

Desarrollo Backend

Clase 5: NPM y el framework Express



¿Ponemos a grabar el taller?



Agenda de hoy

- A. Administración de paquetes y dependencias
- B. NPM
- C. Nodemon
- D. El framework Express



Administración de paquetes y dependencias



Administración de paquetes y dependencias

Esto se refiere al **proceso de gestionar e instalar paquetes de software y sus dependencias** en una aplicación Node.js.

En éste, los paquetes de software se distribuyen a través de **npm** (*Node Package Manager*), que es una herramienta de línea de comandos que permite a los desarrolladores descargar, instalar y actualizar paquetes de software desde un repositorio centralizado.



Administración de paquetes y dependencias

La administración de paquetes en Node.js implica la creación y mantenimiento de un archivo llamado **package.json**, que describe la aplicación y sus dependencias.

Este archivo incluye información sobre los paquetes necesarios para que la aplicación funcione correctamente, así como versiones específicas de cada paquete.



Administración de paquetes y dependencias

Esto permitirá que sumemos a nuestros proyectos Node.js, distintas librerías, frameworks, y otros recursos adicionales que harán que, el desarrollo de una aplicación de backend, sea mucho más fácil.

El término “*dependencias*” viene justamente de las librerías y/o framework de los cuales Node.js depende para agregar nuevas características a la tarea de programar.



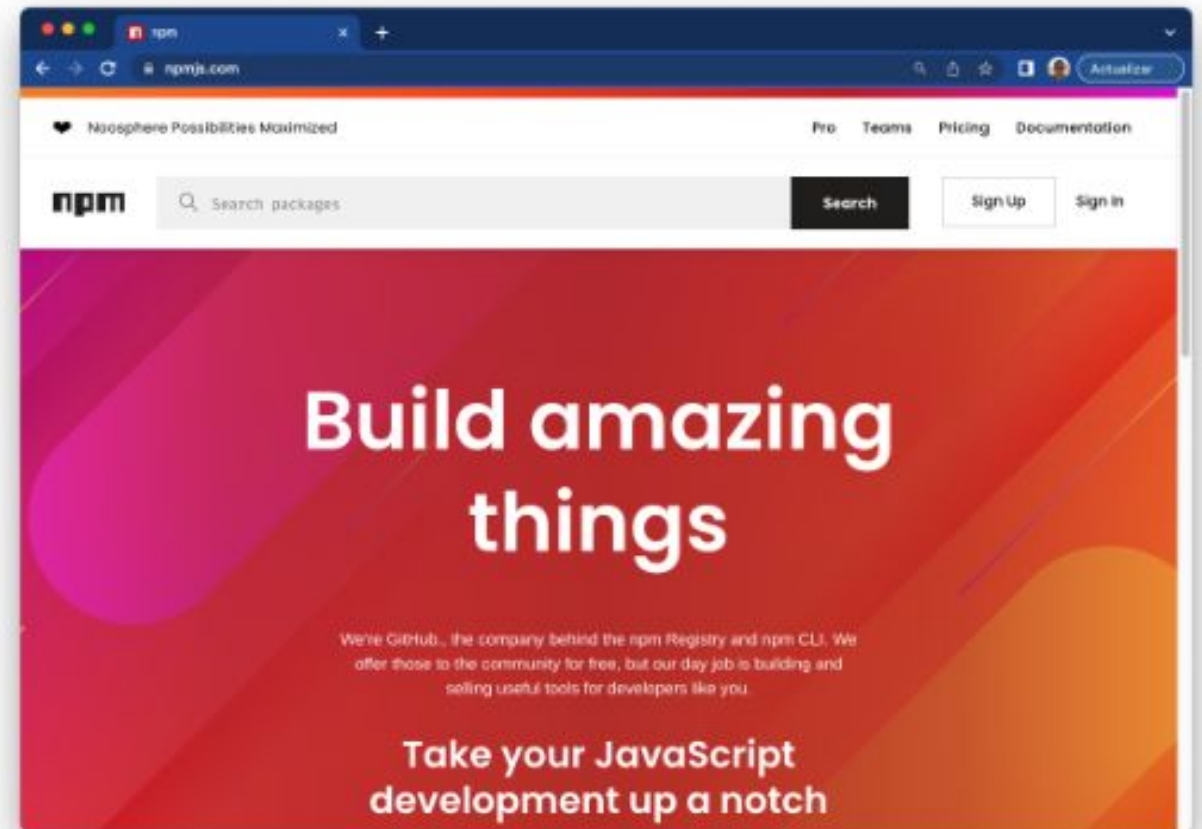
NPM



NPM

En nuestras primeras clases, hablamos de NPM, un gestor de paquetes y dependencias que viene incluido dentro de Node.

Desde hoy, comenzaremos a utilizarlo de forma frecuente.



NPM

NPM se rige por la línea de comandos para administrar los paquetes y dependencias en Node.js. Si bien existen otros tantos gestores como NPM, este es el más utilizado en la comunidad de Node.js, y tuvo estos últimos años, importantes mejoras de performance y manejo de errores.

NPM nos permite descargar, instalar y actualizar paquetes de software desde su repositorio centralizado: <https://www.npmjs.com/>



NPM

NPM gestiona toda la actividad en aplicaciones Node.js de la mano de diferentes comandos. Estos se utilizan a través de la ventana Terminal del S.O. o la ventana integrada a VS Code.

ESTADO	DESCRIPCIÓN
npm init	Crea un archivo "package.json" para describir la aplicación y sus dependencias. Es el comando utilizado para inicializar una aplicación Node.js.
npm install	Descarga e instala las dependencias necesarias en la carpeta node_modules .
npm update	Actualiza los paquetes de software a las últimas versiones disponibles.
npm uninstall	Desinstala un paquete de software específico y lo elimina de package.json .
npm run	Ejecuta un script personalizado especificado en package.json .
npm search	Busca paquetes de software disponibles en el repositorio de NPM.
npm publish	Publica un paquete de software creado por el desarrollador en el repositorio de NPM para que otros puedan descargarlo e instalarlo.

NPM

Estos son solo algunos de los comandos que dispone NPM y que pueden denominarse como los más utilizados.

Veamos a continuación algunos aspectos técnicos de NPM, **package.json** y la carpeta **node_modules**.



NPM

En resumen, NPM cuenta con los siguientes componentes:

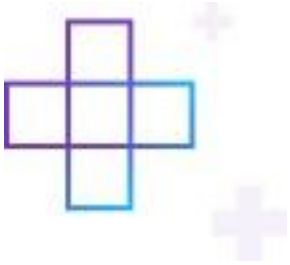
Repositorio: mantiene un repositorio web y descarga desde allí los paquetes de software que utilizaremos.

package.json: utiliza este archivo para describir las características técnicas de la aplicación, y de sus dependencias.

Línea de comandos: utiliza la Terminal para ejecutar comandos que permiten instalar, actualizar y desinstalar paquetes de software.



NPM



Además NPM como administrador de paquetes, contamos con otras alternativas a éste para administrar las dependencias en Node.js. Veamos a continuación cuáles son las más importantes por fuera de NPM.

YARN

Yarn es una alternativa popular a NPM que se centra en mejorar la velocidad y la estabilidad de las descargas y las instalaciones. Al igual que NPM, utiliza a package.json para gestionar las dependencias y cuenta con una amplia gama de paquetes disponibles en su propio repositorio.

PNPM

PNPM es otra alternativa que se centra en mejorar la eficiencia del almacenamiento en caché y la reutilización de dependencias entre proyectos. Utiliza un esquema de almacenamiento en caché compartido al instalar dependencias, lo que significa que cada proyecto no tiene que tener sus propias copias de paquetes duplicados.

RUSH

Rush, herramienta desarrollada por Microsoft, que se centra en la gestión de **proyectos monorepos**. Un monorepo es un repositorio que contiene múltiples proyectos o paquetes, y Rush proporciona una manera fácil de gestionar las dependencias y las versiones de cada uno de ellos. Rush también utiliza un sistema de almacenamiento en caché compartido



node_modules



node_modules

Cada dependencia que instalamos con NPM, descarga una serie de archivos y carpetas a nuestro proyecto. Estas se almacenan en la subcarpeta node_modules.

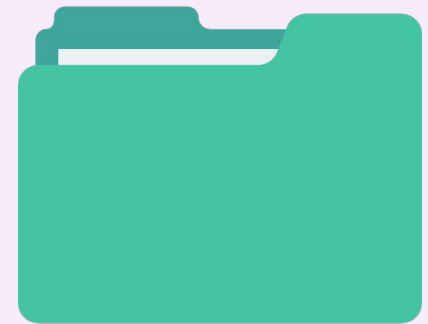
Cuando instalemos algo, veremos que dicha carpeta se crea automáticamente, al utilizar el comando NPM.



node_modules

Su contenido crece significativamente, y habitualmente no necesitamos buscar nada de allí.

Con referenciar la librería o framework desde nuestras aplicaciones JS, Node.js sabrá cómo y dónde ubicar a éste, dentro de la carpeta node_modules.



node_modules

Debemos tener la precaución, cuando subimos nuestros proyectos a un repositorio, de no subir también la carpeta **node_modules**. Esta crece mucho y ocupa demasiado espacio de almacenamiento.

Si debemos descargar el proyecto Node.js en otra computadora, gracias a NPM y el archivo package.json, podremos reinstalar rápidamente sus dependencias, generando nuevamente la carpeta node_modules.



Nodemon

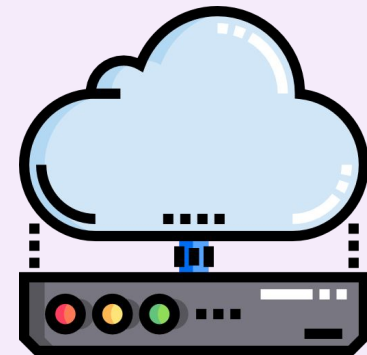


Nodemon

Hasta el momento, ejecutamos nuestros servidores web, utilizando la Terminal mediante los comandos de Node.

Llegó el momento de liberarnos de este proceso repetitivo, automatizando algunos pasos de la mano de las diferentes herramientas que Node nos provee.

Veamos a continuación, tres maneras de automatizar.



NPM y Scripting

NPM y Scripting

1. Editamos **package.json** y ubicamos la propiedad **scripts**, eliminando el valor predeterminado **'test'**.
2. Definimos un parámetro llamado **start**, y su valor será **node server.js**.
3. Luego, al momento de ejecutar el proyecto en la Terminal, tipeamos **npm start**, y se iniciará el proyecto.

```
Package.json

"scripts": {
  "start": "node server.js",
  "end": "killall -9 node"
},
```

NPM y Scripting

1. A continuación podemos definir otro parámetro llamado **end**, y que su valor sea **killall -9 node**.
 - a. Si se cuelga el proceso del servidor web en sistemas Unix, Linux, Mac, podremos interrumpirlo fácilmente.

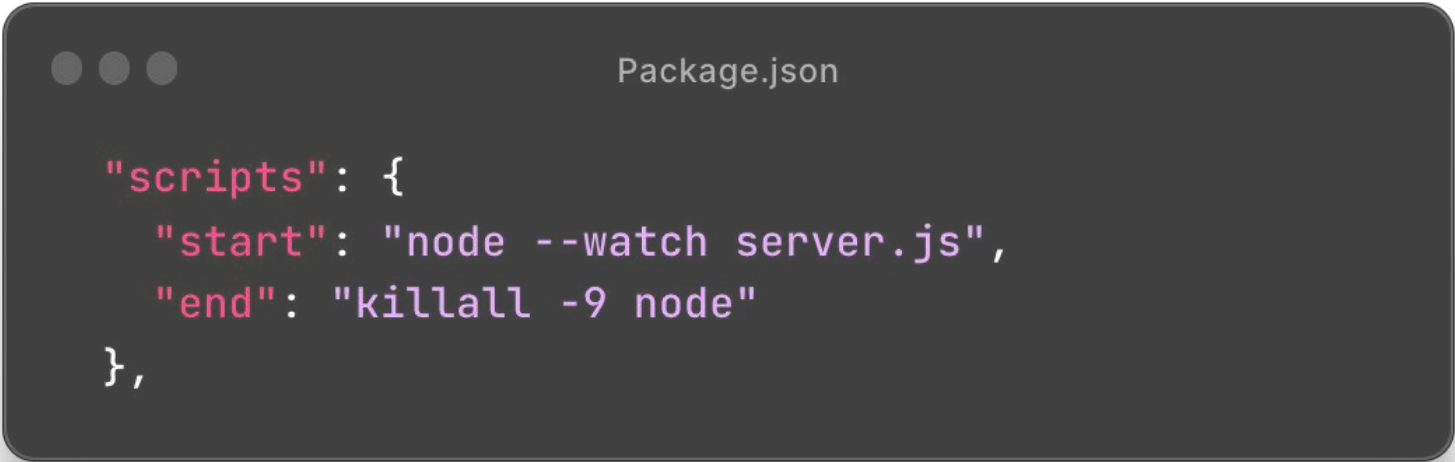
Package.json

```
"scripts": {  
  "start": "node server.js",  
  "end": "killall -9 node"  
},
```


NPM automatizado

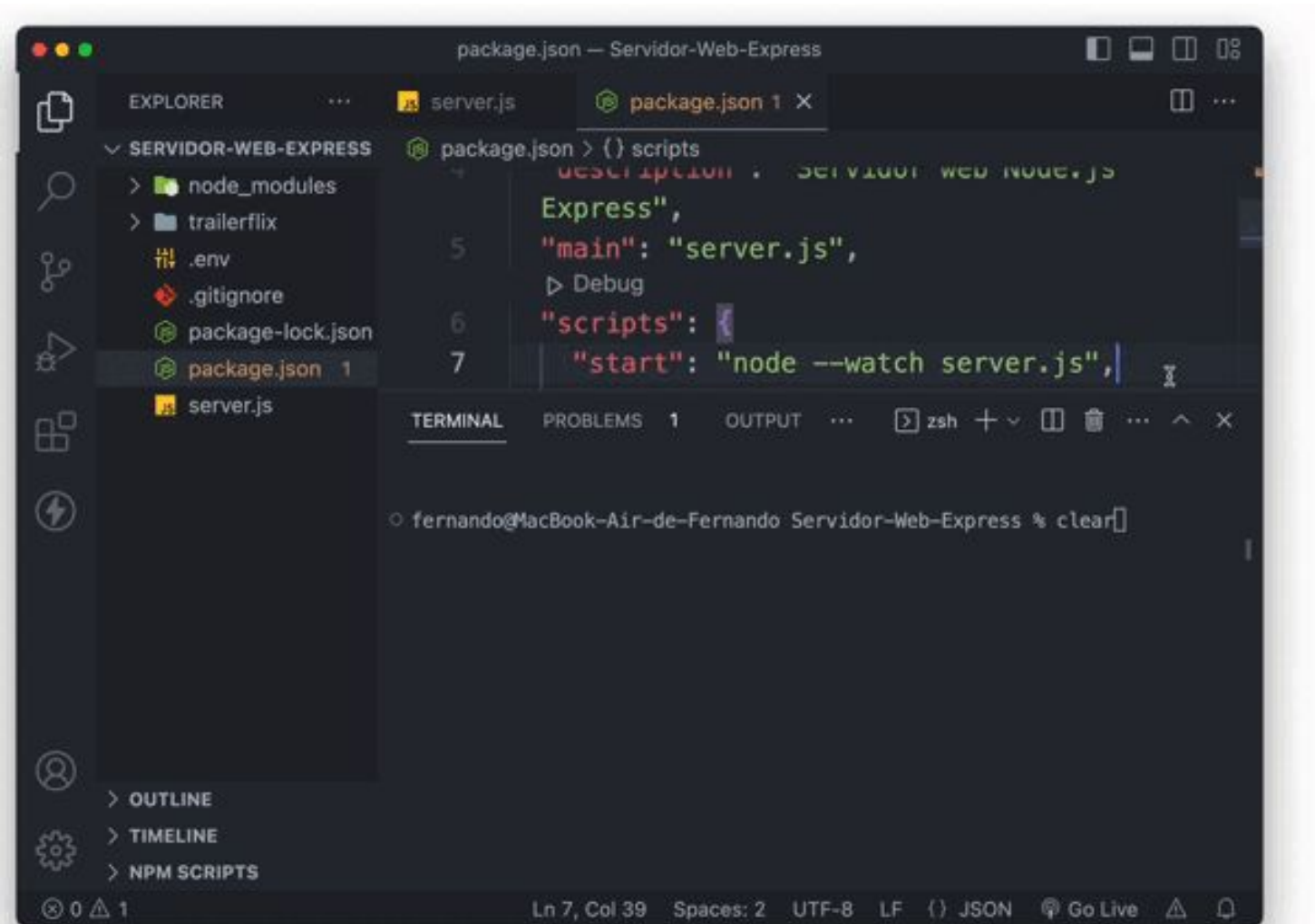
NPM automatizado

Hacia fines de 2022, Node incluyó un parámetro adicional en su línea de comandos, denominado **--watch**. Este permite ejecutar una aplicación Node.js en la Terminal con este parámetro adicional, y así que la aplicación en ejecución se detenga y vuelva a ejecutar cuando se detecta un cambio en el código.



```
"scripts": {  
  "start": "node --watch server.js",  
  "end": "killall -9 node"  
},
```

NPM automatizado



The screenshot shows the Visual Studio Code editor interface. The Explorer sidebar on the left displays the project structure for 'SERVIDOR-WEB-EXPRESS', including 'node_modules', 'trailerflix', '.env', '.gitignore', 'package-lock.json', 'package.json', and 'server.js'. The main editor area shows the 'package.json' file with the following content:

```
package.json > {} scripts
description: SERVIDOR WEB NODE.JS
Express",
"main": "server.js",
  Debug
"scripts": {
  "start": "node --watch server.js",
```

The bottom of the editor shows the TERMINAL panel with the command prompt: `fernando@MacBook-Air-de-Fernando Servidor-Web-Express % clear`. The status bar at the bottom indicates the current position is Line 7, Column 39, with 2 spaces, UTF-8 encoding, LF line endings, and a JSON file type.

De esta forma, seguimos utilizando el clásico comando **npm start**, aunque ahora, con cada cambio en el código, se reiniciará el proyecto Node.js automáticamente.

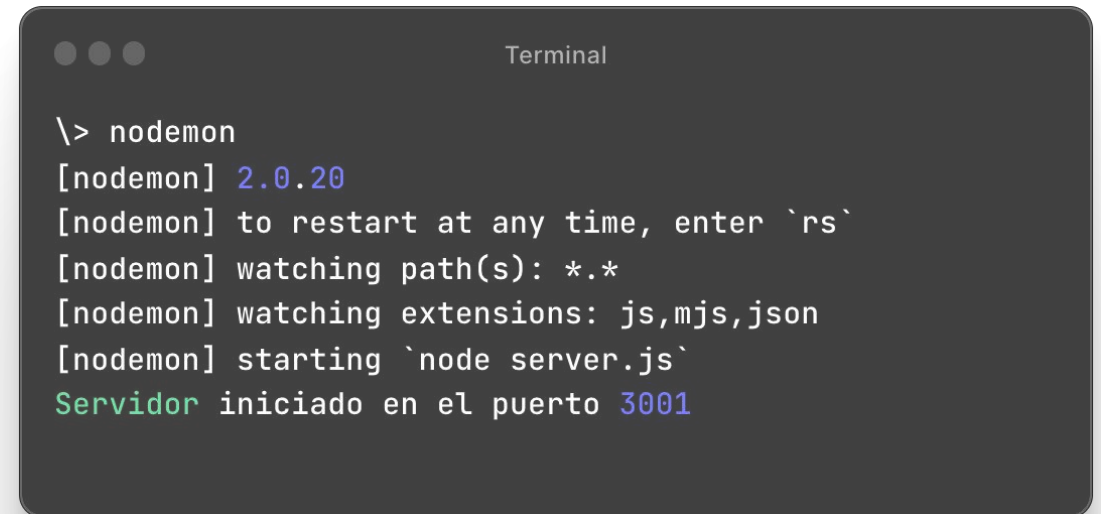
3

Nodemon

Nodemon

La otra alternativa para automatizar la ejecución de nuestra aplicación web, es instalando la dependencia nodemon desde la **Terminal**.

1. instalamos **Nodemon** utilizando el comando **npm install nodemon**.
2. Definimos el parámetro **script start**, tal como vimos en la primera opción de automatización.
3. Luego ejecutamos nodemon para que se inicie nuestra aplicación. Con cada cambio en el código, esta reiniciará automáticamente, disponibilizando de forma inmediata las nuevas características agregadas a la aplicación backend.



```
Terminal
\> nodemon
[nodemon] 2.0.20
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node server.js`
Servidor iniciado en el puerto 3001
```


Automatizar aplicaciones Node.js

Contamos con un sinfín de herramientas que permiten automatizar pequeñas cuestiones durante la etapa de desarrollo de software.

Si bien, **Nodemon** es lo más utilizado hoy, también contamos con la opción de utilizar **NPM** junto al parámetro **--watch**. Es una opción moderna y útil para evitar la instalación de dependencias como **Nodemon**, en pos de utilizar las opciones nativas.





Descanso

Nos vemos en 5 minutos





EL framework Express



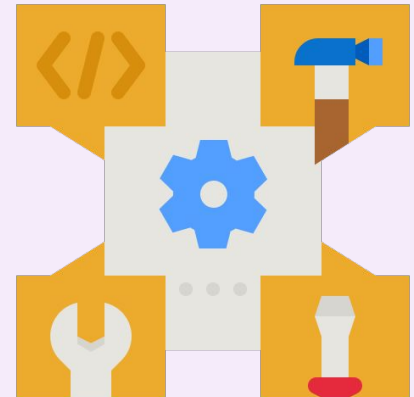
El framework Express

Para el desarrollo de aplicaciones en el backend utilizando Node.js, existen varios frameworks populares como ser:

- Express.js
- Nest.js
- Koa.js
- Hapi.js

y otros.

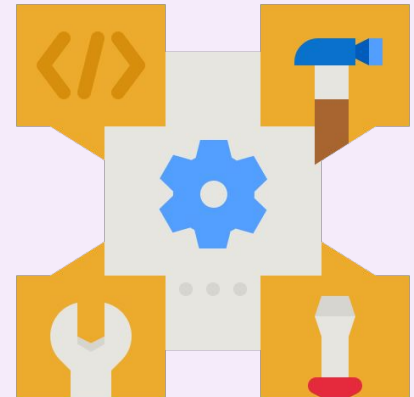
Estos proporcionan una amplia gama de funcionalidades para el manejo de rutas, gestión de peticiones y respuestas HTTP, integración con bases de datos y middleware para validar datos, autenticación, y seguridad, entre otras opciones.



El framework Express

Express.js es uno de los frameworks más populares para el desarrollo de aplicaciones web en Node.js, proporcionando una estructura flexible y minimalista para construir aplicaciones web y APIs.

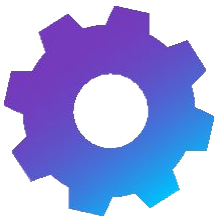
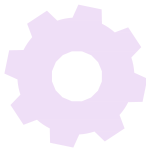
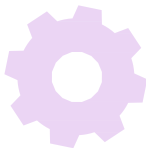
Será el elegido por nosotr@s para poder construir aplicaciones de backend basadas en servidores web, de una forma mucho más cómoda que la que propone el módulo HTTP.



NPM

Algunas de las ventajas destacables de utilizar Express, son:

VENTAJAS	DESCRIPCIÓN
Estructura de código	Proporciona una estructura de código clara y organizada para el desarrollo de aplicaciones web con funciones y métodos predefinidos, que facilitan la creación de rutas, middlewares, y controladores.
Middleware	Tiene un sistema de middleware flexible que permite agregar funcionalidades adicionales a la aplicación web, como autenticación, manejo de errores, compresión de archivos, entre otros.
Escalabilidad	Es altamente escalable, por lo cual puede manejar gran cantidad de solicitudes sin afectar el rendimiento de la aplicación. Además, su estructura modular permite agregar nuevas funcionalidades extendiendo la aplicación según sea necesario.
Comunidad y documentación	Es el gestor más popular utilizado junto a Node.js. Cuenta con una gran comunidad de desarrolladores y documentación exhaustiva y clara, que facilita el aprendizaje y la implementación de la tecnología.
Integración	Se integra fácilmente con otros módulos y librerías, pudiendo agregar nuevas funcionalidades a la aplicación web, como ser bb.dd, servicios de almacenamiento Cloud, servicios de autenticación, entre otros.



Express JS

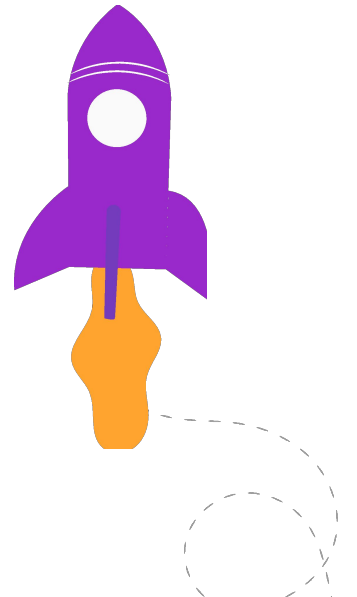
Instalación



Instalación de Express JS

Crearemos una nueva aplicación web backend. Definamos para ello, una **carpeta** en nuestra computadora, la cual llamaremos 📁 **Servidor-Web-Express**.

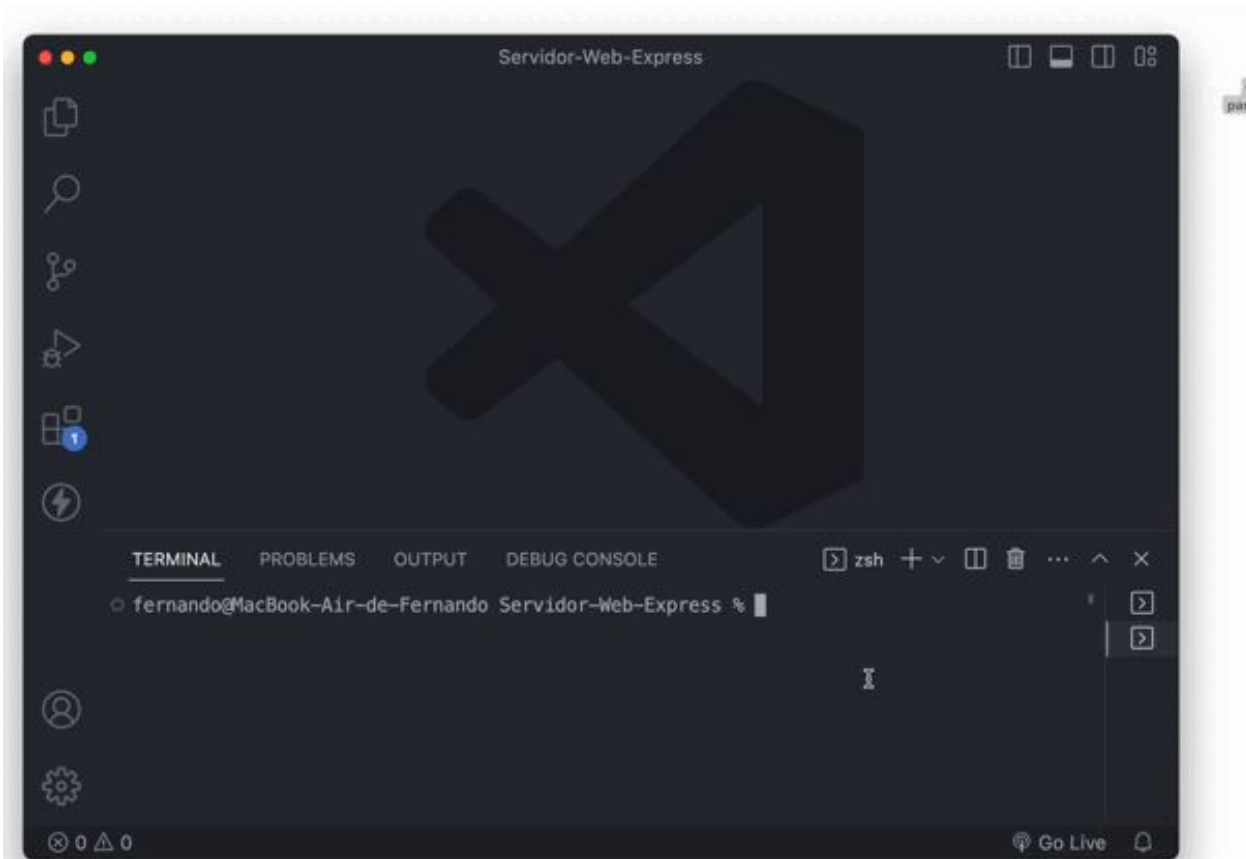
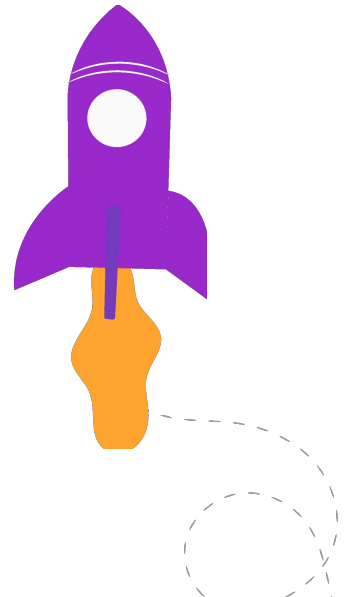
Recuerda crearla junto al resto de los proyectos, en un lugar donde puedas crear una copia de seguridad a tu cuenta de Github.



Instalación de Express JS

Inicializamos nuestro proyecto con **npm init -y**, y completamos la información del archivo **package.json**.

Luego instalamos el framework Express a través del comando **npm install express**. Se instalará Express creando la carpeta **node_modules**.





Express JS

Crear un servidor web con Express JS



Crear un servidor web con Express JS

```
Express

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

La declaración **require('express')** carga el módulo **express** y lo asigna a la constante homónima. Luego, instanciamos el framework dentro de la constante **app**.

```
Express

// Define una ruta básica
app.get('/', (req, res) => {
  res.send('¡Hola, mundo!');
});
```

El método **.get()** se ocupa de “escuchar las peticiones entrantes”, recibiendo dos parámetros: el primero, la ruta peticionada, el segundo, la función de respuesta a la petición, con **request** y **response** como parámetros.

Crear un servidor web con Express JS

Finalmente, el método **send()**, asociado a la respuesta (**response**), se utiliza para enviar la respuesta a dicha petición.

Como vemos, hasta aquí todo es bastante similar a lo que aprendimos trabajando con el módulo HTTP.

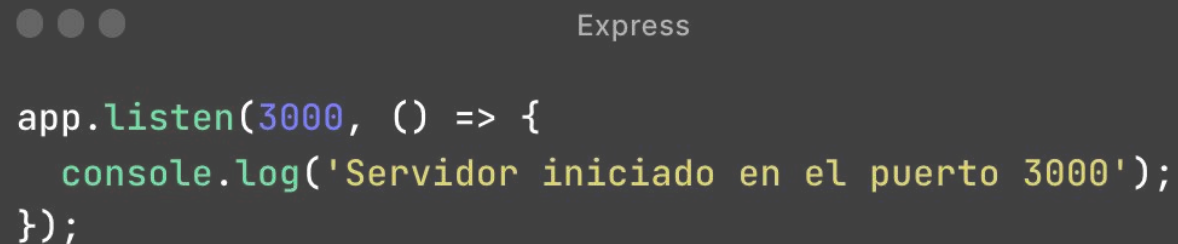


```
// Define una ruta básica
app.get('/', (req, res) => {
  res.send('¡Hola, mundo!');
});
```

Crear un servidor web con Express JS

Nos queda definir el puerto de escucha para nuestro servidor. El mecanismo también es similar al utilizado con el módulo HTTP.

Con todo esto, ya podemos ejecutar nuestro servidor web Express.

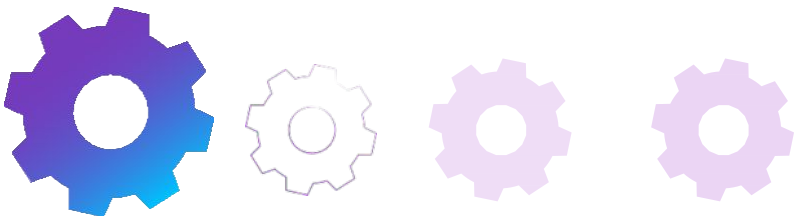


```
app.listen(3000, () => {  
  console.log('Servidor iniciado en el puerto 3000');  
});
```

Crear un servidor web con Express JS

Ya podemos abrir la Terminal, y ejecutar el comando **npm start** o, en su defecto **nodemon**, para poner a funcionar nuestro servidor web Express.

Luego, lo probamos desde un navegador web.

A screenshot of a code editor window titled "server.js — Servidor-Web-Express". The editor shows the following JavaScript code:

```
1  const express = require('express');
2  const app = express();
3
4  // Define una ruta básica
5  app.get('/', (req, res) => {
6    res.send('¡Hola, mundo!');
7  });
8
9  // Inicia el servidor
10 app.listen(3000, () => {
11   console.log('Servidor iniciado en el puerto 3000');
12 });
13
```

The editor interface includes a sidebar on the left with icons for Explorer, Search, Source Control, Run and Debug, Extensions, and User. The bottom status bar shows "Ln 8, Col 1", "Spaces: 4", "UTF-8", "LF", "JavaScript", and "Go Live".

Crear un servidor web con Express JS

Tengamos presente que Express JS, a diferencia del módulo HTTP, responde solo a rutas creadas.

Si petitionamos una ruta inexistente o no controlada por Express, este devolverá un error no controlado.

Debemos definir cada una de las rutas que utilizaremos, y un mensaje de error personalizado para todas aquellas rutas inexistentes en este servidor. 🤔





Express JS

Manejo de rutas



Manejo de rutas

Para que podamos definir rutas en Express JS, debemos utilizar el objeto "*Router*" proporcionado por la librería.

Si miramos el código de ejemplo de la clase anterior, veremos que el control de peticiones que maneja Express JS, además de la función de retorno con **request** y **response**, antepone un parámetro el cual evalúa una ruta.



Manejo de rutas

Al utilizar el parámetro “/”, estamos definiendo que esta petición escuchará la ruta principal, o raíz, de nuestro servidor web. Ante la primera petición a dicha ruta, responderá con un mensaje de texto, tal como vemos en el código contiguo.



Rutas con Express JS

```
// Define una ruta básica
app.get('/', (req, res) => {
  res.send('¡Hola, mundo. Hola, Node.js!');
});
```

Manejo de rutas

En todo servidor web, la primera ruta siempre es definida con esta notación y corresponde a la página principal que responde a la solicitud **HTTP GET** en la raíz del servidor (por ejemplo: **http://localhost:3001/**).

Si queremos crear múltiples rutas, debemos replicar esta misma estructura por cada una de ellas. Así definimos las rutas válidas que devolverán una respuesta.



Rutas con Express JS

```
// Define una ruta básica
app.get('/', (req, res) => {
  res.send('¡Hola, mundo. Hola, Node.js!');
});
```

Manejo de rutas

```
Express JS

// La ruta raíz
app.get('/', (req, res) => {
  res.send('¡Hola, mundo. Hola, Node.js!');
});

app.get('/nosotras', (req, res) => {
  res.send('Aquí tienes información sobre nuestra empresa.');
```

```
});

app.get('/cursos', (req, res) => {
  res.send('Este es el listado de cursos que brindamos.');
```

```
});
```

Aquí tenemos un ejemplo similar al elaborado con el módulo HTTP, pero traducido íntegramente al framework Express JS.

Cada una de las rutas definidas, enviará información como mensaje de respuesta, de acuerdo a lo que dice el ejemplo de código.

Express JS

Control de errores



Control de errores

En Express JS, es posible manejar solicitudes que no coinciden con ninguna de las rutas definidas. Esto se hace utilizando un middleware que se ejecuta después de todas las rutas definidas.

Para manejar rutas inexistentes, podemos agregar una ruta predeterminada (**catch-all**) utilizando **app.use()** y definir el controlador para esta ruta.



Control de errores

El método `.use()` es el que utilizamos para definir un Middleware que se ocupe de realizar tareas o procesos intermedios, por fuera de la lógica de nuestra aplicación.

Agregaremos entonces este método, contenido en el objeto `app`, para controlar toda aquella ruta que sea peticionada y que no exista en nuestra aplicación.



Control de errores

La estructura de este Middleware recibe como parámetro la misma función creada para el resto de las rutas definidas en nuestro servidor web, pero sin esperar un **path** específico.

El controlador para esta ruta devuelve una respuesta con un código de estado **404 - (recurso no encontrado)** y un mensaje para informar al usuario que el recurso solicitado no existe.



Rutas con Express JS

```
// Ruta predeterminada para manejar rutas inexistentes
app.use((req, res) => {
  res.status(404).send('Lo siento, la página que buscas no
  existe.');
```


Control de errores

También es muy común utilizar el método `.get()` seguido del path con el carácter comodín “*” para interceptar rutas inexistentes.

La respuesta hacia al cliente será la misma que en el ejemplo de uso del Middleware y la ubicación en la estructura del código de este ejemplo, también debe respetarse.

```
● ● ● Rutas con Express JS

// Ruta predeterminada para manejar rutas inexistentes
app.get('*', (req, res) => {
  res.status(404).send('Lo siento, la página que buscas no
  existe.');
```

Manejo de errores

Es importante destacar que, para definir correctamente el Middleware o el método **get("*")** que interceptan las peticiones a rutas o recursos inexistentes, sean alojados luego de todas las rutas definidas, para no interferir con las solicitudes que sí coinciden con rutas definidas.





Express JS

Servir un sitio web



Servir un sitio web

Para resolver esta tarea, debemos referenciar al módulo **path**, luego usamos un Middleware junto al método **static()** de Express JS.

Por último, **path.join()** nos ayudará a concatenar la variable global **__dirname** junto a la subcarpeta que contiene el sitio web.

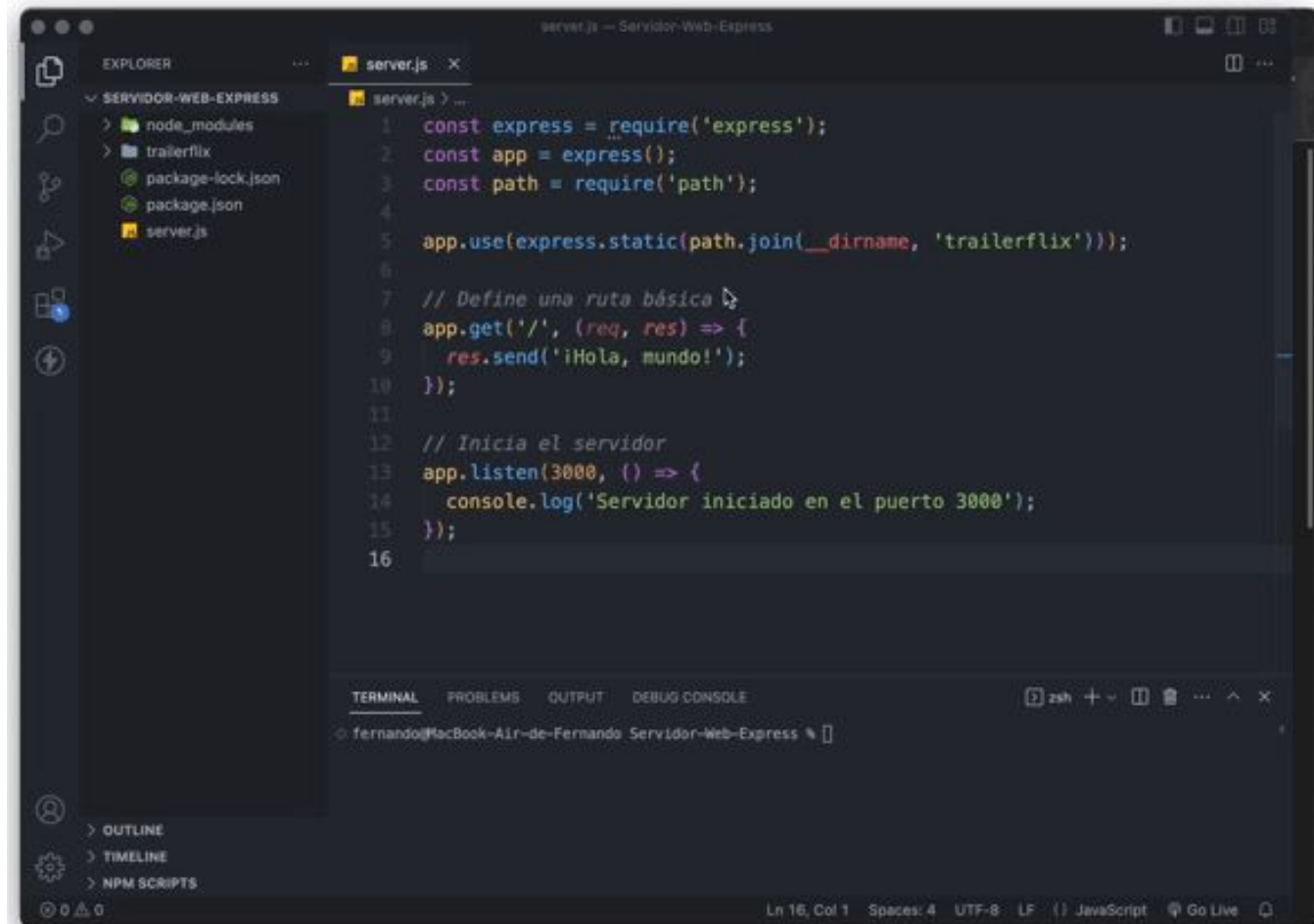
```
Express
const path = require('path');

app.use(express.static(path.join(__dirname, 'trailerflix')));
```

Servir un sitio web

Efectivamente, Express JS, es el Framework para crear servidores web que nos facilita la vida y acorta notoriamente los tiempos de desarrollo web.

Aquí, un claro ejemplo de ello 🙌.



The screenshot shows a Visual Studio Code editor window with a project named 'Servidor-Web-Express'. The Explorer sidebar on the left shows the file structure: 'node_modules', 'trailerflix', 'package-lock.json', 'package.json', and 'server.js'. The main editor area displays the content of 'server.js', which is a JavaScript file for starting an Express server. The code includes imports for 'express' and 'path', sets up a static directory for 'trailerflix', defines a basic GET route for the root path that responds with '¡Hola, mundo!', and starts the server on port 3000. The bottom of the window shows a terminal with the prompt 'fernando@MacBook-Air-de-Fernando: Servidor-Web-Express %'.

```
1 const express = require('express');
2 const app = express();
3 const path = require('path');
4
5 app.use(express.static(path.join(__dirname, 'trailerflix')));
6
7 // Define una ruta básica
8 app.get('/', (req, res) => {
9   res.send('¡Hola, mundo!');
10 });
11
12 // Inicia el servidor
13 app.listen(3000, () => {
14   console.log('Servidor iniciado en el puerto 3000');
15 });
16
```

Desafío

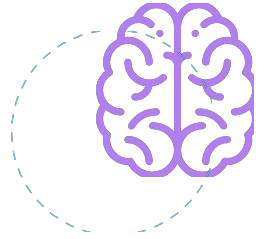
Recuperamos el desafío realizado en la unidad anterior y convertiremos el código del servidor web basado en el módulo HTTP, en un proyecto nuevo utilizando Express como servidor web.

Respetaremos la creación de cada ruta definida previamente y el formato de datos con el que se responde la petición a dicha ruta.

Crearemos también el control de errores sobre rutas inexistentes.



Prácticas



Definiremos una constante **PORT**, con el valor **3050**. En el servidor web, deben seguir las siguientes rutas definidas:

- “/”
- “/cursos”
- “/contacto”

El formato de respuesta para cada ruta, queda exactamente igual a cómo lo definimos en el proyecto que utiliza el módulo HTTP.

En el **control de rutas inexistentes**, crearemos una estructura JSON, la cual debe enviarse como respuesta a las rutas inexistentes que sean peticionadas. Ejemplo:

```
{“error”: “404”, “description”: “No se encuentra la ruta o recurso solicitado.”}
```

Modificaremos también package.json, agregando el script correspondiente para inicializar el proyecto con **Nodemon**.



¿Alguna consulta?





¡Muchas gracias!

FUNDACIÓN
YPF