

10

Node S1 | Express.js

Después de esta lección podrás:

1. Conocer el funcionamiento básico de Express.
2. Usar rutas para adecuar las respuesta de tu servidor a cada caso.

¿Qué es Express.js?

Express es un framework minimalista que nos proporciona las utilidades los mínimas para construir un back-end. Es apenas un esqueleto, y tanto es así que otros muchos frameworks lo usan como cimientos.

- **Link a la web oficial del framework: <https://expressjs.com>**

¿Cómo comenzamos un proyecto con Express.js?

Vamos a comenzar un nuevo proyecto de cero.

Puesto que vamos a incluir módulos de terceros, tenemos primero que inicializar el proyecto de la siguiente forma:

```
$ npm init -y
```

Como no queremos tener que estar rearrancando el servidor cada vez que cambiemos nuestro código, vamos a instalar Nodemon como dependencia de desarrollo:

```
$ npm install -D nodemon
```

Para poder ejecutar nuestra aplicación con Nodemon en vez de como lo hacemos normalmente, tenemos que añadir un nuevo script:

```
"scripts": {  
  "start": "node index.js",  
  "dev": "nodemon index.js",  
  "test": "echo \"Error: no test specified\" && exit 1"  
},
```

Para trabajar con Express.js, lógicamente, tendremos también que instalarlo (en este caso se trata de una dependencia de producción, o sea, es una pieza sin la que nuestra aplicación no va a funcionar):

```
$ npm install express
```

Finalmente, crearemos el fichero que va a ser el punto de entrada de nuestra aplicación (en este ejemplo lo vamos a llamar `index.js` pero se puede llamar como más intuitivo te parezca: `main.js`, `app.js`, `server.js`, etc.). En este fichero escribimos el siguiente código:

```
const express = require('express');

const PORT = 3000;
const server = express();

server.listen(PORT, function() {
  console.log(`Server running in http://localhost:${PORT}`);
});
```

Solo queda poner en marcha nuestra aplicación:

```
$ npm run dev
```

A partir de aquí, como estamos usando Nodemon, podremos escribir código nuevo y el servidor se reiniciará solo.

A screenshot of the Visual Studio Code interface. The left side shows a code editor with a file named 'index.js'. The code defines a simple Express server that responds to the root URL ('/') with the message 'Hello Upgrade Hub!'. Below the code editor is a navigation bar with tabs: PROBLEMS, OUTPUT, DEBUG CONSOLE, and TERMINAL. The TERMINAL tab is active, showing a terminal window titled '1: zsh' with the command line prompt. The right side of the interface includes a sidebar with various icons and a status bar at the bottom.

```
JS index.js  X
JS index.js > ⚡ router.get('/movies') callback
1 const express = require('express');
2
3 const PORT = 3000;
4 const server = express();
5
6 const router = express.Router();
7
8 router.get('/', (req, res) => {
9   res.send('Hello Upgrade Hub!');
10 });
11
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 1: zsh + ⌂ ⌁ ⌂ ⌁ ⌁ ⌁

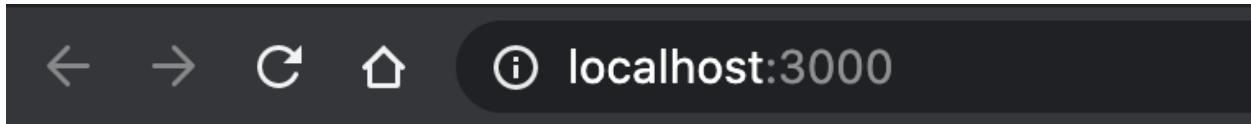
~ /Documents/upgrade/node/day_2/node_express 19:48:27

Creando rutas en nuestro servidor

Vamos a añadir un manejador para las peticiones que se hagan a la ruta `/`. Se codifica de esta forma:

```
server.use('/', function(req, res) {
  res.send('Hello Upgrade Hub!');
});
```

Si lanzamos una petición desde el navegador, veremos el siguiente resultado:

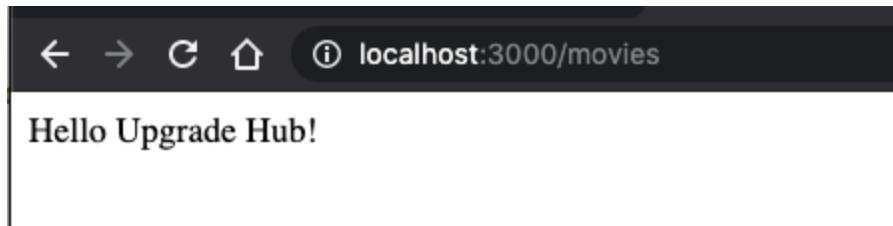


Hello Upgrade Hub!

Vamos a crear una nueva ruta:

```
server.use('/movies', function(req, res) {  
  const movies = ['Harry Potter', 'Titanic', 'Back to the Future'];  
  res.send(movies);  
});
```

Si accedemos a <http://localhost:3000/movies>, veremos lo siguiente:



Pero, ¿y las películas? 😞

Cuando hemos ejecutado nuestro fichero principal, hemos registrado una serie de manejadores mediante el método `use` para el caso de que se produzcan peticiones HTTP desde determinadas rutas. Y, para empezar, el orden en que hemos registrado los manejadores, será el orden en que Express va a comprobar si tiene que ejecutar alguno de estos cuando reciba una petición HTTP.

Pero, si la petición que hemos hecho ha sido con la ruta `/movies`, ¿por qué ha ejecutado el manejador que habíamos registrado para `/`? El manejador que ha ejecutado sí que estaba primero en la lista de manejadores registrados, pero la ruta es diferente ¿no?

En el caso del método `use`, lo que se valora a la hora de ejecutar el manejador o no, es si la ruta que hemos puesto al registrar el manejador coincide con el inicio o con la ruta completa desde la que está llegando la petición: en este caso `/movies` comienza por `/`, así que la condición se cumple.



Cuando decimos que basta con que coincida el inicio, es en realidad un poco más complicado. Por ejemplo, si registro con `use` un manejador para la ruta `/uno` y recibo una petición con la ruta `/uno/dos/`, esta cumpliría la condición del manejador. Pero si la ruta que yo he registrado es `/un`, la condición no se cumpliría para una petición desde `/uno/dos/`: uno basta con que los caracteres iniciales coincidan, tiene que coincidir porciones completas de la ruta. Haz la prueba 😊

¿Y por qué no sigue mirando la lista de manejadores registrados después de ejecutar el manejador `/`? El manejador registrado para la ruta `/movies` también cumple la condición ¿no? Efectivamente, cumple la condición, pero una vez que hemos ejecutado `res.send('Hello Upgrade Hub!');`, hemos respondido al cliente que nos había hecho la petición, y ese es el final del camino.

Es importante conocer cómo trata las rutas el método `use` puesto que habrá casos en lo que necesitemos usarlo. Pero si lo que queremos construir es una API, va a haber una forma específicamente pensada para trabajar no solo con rutas si no también con los métodos HTTP: utilizar el módulo Router de Express.js.

Los métodos HTTP, proporcionan semántica a la hora de utilizar servicios web. Los métodos más comunes son los que coinciden con las operaciones clásicas de manejo de datos (**CRUD, iniciales de Create, Read, Update y Delete**):

- GET → Recopila datos del servidor y los devuelve a los usuarios.
- POST → Envía información al servidor y crea nuevos elementos en la base de datos.
- PUT → Envía información al servidor y sobreescribe un elemento de la base de datos.
- PATCH → Envía información al servidor y modifica un elemento de la base de datos.
- DELETE → Elimina un elemento de la base de datos.

Vamos a comenzar con manejadores que responden al método GET, para recoger información de nuestro servidor.

Para utilizar el módulo gestor de rutas, Router, tenemos que cambiar un poco nuestro código:

```
const express = require('express');

const PORT = 3000;
const server = express();
const router = express.Router();

server.listen(PORT, function() {
  console.log(`Server running in http://localhost:${PORT}`);
});

server.use('/', router);

// A partir de aquí voy a registrar los manejadores de ruta

router.get('/', function(req, res) {
  res.send('Hello Upgrade Hub!');
});

router.get('/movies', function(req, res) {
  const movies = ['Harry Potter', 'Titanic', 'Back to the Future'];
});
```

```
    res.send(movies);
  });
}
```

Si ahora volvemos a hacer la petición de las películas, va a utilizar la ruta exacta para seleccionar el manejador a ejecutar:



En Internet veremos a menudo la palabra **endpoint** para referirse a estas rutas que hemos definido. Un endpoint puede ser muchas cosas en realidad, pero en el contexto de una API, es una ruta que se corresponde con una determinada funcionalidad de la API: por ejemplo, una ruta para obtener todas las películas u otra para añadir una película nueva.

Parámetros de ruta y de búsqueda

Nuestras rutas han sido sencillas hasta el momento, pero es hora de añadir una mejora al sistema de comunicación que hemos creado con el servidor.

Vamos a realizar una petición a <http://localhost:3000/movies/titanic> en nuestro navegador.

```
Cannot GET /movies/titanic
```

Esta ruta no funcionará porque no la hemos creado previamente, pero ¿y si queremos una ruta que nos responda si una película existe o no dado su nombre? Tendríamos que crear un manejador de ruta para cada película que tenemos en el servidor y sería algo infinito.

Para solucionar esto, tenemos los parámetros de ruta, o **route params**, que obtendremos de la **Request (req)** que llega a nuestros manejadores de ruta en el objeto request (concretamente en `req.params`).

Para crear el parámetro de ruta que buscamos, hay que añadir `/:nombre_del_param` en la ruta de la url, como hacemos a continuación:

```
router.get('/movies/:name', function(req, res) {});
```

Con esta ruta, hemos creado un parámetro que esperamos en la ruta llamado **name** que obtendremos de `req.params.name`. Y será la cadena de texto que ocupe ese lugar en la url, por lo tanto:

```
// http://localhost:3000/movies/titanic

router.get('/movies/:name', function(req, res) {
  console.log(req.params.name);
});
```

El nombre que aparecerá en el log en la terminal, será '**titanic**'.

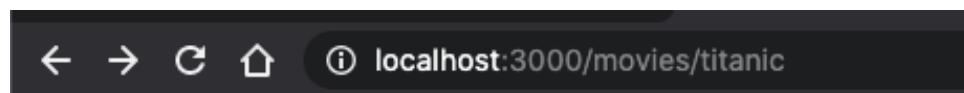
Sigamos:

```
router.get('/movies/:name', function (req, res) {
  const name = req.params.name;

  const movies = ['Harry Potter', 'Titanic', 'Back to the Future'];
```

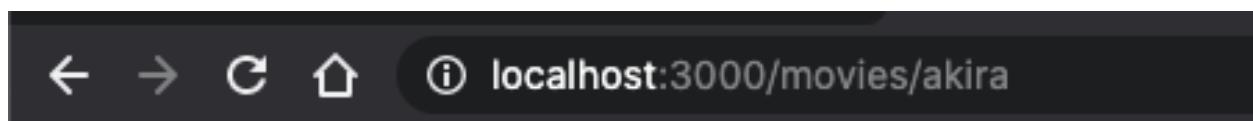
```
// Si existe una película con el nombre que recibimos,  
// hasMovie será true  
const hasMovie = movies.some((movie) => movie.toLowerCase() === name);  
  
if (hasMovie) {  
  res.send(`We have a movie with the name ${name}`);  
} else {  
  res.send('We could not find the movie you are looking for!');  
}  
});
```

Para una película que existe en el array que tenemos en el servidor:



We have a movie with the name titanic

Para una película que no tenemos en el servidor:



We could not find the movie you are looking for!

Ya conocemos el uso de los parámetros de ruta, pero hay otra forma de pasar datos en la petición, utilizando parámetros de búsqueda (**query params**).

Serán creados utilizando una interrogación **?** para comenzar a describir que vamos a utilizar queries, y separando cada uno de estos con una **&**.

```
http://localhost:3000/movies?name=titanic
```

Y ahora, podremos obtener el valor de name tal y como hicimos antes con los parámetros, pero en `req.query`.

El código quedaría finalmente así:

```
router.get('/movies', function(req, res) {
  const movies = ['Harry Potter', 'Titanic', 'Back to the Future'];

  // Ahora utilizamos req.query
  const name = req.query.name;

  // Si no tenemos req.query.name enviaremos todas las películas
  // ya que no habrá búsqueda por parte del usuario
  if (!name) {
    res.send(movies);
  } else {
    const hasMovie = movies.some((movie) => movie.toLowerCase() === name);

    if (hasMovie) {
      res.send(`We have a movie with the name ${name}`);
    } else {
      res.send('We could not find the movie you are looking for!');
    }
  }
});
```

EJERCICIO: API Express

Hagamos una API muy sencilla, emulando las APIs que usábamos en otros módulos como JavaScript o React, en la que los elementos van a estar en un fichero JSON.