En `node` se puede ver un objeto poniendo su nombre:

```
> objeto
{ Madrid: 25,
  Atleti: 33,
  Ponferradina: 44 }
> objeto2
{ Madrid: 25,
  Atleti: 33,
  Ponferradina: { casa: 33, fuera: 44 } }
```

Evidentemente, lo interesante de JSON es que podemos independizar la definición de un objeto del código, pudiendo leerlo de una cadena, un fichero o a través de una conexión web. Supongamos que lo tenemos en una variable, de la forma siguiente (trabajando una vez más en el intérprete):

```
> var valor= "{ Madrid : 25, Atleti: 33, Ponferradina: { casa: 33, fuera: 44} }";
undefined
> valor
'{ Madrid : 25, Atleti: 33, Ponferradina: { casa: 33, fuera: 44} }'
> eval("var con_json ="+valor)
undefined
> con_json
{ Madrid: 25,
  Atleti: 33,
  Ponferradina: { casa: 33, fuera: 44 } }
```

Hay otras formas de convertir cadenas en JSON a variables, pero se verán más adelante.

En realidad, la declaración `var` no es necesaria (es una declaración de ámbito), pero la usamos como buena práctica, siempre se debe declarar el ámbito de las variables JS que se usen. Como contrapartida, el uso de `var` en la consola hace que la declaración devuelva `undefined`, como se ve más arriba. Simplemente se debe al hecho de que la

sentendia `var` (que es una sentencia en JS como otra cualquiera) no devuelve ningún valor. Si se suprime `var` la consola mostrará el valor asignado a la variable.

JSON, precisamente, es uno de los formatos de intercambio, el más simple, el que se usa en la mayoría de aplicaciones AJAX actuales, y un modo de acceder también a servicios web (como Twitter, por ejemplo. Lo usaremos para transferir información entre cliente y servidor más adelante, aunque en general se hará de forma transparente al usuario (es decir, funciones de una librería crearán e interpretarán automáticamente las cadenas JSON por nosotros). Y la gran ventaja es que si sabes definir estructuras de datos en JS lo puedes hacer en JSON, pero por si acaso incluimos en el siguiente ejemplo todas las estructuras posibles:

```
> var json_string = "{ 'vector': [ 0, 1, 2], 'hash': { 'clave':'valor', 'key': 3} }"
undefined
> eval( "var objeto="+json_string );
undefined
> objeto
{ vector: [ 0, 1, 2 ],
  hash: { clave: 'valor', key: 3 } }
```

Ejercicios

Bloque 1

1. Usar el API de búsqueda de Twitter para descargar resultados en JSON y examinarlos o copiarlos dentro de un pequeño programa en JS y trabajar con ellos.

Algunas aplicaciones, como CouchDB, permiten almacenar estructuras de datos en JSON. CouchDB: es un sistema de almacenamiento de documentos (en formato JSON) a los que va asociada una clave. Se parece a las bases de datos tradicionales en que es relativamente fácil recuperar claves o rangos de las mismas, pero lo que nos interesa desde e punto de vista de este curso es la facilidad con que se puede usar para almacenar estructuras de datos JSON. Podemos instalarlo o bien abrirnos una cuenta en CloudAnt, ir al interfaz web (denominado Futon) y empezar a trabajar con él creando objetos directamente desde la web.

Hagamos lo siguiente: una vez creada la cuenta, nos vamos a Dashboard, introducimos un nombre nuevo debajo de "New database" y pulsamos sobre el nombre de la misma. Uno de los botones, "View in futon", nos permite acceder al interfaz web de creación de elementos; "New Document" te presenta un interfaz con un _id (todos los documentos tienen que tenerlo, pero le podemos asignar el valor que queramos). Se pueden añadir nuevos campos con Add Field, y se cambia el valor pulsando dos veces sobre los valores que haya. Finalmente, no se debe olvidar dar a "Save document"; al hacerse así se introduce un campo `_rev` con un valor que empieza en uno. Nos interesa un trapecio de color gris que aparece arriba a la derecha, pulsándolo accedemos directamente al objeto en representación JSON, o podemos copiar el enlace y descargarlo desde el navegador o línea de órdenes.

Ejercicios

Bloque 2

1. Crear "a mano" varios documentos en una base de datos CouchDB creada al efecto, con diferente estructura y valores. Los usaremos más adelante.

Lo interesante de CouchDB, entre otras cosas, es que se puede acceder a ella con un Interfaz REST. Es un ejemplo de aplicación *local* a la que podemos acceder de esta forma; de hecho, se puede acceder tanto el local como remotamente. Veremos a continuación en qué consisten los interfaces REST

3 Introducción al interfaz REST

REST (Representational State Transfer) es una serie de convenciones en la interacción cliente-servidor sobre el protocolo HTTP. En la práctica, un interfaz REST es un interfaz de programación de aplicaciones que usa, para acceder al servidor, el conjunto completo de órdenes del protolo HTTP y confía en los mensajes informativos y de error del mismo.

Aunque se trate de un *hermano menor* de otros tipos de servicios web (que se verán más adelante en este curso), su popularidad se debe sobre todo al poco overhead que añade a las peticiones y a la facilidad de su uso, tanto en el cliente como el servidor. También se puede implementar directamente sobre servidores web estándar como Apache o nginx , lo que facilita su implantación y desarrollo. Crear un cliente para un API REST es tan fácil como crear una cadena; de hecho, se pueden usar desde la línea de órdenes

Y esto sucede así porque aprovecha la expresividad del protocolo HTTP, uno de los protocolos más infrautilizados de la historia. A pesar de que ofrece múltiples posibilidades y versiones, se usa simplemente para enviar y recibir información de un servidor. Para recibir información se usa la orden `GET`, y para enviar, la orden `POST`. Pero también hay otras posibilidades, `PUT` (que envía un recurso determinado al servidor), `DELETE` (que borra un recurso del servidor) e incluso `HEAD` (igual que `GET`, pero sin el cuerpo de la respuests); en realidad hay algunos métodos más, pero sólo se suelen usar los anteriores en REST.

El protocolo HTTP gira alrededor del concepto de *recurso*: un recurso en un servidor está identificado por un URI, y es la mínima acción que un servidor puede realizar. Como características adicionales, la acción de algunas peticiones (`GET` y `HEAD`) debe ser *segura*, es decir, dejar al servidor en el mismo estado que antes de la petición. Otras acciones, como `PUT` y `DELETE`, se denominan *idempotentes*: el hacer varias veces la misma petición tiene el mismo efecto que el hacerla una sola vez.

HTTP funciona puramente como cliente-servidor: se hace una petición, y se espera la respuesta. Lo que no quiere decir que no se puedan hacer peticiones concurrentes y asíncronas; sin embargo, esas peticiones tendrán que estar dentro del marco de una página web (o sea, una aplicación).

A las peticiones el servidor responde con una serie de códigos estándar, que usan la misma presentación que la petición: texto puro y duro. Cuando todo va bien, la respuesta es `200 OK`; los códigos `2xx` corresponden, en general, a una petición hecha, y fuera de los 2xx existe el caos y el descontrol. En especial, un código 500 implica error en el servidor. Evidentemente, estos mensajes están pensados para que los lea un cliente en el navegador; sin embargo, cuando trabajamos directamente sobre este protocolo, nuestro programa deberá ser consciente de ellos y responder de forma adecuada como si se tratara de una llamada a otro procedimiento; dependiendo del código se generará un error en el cliente o se llevará a cabo otra acción.

Las aplicaciones construidas alrededor del protocolo HTTP y sus características se suelen llamar aplicaciones RESTful (REST == REpresentational State Transfer). La idea de REST es que se transfiere el estado del servidor al cliente. Un recurso tiene una representación, que se transfiere al cliente por una petición; esa representación se puede cambiar con diferentes operaciones. Sin embargo, con esto sólo estamos especificando la capa más baja del servicio web; hace falta una capa de mensajería. Y esta capa de mensajería se suele denominar POX, o *Plain Old XML* (XML *de toda la vida*), es decir XML bien formado (como hemos visto en el tema anterior) con algunas ampliaciones, pero sin ningún tipo de validación. En algunos casos se usa texto directamente, aunque también se puede usar JSON o cualquier otro tipo de capa.

De hecho, las aplicaciones REST suelen ser más populares que otros servicios web, por el simple hecho de que es muy fácil construir el interface: simplemente creando una cadena determinada. Eso los hace también más rápidos, aunque sean menos flexibles. Por eso nos concentraremos en este tutorial en este tipo de aplicaciones solamente.

Ejercicios

Bloque 3

1. Buscar un interfaz REST público, y hacer un programa que realice peticiones al mismo.

La idea de REST desde el punto de vista del servidor es usar el URL para representar recursos, y las propias órdenes de HTTP para ejercitar acciones sobre esos recurso. En general, `GET` servirá para transferir la representación de un recurso del cliente al servidor, `POST` cambiará el estado de un recurso, `PUT` (que no se suele usar tan a menudo) directamente cambiaría la representación del recurso, mientras que `DELETE` borraría el recurso; a estas arquitecturas también se les suele denominar también *arquitecturas orientadas al recurso* .

Por eso también se suelen proponer una serie de buenas prácticas para diseñar un interfaz REST:

- La funcionalidad está divida en recursos
- Se usa una sintaxis universal basada en URL
- Todos los recursos tienen un interfaz uniforme, con un conjunto bien definido de operaciones y un conjunto restringido de tipos de ocntenido. En particular, este interfaz esconde los detalles de la implementación.

Por ejemplo, supongamos que hay que diseñar un interfaz REST para una quiniela deportiva. Hay una quiniela por jornada, y cada jornada tiene 15 partidos. Supongamos que se conocen los partidos de antemano, y que sólo se pueden proponer resultados por parte de un usuario. Se podría diseñar el interfaz de la forma siguiente.

- Quiniela de una jornada: `http://jost.com/quiniela/jornada/[número de jornada]`
- Un partido de una quiniela: `http://jost.com/quiniela/jornada/[número de jornada]/partido/[número de partido]`
- Para los resultados, habría que sustituir `quiniela` por `resultados`. Adicionalmente, añadir `usuario/[nombre de usuario]`, para recuperar los resultados propuestos por un usuario determinado. Por ejemplo, Resultados: `http://jost.com/resultados/jornada/22/usuario/foobar`

Las operaciones vienen dadas por el diseño del interfaz. Por ejemplo, para proponer un resultado determinado habría que hacer una petición POST con dos parámetros: el nombre de usuario y el resultado propuesto. El servidor responderá con un mensaje estándar HTTP y un fichero XML si se ha podido hacer correctamente, y con un error HTTP si no.

El principal problema con este diseño RESTful es hacerlo en la práctica. Aunque uno puede interpretar un URL de la forma que quiera (simplemente tratando la cadena), la forma como se pasan los parámetros tiene un montón de & e signos =. Así que hay que *limpiar* el URL de alguna forma. Dependiendo de la implementación del servidor, quizás se puede hacer directamente; por ejemplo, en caso de que se trate de un `.war` en un contenedor de servlets, ya se encarga directamente; sin embargo. Algunos CGIs también permiten interpretar directamente el URL. Pero otra forma de hacerlo es usar `mod_rewrite`, que permite reescribir los URLs, de forma que la petición cambia de forma antes de servirse la petición. Estos *cambios* toman la forma de directivas del servidor; por ejemplo, en Apache podríamos usar la siguiente (dentro del fichero `httpd.conf` o el fichero de configuración de un directorio en particular, `.htaccess`:

```
RewriteRule ^quiniela/(\w+)/(\d+)/(\w+)/(\d+)$ /~jmerelo/REST/quiniela.cgi?$1=$2&$3=$4 [L]
```

Parece un poco complicada, pero no lo es. Para empezar, se cambiará la expresión regular de la izquierda por la de la derecha. La de la izquierda incluye palabras (\w+) y números (\d+), y en la expresión de la derecha aparecen, por orden, representados por `$n`.

El interfaz REST se puede usar directamente desde línea de comandos con aplicaciones como `curl` que son capaces de generar peticiones HTTP directamente. Por ejemplo, usándolo sobre una base de datos CouchDB. Las direcciones son imaginarias, habría que sustituirlo por la dirección correcta (preguntádsela al profesor si lo tenéis a mano, o usad la base de datos creada en el ejercicio de la sección anterior en CloudAnt). Por ejemplo, la siguiente orden REST devolvería todos los documentos existentes

```
jmerelo@penny:~/public_html/tutoriales/servicios-web/ejemplos$  curl https://couchdb.com/bd/_all_docs
{"total_rows":0,"offset":0,"rows":[

]}
```

En este caso devuelve 0, porque no hay ningún documento todavía. `curl` es una utilidad de línea de órdenes que permite, como pocas otras, ejecutar todos los comandos HTTP. En este caso, curl sólo equivale a curl -X GET; `-X` es la opción para introducir, en mayúsculas, el comando HTTP. Recordemos que GET es una petición "segura", por lo que podemos hacerlo todas las veces que queramos, que no afectará al servidor. Sin embargo, sí lo hará un POST como el siguiente:

```
curl -X POST https://couchdb.com/db/ -H "Content-Type: application/json" -d '{ "una": 1 }'
```

Que introduce un documento JSON en la base de datos que hayamos definido; CouchDB le añadirá una revisión (`_rev`) y un id (`_id`) automáticamente. Si ejecutamos la orden GET previa, veremos que ahora existe un documento. Esta orden es idempotente, es decir, que teóricamente se puede ejecutar varias veces, pero en este caso no es así, ya que se crea un recurso: aplicaciones repetidas crearán varios documentos con diferentes id. Esto es una peculiaridad de CouchDB, sin embargo.

El mismo recurso se puede borrar más adelante

```
curl -X DELETE https://usuario:clave@couchdb.com/web30/ -H "Content-Type: application/json" -d '{ "_id": "un_
```

pero hace falta usar el nombre de usuario, la clave y además indicar para evitar conflictos el ID del documento a borrar y también la revisión. En todo caso, se puede observar que desde curl se pueden usar todo tipo de órdenes HTTP y actuar sobre una base de datos que usa el interfaz REST.

Ejercicios

Bloque 4

1. Practicar con la creación de bases de datos (usando PUT y el nombre de usuario y la clave, necesarios para hacer este tipo de acciones), objetos, descarga de los mismos en JSON y su borrado

4 Funciones como objetos

Las funciones son objetos de pleno derecho en JavaScript. Se puede crear una función como cualquier otro objeto, y de hecho ya hemos visto algo parecido cuando hemos definido una clase (que es simplemente un tipo de función). Como tales objetos, podemos pasarlas como parámetros y modificarlas de diferentes formas; algo así hemos visto ya cuando hemos definido objetos también, en los que se asignan los nombres de funciones a métodos de una clase simplemente usando su nombre. Las diferentes formas de definir funciones se explican en este post de StackOverflow (un recurso imprescindible, por otro lado).

Vamos a ver un ejemplo a continuación, usando nuestra conocida quiniela; usaremos una función para imprimir el resultado de la quiniela, de forma que se pueda ver la salida de varias formas diferentes, es decir, que podamos definir en qué formato se escriben los partidos.

```
// Definición de la clase Partido
function Nuevo_partido(local,visitante) {
    this.local = local;
    this.visitante=visitante;
    this.resultado=null;
    this.setResultado = setResultado;
    this.toString = toString;
    this.set_to_string = set_to_string;
    this.impresor = _toString;
}

function setResultado( esteResultado ) {
    if ( esteResultado == '1' || esteResultado=='x' || esteResultado=='2' )
        this.resultado = esteResultado;
}

function toString() {
    return this.impresor(this.local, this.visitante, this.resultado);
}

function _toString( local, visitante ,, resultado ) {
    return ": " + local + " - " + visitante + " = "+ resultado;
}

function set_to_string ( impresor ) {
    this.impresor = impresor;
}
```

En el programa anterior (del que mostramos aquí la primera parte, donde está la definición de la clase) definimos un método `_toString`, que precedemos con un _ de forma convencional para indicar que se trata de una función para consumo "interno". La variable de instancia `impresor` adquiere, en la creación del objeto, ese valor, para que cuando se quiera escribir el valor de la quiniela se use por defecto. También la función `set_to_string` se usa para alterar ese valor, en caso de que queramos hacerlo, como se hará en el programa siguiente.

```
var equipos= new Array('Madrid', 'Barça', 'Atleti', 'Geta', 'Betis', 'Depor', 'Sevilla', 'Graná');

function jornada( estosEquipos ) {
    var equiposAqui = new Array;
    var imprime = function( local, visitante, resultado ) {
```

```
        console.log("Imprimiendo \n");
        return  "- " + local + " vs. " + visitante + " resultado  "+ resultado;
    };
    equiposAqui = equiposAqui.concat(estosEquipos);
    var midsize = equiposAqui.length/2;
    var quiniela = new Array( midsize );
    var unox2 = new Array( '1','x','2');
    for ( var i=0; i < midsize ; i++ ) {
        var equipo1 = equiposAqui.splice(Math.floor( equiposAqui.length*Math.random()) , 1);
        var equipo2 = equiposAqui.splice(Math.floor( equiposAqui.length*Math.random()), 1);
        quiniela[i] = new Nuevo_partido( equipo1, equipo2 );
        quiniela[i].setResultado( unox2[Math.floor( 3*Math.random()) ]);
        quiniela[i].set_to_string( imprime );
    }
    return quiniela;
}

var quinielas = new Array;
for ( var i = 0; i < 10; i ++ ) {
  quinielas[i] = jornada( equipos );
}

var resultados= new Array;
for ( var i in equipos ) {
  resultados[equipos[i]]=0;
}

for ( var i = 0; i < quinielas.length; i ++ ) {
    for ( var j = 0;j < quinielas[i].length; j ++ ) {
      var local = quinielas[i][j].local;
      var visitante = quinielas[i][j].visitante;
      var resultado = quinielas[i][j].resultado;
      console.log(quinielas[i][j].to_string());
      switch (resultado) {
    case '1':
        resultados[local]+=3;
        break;
    case 'x':
        resultados[local]+=1;
        resultados[visitante]+=1;
        break;
    default:
        resultados[visitante]+=3;
    }
  }
}

for ( var i in resultados ) {
  console.log( i + ": " + resultados[i])
}
```

Para empezar, conviene observar que el programa está hecho para ejecutarse con el intérprete node.js. Se podría usar en cualquier otro intérprete cambiando el modo en el que se imprimen cosas en pantalla; en este caso usamos `console.log`. En Rhino y Spidermonkey sería `print`; si queremos que se interprete en el navegador habrá que cambiarlo por document.writeln, aunque console.log también funcionará (escribiendo el resultado en la consola de Javascript, habitualmente) si tienes instalado FireBug o usas Chrome o Safari. Ese es uno de los problemas (relativos) de JS, que no hay una "librería estándar" que permita hacer todas estas cosas fácilmente. Todo esto no es más que un entramado para la parte que nos ocupa en esta sección: la definición de la función `imprime` dentro, a su vez, de la función `jornada`. Esa definición de función *anónima* que asignamos a una variable es similar a las funciones *con nombre* salvo que se suprime el nombre y el paréntesis con los nombres que vamos a asignar a las variables sigue inmediatamente a la palabra clave `function`. Esta función anónima la podemos asignar a otra variable, o dejarla tal cual. Por ejemplo, podríamos haberla puesto directamente en la orden (en este ejemplo) en la que la asignamos a un objeto

```
        quiniela[i].set_to_string( function( local, visitante, resultado ) {
            console.log("Imprimiendo \n");
            return  "- " + local + " vs. " + visitante + " resultado  "+ resultado;
        } );
```

En este caso la función no se *bautiza* hasta que no se usa dentro de `set_to_string`. En realidad, el resultado es el mismo.

Ejercicios

Bloque 5

1. En el programa anterior, añadir a la clase Nuevo_partido una función que asigne puntuación según el resultado (ahora se hace en el penúltimo bucle). Permitir que el usuario pueda definir esa función, cambiando el valor que tiene por defecto.

5 Node.js, un intérprete asíncrono para JS

En los ejemplos anteriores ya hemos usado `node.js` como intérprete de línea de órdenes, pero los ejemplos podrían

haber funcionado exactamente igual en otros intérpretes o en el navegador. Ahora veremos cómo usarlo, y el concepto general del mismo.

La aceptación de JS como un lenguaje de programación vino del hecho de su incorporación en diferentes herramientas de propóxito general, y sobre todo el hecho de desgajarlo del navegador. Una de tales herramientas es Node.js, un sistema para programación de eventos asíncrono que usa como base JS. Se puede usar directamente como intérprete de JS (tal como hemos hecho antes), salvo por el hecho de que está preparado para trabajar de forma asíncrona, por lo que un patrón habitual de comportamiento, que es asignar la salida de una orden a una variable, se convierte aquí en la creación de un callback, es decir, de una función a la que se llama una vez que se complete la acción que se ha solicitado.

En todo caso, node.js convierte JS en un lenguaje de propósito general, algo que le falta a otros intérpretes como Rhino, que necesita usar librerías de Java para poder hacer cosas básicas como abrir ficheros. Empecemos pues, por el principio: instalar node.js, lo que se puede hacer en Linux fácilmente, desde los repositorios, o descargándoselo desde su web.

Seguimos haciendo nuestro primer programa. Aunque es posible que ya lo hayamos hecho en el apartado anterior, hagamos un programa simple (guenas.js) en `node.js` y ejecutémoslo.

```
#!/usr/bin/node

var saludo = new Object;
saludo.hola = 'mundo';
console.log( saludo );
```

La primera línea es exclusivamente para sistemas Linux; en ella habrá que poner el camino completo al intérprete de node; este es una opción, como `/usr/local/bin/node`; con ella y haciendo ejecutable el fichero con `chmod +x node.js` podemos ejecutarlo y obtener el siguiente resultado

```
jmerelo@penny:~/servicios-web/ejemplos$  ./guenas.js
{ hola: 'mundo' }
```

En otro entorno (o si no se quiere hacer al fichero ejecutable), con escribir

```
jmerelo@penny:~/servicios-web/ejemplos$  node guenas.js
```

es suficiente. En cualquier caso, la salida será la misma. Y la explicación también: definimos un objeto `saludo` en la primera línea, y en la segunda le asignamos el valor `mundo` a la variable de instancia `hola`, o visto de otro modo, el valor `mundo` a la clave `hola`. `console.log` imprime la cadena en la salida, escribiendo directamente (y además en JSON) el valor de la misma.

Ejercicios

Bloque 6

1. Escribir y ejecutar un programa en node que imprima varios saludos (tres, por ejemplo) contenido en un array de JS.

Sin embargo, node.js no es un intérprete habitual, tiene una forma particular de hacer las cosas: asíncronamente. Veremos, por ejemplo, como leer un fichero, el de las quinielas que hemos usado hasta ahora.

```
#!/usr/bin/node

fs = require('fs');
eval(fs.readFileSync('./Nuevo_partido.js', 'utf8'));
fs.readFile('quiniela.datos', 'utf8',
            function(err,datos) {
                if (err) {
                    return console.log(err);
                };
                var filas = datos.split("\n");
                for ( var f in filas ) {
                    var cachos = filas[f].split(" ");
                    var este_partido = new Nuevo_partido( cachos[0], cachos[1], cachos[2] );
                    console.log(f + " " + este_partido.toString() );
                }
            });
```

En este programa se usa el intérprete node.js, lo que se ve en la primera línea, que no hace falta en Windows (aunque se tendrá que ejecutar desde línea de órdenes poniendo explícitamente node fichero.js). En la segunda vemos que se carga una librería usando `require`, el mecanismo para cargar un módulo y evaluarlo, que, además, crea un objeto que se puede usar; lo usamos más adelante para leer un fichero. `fs` se refiere a *filesystem*, o sistema de ficheros, y es el módulo que contiene una serie de funciones para interaccionar con el mismo.

La segunda orden es un poco más complicada. Esencialmente, lo que hacemos es evaluar el código que cargamos (de ahí que comience por `eval`). Es similar al `require` de la línea anterior, pero al principal diferencia es que necesitamos que el fichero lo lea de forma síncrona, por eso se usa `readFileSyn`. Para entendernos, esta es una lectura de ficheros "de las de toda la vida", que conviene que usemos en este caso, aunque también se podría leer el código JS y luego evaluarlo, como se hará a continuación.

La tercera línea es la que usa un modo de interacción propio de node.js. Como ya se ha indicado (varias veces),

node funciona de forma asíncrona. En general, el patrón de las funciones en node, en vez de ser

```
haz_a();
 haz_b();
```

que ejecutaría `haz_a`, y, tras terminar, ejecutaría `haz_b`, es

```
haz_a(parametros, haz_b);
haz_c()
```

que viene a decir ejecuta `haz_a` sobre unos `parametros` y, cuando veas que has terminado, llama a la función `haz_b`; fijaros que se trata de un puntero a función, no una llamada a la misma (no lleva paréntesis). Pero dependiendo de lo que tarde `haz_a`, `haz_c` podría ejecutarse antes que `haz_b`. En general, la secuencia de las líneas no tiene por qué ser la secuencia de ejecución de las funciones, eso es precisamente lo que significa la asincronía. Eso no quiere decir que no se pueda usar como cualquier otro lenguaje, sólo que hay que tener cuidado. Y, por otro lado, permite responder muy rápidamente a eventos sin bloquear la operación; cada evento inicia una hebra y se van procesando en paralelo.

Vamos a la orden específica: efectivamente, con `readFile` leemos el fichero. Los dos primeros argumentos son el nombre del fichero y, a continuación, la codificación, que es obligatorio usar. Y continuación el *callback* del que hemos hablado: una función que se ejecuta cuando se termine; se trata de una función anónima tal como las que hemos visto en apartados anteriores. El hecho de que se ejecute asíncronamente quiere decir que fs.readFile se ejecuta y se deja el evento generado; si hubiera una orden a continuación se ejecutaría inmediatamente. Esto le permite a `node.js` leer las cosas con mucha eficiencia, y hacer una serie de operaciones que no se pueden hacer fácilmente con otros lenguajes.

Concentrémonos en la función. Tiene dos argumentos: `err` y `datos`. Si hay un error, estará en la primera variable (que comprobamos) y si no, el resultado irá a la segunda variable. Es decir, cuando se ejecute la acción, se llamará a la función con dos argumentos, uno de los cuales será `null`. Vemos también que se usa `console.log` para escribir en la consola; `console` es un objeto que equivaldría al `document` del DOM, salvo que no tiene ningún tipo de estructura; tiene la ventaja de que si se escribe una estructura de datos compleja, la "desplegará".

El resto del programa es más o menos habitual; usamos la clase que hemos definido anteriormente para genera un objeto de cada tipo e imprimirlo usando `console.log`

Ejercicios

Bloque 7

1. Hacer un programa en node.js que lea de un fichero los resultados numéricos de un partido "EquipoA n EquipoB m" y los incluya en una estructura de datos, que sea eventualmente escrita..

6 npm, instalación de módulos en Node

`fs` es sólo el principio de una serie de módulos muy interesantes. Hay un módulo para crear servidores web, pero lo veremos más adelante. Sólo para tener una idea se puede visitar nodetuts. En el sitio de descarga de nodejs vienen también todos los módulos disponibles, que permiten trabajar con el sistema operativo y cosas más avanzadas como una interfaz criptográfica. Pero si se quieren instalar más módulos, una de las características más interesantes de node es que tiene su propio gestor de paquetes, `npm`. Hay que seguir las instrucciones para instalarlo,y una vez hecho tener en cuenta que los módulos se instalan por omisión en el directorio superior al que uno está trabajando. La lista de todos los paquetes está en línea, y contiene módulos para la mayoría de los servicios web y aplicaciones actuales.

Instalemos por ejemplo `request`, una de las librerías más populares, que actúa como cliente de HTTP. Una vez instalado npm, se escribe (en el directorio donde lo vayamos a usar, en general; la política de módulos de Node es tener los módulos instalados junto con la aplicación que los usa, en vez de en un sitio centralizado) `npm install request`. Esta orden, si la conexión a Internet está disponible, descargará e instalará el módulo en el directorio desde la que la llamemos. Si se ejecuta por primera vez, creará un directorio `node_modules`, dentro del cual habrá un directorio `request`.

Con `request` podemos codificar todo tipo de peticiones REST. A un nivel muy básico se puede usar de la forma siguiente, en el programa request.js, que pide la línea temporal pública de Twitter:

```
var request = require('request');
var url ='http://api.twitter.com/1/statuses/public_timeline.json';
request(url, function (error, response, body) {
  if (!error && response.statusCode == 200) {
      console.log(JSON.parse(body));
  } else {
      console.log(error);
  }
})
```

Usamos la librería recién instalada para descargarnos la línea temporal de Twitter, usando la librería llamada `JSON`, que se instala con Node. La forma de habitual es la asíncrona habitual en Node: se hace la petición y se le pasa la función a la que hay que llamar cuando se reciba la respuesta, como en el caso anterior de apertura de un fichero. A la función se la llama con tres parámetros: o bien `err`, en caso de que se produzca un error, o bien `response` y `body` en caso de que la respuesta sea correcta. `body` contendrá el texto de la respuesta, que habrá que decodificar (o imprimir tal cual, en caso de que se trate de HTML); `response` es una estructura de datos compleja, que podemos imprimir con `console.log` (y saldrá un montón de cosas, incluyendo la versión de HTTP, las cabeceras, y mucha información más), pero que contiene, entre otras cosas, el estado de la petición, con un código del protocolo HTTP. En el programa anterior se comprobaba sólo si había error o no; ahora demás comprobamos que el código devuelto es el correcto, es decir, 200. Si hubiera un código 400, o 500, o incluso un 201, tendríamos que interpretar la respuesta de otra forma.

Ejercicios

Bloque 8

1. Sobre el programa anterior, imprimir de cada uno de los tuits descargados el id y el autor. Para ello, habrá que examinar (y comprender) la estructura de datos que se devuelve.

7 CommonJS, una infraestructura común

Uno de los problemas de JS es que, al haber sido desarrollado principalmente para trabajar en el navegador, carece de una serie de librerías comunes para trabajar en el servidor o en aplicaciones de escritorio. CommonJS es un intento de dar tal infraestructura. Principalmente se trata de proveer una serie de especificaciones para hacer cosas comunes, desde o más simple, que es crear un módulo o librería hasta cosas más complejas: interacción con consola o con línea de órdenes.

Por lo pronto la especificación que ha tenido más éxito es la de módulos, que se resume en este artículo; se trata de que un módulo escrito para un intérprete (Rhino, por ejemplo) pueda funcionar en otro (tal como node.js). Vamos a ver cómo adaptaríamos alguna de las cosas hechas a este estándar, por ejemplo, cambiando esto sobre la clase Nuevo_partido.js creada anteriormente (la llamamos Un_Partido):

```
exports.Un_Partido = function (local,visitante,resultado) {
    this.local = local;
    this.visitante=visitante;
    this.resultado=resultado;
    this.setResultado = setResultado;
    this.toString = toString;
    this.set_to_string = set_to_string;
    this.impresor = _toString;
}
```

El único cambio ha sido que en vez de definir la función directamente, se define como un atributo de `exports`. El resto, al ser atributos de ese objeto, no hace falta que lo definamos de la misma forma. Al llamarlo también habrá un pequeño cambio. Mientras que antes teníamos que hacer un eval sobre lo cargado, ahora basta con (programa usa_partido.js):

```
var un_partido = require('./Un_Partido.js');

var este_partido = new un_partido.Un_Partido( 'este','otro','1');
console.log('Resultado ' + este_partido.toString());
```

Este módulo ya se comporta como el resto de los módulos de Node, haciendo falta usar sólo `require` (con el camino completo) para cargarlo. Con el uso de CommonJS, con require lo que definimos es un objeto, y las funciones son atributos de ese objeto; por lo que a la hora de declarar nuevos objetos de esa clase tendremos que hacerlo con `new un_partido.Un_Partido`. A partir de ahí el objeto generado se comporta exactamente igual que cualquier otro objeto, como podemos ver usando `console`.

8 Usando un servidor web

Para escribir posibles respuestas a una petición web hay que hacerlo desde un servidor. La tendencia moderna es hacerlo desde un entorno integrado, sin embargo los servidores web multifunción permiten tanto ofrecer páginas web estáticas como webs dinámicas, y además hacerlo desde una variedad de lenguajes de programación; por eso conviene conocer, al menos, cómo instalar un servidor web simple y hacer programas que funcionen desde él con facilidad.

El clásico Apache sigue usándose extensivamente, aunque últimamente se están empezando a usar otras opciones como el nginx, un servidor web de altas prestaciones que se puede isntalar, además, en todo tipo de plataformas. Tanto uno como otro están disponibles en los repositorios de las distribuciones Linux más comunes.

Un servidor web se instala como un servicio y *escucha* un puerto TCP/IP, normalmente el 80; este puerto, en Linux, está reservado (como todos hasta el 999) al superusuario, así que hay que ejecutarlo con esos privilegios. Una vez instalado se pueden servir tanto páginas estáticas (habrá que consultar en la documentación para ver cuál es el directorio configurado para hacerlo) como dinámicas (una vez más, también hay que consultar cuál es el directorio por omisión). Las páginas estáticas se sirven (más o menos) tal cual, y las dinámicas se generan a partir de la ejecución de un programa desde el servidor, con los privilegios del mismo o los que tenga configurados. Esto lo veremos más adelante, pero la idea principal es que los recursos accesibles al servidor web están en una serie de directorios cuyo valor lo calcula el servidor a partir del URL que se le solicita.

Para servir contenidos desde un programa, la forma habitual es copiar el programa con la extensión .cgi al directorio que se haya configurado para ello. De la forma más simple posible un CGI escrito en node.js podría ser el siguiente:

```
#!/usr/bin/node

//cabecera
console.log('Content-Type: text/plain; charset=UTF-8');

//contenido
var una_variable=['uno','dos',{ tres: 'tres'}];
console.log('');
console.log(una_variable);
```

Para ejecutarlo no hay más que copiarlo a un directorio determinado con permisos de ejecución para otros (`chmod +x hola-js.cgi`).La primera envía una cabecera al cliente que le indica el tipo que se usa; la segunda parte es la que efectivamente envía el contenido, en este caso una variable en JSON (recordad que console.log escribe en salida estándar, y convierte las estructuras de datos a JSON).

Node, por su naturaleza asíncrona, realmente no es el mejor sistema para trabajar con JavaScript en un servidor que incluya otros lenguajes. Sin embargo, se puede usar Javascript de muchas maneras diferentes: SilkJS, por ejemplo, es un intérprete de JS que incluye también un servidor web; o V8 CGI es un sistema para crear CGIs basado en el intérprete rápido de JS de Google. Por no introducir más herramientas, no los vamos a ver aquí, pero conviene tener en cuenta que existen este tipo de soluciones que pueden convivir en un servidor como Apache o NGINX con otros lenguajes como Ruby o Perl.

9 node.js como servidor

Crear un servidor web con node.js es tan simple que viene directamente en la página principal del mismo

```
var http=require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Ahí estamos\n');
}).listen(8080, '127.0.0.1');
console.log('Server running at http://127.0.0.1:8080/');
```

Este programa simplemente escribirá "Ahí estamos" en el navegador cuando se solicite el URL. Nada complicado, pero tampoco lo es el programa: se usa un módulo `http` que es estándar en Node en la primera línea del programa; se crea un servidor con `createServer`. Esta orden recibe como parámetro la función a la que hay que llamar cada vez que se reciba una petición. Cuando se recibe una petición, se llama a una función que escribe primero la cabecera HTTP (`writeHead`) y termina (`end`) el servicio de la misma escribiendo el contenido que nos aparecerá en el navegador.

`http.createServer` crea un objeto y lo devuelve; en este caso, no lo asignamos a ninguna variable, sino que sobre el mismo objeto (anónimo) le decimos con `listen` en qué puerto (8080) y dirección (la del propio ordenador, *there's no place like 127.0.0.1*) va a escuchar el servidor. Es una orden que se ejecuta de forma asíncrona, con lo que lo que crea es un *callback* que se llamará cada vez que se llame a ese URL. Sólo los puertos por encima de 1024 están accesibles al usuario, así que tendréis que usar un número en ese rango (como 8080 o 12121, todos por debajo de 65535). El mensaje se escribe en pantalla de forma síncrona, es decir que a partir de que se escriba ese mensaje sabremos que podemos usar el servidor.

Evidentemente, si queremos crear un servidor que haga *algo* tendremos que usar las peticiones que se reciban para dar una respuesta variable. En el

```
var http=require('http');
http.createServer(function (req, res) {
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.end('Ahí estamos ' + req.url);
}).listen(8081, '127.0.0.1');
console.log('Servidor ejecutándose en http://127.0.0.1:8081/');
```

La principal diferencia entre este programa y el anterior es, aparte del puerto usado (8081 en vez de 8080) la línea en la que escribe algo, y en la que usa la variable `req`, un objeto que contiene información sobre la petición, y entre otras cosas el URL (una vez eliminada la parte del servidor) que se ha usado; este URL es el que se escribe a continuación de "Ahí estamos", tal cual.

En general, para programar un servicio web habrá que trabajar con esa petición (que será la que reciba la orden del API) y actuar según la misma, y teniendo en cuenta también la orden HTTP que se use (PUT, GET o la que sea). Esto lo veremos un poco más adelante.

Ejercicios

Bloque 9

1. Crear un servidor que escriba una página web (es decir, un documento en HTML). Si además, la página web cambia un contador cada vez que se visita, mejor.

A partir de ahí se puede construir un mínimo interfaz REST para responder a una serie de peticiones. La idea básica es que las funciones a las que tendremos que llamar estarán indentificadas por el URL que se use para pedirlas. Por ejemplo, el programa rest-minimo.js

```
var http=require('http');
var puerto=process.argv[2]?process.argv[2]:8080;
http.createServer(function (req, res) {
    res.writeHead(200, {'Content-Type': 'text/plain'});
    var split_url=req.url.split("/");
    if ( split_url[1] == '' ) {
        res.end('Portada');
    } else if ( split_url[1] == 'proc' ) {
        res.end('No es la portada');
    } else {
        res.end('No entiendo la petición');
    }
}).listen(puerto, '127.0.0.1');
console.log('Server running at http://127.0.0.1:'+puerto+'/');
```

En este programa procesamos, no sólo imprimimos, la variable `req`. Es una estructura de datos con un montón de cosas (insertad un console.log si queréis verlo), pero de la que vamos a usar solamente el camino. La idea es que el URL lo que describe es un recurso, no un fichero, así que nosotros procesamos el URL partiéndolo en sus diferentes componentes. Si en el primer componente no hay nada, damos la portada; si hay, por ejemplo, `proc`, haríamos otra cosa diferente, y eventualmente si se trata de un URL desconocido devolvemos un mensaje diferente.

Adicionalmente, hemos introducido en este programa un puerto que se toma de la línea de órdenes. `process.argv` contiene información sobre la línea de órdenes y otras cosas; en el 2º elemento es donde está, precisamente, el primer argumento de la línea de órdenes. El puerto por omisión será 8080 (lo que se ve en la segunda línea), pero si se pasa algún argumento (y es un puerto válido) se usará ese valor.

Ejercicios

Bloque 10

1. Modificar el programa anterior para que, en caso de que sea un URL desconocido, devuelva una cabecera de error tal como se debe hacer en las peticiones REST.

Algunos sitios web como Heroku o Nodester permiten publicar de forma gratuita aplicaciones web hechas con node.js. Pueden ser bastante útiles para crear prototipos o para hacer pruebas, incluso para alojar prácticas de alguna asignatura.

10 Interfaces REST simples con express

Para diseñar interfaces REST con Node de forma directa hay un módulo llamado express. La idea de este módulo es reflejar en el código, de la forma más natural posible, el diseño del interfaz REST.

Pero primero hay que instalarlo. Hasta ahora hemos usado solo módulos estándar, que se instalan con Node. Para buscar e instalar otros módulos hay un sistema para gestionarlo bastante simple (sí, casi todo es simple y directo en Node) llamado npm. Tras seguir las instrucciones en el sitio para instalarlo (o, en el caso de ubuntu, instalarlo desde Synaptic, el centro de software o con apt-get), vamos al directorio en el que vayamos a crear el programa y escribimos

```
npm install express
```

en general, no hace falta tener permiso de administrador, sólo el necesario para crear, leer y ejecutar ficheros en el directorio en el que se esté trabajando y desde el que vayamos a ejecutar el programa.

Tras la instalación, el programa que hemos visto más arriba se simplifica, transformándose en el siguiente (rest-server.js):

```
var puerto=process.argv[2]?process.argv[2]:8080;
var express=require('express');
var app = express.createServer();
app.get('/', function (req, res) {
    res.send('Portada');
});
app.get('/proc', function (req, res) {
    res.send('No es la portada');
});

app.listen(puerto);
console.log('Servidor en http://127.0.0.1:'+puerto+'/');
```

Para empezar, `express` nos evita todas las molestias de tener que procesar nosotros la información que hay en el URL y el protocolo que se está usando: directamente escribimos una función para cada respuesta que queramos tener, lo que facilita mucho la programación. Las órdenes reflejan directamente las órdenes de HTTP a las que queremos responder, en este caso `get`; por otro lado se pone directamente la función para cada una de ellas. Dentro de cada función de respuesta podemos procesar las órdenes que queramos.

Por otro lado, se usa `send` en vez de `end` para enviar el resultado. Lo que viene a ser lo mismo, `s` más o menos, aunque send es más flexible, admitiendo todo tipo de datos que son procesados para enviar al cliente la respuesta correcta. Tampoco hace falta establecer explícitamente el tipo MIME que se devuelve, encargándose `send` del mismo.

Con el mismo `express` se pueden generar aplicaciones no tan básicas ejecutándolo de la forma siguiente:

```
node_modules/express/bin/express prueba-rest
```

Se indica el camino completo a la aplicación binaria, que sería el puesto. Con esto se genera un directorio prueba-rest. Cambiándoos al mismo y escribiendo simplemente `npm install` se instalarán las dependencias necesarias. La aplicación estará en el fichero `app.js`, lista para funcionar, pero evidentemente habrá que adaptarla a nuestras necesidades particulares.

Ejercicios

Bloque 11

1. Crear una aplicación node.js que devuelva un contador que se incremente cada vez que se llame. El contador deberá estar en JSON.
2. Hacer que, según la orden REST, se devuelva el contador en HTML o en JSON.

El acceso a los parámetros de la llamada y la realización de diferentes actividades según el mismo se denomina enrutado. En express se pueden definir los parámetros de forma bastante simple, usando marcadores precedidos por `:`. Por ejemplo, si queremos tener diferentes contadores podríamos usar el programa siguiente:

```
var express=require('express');
var app = express.createServer();
```

```
var contadores = new Array;
var puerto=process.argv[2]?process.argv[2]:8080;

app.get('/', function (req, res) {
        res.send('Portada');
});

app.put('/contador/:id', function( req,res ) {
    contadores[req.params.id] = 0;
    res.send( { creado: req.params.id } );
});

app.get('/contador/:id', function (req, res) {
    res.send( "{ "+req.params.id+": "+ contadores[req.params.id] + "}"  );
});

app.post('/contador/:id', function (req, res) {
    contadores[req.params.id]++;
    res.send( "{ "+req.params.id+": "+ contadores[req.params.id] + "}"  );
});

app.listen(puerto);
console.log('Server running at http://127.0.0.1:'+puerto+'/');
```

Este programa (express-count.js) introduce otras dos órdenes REST: PUT, que, como recordamos, sirve para crear nuevos recurso y es idempotente (se puede usar varias veces con el mismo resultado), y además POST. Esa orden la vamos a usar para crear contadores a los que posteriormente accederemos con get. PUT no es una orden a la que se pueda acceder desde el navegador, así que para usarla necesitaremos hacer algo así desde la línea de órdenes: `curl -X PUT http://127.0.0.1:8080/contador/primero` para lo que previamente habrá que haber instalado `curl`, claro. Esta orden llama a PUT sobre el programa, y crea un contador que se llama `primero`. Una vez creado, podemos acceder a él desde la línea de órdenes o desde el navegador (desde el navegador se generan peticiones GET y POST solamente).

Ejercicios

Bloque 12

1. Modificar la portada para que muestre en dos formatos (dependiendo del URL que se le pase), en HTML o JSON, todos los contadores y sus valores.
2. (Opcional) Crear un formulario HTML que permita actualizar, desde el navegador, un contador determinado.

11 Clientes REST

Tampoco es complicado escribir con node.js un cliente REST. Se puede hacer mediante peticiones HTTP (el mismo módulo `http` que hemos visto anteriormente también sirve para crear clientes), pero por supuesto es más fácil escribir un cliente usando la librería restler, que se instala de la misma forma que hemos visto anteriormente con `npm`. Una vez instalada, se puede escribir un cliente como este al utilísimo crea-contadores anterior.

```
#!/usr/local/bin/node

var rest = require('restler');
var url = 'http://127.0.0.1:8080/contador/';
process.argv.forEach(function (val, index, array) {
    if ( index > 1 ) {
        rest.put( url + val ).on('complete', function( data ) {
            console.log( data );
        } );
    }
});
```

El cliente es bastante simple, y lo que hace es crear tantos contadores como argumentos le pasamos por la línea de órdenes. Tras definir un par de variables (ojo con la segunda, tiene que contener el URL del sitio donde vamos a hacer la consulta, el número y la dirección puede ser otro cualquiera, lo que no variará será `contador` si estamos usando el programa anterior), usamos la variable `process.argv` que contiene los argumentos de la línea de órdenes.

Sobre ese objeto ejecutamos un bucle, `forEach` recorre los elementos de un objeto llamando sobre cada uno de ellos una función con tres argumentos: el índice y el elemento que se está recorriendo en ese momento, y el array completo, que en este caso no vamos a usar. Además, los argumentos están en realidad a partir del segundo elemento; los dos primeros contienen el camino a node y el camino completo al programa que se está ejecutando.

La clientela REST se usa con `rest.put`. Vamos a crear un contador con el nombre `val` que se envía desde la línea de órdenes, para lo que creamos el URL del mismo simplemente concatenando las dos cadenas.

Recordemos que node.js actúa de forma asíncrona, por lo que lo que hacemos con esa orden es crear un callback cuando (*on*) la petición se haya completado (*complete*). Ese callback simplemente te dice cual ha sido la respuesta y la imprime.

El resultado, de todas formas, es el siguiente:

```
jmerelo@penny:~/servicios-web/ejemplos$ node crea-contadores.js este aquel el-de-mas-alla otro_mas
{ creado: 'otro_mas' }
{ creado: 'el-de-mas-alla' }
```

```
{ creado: 'aquel' }
{ creado: 'este' }
```

Curiosamente, en orden contrario a como se pasa por la línea de órdenes. Conviene tener en cuenta que hay que ejecutar antes el servidor REST que hemos visto en la sección anterior, y hacerlo en el puerto que tenemos aquí, el 8080 (si se ejecuta en otro, simplemente se cambia).

Ejercicios

Bloque 13

1. Modificar el servidor para que responda a la orden `suma` a la que se le pasen los IDs de dos contadores, y modificar el cliente para que se le pasen por línea de órdenes las ids de dos contadores y muestre la suma.

12 Para finalizar

A lo largo de este tutorial se ha explicado, paso a paso, como diseñar, crear y consumir un servicio web de la manera más común hoy en día: usando REST. En la mayoría de las aplicaciones, REST no presenta ningún problema de escalado ni de implementación, y por tanto resulta fácil de usar. El usar node.js nos ha permitido usar el mismo lenguaje en cliente y servidor, además de la representación (JSON), lo que hace que nos concentremos más en lo que hacemos que en aprender un lenguaje nuevo (Java, XML) para cada tarea.

Evidentemente, hay mucho material para ir más allá. Las arquitecturas orientadas a servicio son todo un campo de aplicación, que usa principalmente XML, SOAP y clientes y servidores basados en Java. Para aplicaciones con cierta entidad, cambio dinámico de propiedades, versionado, y en general características requeridas por grandes empresas son la mejor opción.

Si se quiere trabajar básicamente en el navegador, jQuery funciona de forma muy similar a node: es un entorno asíncrono para trabajar desde el navegador fácilmente, sin tener que escribir demasiado código JavaScript. Trasladar un programa de node a JavaScript es bastante directo, y existen diversidad de ampliaciones (plugins) para jQuery que hacen la vida (todavía) más fácil.

Por otro lado, cualquier lenguaje de scripting como Python o Perl permite crear también arquitecturas cliente y servidor, sólo que no se pueden incluir en el navegador (o usar la experiencia que tenemos con el mismo). Sin embargo, especialmente cuando se trate sólo de consumir servicios web, pueden ser la opción más adecuada.

En cuanto a recurso para hacer preguntas y obtener respuestas interesantes, StackOverflow es un recurso imprescindible. Recuerda que tu karma aumentará también cuando contestes preguntas.

13 Agradecimientos

Agradezco a los lectores en Twitter, especialmente @danielribes, sugerencias sobre este material.

14 Bibliografía

Hay dos libros fundamentales para aprender JS, aunque están muy enfocados a JS en el navegador: *Javascript: The Definitive Guide*, el libro del rinoceronte, editado por O'Reilly, que está disponible como recurso eletrónico en la UGR y *JavaScript Bible, de Danny Goodman*, un tocho considerable, en el que hay de todo, y que viene con un útil CD con ejemplos. También está en la Biblioteca de la UGR. Y, por si fuera poco, una referencia de bolsillo de JS y un libro llamado JS profesional projects. Es que en la biblioteca de la UGR hay de todo.

Como recursos adicionales, las páginas de Javascript en Mozilla.org, el estándar completo, Eloquent Javascript y el curso de Javascript de Víctor Rivas Santos.

Por último, Javascript: The good parts es un manual bastante completo que menciona muchos trucos para trabajar con este lenguaje.