

HTTP

Contenido

- TCP y SOCKET
- HTTP

TCP

- Transmission Control Protocol, es el protocolo de comunicación más importante de los Internet protocol (IP). Proporciona un mecanismo de transporte a la capa de aplicaciones.
- En TCP se basan otros protocolos como: HTTP, iRC, SMTP e IMAP.
- Node implementa un servidor HTTP en forma de pseudo-class en `http.Server`, que proviene de la pseudo-clase `net.Server` del servidor TCP.

TCP: Crear un servidor

- require el modulo ('**net**');

```
require('net').createServer(function(socket) {  
  // new connection  
  socket.on('data', function(data) {  
    // got data  
  });  
  socket.on('end', function(data) {  
    // connection closed  
  });  
  socket.write('Some string');  
}).listen(4001);
```

- Usa el método **createServer()** del package **net**, que se vincula al puerto TCP 4001.
- Al método `createServer()` se pasa una función de callback que se invoca cada vez que se produce un evento 'connection'.
- Dentro del callback se maneja el objeto **socket**, que se puede utilizar para enviar y recibir datos a y desde un cliente.
- El objeto server es un *event emitter*, que emite los siguientes eventos:
 - “listening”: Cuando el servidor escucha en el Puerto y dirección especificados
 - “connection”: Cuando se establece una nueva conexión. El callback recibe el objeto socket
 - “close”: Cuando se cierra el servidor, termina el enlace al puerto.
 - “error” — Cuando ocurre un error a nivel de servidor. Ej: Puerto ocupado o no hay permisos para hacer el binding al puerto

TCP: Ciclo de vida de un servidor

```
var server = require('net').createServer();
var port = 4001;
server.on('listening', function() {
  console.log('Server is listening on port', port);
});
server.on('connection', function(socket) {
  console.log('Server has a new connection');
  socket.end();
  server.close();
});
server.on('close', function() {
  console.log('Server is now closed');
});
server.on('error', function(err) {
  console.log('Error occurred:', err.message);
});
server.listen(port);
```

- Cliente para conectarse al servidor:

telnet localhost 4001 ó nc localhost 4001

SOCKET

- Es el primer argumento de la función callback que devuelve el evento “connection”
- El objeto **socket** es un stream duplex (lectura y escritura). Emite el evento “data” cuando devuelve un paquete de datos y el evento “end” cuando se cierra la conexión.
- También es un stream de escritura, por lo que puede escribir buffers o strings mediante el método `socket.write()`.
- Otros métodos accesibles, por ser stream son: `socket.pause()`, `socket.resume()`, o incluso se puede hacer pipe a cualquier stream de escritura.
- Para otras características mirar la documentación de `node.js`

Ejemplo de server tcp

```
var server =
require('net').createServer(function(socket
) {
  console.log('new connection');
  socket.setEncoding('utf8');
  socket.write("Hello! You can start typing.
  Type 'quit' to exit.\n");
  socket.on('data', function(data) {
    console.log('got:', data.toString())
    if (data.trim().toLowerCase() === 'quit') {
      socket.write('Bye bye!');
      return socket.end();
    }
    socket.write(data);
  });
  socket.on('end', function() {
    console.log('Client connection ended');
  });
}).listen(4001);
```

CHAT SERVER

- Instanciar el servidor, bindear eventos importantes (error, close) y unirlo al puerto 4001

```
var net = require('net');  
var server = net.createServer();  
server.on('error', function(err) {  
  console.log('Server error:', err.message);  
});  
server.on('close', function() {  
  console.log('Server closed');  
});  
server.listen(4001);
```

- Aceptar peticiones de clientes (evento connection):

```
server.on('connection', function(socket) {  
  console.log('got a new connection');  
});
```


CHAT SERVER

- Leer datos de la conexión

```
server.on('connection', function(socket) {  
  console.log('got a new connection');  
  socket.on('data', function(data) {  
    console.log('got data:', data);  
  });  
});
```

- Recopilar todos los clientes para transmitirles los datos de un cliente.
Para ello hay que almacenar todas las conexiones en un repositorio

```
var sockets = [];  
server.on('connection', function(socket) {  
  console.log('got a new connection');  
  sockets.push(socket);
```

CHAT SERVER

- Transmitir los datos a todos los clientes

```
socket.on('data', function(data) {  
  console.log('got data:', data);  
  sockets.forEach(function(otherSocket) {  
    if (otherSocket !== socket) {  
      otherSocket.write(data);  
    }  
  });  
});
```

Ejercicio 14

- Sobre el servidor de chat realizar las siguientes funciones:
 - Registrar la eliminación de una conexión cerrada
 - Pedir un nombre a cada cliente y registrarlo asociado a la conexión. Si el nombre ya existe pedir otro nuevo.
 - Cuando se conecte un cliente nuevo comunicarlo al resto de clientes, así como el número de clientes registrados.
 - Imprimir el prompt de otro cliente, delante del texto.

HTTP

- Hypertext Transfer Protocol, o HTTP, es un protocolo a nivel de aplicación que permite la transmisión de contenido. Es el fundamento de la World Wide Web.
- HTTP es un protocolo basado en texto que trabaja sobre la capa de TCP.
- La versión encriptada de HTTP se denomina HTTP Secure, o HTTPS
- HTTP es un protocolo de petición-respuesta desarrollado sobre las bases de la programación cliente-servidor. Normalmente, un navegador se utiliza como cliente en una transacción HTTP
- Cuando se introduce una URL en un navegador, se realiza una petición HTTP al servidor que hospeda la URL, suele hacerse a través del puerto 80 (o 443 si es HTTPS). El servidor procesa la petición y responde al cliente.

Servidor HTTP en Node.JS

```
var http = require('http');  
var server = http.createServer();  
server.on('request', function(req, res) {  
  res.writeHead(200, {'Content-Type':  
    'text/plain'});  
  res.write('Hello World!');  
  res.end();  
});  
server.listen(4000);
```

- Cuando un cliente hace una petición, el servidor HTTP emite un evento de petición pasando un objeto petición HTTP y uno respuesta HTTP. El objeto petición HTTP permite consultar propiedades de la petición, mientras que el objeto respuesta HTTP permite construir la respuesta que se enviará al cliente

HTTP: El objeto `HTTPServerRequest`

- Contiene las siguientes propiedades:
 - `req.url`: Contiene el string de la URL demandada desde el cliente
 - `req.method`: Contiene el método HTTP usado en la petición: GET, POST, DELETE o HEAD. Se puede analizar lo que.
 - `req.headers`: Contiene las propiedades de la cabecera de la petición
`res.end(util.inspect(req.headers));`
- Cuando se realiza una petición a un servidor el body de la petición no se recibe inmediatamente. Pero se puede "escuchar" un evento `data` que maneje la información cuando llegue. El objeto `request` es un **stream de lectura**

```
var writeStream = ...
require('http').createServer(function(req, res) {
  req.on('data', function(data) {
    writeStream.write(data);
  });
}).listen(4001);
```

HTTP: El objeto `HTTPServerResponse`

- En la respuesta se puede escribir: Head y Body.

- Para escribir la cabecera usar: `res.writeHead(status, headers)`

```
require('http').createServer(function(req, res)
{
  res.writeHead(200, {
    'Content-Type': 'text/plain',
    'Cache-Control': 'max-age=3600' });
  res.end('Hello World!');
}).listen(4000);
```

- Las cabeceras se pueden modificar mientras no se haya enviado el body de la respuesta (`res.write` o `res.end`):

`res.setHeader(status, headers)``res.removeHeader('Cache-Control');`

- El body se escribe directamente:

`res.write('Hello');`

o utilizando un buffer:

```
var buffer = new Buffer('Hello World');
res.write(buffer);
```

Respuesta troceada STREAMING HTTP

- HTTP *chunked encoding* permite al servidor mantener el envoi de datos al cliente sin tener que enviar el tamaño del cuerpo de la respuesta (body size). Si no se ha especificado la propiedad Content-Length del header, por defecto, El servidor Node HTTP envía el siguiente header al cliente:

Transfer-Encoding: chunked

- Esta cabecera permite al cliente recibir los datos de forma troceada, enviando un ultimo trozo de longitud 0 antes de que el cliente termine de procesar la respuesta. Esto tiene utilidad para el envoi de streaming de texto, audio, o vídeo
- Algunos ejemplos de streaming usando estas características son el piping de un fichero al stream.

Ejercicio 15

- Crear un stream sobre un vídeo '.mp4'
- Fijar la propiedad Content-Type de la cabecera de la respuesta
- Hacer un pipe a la respuesta HTTP
- Abrir el navegador y comprobar que el vídeo empieza a visualizarse sin haberse cargado completamente.

Ejercicio 16

- Crear un servidor HTTP que devuelva texto plano con 100 líneas separadas de timestamp cada segundo, durante 10 segundos.

Peticiones HTTP

- HTTP ocupa el centro de muchas infraestructuras no sólo como servidor de contenido estático sino también como servidor de información a través de llamadas a una API pública
- Una de las fortalezas de Node es el manejo de las E/S, esto lo convierten en una buena herramienta para proveer información y manejar los servicios HTTP.
- El protocolo HTTP presenta dos propiedades: la URL y el método.
- El método GET es el método principal más usado, otros métodos como POST (principal método para enviar formularios web), PUT, DELETE, and HEAD.

Métodos o Verbos en una petición

Método	Descripción
GET	Operación de lectura, que obtiene una representación de los recursos especificados.
HEAD	Equivalente a la petición GET, except que no se devuelve el cuerpo de la respuesta Útil para obtener respuestas rápidas sin la sobrecarga de transferir todo el cuerpo
POST	Crea un nuevo recurso en el servidor. Se usa normalmente para enviar formularios HTML y datos a la base de datos.
PUT	Similares a peticiones POST. Se utilizan para actualizar recursos existentes. Si el recurso no existe, puede crearlo.
DELETE	Utilizado para borrar un recurso del servidor.
TRACE	Realiza una prueba de bucle de retorno de mensaje a lo largo de la ruta al recurso de destino. Se utiliza para detectar cambios hechos por servidores intermediarios OPTIONS Returns a list of the verbs supported for the given URL.
OPTIONS	Devuelve una lista de los verbos soportados por una URL
CONNECT	Crea un túnel a través del servidor proxy. Permite tráfico encriptado HTTPS a través de un canal HTTP sin encriptación
PATCH	Similar a PUT. Realiza una actualización parcial de un recurso existente. PUT reenvía el recurso entero durante la actualización.

Códigos de respuesta (STATUS_CODE)

Status Code and Reason

Phrase

Description

200

OK La petición tuvo éxito

201

La petición se ha cumplido y se ha creado un nuevo recurso en el servidor

301

Moved Permanently, el recurso se ha movido a una nueva URL, El header de la respuesta debería tener información de la nueva URL

303

See Other. El recurso solicitado se puede encontrar via GET a la URL especificada en el header

304

Not Modified, indica que el recurso en caché no ha sido modificado. Este recurso no debería llevar body

400

Bad Request, la petición está mal construida y no se entiende. Ej. falta algún parámetro

401

Unauthorized. El recurso requiere autenticación y las credenciales proporcionadas se han rechazado.

404

Not Found, El servidor no puede localizar la URL solicitada

418

I'm a Teapot This status code was introduced as an April Fools' Day joke. Actual servers should not return this status code.

500

Internal Server Error. El servidor encontró un error mientras intentaba cumplir la petición.

Creando Peticiones HTTP

- Además de crear servidores, el modulo http permite crear peticiones mediante el método **request()**
- El método **request()** toma dos argumentos, options y callback.
 - options se usa para parametrizar la petición HTTP
 - callback función que es invocada cuando se recibe una respuesta a la petición
 - **IncomingMessage** es el único argumento que se le pasa al callback
- request() Tambien devuelve una instancia de http.ClientRequest, que es un stream de escritura.

Ejemplo de Petición GET HTTP (Cliente HTTP)

```
var http = require("http");
var request = http.request({
  hostname: "localhost",
  port: 8000,
  path: "/",
  method: "GET",
  headers: {"Host": "localhost:8000"}
}, function(response) {
  var statusCode = response.statusCode;
  var headers = response.headers;
  var statusLine = "HTTP/" + response.httpVersion + " " + statusCode + " " + http.STATUS_CODES[statusCode];
  console.log(statusLine);
  for (header in headers) {
    console.log(header + ": " + headers[header]);
  }
  console.log();
  response.setEncoding("utf8");
  response.on("data", function(data) {
    process.stdout.write(data);
  });
  response.on("end", function() {
    console.log();
  });
});
request.end();
```

Ejemplo de Petición HTTP

- Versión reducida, no se puede fijar la cabecera

```
var http = require("http");
var request = http.request("http://localhost:8000/", function(response) {
    response.setEncoding("utf8");
    response.on("data", function(data) {
        process.stdout.write(data);
    });
    response.on("end", function() {
        console.log();
    });
});
request.end();
```


Ejemplo de Petición POST HTTP (Cliente HTTP)

```
var http = require("http");
var qs = require("querystring");
var body = qs.stringify({
  foo: "bar",
  baz: [1, 2]
});
var request = http.request({
  hostname: "localhost",
  port: 8000,
  path: "/",
  method: "POST",
  headers: {
    "Host": "localhost:8000",
    "Content-Type": "application/x-www-form-urlencoded",
    "Content-Length": Buffer.byteLength(body)
  },
  function(response) {
    response.setEncoding("utf8");
    response.on("data", function(data) {
      process.stdout.write(data);
    });
    response.on("end", function() {
      console.log();
    });
  });
request.end(body);
```

Ejemplo de Servidor que procesa POST HTTP (Cliente HTTP)

```
var http = require("http");
var qs = require("querystring");
var server =
http.createServer(function(request, response) {
var bodyString = "";
request.setEncoding("utf8");
request.on("data", function(data) {
bodyString += data;
});
request.on("end", function() {
var body = qs.parse(bodyString);
for (var b in body) {
response.write(b + ' = ' + body[b] + "\n");
}
response.end();
});
});
server.listen(8000);
```

MIDDLEWARES

- Bloque de código que se ejecuta entre la petición que hace el usuario (request) hasta que la petición llega al servidor.
- Una parte del middleware recibe una petición entrante y la puede procesar completamente o pasarla a otra parte del middleware para un procesamiento adicional antes de enviar la respuesta al cliente.
- Las funciones de middleware, por tanto, tienen acceso al objeto de solicitud (req), al objeto de respuesta (res) y a la siguiente función de middleware en el ciclo de solicitud/respuestas de la aplicación. La siguiente función de middleware se denota normalmente con una variable denominada **next**.
- Ej:

```
function middleware(request, response, next) {  
  return next();  
}
```

MIDDLEWARES

- Las funciones de middleware pueden realizar las siguientes tareas:
 - Ejecutar cualquier código.
 - Realizar cambios en la solicitud y los objetos de respuesta.
 - Finalizar el ciclo de solicitud/respuestas.
 - Invocar el siguiente middleware en la pila.
- Si la función de middleware actual no finaliza el ciclo de solicitud/respuestas, debe invocar `next()` para pasar el control a la siguiente función de middleware. De lo contrario, la solicitud quedará colgada.

EJEMPLO SERVIDOR CON MIDDLEWARE

- Instalamos e importamos el módulo **connect**
- Función middleware que evalúe si estamos conectados
- Función middleware que genere la respuesta a la petición:

```
const http = require('http');
const connect = require('connect');
const app = connect();

let isLogin = false;
app.use((req,res,next) => {
  if(req.url.indexOf("favicon.ico")>0){ return; } next()})
app.use((req,res,next)=>{
  if(isLogin) return next();
  else{
    console.log('No estás logado');
    res.end('No estas logado')}
  })
app.use((request,response,next)=>{
  response.setHeader('Content-Type','text/html' );
  response.end('Hello <strong>HTTP</strong>!');
  })

app.listen(4000);

http.createServer(app);
```

Ejercicio 17

- Rehacer el servidor de la petición post con la capa de middleware:

- Utilizar el package

```
var bodyParser = require('body-parser');  
app.use(bodyParser.urlencoded({extended: false}));
```

CREANDO UN MIDDLEWARE CON EXPRESS

Express

Inicio

Cómo empezar

Guía

Referencia de API

Temas avanzados

Recursos

Express

Infraestructura web rápida, minimalista y flexible para **Node.js**

```
$ npm install express --save
```

Aplicaciones web

Express es una infraestructura de aplicaciones web Node.js mínima y flexible que proporciona un conjunto sólido de características para las aplicaciones web y móviles.

API


Con miles de métodos de programa de utilidad HTTP y middleware a su disposición, la creación de una API sólida es rápida y sencilla.

Rendimiento

Express proporciona una delgada capa de características de aplicación web básicas, que no ocultan las características de Node.js que tanto ama y conoce.

LoopBack

[Desarrolle aplicaciones basadas en modelos con una infraestructura basada en Express.](#)
[Encontrará más información en \[loopback.io\]\(http://loopback.io\).](#)



MIDDLEWARE CON EXPRESS

- Creamos un proyecto con archivo app.js y generamos package.json
- Instalamos express: `npm install express --save`
- De la página oficial bajamos el ejemplo "hola mundo":
 - <https://expressjs.com/es/starter/hello-world.html>

MIDDLEWARE CON EXPRESS

- Insertamos el resto de operaciones http: post, put, delete
- Probamos con Postman
- Incorporamos dos métodos de middleware que evalúen:
 - Si estamos logados
 - Se guarde una traza de la url desde donde se conecta el cliente y el método http

Ejercicio 18

- Incorporar la fecha actual a la respuesta a la petición mediante un middleware.

ESTRUCTURA DE UN PROYECTO EN NODE.JS

- **/models** contiene todos los modelos del ORM (*Schemas* en mongoose)
- **/views** contiene las vistas-templates (usando cualquier motor de plantillas de express)
- **/public** contiene todo el contenido estático (imágenes, hojas de estilo, JavaScript del lado del cliente)
 - **/assets/images** archivos de imágenes
 - **/assets/pdf** archivos pdfs estáticos
 - **/css** hojas de estilos (o compilados por un motor de css)
 - **/js** JavaScript del lado cliente
- **/controllers** contiene las rutas de express, separadas por módulo/área de la aplicación (si se utiliza la funcionalidad de generación de estructuras de express, esta carpeta se llama **/routes**)

DIRECCIONAMIENTO (ROUTING)

- Forma de responder una aplicación a una solicitud de un cliente en un endpoint determinado, formado por un URI (o *path*) y un método de solicitud HTTP específico (GET, POST, etc.).
- Cada ruta puede tener una o varias funciones que gestionen la petición, siendo excluyentes entre sí.

```
app.METHOD(PATH, HANDLER)
```

Donde:

- app es una instancia de express.
- METHOD es un método de solicitud HTTP.
- PATH es una vía de acceso en el servidor.
- HANDLER es la función que se ejecuta cuando se correlaciona la ruta.

DIRECCIONAMIENTO (ROUTING)

- Ej.:

```
// GET method route
app.get('/', function (req, res) {
  res.send('GET request to the homepage');
});
```

```
// POST method route
app.post('/', function (req, res) {
  res.send('POST request to the homepage');
});
```

```
//POST method route
app.post('/user', function (req, res) {
  res.send('POST request to the homepage');
});
```

MÉTODOS (ROUTING)

- En Express se da soporte al direccionamiento de los siguientes métodos:
 - get, post, put, head, delete, options, trace, copy, lock, mkcol, move, purge, propfind, proppatch, unlock, report, mkactivity, checkout, merge, m-search, notify, subscribe, unsubscribe, patch, search y connect.
- Existe un direccionamiento especial: **app.all** para cargar funciones de middleware accesibles a todos los métodos de solicitud. Ej:

```
app.all('/secret', function (req, res, next) {  
  console.log('Accessing the secret section ...');  
  next(); // pass control to the next handler  
});
```

PATHS (ROUTING)

- Los paths pueden ser strings, patrones de caracteres o expresiones regulares.
- Los caracteres:?, +, *, y () son parte de expresiones regulares. El guión o (-) y el punto (.) se interpretan literalmente.
- Ej.:

```
app.get('/', function (req, res) { res.send('root') })
This route path will match requests to /about.
app.get('/about', function (req, res) { res.send('about') })
This route path will match requests to /random.text.
app.get('/random.text', function (req, res) { res.send('random.text') })
Here are some examples of route paths based on string patterns.
This route path will match acd and abcd.
app.get('/ab?cd', function (req, res) { res.send('ab?cd') })
This route path will match abcd, abbcd, abbbcd, and so on.
app.get('/ab+cd', function (req, res) { res.send('ab+cd') })
This route path will match abcd, abxcd, abRANDOMcd, ab123cd, and so on.
app.get('/ab*cd', function (req, res) { res.send('ab*cd') })
This route path will match /abe and /abcde.
app.get('/ab(cd)?e', function (req, res) { res.send('ab(cd)?e') })
Examples of route paths based on regular expressions:
This route path will match anything with an "a" in it.
app.get(/a/, function (req, res) { res.send('/a/') })
This route path will match butterfly and dragonfly, but not butterflyman, dragonflyman, and so on.
app.get(/.*fly$/, function (req, res) { res.send('/.*fly$/') })
```

PARÁMETROS (ROUTING)

- Los parámetros de la dirección son segmentos de una URL utilizados para capturar valores en una posición de la URL.
 - A los valores capturados se accede mediante **req.params object**, con el nombre del parámetro especificado en la dirección
 - Ej.:

```
Route path: /users/:userId/books/:bookId  
Request URL: http://localhost:3000/users/34/books/8989  
req.params: { "userId": "34", "bookId": "8989" }
```

```
app.get('/users/:userId/books/:bookId', function (req, res) {  
  res.send(req.params)  
})
```


Ejercicio 19

- Crear un middleware con Express que asocie un método auth a una URI /admin y a las subrutas. De manera que si no se obtiene un usuario y una password determinados en el request de la petición devolver status_code = Not authorized.
- Si se autoriza devolver en /admin mensaje "Estás validado como admin"
- Si entra a /admin/user devolver los datos de usuario y password

TIP: El usr y pssw enviarlos como key/value desde POSTMAN.

HANDLERS (ROUTING)

- Conjunto de callbacks que se comportan como un único middleware para manejar la respuesta a una petición. Requieren que cada una de las funciones callbacks implemente el método next().
- Ej.

```
app.get('/example/b', function (req, res, next) {  
  console.log('the response will be sent by the next function ...')  
  next()  
}, function (req, res) {  
  res.send('Hello from B!')  
})
```

app.route() (ROUTING)

- Permite encadenar handlers de ruta. Facilita la lectura del código, reduce la redundancia y errores
- Ej.

```
app.route('/book')
  .get(function(req, res) {
    res.send('Get a random book');
  })
  .post(function(req, res) {
    res.send('Add a book');
  })
  .put(function(req, res) {
    res.send('Update the book');
  });
```

express.Router (ROUTING)

- Permite crear handlers modulares. Una instancia de Router es un sistema middleware y direccionamiento completo. Funciona como una miniaplicación.
- Ej. para manejar la ruta /birds y las subrutas se crea un módulo bird.js. Después se carga en la aplicación:

```
//módulo bird.js
var express = require('express');
var router = express.Router();
// middleware that is specific to this router
router.use(function timeLog(req, res, next) {
  console.log('Time: ', Date.now()); next(); });
// define the home page route
router.get('/', function(req, res) {
  res.send('Birds home page'); });
// define the about route router.get('/about',
function(req, res) { res.send('About birds');
});

module.exports = router;
```

```
//app.js

var birds = require('./birds');
...
app.use('/birds', birds);
```

Métodos de respuesta (ROUTING)

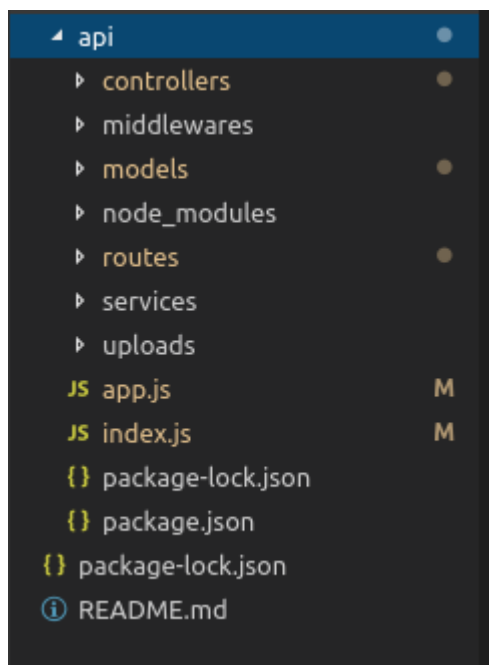
Método	Descripción
<u>res.download()</u>	Solicita un archivo para descargarlo.
<u>res.end()</u>	Finaliza el proceso de respuesta.
<u>res.json()</u>	Envía una respuesta JSON.
<u>res.jsonp()</u>	Envía una respuesta JSON con soporte JSONP.
<u>res.redirect()</u>	Redirecciona una solicitud.
<u>res.render()</u>	Representa una plantilla de vista.
<u>res.send()</u>	Envía una respuesta de varios tipos.
<u>res.sendFile</u>	Envía un archivo como una secuencia de octetos.
<u>res.sendStatus()</u>	Establece el código de estado de la respuesta y envía su representación de serie como el cuerpo de respuesta

Ejercicio 20

- Crear un controlador `admin.js` para el ejercicio anterior, añadiéndole un middleware que imprima un timestamp en la respuesta.

Proyecto

- Crea las rutas, middlewares y controllers para las aplicaciones: Red Social y e-commerce.
 - Tienes que analizar que peticiones se harán desde el frontend a las APIs teniendo en cuenta las funcionalidades discutidas en la estructuración de las aplicaciones.
 - Comenzar a estructurar los proyectos:



index.js: conexión a la base de datos y configuración general de mongoose,

app.js servidor web con NodeJS y configuración de express,
middlewares (middleware de autenticación,...)

models: crear los modelos y esquemas,

controllers: acciones y operaciones sobre la bd

routes: directorio de rutas, se definen las rutas a las que responderá la aplicación.