



JavaScript en el lado servidor: NodeJS

Índice



1 JavaScript del <u>lado</u> <u>servidor</u>	3
1.1 <u>Instalación</u> y <u>utilización</u>	4
<u>Instalación</u>	4
<u>Utilización</u>	5
1.2 <u>Creación</u> de un <u>servidor</u> <u>HTTP</u>	6
<u>Manipulación</u> de <u>peticiones</u>	8
¿Qué es el <u>módulo</u> <u>express</u> ?	8
<u>Instalación</u>	8
<u>Creando</u> <u>rutas</u>	9
<u>Recibiendo</u> <u>parámetros</u>	10
<u>Recibiendo</u> <u>POST</u>	10
<u>Usando</u> <u>expresiones</u> <u>regulares</u> <u>como</u> <u>ruta</u>	11
1.3 <u>Módulos</u> <u>principales</u>	13

1. JavaScript del lado servidor

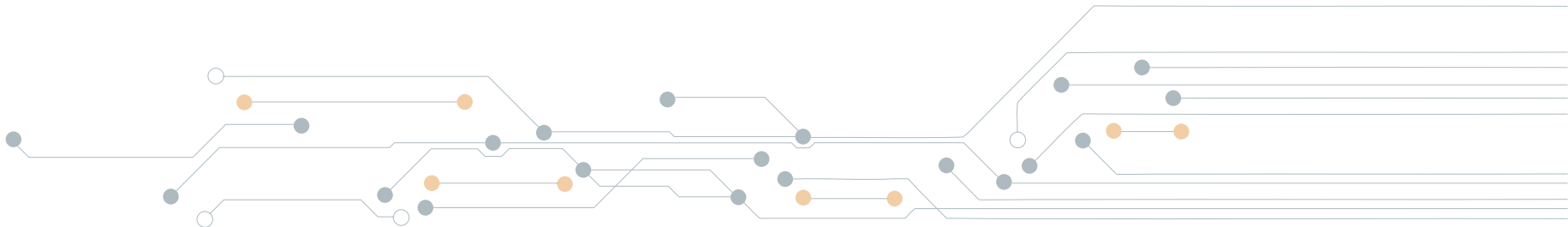
Las primeras evoluciones de *JavaScript* vivían en los navegadores. Pero esto es sólo un contexto. Debemos tener claro que JavaScript es un lenguaje “completo”: Se utiliza en muchos otros contextos. **Node.js** realmente es sólo otro contexto: te permite **ejecutar** e **interpretar** código JavaScript en el **backend**, es decir, fuera del navegador.

Para utilizar el **código JavaScript** en el **backend**, éste **necesita** ser **interpretado** y, bueno, **ejecutado**. Esto es lo que Node.js hace, con el **uso** de la **Máquina Virtual V8** de **Google**, el **mismo entorno** de **ejecución** para **JavaScript** que el **navegador** de **Google**, **Chrome**, usa.

Aparte, **Node.js** **aporta** muchos **módulos útiles**, de manera que **no tienes que escribir todo de cero**, como por **ejemplo**, algo que ponga un **String** a la **consola**.

Por tanto, **Node.js** es ciertamente **dos cosas**: una **librería** y un **entorno de ejecución**.

Para hacer **uso** de éstas (la **librería** y el **entorno**), necesitas **instalar Node.js** en **tu equipo**.

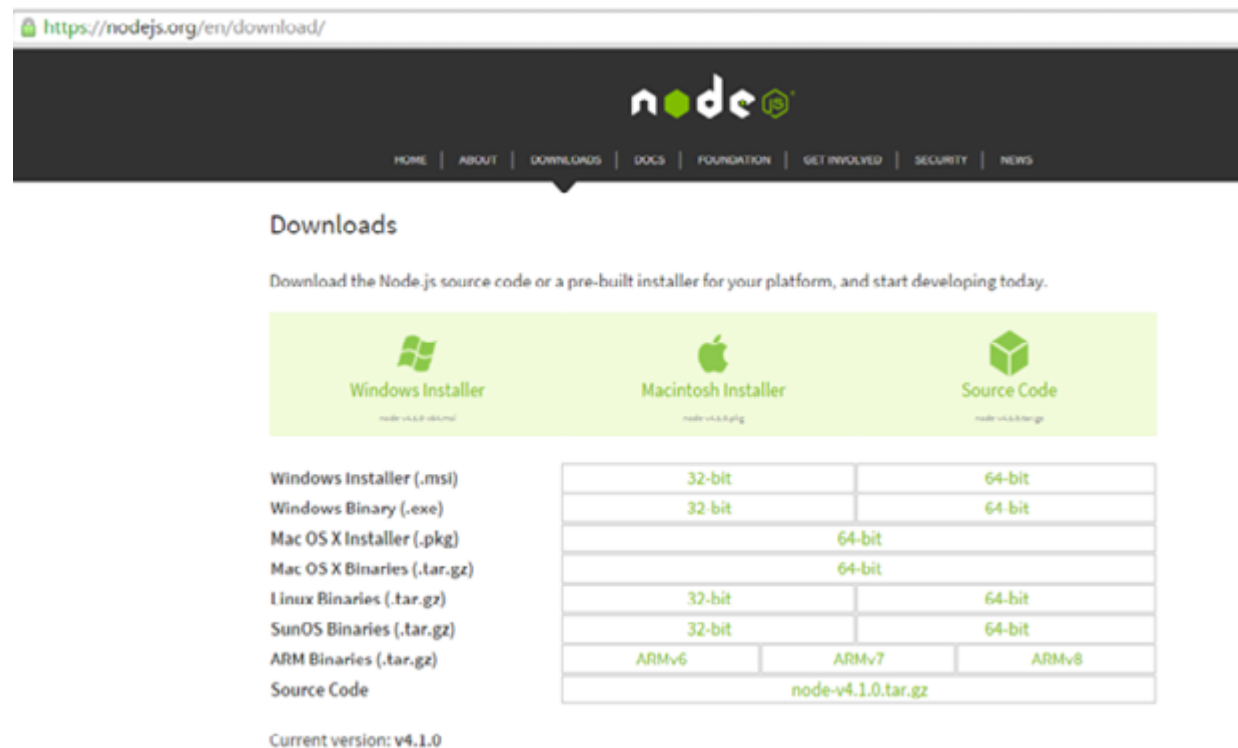


1.1 | Instalación y utilización

INSTALACIÓN

Entrar en NodeJS (<https://nodejs.org/en/download/>) y descargar el ejecutable para instalar NodeJS según nuestro SO.

La instalación es muy sencilla y una vez instalado (a partir de nuestro ejecutable) ya podemos ejecutar NodeJS para que aparezca una **consola de comandos**.



The screenshot shows the Node.js download page with the following content:

Downloads

Download the Node.js source code or a pre-built installer for your platform, and start developing today.

Three main download options are highlighted:

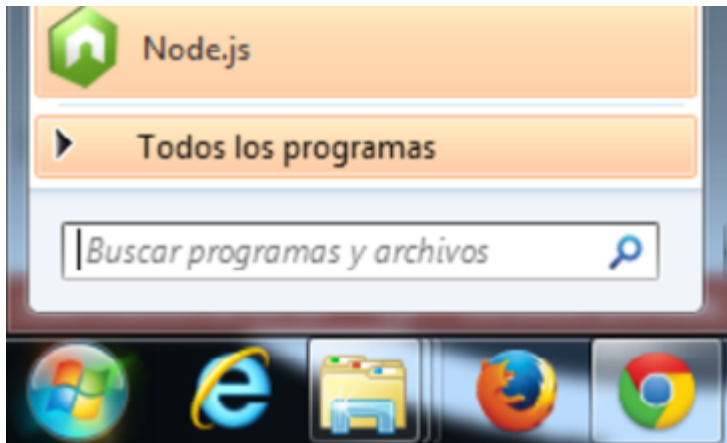
- Windows Installer (node-v4.1.0.msi)
- Macintosh Installer (node-v4.1.0.pkg)
- Source Code (node-v4.1.0.tar.gz)

A list of download links is provided:

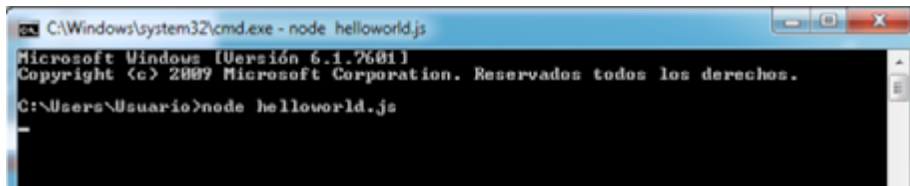
- Windows Installer (.msi)
- Windows Binary (.exe)
- Mac OS X Installer (.pkg)
- Mac OS X Binaries (.tar.gz)
- Linux Binaries (.tar.gz)
- SunOS Binaries (.tar.gz)
- ARM Binaries (.tar.gz)
- Source Code

Current version: v4.1.0

32-bit		64-bit	
32-bit		64-bit	64-bit
64-bit			
64-bit			
32-bit		64-bit	64-bit
32-bit		64-bit	64-bit
ARMv6	ARMv7	ARMv8	
node-v4.1.0.tar.gz			



Se puede desarrollar en cualquier IDE de desarrollo que permita escribir código JS de tal manera que podemos ejecutar un js desde línea de comandos (se presupone un archivo js que llamado helloworld.js)



UTILIZACIÓN

Vamos, pues, a realizar nuestra primera aplicación Node.js: "Hola Mundo".

Podemos escribirla con cualquier editor de textos.

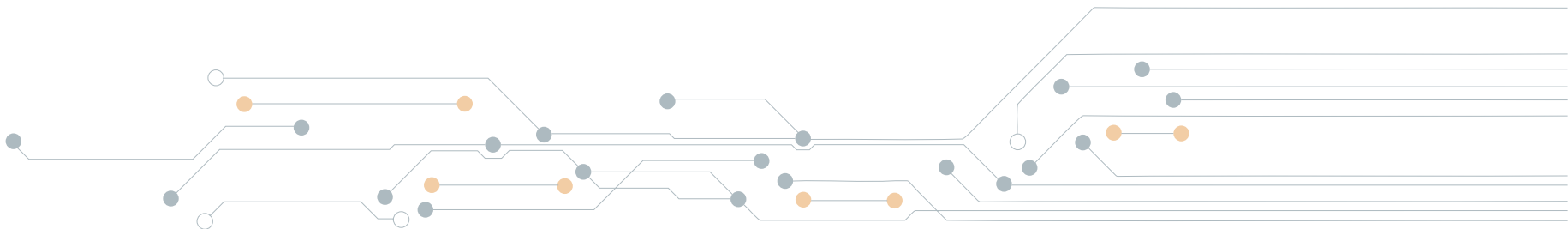
Creamos un archivo llamado holamundo.js y lo rellenamos con este código:

```
console.log("Hola Mundo");
```

Salvamos el archivo, y lo ejecutamos a través de Node.js (en consola dentro de nuestro equipo):

```
node holamundo.js
```

Este debería devolver Hola Mundo en tu monitor.



1.2 | Creación de un servidor HTTP

Aunque más adelante hablaremos de la introducción de módulos en NodeJS, es necesario abordar un primer módulo imprescindible para la creación de un servidor de HTTP. Este es el módulo http.

Aunque más adelante hablaremos de la introducción de módulos en NodeJS, es necesario abordar un primer módulo imprescindible para la creación de un servidor de HTTP. Este es el módulo http.

Para usar el servidor y el cliente HTTP se debe añadir require('http').

Las interfaces HTTP en Node están diseñadas para soportar muchas de las características del protocolo que tradicionalmente han sido difíciles de usar. En particular, los mensajes grandes, seguramente fragmentado. La interfaz se asegura de que las peticiones o respuestas nunca se almacenen completamente en un búfer--se permite al usuario hacer stream de datos.

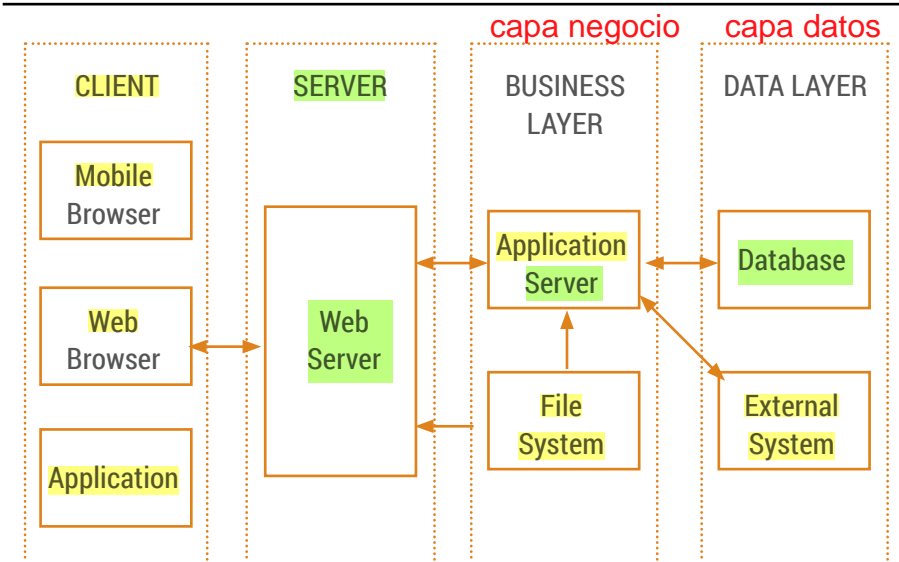
Las cabeceras de los mensajes HTTP se representan por un objeto como este:

```

clave      valor
{ 'content-length': '123', 'content-type': 'text/plain',
  'connection': 'keep-alive', 'accept': '/' }
```

Las claves se convierten a minúsculas. Los valores no se modifican.

Para soportar el espectro completo de las posibles aplicaciones HTTP, la API HTTP de Node es de muy bajo nivel. Se encarga únicamente de manejar el stream y del parsing del mensaje. Parsea el mensaje en sus cabeceras y body pero no parsea las cabeceras o el body.



Como hemos comentado, para crear un servidor HTTP en NodeJS necesitamos la carga del módulo http pero, aparte, podemos necesitar la carga de otros módulos de node como el de filesystem y el de url (ambos en el ejemplo inferior). El primero serviría para buscar en el sistema de ficheros la página a devolver con la petición y el segundo para coger la url del envío de nuestro cliente.

```
//Carga de los módulos requeridos para el programa
var http = require('http');
var fs = require('fs');
var url = require('url');
//Creación del servidor
http.createServer( function (request, response) {
  // Cogemos el path que nos ha entrado por la request
  var pathname = url.parse(request.url).pathname;

  // Leemos el fichero requerido para que sea enviado
  fs.readFile(pathname.substr(1), function (err, data) {
    if (err) {
      console.log(err);
      // Página no encontrada
      // HTTP Status: 404 : NOT FOUND
      // Content Type: text/plain
      response.writeHead(404, {'Content-Type': 'text/html'});
    }else{
      // Página encontrada
      // HTTP Status: 200 : OK
      // Content Type: text/plain
      response.writeHead(200, {'Content-Type': 'text/html'});

      // Damos la respuesta a nuestra petición
      response.write(data.toString());
    }
    // Enviamos la respuesta
    response.end();
  });
}).listen(8081);
```

Una vez tengamos nuestro servidor lanzado vemos que lo hemos puesto a escuchar en el puerto 8081 por lo que un cliente desde el navegador podría acceder a él de esta manera:

```
protocol url      port  request
http://127.0.0.1:8081/index.htm
```

Esta petición se haría sobre localhost (127.0.0.1) al puerto 8081 y pidiendo la página index.html que se encontraría en los recursos del servidor.

MANIPULACIÓN DE PETICIONES

Para poder realizar la manipulación de peticiones es necesario instalar el módulo de express en nuestra aplicación. Pero, ¿para qué sirve este módulo y que es lo que nos ofrece?

¿QUÉ ES EL MÓDULO EXPRESS?

Express es un módulo de NodeJS que se puede instalar a través de la herramienta npm.

En una definición exacta sería: **el framework que se lanza sobre un servidor http de NodeJS** para manipular las rutas y dar acceso de un modo sencillo al cliente para acceder al ciclo de vida de la aplicación.

INSTALACIÓN

Para instalar dicho módulo solo sería necesario escribir esta línea desde consola de comandos:


```
$ npm install -g express
$ npm install -g express-generator
```

Si escribimos la siguiente línea nos devolverá la versión de express que se ha descargado (en el ejemplo la 3.3.5):

```
$ express --version
3.3.5
```

Ahora que ya tenemos los paquetes que necesitamos, podemos empezar a escribir nuestra aplicación. Creamos un nuevo archivo llamado app.js en el directorio del proyecto, este archivo será el que inicie el servidor. Empezamos, escribiendo las dependencias que necesitamos:

```
var express = require('express');
var http = require('http');
```



`express` es el `framework`, como ya hemos comentado.

`http` es el `módulo` del `servidor` para `NodeJS`.

Ahora `creamos` nuestra `aplicación`:

```
var app = express();
```

Le `indicamos` a `express` en que `puerto` `vamos` a estar `escuchando`:

```
app.set('port', process.env.PORT || 3000);
```

`process.env.PORT` es una `variable de entorno`, si no está configurada para guardar el puerto en que debe correr la aplicación, entonces toma el 3000.

Por último `creamos` e `iniciamos` el `servidor`:

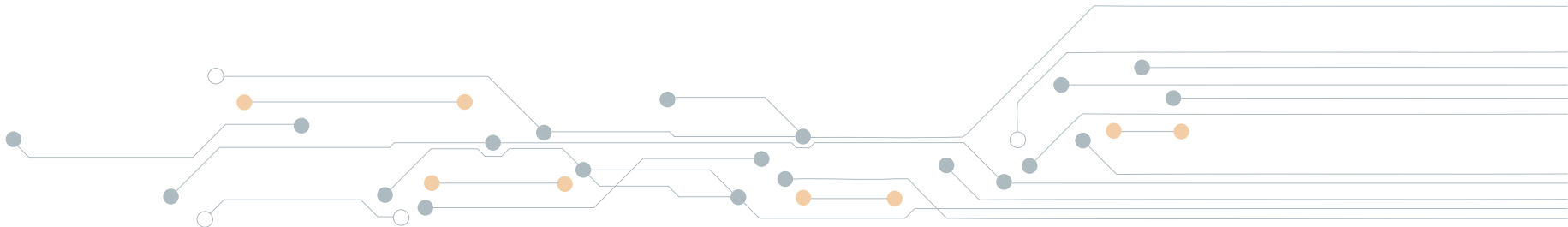
```
http.createServer(app).listen(app.get('port'), function(){  
  console.log('Express server listening on port ' + app.  
    get('port'));  
});
```

Ahora ya `tenemos` lo `mínimo necesario` para `iniciar` la `aplicación`.

CREANDO RUTAS

Las `rutas` `nos permiten` `direccionar` `peticiones` a los `controladores correctos`. Vamos a `empezar` `agregando` el `código` de un `controlador` para una `ruta`:

```
app.get('/', function(request, response) {  
  response.send('¡Hola, Express!');  
});
```



Si **corremos** nuestra **app** en la **consola** (parados en directorio de la aplicación) **node app.js** y vamos a `"http://localhost:3000/"` en nuestro **explorador** de **preferencia**, **debemos** ver el **mensaje** "¡Hola, Express!"

RECIBIENDO PARÁMETROS

Si queremos **recibir** algún **parámetro** en una **ruta** **debemos** **especificar** en el **String** el **nombre** del **parámetro** son ":" adelante:

```
app.get('/users/:userName', function(request, response) {  
  {  
    var name = request.params.userName;  
    response.send('¡Hola, ' + name + '!');  
  });  
});
```

Ahora si corremos la app y vamos a `"http://localhost:3000/users/oscar"` veremos que se despliega el mensaje "¡Hola, Òscar!".

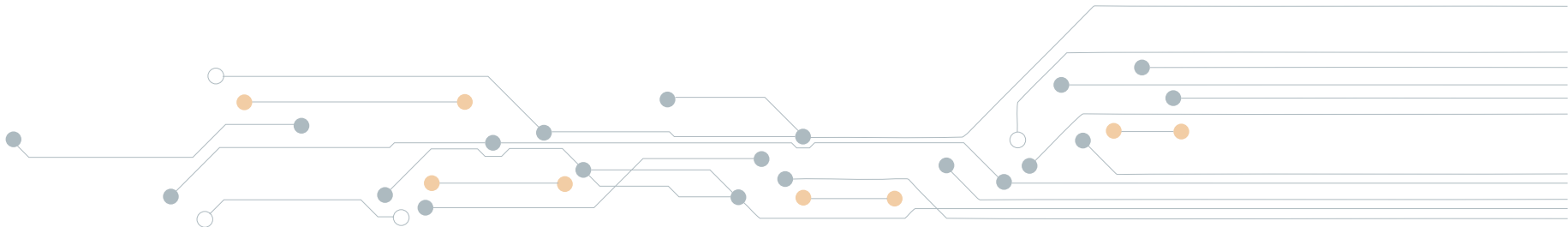
RECIBIENDO POST

También podemos **recibir** **requests** de **tipo POST** de la siguiente manera:

```
app.post('/users', function(request, response) {  
  var username = request.body.username;  
  response.send('¡Hola, ' + username + '!');  
});
```

Antes de correr este código debemos agregar bodyParser fuera del método, porque express no parsea el cuerpo del request por defecto:

```
app.use(express.bodyParser());
```



Ahora podemos hacerle un `post` desde cualquier app que nos permita hacerlo. Se puede utilizar una extensión de Chrome llamada Postman, desde ahí le podemos enviar lo siguiente a `"http://localhost:3000/users"`:

```
POST /users HTTP/1.1
Host: localhost:3000
Authorization: ApiKey
appClient:xxxxxxxxxxxxxxxxxxxxxxxxxxxx
Cache-Control: no-cache
----WebKitFormBoundaryE19zNvXGzXaLvS5C
Content-Disposition: form-data; name="username"
oscar1234
----WebKitFormBoundaryE19zNvXGzXaLvS5C
Y deberá retornar:
¡Hola, oscar1234!
```

De esta misma manera también podemos recibir requests `PUT` y `DELETE` utilizando `app.put()` y `app.delete()` respectivamente.

USANDO EXPRESIONES REGULARES COMO RUTA

También podemos usar expresiones regulares como rutas, por ejemplo, podríamos usar `"\\/users\\/(\\d*)\\/?(edit)?/"` como una ruta, especificando así que debe haber un dígito en el medio y que la palabra "edit" es opcional.

```
app.get(/\\/personal\\/(\\d*)\\/?(edit)?/, function
(request, response) {
  var message = 'el perfil del empleado #' +
  request.params[0];
  if (request.params[1] === 'edit') {
    message = 'Editando ' + message;
  } else {
    message = 'Viendo ' + message;
  }
  response.send(message);
});
```



Si corremos la app y vamos a *"http://localhost:3000/personal/15"* veremos que se despliega el mensaje "Viendo el perfil del empleado #15", y si agregamos *" /edit"* al final veremos que el mensaje cambia a "Editando el perfil del empleado #15".

Luego de todos estos cambios tu archivo `app.js` debe lucir así:

```
var express = require('express');
var http = require('http');
var app = express();
// all environments
app.set('port', process.env.PORT || 3000);
app.use(express.bodyParser());
app.get('/', function(request, response) {
    response.send('¡Hola, Express!');
});

app.post('/users', function(request, response) {
    var username = request.body.username;
    response.send('¡Hola, ' + username + '!');
});
app.get(/\/personal\/(\d*)\/?(edit)?/, function (request, response) {
    var message = 'el perfil del empleado #' + request.params[0];
    if (request.params[1] === 'edit') {
        message = 'Editando ' + message;
    } else {
        message = 'Viendo ' + message;
    }
    response.send(message);
});
http.createServer(app).listen(app.get('port'), function(){
    console.log('Express server listening on port ' + app.get('port'));
});
```

Modulos - Subprogramas , pequeñas rutinas , abstractas , encapsuladas

1.3 | Módulos principales

un módulo es una porción de un programa de ordenador.

De las varias tareas que debe realizar un programa para cumplir con su función u objetivos, un módulo realizará, comúnmente, una de dichas tareas (o varias, en algún caso).

NodeJS posee varios módulos compilados en binario. Estos módulos son descritos con más detalle en las siguientes secciones del documento. Los módulos básicos son definidos en el código fuente de *NodeJS* en la carpeta lib/.

Los módulos básicos tienen la preferencia de cargarse primero si su identificador es pasado desde *require()*. Por ejemplo, *require('http')* siempre devolverá lo construido en el módulo HTTP, incluso si hay un fichero con ese nombre.

A continuación vamos a pasar a reseñar algunos de los módulos más importantes que podemos agregar a nuestras aplicaciones de los centenares y centenares que hay. Como nota, se debe comentar, que podemos construirnos nuestros propios módulos dentro de nuestra aplicación, con el objetivo de modularizar nuestro código.

▪ console

Marcado en el API como STDIO, ofrece el objeto console para imprimir mensajes por la salida estándar: stdout y stderr. Los mensajes van desde los habituales info o log hasta trazar la pila de errores con trace.

▪ timers

Ofrece las funciones globales para el manejo de contadores que realizarán la acción especificada pasado el tiempo que se les programa. Debido a la cómo está diseñado Node, relacionado con el bucle de eventos del que se hablará en un futuro, no se puede garantizar que el tiempo de ejecución de dicha acción sea exactamente el marcado, sino uno aproximado cercano a él, cuando el bucle esté en disposición de hacerlo.

▪ module

Proporciona el sistema de módulos según impone CommonJS. Cada módulo que se carga o el propio programa, está modelado según module, que se verá como una variable, module, dentro del mismo módulo. Con ella se tienen disponibles tanto el mecanismo de carga *require()* como aquellas funciones y variables que exporta, en module.exports, que destacan entre otras menos corrientes que están a un nivel informativo: módulo que ha cargado el actual (*module.parent*), módulos que carga el actual (*module.children*)..

Buffer : es un espacio de memoria, en el que se almacenan datos de manera temporal, normalmente para un único uso (generalmente utilizan un sistema de cola FIFO);
 Su principal uso es para evitar que el programa o recurso que los requiere, ya sea hardware o software, se quede sin datos durante una transferencia (entrada/salida) de datos irregular o por la velocidad del proceso.

■ buffer

Es el objeto por defecto en *Node* para el manejo de datos binarios. Sin embargo, la introducción en *JavaScript* de los *typedArrays* desplazará a los Buffers como manera de tratar esta clase de datos.

Los módulos siguientes, listados por su identificador, también forman parte del núcleo de *Node*, aunque no se cargan al inicio, pero se exponen a través del API:

■ util

Conjunto de utilidades principalmente para saber de si un objeto es de tipo array, error, fecha, expresión regular...También ofrece un mecanismo para extender clases de JavaScript a través de herencia:

```
inherits(constructor, superConstructor);
```

■ events

Provee la fundamental clase *EventEmitter* de la que cualquier objeto que emite eventos en *Node* hereda. Si alguna clase del código de un programa debe emitir eventos, ésta tiene que heredar de *EventEmitter*.

■ stream

Interfaz abstracta que representa los flujos de caracteres de *Unix* de la cual muchas clases en *Node* heredan.

■ crypto

Algoritmos y capacidades de cifrado para otros módulos y para el código de programa en general.

■ tls

Comunicaciones cifradas en la capa de transporte con el protocolo *TLS/SSL*, que proporciona infraestructura de clave pública/privada.

■ string_decoder

Proporciona una manera de, a partir de un Buffer, obtener cadenas de caracteres codificados en utf-8.

■ fs File System

Funciones para trabajar con el sistema de ficheros de la manera que establece el estándar *POSIX*. Todos los métodos permiten trabajar de forma asíncrona (el programa sigue su curso y *Node* avisa cuando ha terminado la operación con el fichero) o síncrona (la ejecución del programa se detiene hasta que se haya completado la operación con el fichero).

▪ path

Operaciones de manejo y transformación de la ruta de archivos y directorios, a nivel de nombre, sin consultar el sistema de ficheros.

▪ net

Creación y manejo asíncrono de servidores y clientes, que implementan la interfaz *Stream* mencionada antes, sobre el protocolo de transporte *TCP*.

▪ dgram

Creación y manejo asíncrono de datagramas sobre el protocolo transporte *UDP*.

▪ dns

Métodos para tratar con el protocolo *DNS* para la resolución de nombres de dominio de Internet.

▪ http

Interfaz de bajo nivel, ya que sólo maneja los *Streams* y el paso de mensajes, para la creación y uso de conexiones bajo el protocolo *HTTP*, tanto del lado del cliente como del servidor. Diseñada para dar soporte hasta a las características más complejas del protocolo como chunk-encoding.

▪ https

Versión del protocolo *HTTP* sobre conexiones seguras *TLS/SSL*.

▪ url

fragmenta

Formateo y análisis de los campos de las *URL*.

▪ querystrings

Utilidades para trabajar con las *queries* en el protocolo *HTTP*. Una *query* son los parámetros que se envían al servidor en las peticiones *HTTP*. Dependiendo del tipo de petición (*GET* o *POST*), pueden formar parte de la *URL* por lo que deben codificarse o escaparse y concatenarse de una manera especial para que sean interpretadas como tal.

▪ readline

Permite la lectura línea por línea de un *Stream*, especialmente indicado para el de la entrada estándar (*STDIN*).

▪ repl

Bucle de lectura y evaluación de la entrada estándar, para incluir en programas que necesiten uno. Es exactamente el mismo módulo que usa *Node* cuando se inicia sin argumentos, en el modo *REPL* comentado con anterioridad.

→ La codificación de STREAMING fragmentada es un mecanismo de transferencia de datos de transmisión por secuencias disponible en la versión 1.1 del Protocolo de transferencia de hipertexto (*HTTP*). En la codificación de transferencia fragmentada, el flujo de datos se divide en una serie de 'fragmentos' que no se superponen. Los trozos se envían y se reciben de forma independiente. No es necesario conocer el flujo de datos fuera del fragmento que se procesa actualmente para el remitente y el receptor en un momento dado



- **vm**

Compilación y ejecución bajo demanda de código.

- **child_process**

Creación de procesos hijos y comunicación y manejo de su entrada, salida y error estándar con ellos de una manera no bloqueante.

- **assert**

Funciones para la escritura de tests unitarios.

- **tty**

Permite ajustar el modo de trabajo de la entrada estándar si ésta es un terminal.

- **zlib**

Compresión/descompresión de Streams con los algoritmos *zlib* y *gzip*. Estos formatos se usan, por ejemplo, en el protocolo HTTP para comprimir los datos provenientes del servidor. Es conveniente tener en cuenta que los procesos de compresión y descompresión pueden ser muy costosos en términos de memoria y consumo de CPU.

- **os**

Acceso a información relativa al sistema operativo y recursos hardware sobre los que corre Node.

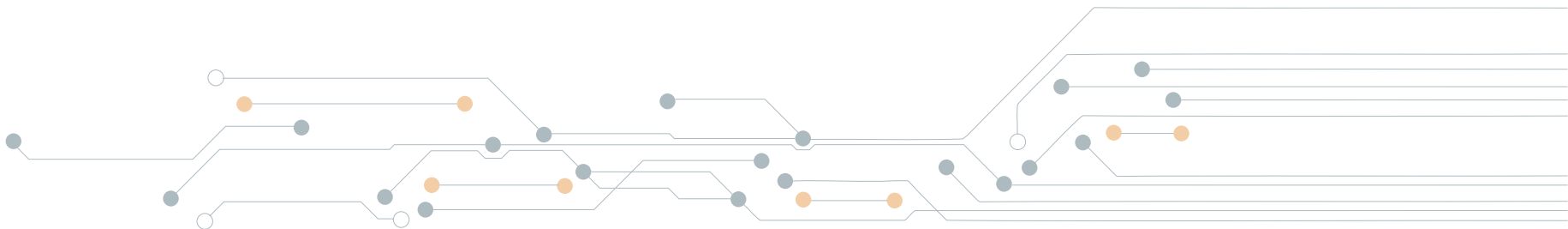
- **_debugger**

Es el depurador de código que Node tiene incorporado, a través de la opción *debug* de la línea de comandos. En realidad es un cliente que hace uso de las facilidades de depuración que el intérprete de Javascript que utiliza

Node ofrece a través de una conexión TCP al puerto 5858. Por tanto, no es un módulo que se importe a través de *require()* sino el modo de ejecución *Debug* del que se ha hablado antes.

- **cluster**

Creación y gestión de grupos de procesos Node trabajando en red para distribuir la carga en arquitecturas con procesadores multi-core.



▪ punycode

Implementación del algoritmo Punycode, disponible a partir de la versión 0.6.2, para uso del módulo url. El algoritmo Punycode se emplea para convertir de una manera unívoca y reversible cadenas de caracteres Unicode a cadenas de caracteres ASCII con caracteres compatibles en nombres de red.

El propósito es que los nombres de dominio internacionalizados (en inglés, *IDNA*), aquellos con caracteres propios de un país, se transformen en cadenas soportadas globalmente.

▪ domain

Módulo experimental en fase de desarrollo y, por tanto, no cargado por defecto para evitar problemas, aunque los autores de la plataforma aseguran un impacto mínimo. La idea detrás de este él es la de agrupar múltiples acciones de Entrada/Salida diferentes de tal manera que se dotan de un contexto definido para manejar los errores que puedan derivarse de ellas.

De esta manera el contexto no se pierde e incluso el programa continua su ejecución.

Quedan una serie de librerías, que no se mencionan en la documentación del API pero que existen en el directorio *lib/* del código fuente. Estas librerías tienen propósitos auxiliares para el resto de los módulos, aunque se pueden utilizarlas a través de *require()*:

▪ _linklist

Implementa una lista doblemente enlazada. Esta estructura de datos se emplea en *timers.js*, el módulo que provee funcionalidad de temporización.

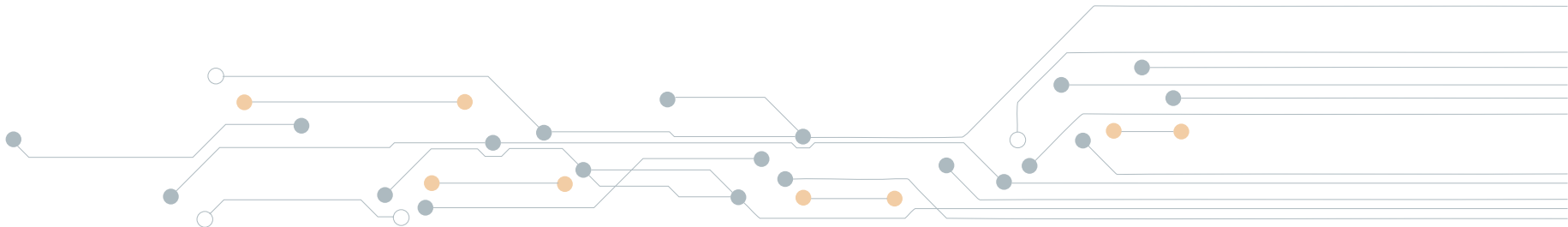
Su función es encadenar temporizadores que tengan el mismo tiempo de espera, *timeout*. Esta es una manera muy eficiente de manejar enormes cantidades de temporizadores que se activan por inactividad, como los *timeouts* de los *sockets*, en los que se reinicia el contador si se detecta actividad en él. Cuando esto ocurre, el temporizador, que está situado en la cabeza de la lista, se pone a la cola y se recalcula el tiempo en que debe expirar el primero.

▪ buffer_ieee754

Implementa la lectura y escritura de números en formato de coma flotante según el estándar *IEEE754* del *IEEE16* que el módulo *buffer* emplea para las operaciones con *Doubles* y *Floats*.

▪ constants

Todas las constantes posibles disponibles de la plataforma como, por ejemplo, las relacionadas con *POSIX* para señales del sistema operativo y modos de manejo de ficheros. Sólo realiza un *binding* con *node_constants.cc*.



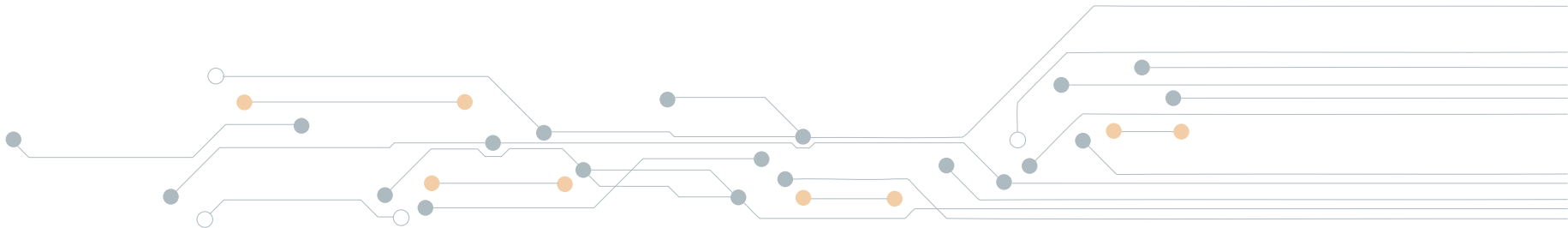
▪ freelist

Proporciona una sencilla estructura de pool o conjunto de objetos de la misma clase (de hecho, el constructor de los mismos es un argumento necesario).

Su utilidad se pone de manifiesto en el módulo `http`, donde se mantiene un conjunto de parsers HTTP reutilizables, que se encargan de procesar las peticiones `HTTP` que recibe un `Servidor`.

▪ sys

Es un módulo deprecado, en su lugar se debe emplear el módulo `utils`.



Telefonica

EDUCACIÓN DIGITAL