



Curso de Back-End con Node.js (Inicial)

Clase 03



Temario



Temario

- npm.
- ES+ y JavaScript exclusivo de Node.
- Uso del `require` (y diferencias con el `import`).
- Express.
- Rutas.
- Ejercicios.



npm

npm (1/5)

Es equivalente a **Composer** en PHP o **NuGet** en .NET.



- npm es el **gestor (o manejador) de paquetes** que viene con Node.
- Tiene un **registro público** donde desarrolladores pueden publicar paquetes, para que el resto de la comunidad pueda usarlos.
- Además, incluye una **cli** (command-line interface) que permite usar npm desde la consola para instalar y publicar dichos paquetes.
- npm Inc. es una empresa privada.
- La palabra npm de por sí, no es una sigla y no obedece a nada. Pero se le suele decir *node package manager*.
- Link: <https://www.npmjs.com>.





npm (2/5)

- npm también sirve para **dar comienzo a un proyecto** de Node.
- Esto se hace mediante el comando: `npm init`.
- Al hacerlo, el cli nos hará algunas preguntas para facilitar la creación del proyecto, aquellas que no queremos responder o que no sabemos cómo, simplemente las ignoramos.
- Al terminar, tendremos un nuevo archivo, llamado `package.json`.



npm (3/5) – package.json

El archivo `package.json` contendrá toda la **información necesaria sobre nuestro proyecto** como ser nombre, versión, descripción, autores, contribuidores, licencia, *punto de entrada*, y principalmente: **dependencias** y **scripts**.

Aquí un ejemplo de un archivo `package.json` creado con el comando `npm init`:

```
{
  "name": "test",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```



npm (4/5)

- Se utilizará npm para instalar **dependencias**.
- Una dependencia es un **módulo externo** a nuestra aplicación la cual sólo funcionará en tanto dicho módulo este instalado. En nuestro caso, éstos módulos externos provienen del registro público de npm.
- Las dependencias en los proyectos Node se instalan en un directorio, ubicado en la raíz del proyecto, llamado **node_modules**.
- El directorio **node_modules** no debe ser editado jamás por nosotros. La estructura de carpetas y los archivos que hay allí dentro sólo deben ser manejados por npm.



npm (5/5)

Para **instalar dependencias**, tenemos que estar ubicados en un proyecto de Node ya iniciado (es decir, que fue creado previamente usando el comando `npm init`) y luego ejecutar el comando:

```
npm install <nombre_del_modulo_externo>
```

Este comando consiste de:

- Invocar `npm`.
- Comando `install` (o simplemente `i`).
- Nombre del módulo externo (dependencia) que se desea instalar, tal como aparece en npmjs.com.

👉 Este comando instalará la dependencia en el directorio `node_modules` y agregará una entrada en el archivo `package.json`.



ES+ y JavaScript exclusivo de Node



ES+ y JavaScript exclusivo de Node (1/2)

Anteriormente se usaron los *módulos* `http` y `fs`. Se accedió a los mismos usando la función `require`, que es exclusivo de Node.

En un entorno Node, la función `require` siempre está disponible globalmente. Sirve para **importar** objetos, funciones, variables y otros elementos desde:

- Módulos de Node (cómo `http`, `fs` y otros que iremos viendo a lo largo del curso).
- Otros archivos (nuestros).
- Dependencias externas, que gestionaremos con `npm`.

Nota: Técnicamente hablando, la función `require` es local al módulo que está haciendo la importación, aunque la experiencia para el desarrollador es como si se tratase de una función global. Ver [más información aquí](#).



ES+ y JavaScript exclusivo de Node (2/2)

Vale la pena recordar que desde la versión ES2015 (ES6) de JavaScript, fue incorporada la palabra clave `import`, cuya funcionalidad es muy similar a la del `require` (que es específica de Node).

Sin embargo, en general, la sintaxis `import` de ES6 no está habilitada en un entorno Node y por lo tanto se debe usar `require`.

En todos los casos la extensión de los archivos se da por entendida (`.js`) y se omite.

```
const modulo = require("modulo");
```

En el curso usaremos esta opción.

0

```
import modulo from "modulo";
```

⚠ La sintaxis `import` de ES6 puede no funcionar en Node.js por defecto: <https://nodejs.org/api/esm.html>



`require / import`



require / import (1/6)

Para **importar módulos (integrados) de Node**, simplemente se elige el módulo que se quiere utilizar y se lo *requiere* o *importa*.

Ejemplo con el módulo `http`:

```
const http = require("http");
```

En el curso usaremos esta opción.

o

```
import http from "http";
```

⚠ La sintaxis `import` de ES6 puede no funcionar en Node.js por defecto: <https://nodejs.org/api/esm.html>



require / import (2/6)

Para **importar módulos que están disponibles en otros archivos** (nuestros), debemos indicar con `require` ó `import` la ruta (*path* en inglés) en donde se encuentra el mismo, relativa a la posición del archivo actual.

Para indicar que el archivo está en el mismo directorio, se usa `./`.

```
const nuestroModulo = require("./nuestro-modulo");
```

En el curso usaremos esta opción.

```
import nuestroModulo from "./nuestro-modulo";
```

⚠ La sintaxis `import` de ES6 puede no funcionar en Node.js por defecto: <https://nodejs.org/api/esm.html>



require / import (3/6)

Para indicar que el archivo está un directorio “antes” (o “más afuera”) que éste archivo, se usa “../”.

```
const nuestroModulo = require("../nuestro-modulo");
```

En el curso usaremos esta opción.

```
import nuestroModulo from "../nuestro-modulo";
```

⚠ La sintaxis `import` de ES6 puede no funcionar en Node.js por defecto: <https://nodejs.org/api/esm.html>



require / import (4/6)

Para indicar que el archivo está un directorio “después” (o “más adentro”) que éste archivo, se usa `../directorio/`.

```
const nuestroModulo = require("../directorio/nuestro-modulo");
```

En el curso usaremos esta opción.

```
import nuestroModulo from "../directorio/nuestro-modulo";
```

⚠ La sintaxis `import` de ES6 puede no funcionar en Node.js por defecto: <https://nodejs.org/api/esm.html>



require / import (5/6)

Siempre que se importen/requieran módulos (*nuestros*), es necesario que dichos módulos hayan sido **exportados** previamente. Esto se hace así:

Dado un módulo (que bien puede ser una función, un objeto, una variable, etc)

```
const nuestroModulo = function() {};
```

Usando la sintaxis exclusiva de Node, se exporta así:

```
module.exports = nuestroModulo;
```

En el curso usaremos esta opción.

O, usando la sintaxis standard de JavaScript, se exportará así..

```
export default nuestroModulo;
```

⚠ La sintaxis `export` de ES6 puede no funcionar en Node.js por defecto: <https://nodejs.org/api/esm.html>



require / import (6/6)

Para **importar módulos instalados vía npm**, al igual que con los módulos integrados en Node (como `http` y `fs`), no es necesario utilizar un prefijo de ruta. Simplemente se escribe el nombre del paquete entre comillas.

Ej: el framework *Express* no viene instalado por defecto con Node. Se debe instalar vía npm. Luego se importa en nuestro proyecto de la siguiente manera:

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

En el curso usaremos esta opción.

o

```
import express from "express";
```

⚠ La sintaxis `import` de ES6 puede no funcionar en Node.js por defecto: <https://nodejs.org/api/esm.html>



En resumen...



En resumen...

“`npm` permite instalar dependencias, las cuales se persisten como tales en `package.json` y en `node_modules`. Luego se utilizan en nuestro código importándolas con la función `require`”



Express



Express

- Es un **framework** para Node.js.
- Diseñado para construir aplicaciones web y APIs.
- Es muy popular. Y es open-source.
- Es **minimalista** → es **rápido**.
- Sirve como base para otros frameworks más “grandes” como [Sails.js](#) o [Adonis.js](#) (este último muy similar a Laravel para PHP).
- Documentación: <https://expressjs.com>.

Express

👉 Express se instalará como una **dependencia** en nuestro proyecto, usando el comando `npm install express`.



Ejemplo de uso de Express



Ejemplo de uso de Express (1/3)

1. Crear un proyecto de Node (usando `npm init`).
2. **Instalar Express** como dependencia del proyecto: `npm install express`
3. Al terminar, el archivo `package.json` debería contener una nueva entrada `dependencies` con `express` dentro.

Algo así:

```
"dependencies": {  
  "express": "^4.17.1"  
}
```

** La versión puede ser distinta a la que aparece en esta diapositiva, Express está en constante actualización*

Ejemplo de uso de Express (2/3)



Podría tener otro nombre, por ejemplo: `server.js`.

Luego, crear un archivo `index.js` con el siguiente código:

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

const app = express(); // Crea una instancia de express.

app.get("/", (req, res) => res.send("!Hola Hack Academy!"));

app.get("/productos", (req, res) => res.send("Página de productos"));

app.listen(3000, () => console.log("!Servidor corriendo en el puerto 3000!"));
```

Luego se deberá correr el comando `node index.js` para que el servidor quede “prendido” (ejecutándose).

Notar que si realiza un cambio en el archivo `index.js` se deberá “cortar” la ejecución anterior cerrando la terminal o con CTRL+C, y luego se deberá correr nuevamente el comando `node index.js`.



Ejemplo de uso de Express (3/3)

Es común crear, al lado de `index.js`, un archivo **routes.js** que contendrá lo que refiere a manejo de rutas (que previamente estaba en `index.js`).

`routes.js` deberá tener un *export*, el cual será una función que tan solo recibirá la instancia de `express` para poder crear los *handlers (callbacks)*.

Luego, en `index.js`, se deberá importar esa función y ejecutarla.

El código debería funcionar igual que antes, pero ahora las **responsabilidades** están mejor repartidas gracias a la modularización.



Rutas en Express



Rutas en Express (1/5)

En el siguiente código se está definiendo una **ruta**, la cual es una “especie” de URL de la aplicación web. Cuando la aplicación recibe un *request* para cierta ruta, se ejecuta determinado bloque de código.

En archivo `index.js` o `server.js`

```
app.get("/contacto", (req, res) => res.send("¡Hola Hack Academy!"));
```

- `app` es una instancia de Express.
- `get` es un método de Express, cuyo nombre coincide con el **método HTTP** GET. Análogamente, se pueden usar los métodos `.post`, `.put`, `.patch`, `.delete`, para aceptar distintos tipos de *requests* en la aplicación. 🖱 En breve detallaremos sobre esto.
- `"/contacto"` (primer parámetro de la función `get`) es una **ruta** (*route*). También se le puede decir *path*.
- El segundo parámetro de la función `get` es una función (llamada **handler** o **callback**) que se ejecuta cada vez que se reciba un *request* de tipo GET en la ruta especificada.



Rutas en Express (2/5)

La sintaxis genérica de una **ruta** es:

En archivo `index.js` o `server.js`

```
APP.METHOD(PATH, HANDLER);
```

- **APP** es una instancia de Express.
- **METHOD** es un método HTTP (en minúscula). Ej: `get` o `post`.
- **PATH** es una ruta ≈ la porción de la URL seguida del dominio. Ej: `“/productos”`.
- **HANDLER** es la función que se ejecuta cada vez que se reciba un *request* en la ruta especificada. También se le dice **callback**.

👉 Por detalles, ver la documentación oficial sobre [Routing](#).



Rutas en Express (3/5)

Al lado de `index.js`, es común crear un archivo `routes.js` que contendrá lo que refiere a manejo de rutas (que previamente lo habíamos colocando en `index.js` o `server.js`). Se le llama “archivo de rutas”.

`routes.js` deberá tener un *export*, el cual será una función que tan solo recibirá la instancia de `express` para poder crear los *handlers* (*callbacks*). También es posible crear un `express.Router` (como se muestra [al final de esta página](#)).

Luego, en `index.js`, se deberá importar esa función y ejecutarla.

El código debería funcionar igual que antes, pero ahora las **responsabilidades** están mejor repartidas gracias a la modularización.



Rutas en Express (4/5)

La función *handler* (o *callback*) puede recibir como parámetros dos objetos a los cuales se les llama `request` y `response`, o simplemente `req` y `res`.

```
app.get("/saludo", function(req, res) {  
    res.send("¡Hola Mundo!");  
});
```

- `req` es un objeto que contiene información sobre el *request* (petición).
- `res` es un objeto que permite retornar una respuesta, usando el método `send`.



Rutas en Express (5/5)

En caso de preferirse, la función *handler* (o *callback*), se puede definir por fuera de la configuración de la ruta. Incluso la función se podría mover a otro archivo.

```
function helloPage(req, res) {  
  res.send(";Hola Mundo!");  
}
```

```
app.get("/saludo", helloPage);
```



Express – Responder un archivo HTML



Responder un archivo HTML

Express permite responder un *request* con un archivo HTML:

```
app.get("/productos", (req, res) => {  
    res.sendFile(__dirname + "/productos.html");  
});
```

`__dirname` es una variable que siempre está presente en Node y lo que contiene es el *path* absoluto del directorio donde está ubicado el archivo que está llamado a dicha variable.

⚠ Notar que no es posible escribir `res.sendFile("./productos.html")` ya que daría error.
[Más info aquí.](#)



Ejercicio 1

Ejercicio 1

1. Crear un sitio web con **Express** que contenga las siguientes rutas:
 - a. [GET] <http://localhost:3000/>.
 - b. [GET] <http://localhost:3000/productos>.
 - c. [GET] <http://localhost:3000/sobre-nosotros>.
 - d. [GET] <http://localhost:3000/contacto>.
 2. La idea es que cada una de esas rutas muestre un HTML básico, diferente para cada URL. Colocar links para poder “navegar” entre las 4 páginas. 🖱️ Para enviar un archivo como parte de una respuesta HTTP pueden usar el método [sendFile](#).
 3. Levantar el servidor en el puerto 3000 y verificar que el sitio sea correctamente navegable.
- ⚠️ En lugar de definir los *handlers* de las rutas como funciones anónimas, mover dichas funciones a un archivo aparte llamado `pagesController.js`.



Ejercicio 2



Ejercicio 2

1. Crear una carpeta llamada `ejercicio_slugify`.
2. Inicializar un proyecto con el comando `npm init`.
3. Pueden ignorar todas las preguntas haciendo “Enter” en cada una.
4. Verificar que se haya creado el archivo `package.json`.
5. Instalar un paquete (dependencia) llamado **Slugify**.
6. Verificar que se haya creado la carpeta `node_modules` y que se haya actualizado el archivo `package.json`.
7. Crear un archivo `index.js` dentro de la carpeta `ejercicio_slugify`.



Ejercicio 2 (cont)

8. Importar (requerir) en dicho archivo la dependencia:

```
const slugify = require("slugify");
```

9. Usar Slugify para convertir el texto:

"¡Quiero viajar a Bélgica & España!  "

a:

"quiero-viajar-a-belgica-y-espana".

10. Al llamar al archivo `index.js` (con el comando `node index.js`) deberá aparecer el nuevo texto impreso en la consola.



Ejercicio 3



Ejercicio 3

Como ya hemos visto, con Node.js es posible acceder al disco duro de la máquina donde está corriendo, es decir, al sistema de archivos y carpetas, que es lo que se conoce como **File System**. Eso es lo que haremos en este ejercicio, utilizando un módulo llamado `fs`, que contiene funcionalidades como crear, leer y editar archivos.

1. Crear una carpeta llamada `ejercicio_filesystem`.
2. Inicializar un proyecto con el comando `npm init`.
3. Pueden ignorar todas las preguntas haciendo “Enter” en cada una.
4. Verificar que se haya creado el archivo `package.json`.



Ejercicio 3 (cont)

5. Importar (requerir) el módulo `fs`.
6. Crear un servidor escuchando en el puerto 8000.
7. Utilizar el método `appendFile` del módulo `fs` para crear un archivo llamado `access_log.txt`. El archivo se debe crear en el primer llamado al servidor y para cada nuevo llamado se debe crear un línea de texto que contenga la fecha actual, con este formato:

`Se llamó al servidor el 17 de marzo de 2021 a las 21:56:57 (miércoles).`

`Se llamó al servidor el 17 de marzo de 2021 a las 22:29:58 (miércoles).`

Instalar la librería [date-fns](#) ~~Moment~~ para manipular las fechas con mayor facilidad.