

5 - Módulo http

Un módulo fundamental para implementar aplicaciones en un servidor web es el 'http'. 'http' es un módulo del core de Node.js e implementado en C para una mayor eficiencia en las conexiones web.

Protocolo HTTP

Lo primero que haremos es repasar el funcionamiento de este protocolo tan importante en Internet.

HTTP (HyperText Transfer Protocol) o (Protocolo de transferencia de hipertexto) permite la transferencia de datos entre un servidor web y normalmente un navegador.

Cuando accedemos a un sitio web desde un navegador escribimos entre otras cosas:

`http://host[:puerto][/ruta y archivo][?parámetros]`

- http (indica el protocolo que utilizamos para conectarnos con un servidor web)
- host (es el nombre del dominio por ej. google.com.ar)
- puerto (es un número que generalmente no lo disponemos ya que por defecto el protocolo http utiliza el nro 80, salvo que nuestro servidor escuche peticiones en otro puerto que ya en este caso si debemos indicarlo)
- [/ruta y archivo] (indica donde se encuentra el archivo en el servidor)
- ?parámetros (datos que se pueden enviar desde el cliente para una mayor identificación del recurso que solicitamos)

Desde el navegador parte el pedido y el servidor tiene por objetivo responder a esa petición.

Servidor web con Node.js

Cuando trabajamos con la plataforma Node.js debemos codificar nosotros el servidor web (muy distinto a como se trabaja con PHP, Asp.Net, JSP etc. donde disponemos en forma paralela un servidor web como puede ser el Apache o IIS)

Veremos inicialmente la implementación de un servidor web utilizando solo el módulo 'http' pero más adelante incorporaremos otros módulos que nos facilitan el desarrollo de sitios web complejos (de todos modos siempre utilizando y extendiendo las funcionalidades del módulo 'http')

Problema

Implementar un servidor web con Node.js que retorne HTML con un mensaje indicando que el sitio esta en desarrollo. No importa que archivo pidamos del servidor retornar siempre el mismo HTML.

Creamos el archivo ejercicio7.js y lo guardamos en la carpeta donde estamos haciendo nuestros prácticos para poder probarlo:

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

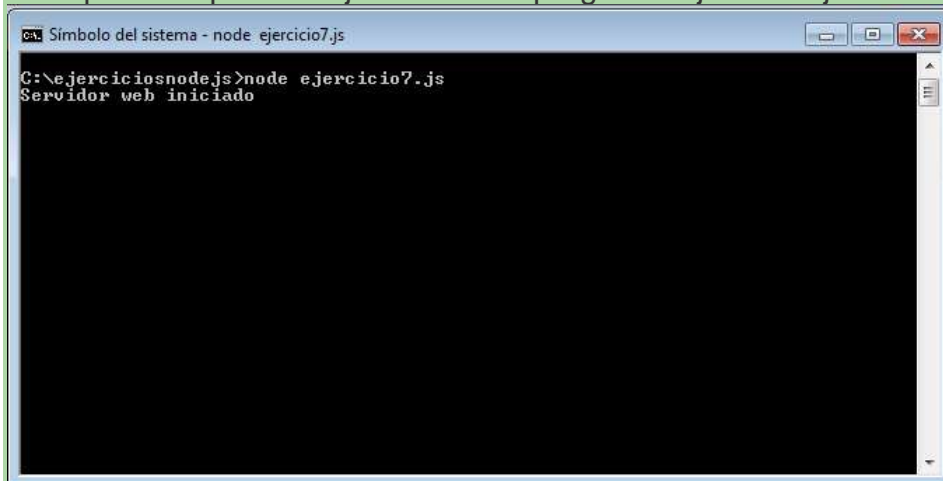
var servidor=http.createServer(function(pedido,respuesta){
  respuesta.writeHead(200, {'Content-Type': 'text/html'});
  respuesta.write('<!doctype html><html><head></head>'+
```

```
'<body><h1>Sitio en
desarrollo</h1></body></html>');
  respuesta.end();
});

servidor.listen(8888);

console.log('Servidor web iniciado');
```

Para probarlo primero ejecutamos el programa ejercicio7.js:

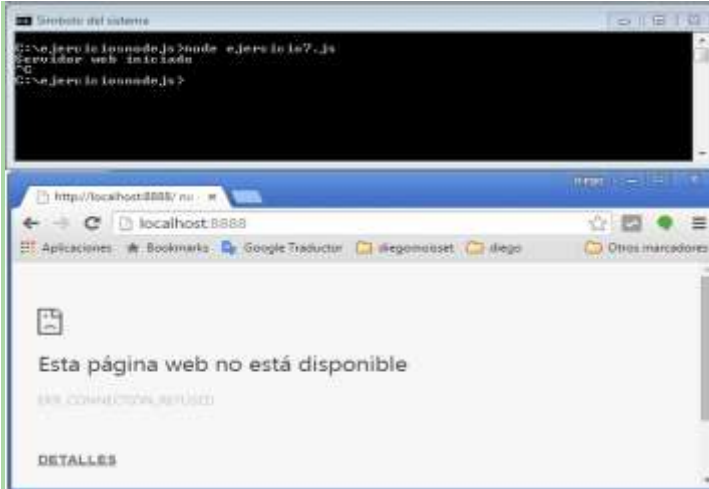


Como vemos luego de ejecutarlo el programa muestra el mensaje 'Servidor web iniciado' y no finaliza, esto debido a que esta esperando peticiones en el puerto 8888.

Podemos abrir nuestro navegador y acceder al servidor local (localhost) en el puerto 8888 y ver que ahora tenemos como resultado:



Si detenemos el servidor que creamos desde la consola (Presionamos las teclas Ctrl C que aborta el programa) y solicitamos nuevamente datos al servidor podremos ver que ahora el servidor no responde:



Es importante entonces tener en cuenta que el programa Node.js este ejecutándose para poder pedirle recursos. Otra cosa muy importante es que cada vez que hagamos cambios en el código fuente de nuestra aplicación en JavaScript debemos detener y volver a lanzar el programa para que tenga en cuenta las modificaciones.

Pasemos ahora a analizar el código de nuestro servidor. Primero requerimos el módulo 'http' y guardamos una referencia en la variable http:

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

El módulo 'http' tiene una función llamada `createServer` que tiene por objetivo crear un servidor que implementa el protocolo HTTP.

La función `createServer` debemos enviarle una función anónima (o podemos implementar una función definida como vimos en el concepto anterior) con dos parámetros que los hemos llamado `pedido` y `respuesta`.

Los objetos `pedido` y `respuesta` los crea la misma función `createServer` y los pasa cuando se dispara el pedido de una página u otro recurso al servidor.

Igual que cuando vimos archivos de texto estamos utilizando programación asincrónica, la función `createServer` se ejecuta en forma asíncrona lo que significa que no se detiene, sino que sigue con la ejecución de la siguiente función:

```
servidor.listen(8888);
```

La función `listen` (escucha) que también es asíncrona se queda esperando a recibir peticiones.

Antes que solicitemos una página desde el navegador podemos ver en la consola el mensaje de: 'Servidor web iniciado'.

El programa como podemos ver desde la consola no ha finalizado sino que esta ejecutandose un ciclo infinito en la función `listen` esperando peticiones de recursos.

Dijimos que cuando hay una solicitud de recursos al servidor se dispara la función anónima llegando dos objetos como parámetro:

```
var servidor=http.createServer(function(pedido,respuesta){
    respuesta.writeHead(200, {'Content-Type': 'text/html'});
    respuesta.write('<!doctype html><html><head></head>'+
                    '<body><h1>Sitio en
desarrollo</h1></body></html>');
    respuesta.end();
});
```

El primer parámetro que lo iremos viendo a medida que avance el tutorial contiene entre otros datos el nombre del archivo que solicitamos, información del navegador que hace la petición etc.

En nuestro programa actual al parámetro `pedido` no lo utilizamos

El parámetro `respuesta` es el que tenemos que llamar a los métodos:

- writeHead: es para indicar la cabecera de la petición HTTP (en esta caso indicamos con el código 200 que la petición fue Ok y con el segundo parámetro inicializamos la propiedad Content-Type indicando que retornaremos una corriente de datos de tipo HTML)
- write: mediante la función write indicamos todos los datos propiamente dicho del recurso a devolver al cliente (en este caso indicamos en la cabecera de la petición que se trataba de HTML)
- end: finalmente llamando a la función end finalizamos la corriente de datos del recurso (podemos llamar varias veces a la función write previo a llamar por única vez a end)

Otra cosa importante de notar de nuestro servidor web elemental es que no importa que archivo pidamos a nuestro servidor web siempre nos devolverá el código HTML que indicamos en la función anónima que le pasamos a createServer (en este ejemplo solicito el archivo pagina1.html, lo mismo sucedería si pido otras páginas: pagina2.html, pagina3.php etc.):

6 - Modulo http:datos que envía el navegador.

Hemos visto que en Node.js debemos crear en nuestra aplicación un servidor web y mediante una función asincrónica capturar las peticiones que envíe el navegador. Recordemos el programa del concepto anterior: ejercicio7.js

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

var servidor=http.createServer(function(pedido,respuesta){
    respuesta.writeHead(200, {'Content-Type': 'text/html'});
    respuesta.write('<!doctype html><html><head></head>'+
        '<body><h1>Sitio en
desarrollo</h1></body></html>');
    respuesta.end();
});

servidor.listen(8888);

console.log('Servidor web iniciado');
```

Cada vez que sucede una petición de recurso (puede ser una página, una imagen, un archivo de sonido etc.) se ejecuta la función anónima que le pasamos a createServer.

No vimos ni hablamos nada del primer parámetro de la función: el cual lo llamamos "pedido"

Este objeto tiene muchos datos que llegan al servidor. Analizaremos la propiedad url de este objeto.

Se trata de un string con la ubicación exacta del recurso que pide el navegador al servidor, por ejemplo:

`http://localhost:8888/carpeta1/pagina1.html?parametro1=10&metro2=20`

Estamos solicitando al servidor localhost el archivo pagina1.html que se encuentra en la carpeta 'carpeta1' del servidor y tiene dos parámetros.

Veamos como podemos recuperar estos datos en la función anónima que debe responder la petición.

Codifiquemos el archivo ejercicio8.js:

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

var servidor=http.createServer(function(pedido,respuesta){
    var objetourl = url.parse(pedido.url);
    console.log('camino completo del recurso y
parametros:'+objetourl.path);
    console.log('solo el camino y nombre del recurso
:'+objetourl.pathname)
    console.log('parametros del recurso
:'+objetourl.query)
    respuesta.writeHead(200, {'Content-Type': 'text/html'});
```

```
respuesta.write('<!doctype
html><html><head></head><body>Hola mundo</body></html>');
respuesta.end();
});

servidor.listen(8888);

console.log('Servidor web iniciado');
```

Requerimos otro módulo llamado 'url' (este módulo nos facilita el análisis de las distintas partes de una url) además del módulo 'http' que ya conocemos:

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

En la función anónima llamamos al método parse del objeto 'url' y le pasamos como parámetro la propiedad url del objeto pedido que llega a la función:

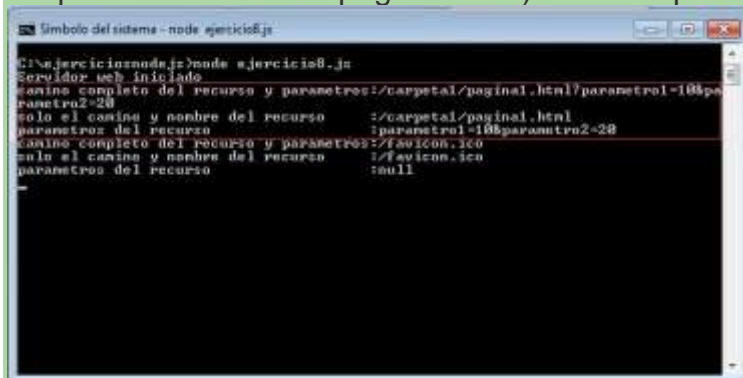
```
var objetourl = url.parse(pedido.url);
```

Esta actividad lo que hace es parsear (procesar el string que contiene pedido.url) y descomponernos las distintas partes de una url en un objeto literal para facilitar analizar su contenido.

Ejecutemos este programa (ejercicio8.js) y dispongamos desde el navegador la url:

<http://localhost:8888/carpeta1/pagina1.html?parametro1=10¶metro2=20>

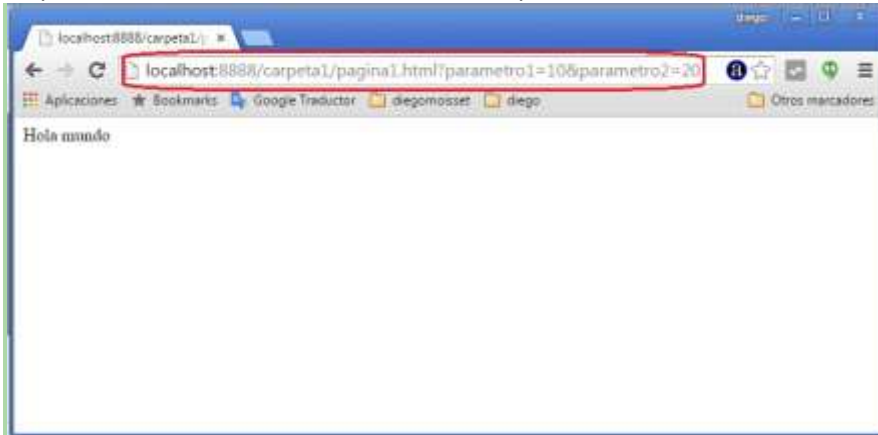
Luego de solicitar la página desde el navegador (recordar que no hace falta que existe la carpeta 'carpeta1' ni el archivo 'pagina1.html') veamos que nos muestra la consola:



El objeto literal objetourl tiene tres propiedades fundamentales para recuperar los datos en forma parcial y agrupada de la petición HTTP que hizo el navegador:

```
console.log('camino completo del recurso y
parametros:'+objetourl.path);
console.log('solo el camino y nombre del recurso
:'+objetourl.pathname)
console.log('parametros del recurso
:'+objetourl.query)
```

Con estos datos mediante if podemos redirigir el pedido al recurso respectivo (tengamos en cuenta que por el momento siempre nuestro sitio devuelve el 'Hola mundo'):



Hay otra cosa que debemos nombrar: los navegadores la primera vez que acceden a un sitio piden un recurso (el archivo favicon.ico) y es eso por lo que se ejecuta dos veces la función anónima, una para solicitar el archivo 'archivo1' y otra para solicitar el archivo 'favicon.ico' (más adelante veremos como retornar dicho archivo y otros que se soliciten)

7 - Servidor web con Node.js que sirve páginas estáticas HTML

Hasta ahora hemos visto como podemos recuperar los datos de la petición HTML. También hemos visto como podemos leer un archivo con el módulo 'fs'.

En este concepto nos concentraremos en como dada una petición de una página HTML proceder a verificar si existe dicha página, leerla y retornarla al navegador que la solicitó.

Vamos a crear un subdirectorio llamado 'static' (puede tener cualquier nombre) en la carpeta donde crearemos nuestra aplicación Node.js y almacenaremos tres páginas HTML: index.html, pagina1.html y pagina2.html

El contenido de estos tres archivos será:

index.html

```
<!doctype html>
<html>
<head>
</head>
<body>
  <h1>Principal</h1>
  <p>Pagina 1:<a href="pagina1.html">Entrar</a></p>
  <p>Pagina 2:<a href="pagina2.html">Entrar</a></p>
</body>
</html>
```

pagina1.html

```
<!doctype html>
<html>
<head>
</head>
<body>
  <h1>Pagina 1</h1>
  <p><a href="index.html">Retornar</a></p>
</body>
</html>
```

pagina2.html

```
<!doctype html>
<html>
<head>
</head>
<body>
  <h1>Pagina 2</h1>
  <p><a href="index.html">Retornar</a></p>
</body>
</html>
```


Tener bien en cuenta de grabar estos tres archivos HTML en una subcarpeta llamada static. Ahora creemos nuestro programa en Node.js que se encargará de responder a peticiones de páginas HTML.

Creemos el archivo:

`ejercicio9.js`

```
var http=require('http');
var url=require('url');
var fs=require('fs');

var servidor=http.createServer(function(pedido, respuesta) {
  var objetourl = url.parse(pedido.url);
  var camino='static'+objetourl.pathname;
  if (camino=='static/')
    camino='static/index.html';
  fs.exists(camino,function(existe) {
    if (existe) {
      fs.readFile(camino,function(error, contenido) {
        if (error) {
          respuesta.writeHead(500, {'Content-
Type': 'text/plain'}));
          respuesta.write('Error interno');
          respuesta.end();
        } else {
          respuesta.writeHead(200, {'Content-
Type': 'text/html'}));
          respuesta.write(contenido);
          respuesta.end();
        }
      });
    } else {
      respuesta.writeHead(404, {'Content-Type':
'text/html'}));
      respuesta.write('<!doctype
html><html><head></head><body>Recurso
inexistente</body></html>');
      respuesta.end();
    }
  });
});

servidor.listen(8888);

console.log('Servidor web iniciado');
```

Lo primero que hacemos es requerir los módulos 'http', 'url' y 'fs' que hemos analizado en conceptos anteriores:

```
var http=require('http');
var url=require('url');
var fs=require('fs');
```

Procedemos a crear un servidor de peticiones HTTP, tema ya visto:

```
var servidor=http.createServer(function(pedido,respuesta) {
    ....
});
```

En la función anónima llamamos al método parse del objeto 'url' y le pasamos como parámetro la propiedad url del objeto pedido que llega a la función:

```
var objetourl = url.parse(pedido.url);
```

Obtenemos las distintas partes de la url en un objeto literal para facilitar extraer solo el camino y nombre del archivo HTML.

Inicializamos una variable con el nombre de la subcarpeta que contiene los archivos HTML y le concatenamos el camino y nombre del archivo HTML solicitado:

```
var camino='static'+objetourl.pathname;
```

Por ejemplo podrían ser:

```
static/
static/index.html
static/pagina1.html
static/pagina2.html
```

También podrían ser peticiones de páginas que no existen, por ejemplo:

```
static/pagina5.html
static/carpetal/pagina1.html
etc.
```

El primer control que hacemos es verificar si en la url no viene ninguna página, en dicho caso retornamos el archivo index.html que es el principal del sitio, para verificar hacemos un if:

```
if (camino=='static/')
    camino='static/index.html';
```

Por ejemplo si disponemos:

```
http://localhost:8888/
o
http://localhost:8888
```

Estaríamos para este caso retornando el archivo index.html.

Mediante el objeto fs procedemos a verificar si existe el archivo HTML, el método exists tiene como primer parámetro el nombre del archivo que debemos indicarlo con todo el camino y el segundo parámetro es una función anónima que llega como parámetro si existe o no el archivo:

```
fs.exists(camino,function(existe){
    if (existe) {
        ....
    } else {
        ....
    }
});
```

Veamos primero si no existe el archivo, en dicho caso se ejecuta el else del if y procedemos a devolver al navegador un mensaje y el código 404 de recurso inexistente:

```
fs.exists(camino,function(existe){
    if (existe) {
        ....
    }
});
```

```

    } else {
        respuesta.writeHead(404, {'Content-Type':
'text/html'});
        respuesta.write('<!doctype
html><html><head></head><body>Recurso inexistente</body></html>');
        respuesta.end();
    }
});

```

Veamos que sucede si el if se verifica verdadero, es decir si existe el archivo HTML:

```

fs.exists(camino, function(existe) {
    if (existe) {
        fs.readFile(camino, function(error, contenido) {
            if (error) {
                respuesta.writeHead(500, {'Content-Type':
'text/plain'});
                respuesta.write('Error interno');
                respuesta.end();
            } else {
                respuesta.writeHead(200, {'Content-Type':
'text/html'});
                respuesta.write(contenido);
                respuesta.end();
            }
        });
    } else {
        ....
    }
});

```

Si existe el archivo procedemos a llamar al método readFile para leer su contenido. El método readFile tiene dos parámetros, el primero es el nombre del archivo HTML a leer (que debemos indicar siempre todo el path) y el segundo parámetro es una función anónima que tiene dos parámetros que son si hubo error y el contenido del archivo:

```

fs.readFile(camino, function(error, contenido) {
    ....
});

```

Cuando se ejecuta la función anónima que ocurre luego de traer a memoria el contenido del archivo verificamos si no hubo un error en la lectura y en caso negativo procedemos mediante el objeto 'respuesta' a devolver al navegador el contenido completo del archivo indicando que se trata de un archivo HTML:

```

fs.readFile(camino, function(error, contenido) {
    if (error) {
        ....
    } else {
        respuesta.writeHead(200, {'Content-Type':
'text/html'});
        respuesta.write(contenido);
        respuesta.end();
    }
});

```

Si hubo error interno en el servidor cuando se lee el archivo HTML procedemos a retornar el código 500 para que el navegador conozca tal situación (tengamos en cuenta que el archivo existe pero por algún motivo luego de verificar que existía no se ha podido leer):

```
fs.readFile(camino, function(error, contenido) {  
    if (error) {  
        respuesta.writeHead(500, {'Content-Type':  
'text/plain'});  
        respuesta.write('Error interno');  
        respuesta.end();  
    } else {  
        ....  
    }  
});
```

8 - Servidor web de archivos estáticos html, css, jpg, mp3, mp4, ico etc.

Vimos en el concepto anterior como crear un servidor web solo de páginas HTML. Ahora extenderemos nuestro servidor web de páginas estáticas a otros formatos de archivos.

Un sitio web como sabemos está constituido entre otro por archivos html, js, css, jpg, gif, mp3, mp4, ico etc.

Siempre que devolvemos un recurso a un cliente (normalmente el navegador) en la cabecera de respuesta indicamos el tipo de recurso devuelto:

```
respuesta.writeHead(200, {'Content-Type':
'text/html'});

respuesta.write(contenido);
respuesta.end();
```

Debemos inicializar el valor 'Content-Type' con el tipo de recurso a devolver, como en el concepto anterior solo devolvíamos archivos html nos bastaba con indicar siempre el valor: 'text/html'

Protocolo MIME

Los MIME Types son la manera estándar de mandar contenido a través de internet. Especifican a cada archivo con su tipo de contenido.

Según el tipo de archivo que retornamos indicamos un valor distinto a Content-Type. Ejemplos de valores:

```
Content-type: text/html
Content-type: text/css
Content-type: image/jpg
Content-type: image/x-icon
Content-type: audio/mpeg3
Content-type: video/mp4
etc.
```

Un listado bastante completo de tipos MIME lo podemos ver en este [sitio](#).

Problema

Confeccionaremos un sitio que contenga una serie de archivos html, css, jpg, mp3, mp4 e ico.

Crearemos un directorio llamado ejercicio10 y dentro de este otro directorio interno llamado static donde dispondremos todos los archivos html,css, jpg, mp3, mp4 e ico.

En el directorio ejercicio10 tipearemos nuestra aplicacion Node.js que tiene por objetivo servir las páginas HTML y otros recursos que pidan los navegadores web.

La aplicación Node.js la llamamos ejercicio10.js

ejercicio10.js

```
var http=require('http');
var url=require('url');
var fs=require('fs');

var mime = {
  'html' : 'text/html',
  'css' : 'text/css',
  'jpg' : 'image/jpg',
  'ico' : 'image/x-icon',
```

```
'mp3'    : 'audio/mpeg3',
'mp4'    : 'video/mp4'
};

var servidor=http.createServer(function(pedido,respuesta) {
  var objetourl = url.parse(pedido.url);
  var camino='static'+objetourl.pathname;
  if (camino=='static/')
    camino='static/index.html';
  fs.exists(camino,function(existe) {
    if (existe) {
      fs.readFile(camino,function(error,contenido) {
        if (error) {
          respuesta.writeHead(500, {'Content-
Type': 'text/plain'}));
          respuesta.write('Error interno');
          respuesta.end();
        } else {
          var vec = camino.split('.');
          var extension=vec[vec.length-1];
          var mimearchivo=mime[extension];
          respuesta.writeHead(200, {'Content-
Type': mimearchivo});
          respuesta.write(contenido);
          respuesta.end();
        }
      });
    } else {
      respuesta.writeHead(404, {'Content-Type':
'text/html'});
      respuesta.write('<!doctype
html><html><head></head><body>Recurso
inexistente</body></html>');
      respuesta.end();
    }
  });
});

servidor.listen(8888);

console.log('Servidor web iniciado');
```

Este proyecto lo puede descargar en un zip con todos los archivos js, html, jpg etc. desde este enlace : [ejercicio10](#)

Veamos los cambios que debemos hacer con respecto al concepto anterior que servíamos solo archivos HTML.

Primero definimos un objeto literal asociando las distintas extensiones de archivos y su valor MIME:

```
var mime = {  
  'html' : 'text/html',  
  'css' : 'text/css',  
  'jpg' : 'image/jpg',  
  'ico' : 'image/x-icon',  
  'mp3' : 'audio/mpeg3',  
  'mp4' : 'video/mp4'  
};
```

Todo el resto del código es idéntico al concepto anterior salvo cuando tenemos que retornar el recurso que solicita el navegador:

```
      var vec = camino.split('.');  
      var extension=vec[vec.length-1];  
      var mimearchivo=mime[extension];  
      respuesta.writeHead(200, {'Content-Type':  
mimearchivo});  
  
      respuesta.write(contenido);  
      respuesta.end();
```

Descomponemos en un vector separando por el punto la variable camino:

```
      var vec = camino.split('.');
```

Si tenemos en la variable camino el valor:

/pagina1.html

Luego de ejecutarse el método split pidiendo que separe por el caracter punto tenemos en la variable vec dos elementos:

```
vec[0] tiene almacenado /pagina1  
vec[1] tiene almacenado html
```

Sacamos el último elemento del vector que en definitiva almacena la extensión del archivo:

```
      var extension=vec[vec.length-1];
```

Seguidamente rescatamos la propiedad del objeto literal mime:

```
      var mimearchivo=mime[extension];
```

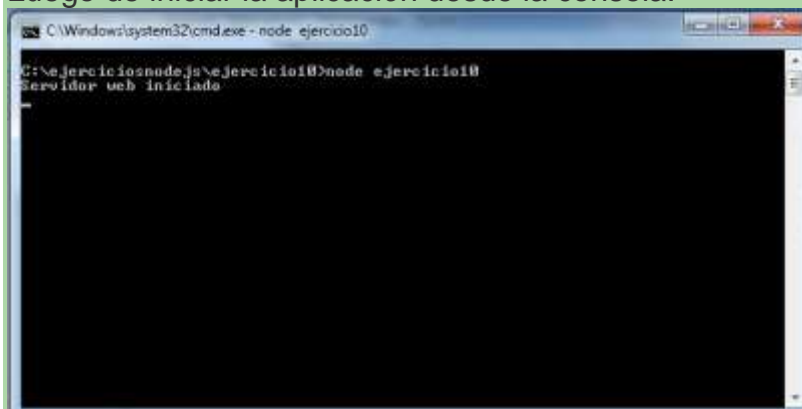
Por ejemplo si tenemos en la variable extension el valor 'html' luego en la variable mimearchivo se almacena: 'text/html' que es el valor para dicha propiedad definida en la variable mime.

Ahora llamamos al método writeHead donde retornamos el tipo MIME para dicha extensión:

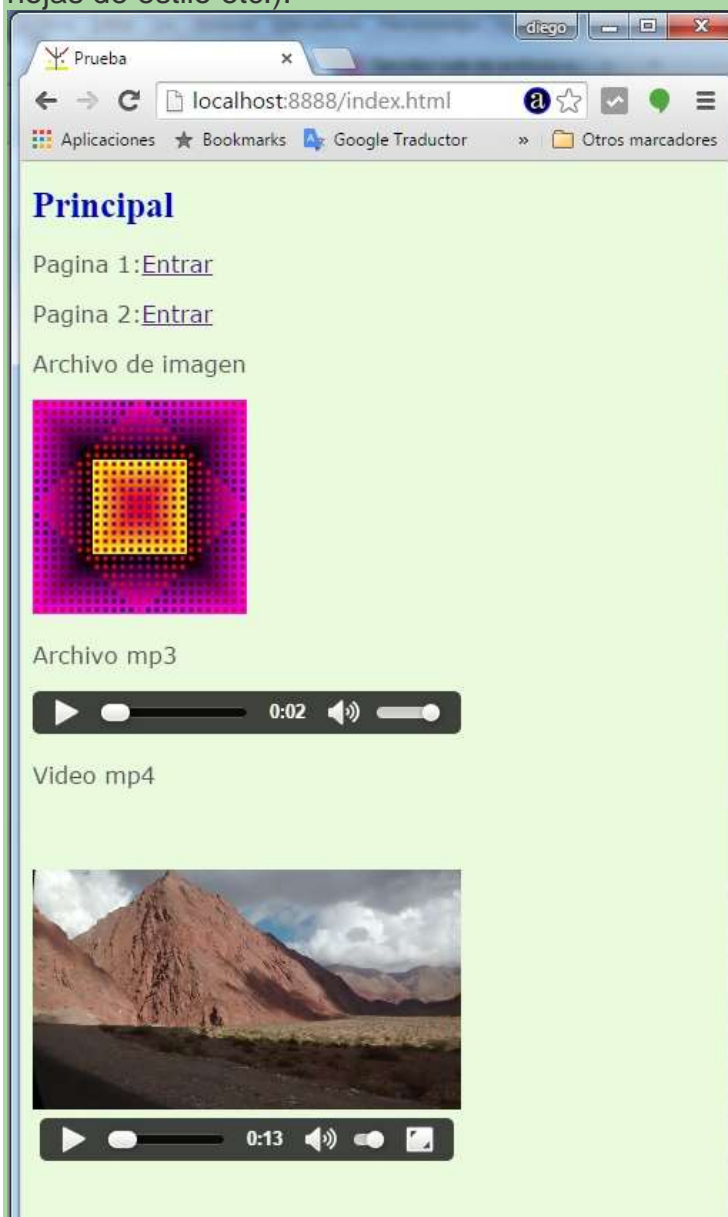
```
      respuesta.writeHead(200, {'Content-Type':  
mimearchivo});
```

El resto de este programa no sufre cambios con respecto a devolver siempre archivos HTML.

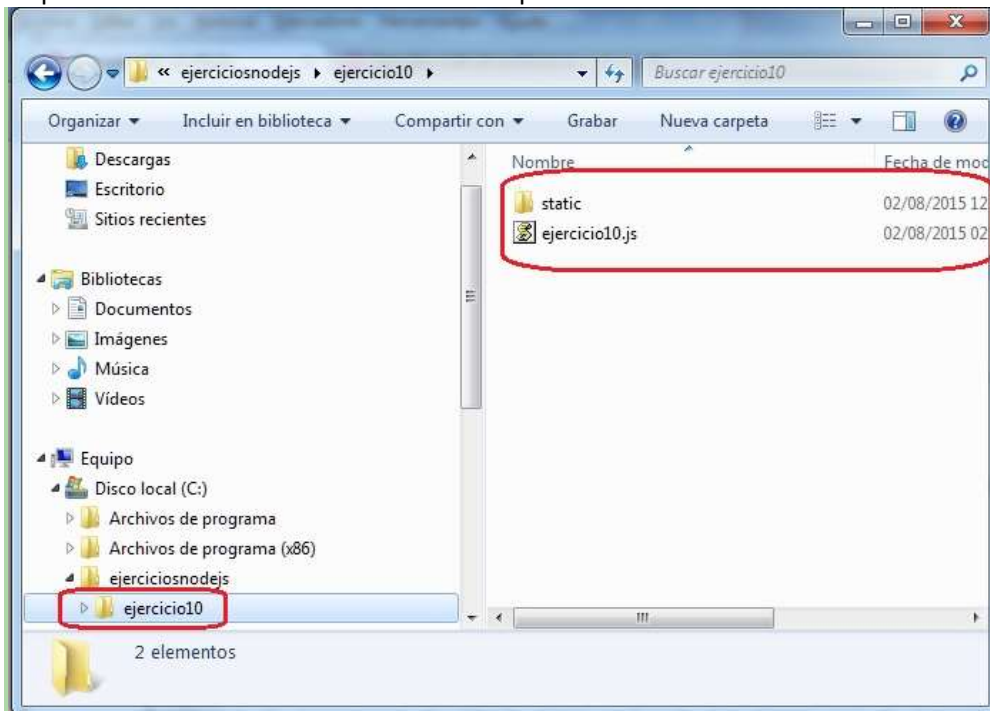
Luego de iniciar la aplicación desde la consola:



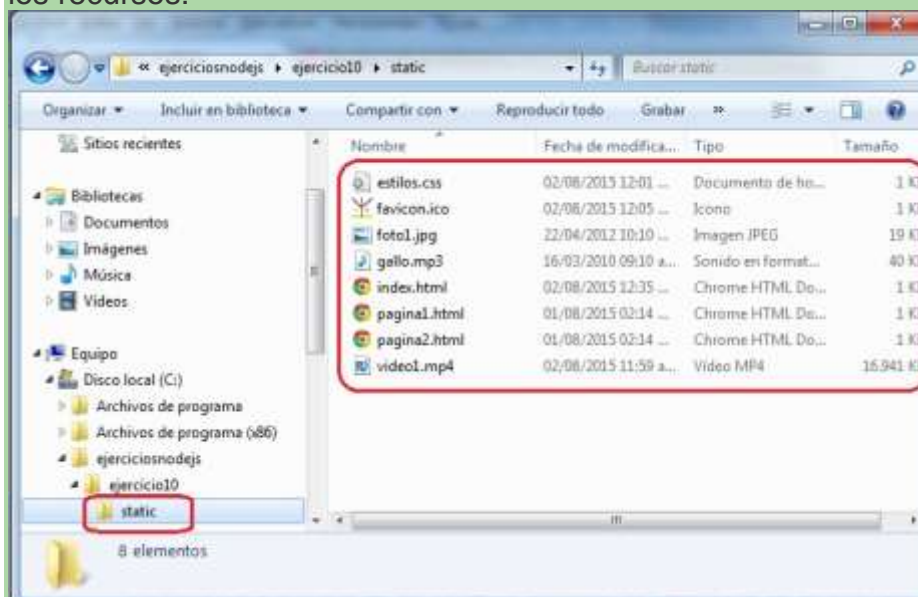
Ya podemos solicitar páginas al servidor que acabamos de inicializar y comprobar que nos retorna los distintos recursos enumerados dentro de cada página (imágenes, audios, videos, hojas de estilo etc.):



La organización de los archivos en el disco duro es:



Como vemos en la carpeta ejercicio10 se ubica la aplicación Node.js y la carpeta static con todos los recursos:



En nuestro ejemplo hemos limitado a servir 6 formatos de archivos distintos:

```
var mime = {  
  'html' : 'text/html',  
  'css'  : 'text/css',  
  'jpg'  : 'image/jpg',  
  'ico'  : 'image/x-icon',  
  'mp3'  : 'audio/mpeg3',  
  'mp4'  : 'video/mp4',  
};
```

Para servir otros formatos de archivos deberíamos agregar al objeto literal mime los valores respectivos que define el protocolo MIME.

9 - Servidor web de archivos y creación de una cache.

Hemos visto en los dos conceptos anteriores como crear un servidor web con Nodo.js que nos retorna distintos formatos de archivos del servidor (html, js, jpg, gif, png, css, etc.)

La mecánica que empleamos es que en cada solicitud del cliente (navegador) analizamos primero si dicho recurso se encuentra en el servidor y en caso afirmativo procedemos a leerlo mediante las funciones que provee el módulo 'fs'. Finalmente lo retornamos al navegador que solicitó el recurso indicando del tipo de archivo que se trata para que el navegador pueda proceder a mostrarlo si es una imagen, renderizar si es un HTML etc.

Vamos a hacer una modificación al programa que hemos planteado hasta el momento para reducir drásticamente los tiempos requeridos de acceso al disco duro del servidor donde se almacenan los archivos. Tengamos en cuenta que el acceso a disco por operaciones de I/O son muy costosas en tiempo.

Lo que vamos a implementar es un sistema de cache en el servidor para almacenar el contenido de los archivos estáticos (es decir aquellos que no cambian) y tenerlos almacenados en la memoria RAM.

Problema

Desarrollar un sitio en Node.js que permita servir archivos estáticos y haga más eficiente su trabajo implementando un sistema de cache de los archivos servidor.

Crearemos una nueva carpeta para este proyecto con el nombre de ejercicio11 y en la misma crearemos una subcarpeta llamada static que contendrá todos los archivos html, css, jpg, mp3, mp4 etc.

En la carpeta ejercicio11 procederemos a codificar el programa en Node.js que arranca un servidor web de páginas estáticas y almacena las páginas pedidas en una cache para no tener que leerlas nuevamente del disco.

ejercicio11.js

```
var http=require('http');
var url=require('url');
var fs=require('fs');

var mime = {
  'html' : 'text/html',
  'css'  : 'text/css',
  'jpg'  : 'image/jpg',
  'ico'  : 'image/x-icon',
  'mp3'  : 'audio/mpeg3',
  'mp4'  : 'video/mp4'
};

var cache={};

var servidor=http.createServer(function(pedido,respuesta) {
  var objetourl = url.parse(pedido.url);
  var camino='static'+objetourl.pathname;
  if (camino=='static/')
    if (fs.existsSync(camino)) {
      respuesta.writeHead(200, {'Content-Type': mime[objetourl.pathname.substr(1, objetourl.pathname.length-1)]});
      fs.readFile(camino, function(err, data) {
        respuesta.end(data);
      });
    } else {
      respuesta.writeHead(404, {'Content-Type': 'text/plain'});
      respuesta.end('No encontrado');
    }
}
```

```

        camino='static/index.html';
    if (cache[camino]) {
        var vec = camino.split('.');
        var extension=vec[vec.length-1];
        var mimearchivo=mime[extension];
        respuesta.writeHead(200, {'Content-Type':
mimearchivo});
        respuesta.write(cache[camino]);
        respuesta.end();
        console.log('Recurso recuperado del cache:'+camino);
    } else {
        fs.exists(camino,function(existe) {
            if (existe) {

fs.readFile(camino,function(error,contenido) {
                if (error) {
                    respuesta.writeHead(500, {'Content-
Type': 'text/plain'}));
                    respuesta.write('Error interno');
                    respuesta.end();
                } else {
                    cache[camino]=contenido;
                    var vec = camino.split('.');
                    var extension=vec[vec.length-1];
                    var mimearchivo=mime[extension];
                    respuesta.writeHead(200, {'Content-
Type': mimearchivo});
                    respuesta.write(contenido);
                    respuesta.end();
                    console.log('Recurso leído del
disco:'+camino);
                }
            });
        } else {
            respuesta.writeHead(404, {'Content-Type':
'text/html'});
            respuesta.write('<!doctype
html><html><head></head><body>Recurso
inexistente</body></html>');
            respuesta.end();
        }
    });
}
});

```

```
servidor.listen(8888);
```

```
console.log('Servidor web iniciado');
```

Veamos lo nuevo que le agregamos a nuestro programa para poder almacenar los contenidos estáticos de los recursos de nuestro sitio web mediante una cache en memoria.

Primero definimos un objeto vacío donde almacenaremos los nombres de los recursos y los contenidos de los mismos:

```
var cache={};
```

Cuando se dispara un pedido de recurso y se ejecuta la función anónima que le pasamos al método createServer procedemos mediante un if a verificar si el objeto cache almacena una propiedad que coincide con el camino del recurso:

```
if (cache[camino]) {
    var vec = camino.split('.');
    var extension=vec[vec.length-1];
    var mimearchivo=mime[extension];
    respuesta.writeHead(200, {'Content-Type': mimearchivo});
    respuesta.write(cache[camino]);
    respuesta.end();
    console.log('Recurso recuperado del cache:'+camino);
} else {
```

Como vemos si el contenido está almacenado en la cache cuando escribimos dentro de la respuesta el dato a devolver procedemos a extraerlo del objeto 'cache' y accedemos a la propiedad que coincide con el recurso pedido:

```
    respuesta.write(cache[camino]);
```

En un principio debemos tener en cuenta que la cache está vacía por lo que el if analizado anteriormente se verifica falso por lo que se ejecuta el otro bloque donde verificamos que el recurso exista y procedemos a su lectura del disco:

```
        fs.readFile(camino,function(error,contenido){
            if (error) {
                respuesta.writeHead(500, {'Content-Type':
'text/plain'});

                respuesta.write('Error interno');
                respuesta.end();
            } else {
                cache[camino]=contenido;
                var vec = camino.split('.');
                var extension=vec[vec.length-1];
                var mimearchivo=mime[extension];
                respuesta.writeHead(200, {'Content-Type':
mimearchivo});

                respuesta.write(contenido);
                respuesta.end();
                console.log('Recurso leído del
disco:'+camino);
            }
        });
```

Luego de leído procedemos a almacenar el contenido del recurso en la variable 'cache' y como valor de propiedad indicamos el camino del recurso (esto hace que quede en la cache para cualquier otra petición que surja en el servidor del mismo archivo):

```
cache[camino]=contenido;
```

Este proyecto lo puede descargar en un zip con todos los archivos js, html, jpg etc. desde este enlace : [ejercicio11](#)

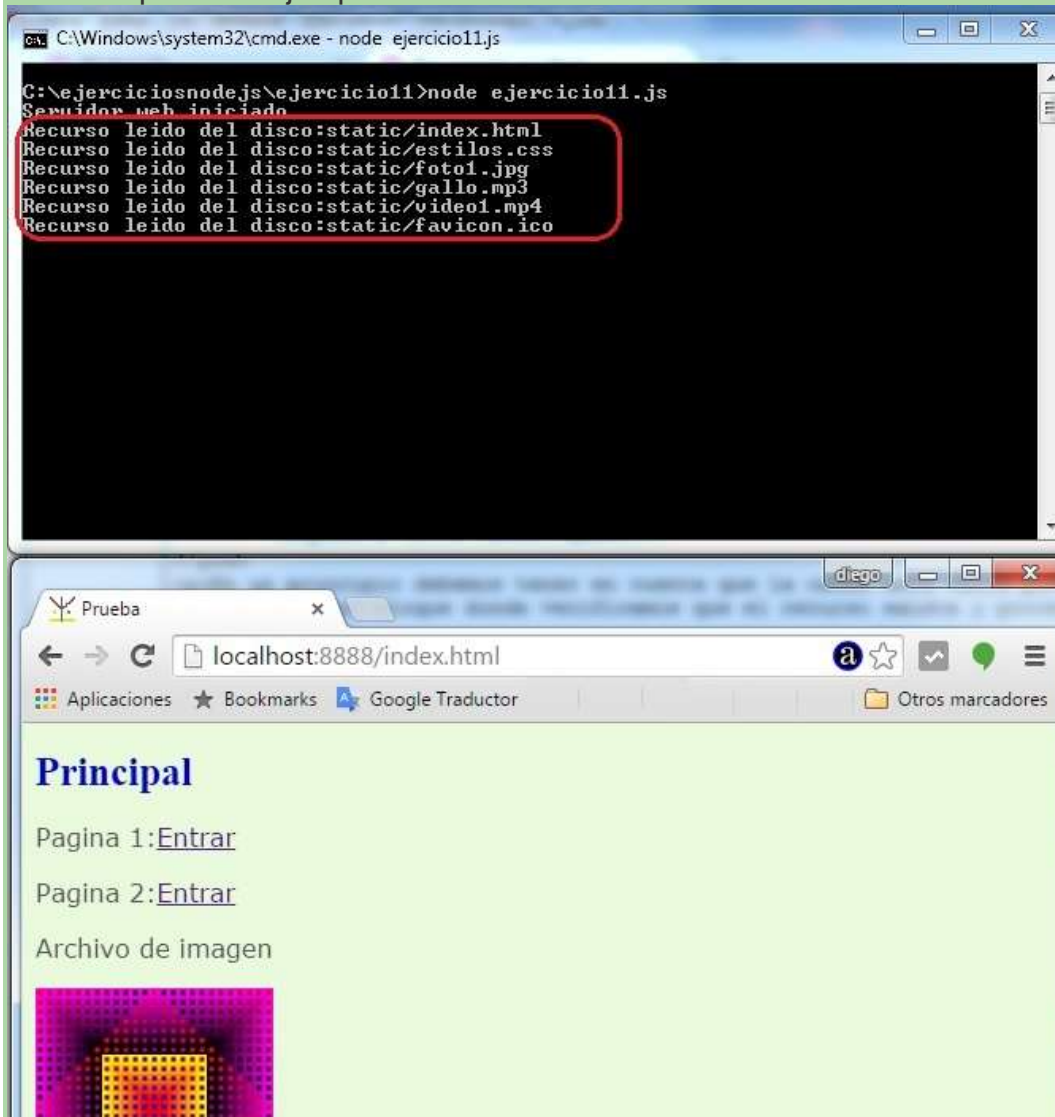
En el código hemos dispuesto la impresión de dos mensajes uno cuando se lee el recurso desde el disco:

```
console.log('Recurso leído del  
disco: '+camino);
```

Y otro cuando se recupera el recurso desde la cache:

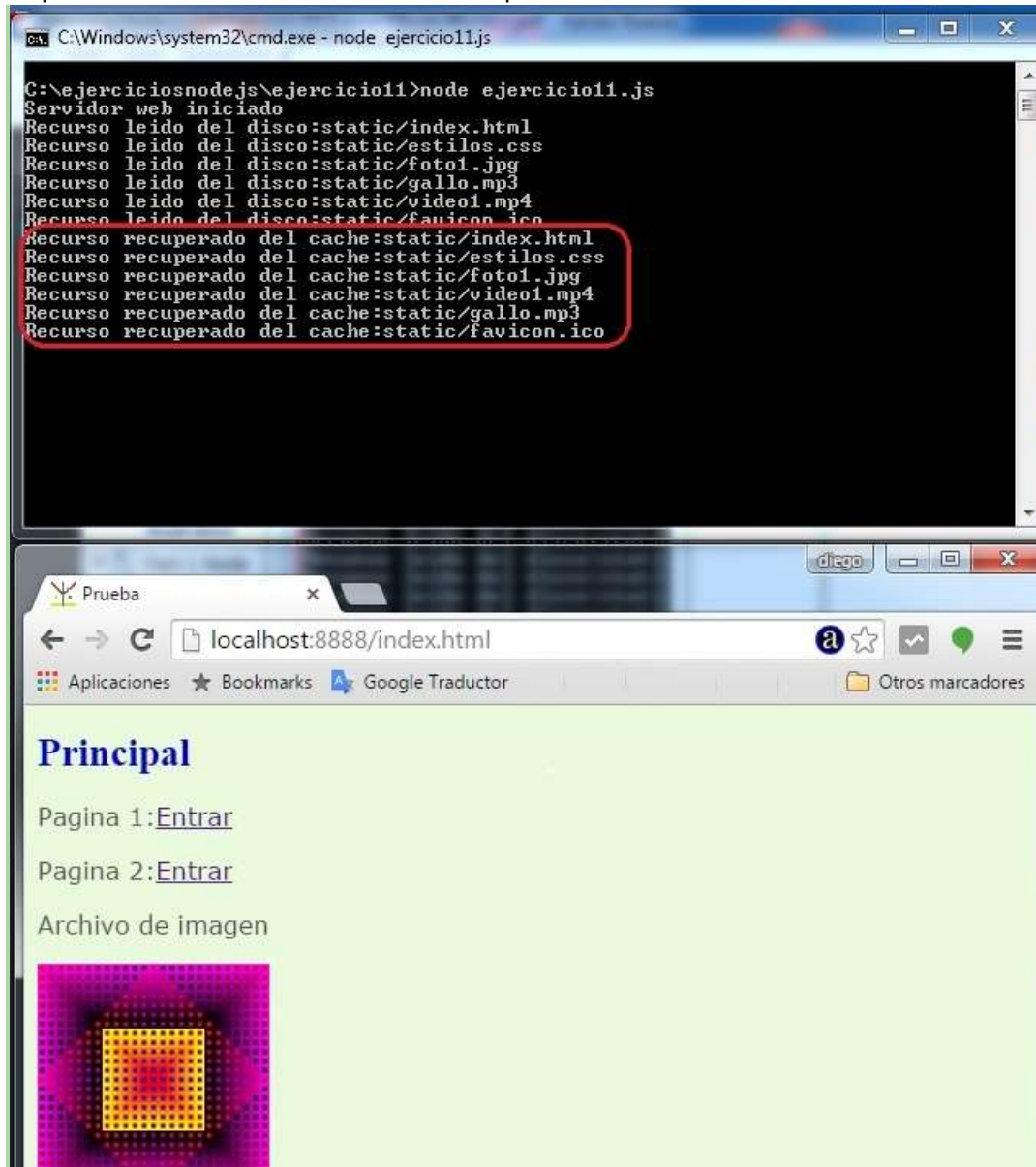
```
console.log('Recurso recuperado del cache: '+camino);
```

Procedamos a iniciar nuestro servidor desde la consola y solicitemos la página index.html, veamos que mensaje aparece en la consola:

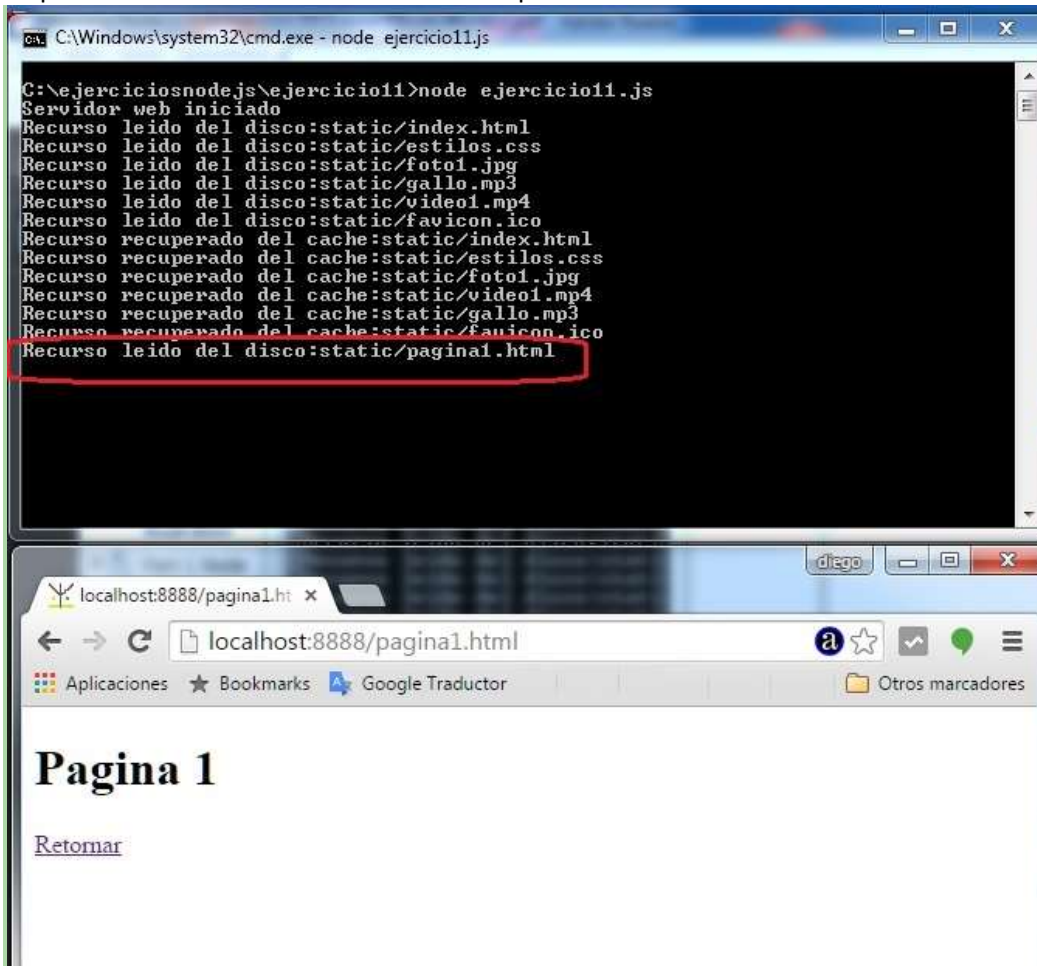


Como podemos observar los seis archivos han sido leídos del disco.

Refresquemos la página para solicitar nuevamente los recursos y tenemos como salida en la consola:



Ahora vemos que los recursos se recuperan de la cache. Por último presionemos el enlace a la página 1:



The image shows two windows. The top window is a command prompt running a Node.js server. The output shows that several resources (index.html, estilos.css, foto1.jpg, gallo.mp3, video1.mp4, favicon.ico) have been read from the disk and then recovered from the cache. The last line, 'Recurso leído del disco:static/pagina1.html', is highlighted with a red box. The bottom window is a web browser showing the page 'localhost:8888/pagina1.html'. The page content includes the title 'Pagina 1' and a link 'Retornar'.

```
C:\Windows\system32\cmd.exe - node ejercicio11.js
C:\ejerciciosnodejs\ejercicio11>node ejercicio11.js
Servidor web iniciado
Recurso leído del disco:static/index.html
Recurso leído del disco:static/estilos.css
Recurso leído del disco:static/foto1.jpg
Recurso leído del disco:static/gallo.mp3
Recurso leído del disco:static/video1.mp4
Recurso leído del disco:static/favicon.ico
Recurso recuperado del cache:static/index.html
Recurso recuperado del cache:static/estilos.css
Recurso recuperado del cache:static/foto1.jpg
Recurso recuperado del cache:static/video1.mp4
Recurso recuperado del cache:static/gallo.mp3
Recurso recuperado del cache:static/favicon.ico
Recurso leído del disco:static/pagina1.html
```

localhost:8888/pagina1.html

localhost:8888/pagina1.html

Aplicaciones Bookmarks Google Traductor Otros marcadores

Pagina 1

[Retornar](#)

La pagina1.html como todavía no se había leído no se encontraba en la cache por lo que en ese momento pasa a estar en la cache para futuros pedidos.

Tener en cuenta que estamos hablando de una cache en el servidor y no la cache del navegador. Es decir que cuando un cliente1 accede al sitio web y solicita el archivo: index.html a partir de este momento dicho archivo queda almacenado en la cache y cuando un cliente2 acceda al sitio y solicite el archivo: index.html su contenido lo recuperará de la cache y no del disco (esto hace mucho mas eficiente a nuestro programa)

Cuando detenemos nuestro programa Node.js la cache es una variable de memoria por lo que su contenido se pierde. Al arrancar nuevamente el servidor a medida que pidamos recursos la cache si irá cargando.