

TEXT CLASS REVIEW

TEMAS A TRATAR EN EL CUE:

- Descripción de Npm.
- Ecosistema de JavaScript y Node.js.
- Uso de Npm por consola.
- Descripción de archivo package.json.
- Manejo de versiones de paquetes.
- Cómo utilizar paquetes en nuestro código.
- ¿Qué es Express?
- Separando archivos para orden y optimización.

¿QUÉ ES NPM?

Es el gestor de paquetes de **Node**. Un paquete en **node** es un conjunto de código que puedes utilizar en tu programa, y que generalmente corresponden a utilidades que harán más fácil tu trabajo como programador. Para hacer uso de éstos, primero debes realizar una instalación de ellos, y es ahí donde entra **npm**.

Una de las grandes bondades de JavaScript es su extenso ecosistema; éste ofrece una cantidad gigante de paquetes para utilizar.

Npm Inc. (www.npmjs.com), es la organización encargada de mantener esta colección de paquetes con código open-source para **Node**.

Cuando instalas Node, esta herramienta se instala por defecto. Para comprobar que esto es así, iremos al terminal, y escribiremos **"npm -v"** sin comillas. Se debería ver la versión de npm.



```
C:\Users>npm -v
6.14.11
C:\Users>
```

El comando **npm** en tu terminal es un CLI (Command Line Interpreter), el cual te permite llevar a cabo todas las acciones referentes a la gestión de paquetes dentro de tu proyecto, ya sea: instalar, actualizar, desinstalar, entre otras opciones.

COMANDOS BÁSICOS

- **Npm init:** permite inicializar un nuevo proyecto. Éste creará un archivo **package.json**, que es un tipo de archivo especial que necesita su propia sección, y por lo tanto, lo estudiaremos en el próximo CUE.

```
C:\Users\Proyecto>npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help init` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (proyecto)
```

- **npm install nombre-paquete:** da la instrucción para la instalación de un paquete determinado. Ejemplo: **npm install express**.

```
C:\Users\Proyecto>npm install express
[.....] / rollbackFailedOptional: verb npm-session cbd3688f3af55046
```

- **npm uninstall nombre-paquete:** indica la desinstalación de un paquete determinado. Ejemplo: **npm uninstall express**.

```
C:\Users\Proyecto>npm uninstall express
[.....] | postinstall: sill install executeActions
```



- **npm list:** da la instrucción para mostrar por la terminal, la lista de paquetes instalados.

```
C:\Users\Proyecto>npm list
C:\Users\Proyecto
|-- express@4.17.1
  +-- accepts@1.3.7
    | +-- mime-types@2.1.31
    | | `-- mime-db@1.48.0
    | `-- negotiator@0.6.2
  +-- array-flatten@1.1.1
  +-- body-parser@1.19.0
    | +-- bytes@3.1.0
    | +-- content-type@1.0.4 deduped
    | +-- debug@2.6.9 deduped
    | +-- depd@1.1.2 deduped
    | +-- http-errors@1.7.2
    | | +-- depd@1.1.2 deduped
    | | +-- inherits@2.0.3
    | | +-- setprototypeof@1.1.1 deduped
    | | +-- statuses@1.5.0 deduped
    | | `-- toidentifier@1.0.0
    | +-- iconv-lite@0.4.24
    | | `-- safer-buffer@2.1.2
    | +-- on-finished@2.3.0 deduped
```

- **npm Audit:** da la instrucción para auditar los paquetes instalados, e informar sobre sus posibles vulnerabilidades.

```
C:\Users\Proyecto>npm audit

=== npm audit security report ===

found 0 vulnerabilities
in 50 scanned packages
```

- **npm update:** se utiliza para actualizar los paquetes que hayamos instalado.

```
npm update
```

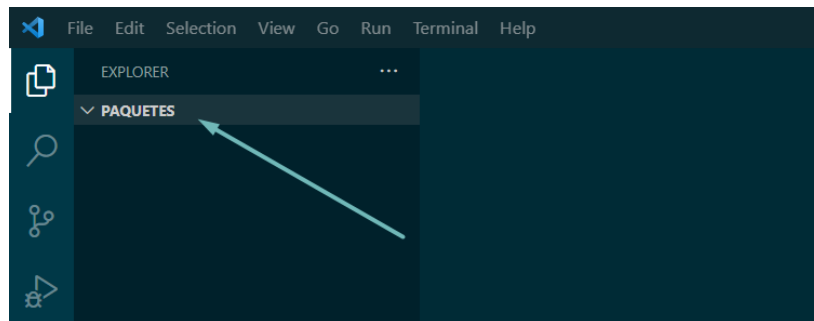
- **npm outdated:** imprime una lista con los paquetes desactualizados.

```
npm outdated
```

Ahora que conocemos estos comandos, procedemos a realizar la instalación, y usaremos nuestro primer paquete.

Este paquete se llama **nodemon**, y nos permite reiniciar nuestro código de manera automática, cada vez que detecta cambios en él; nos evita la tediosa tarea de volver a ejecutar manualmente nuestro programa, cada vez que realizamos un cambio a nuestro código.

Crearemos una nueva carpeta llamada paquetes, y abriremos Visual Studio Code usándola para trabajar.



Luego, abriremos una nueva terminal, e ingresaremos el comando **npm init**.

Una vez ingresado este comando, la terminal irá requiriendo información: el nombre de tu proyecto, la versión, punto de entrada (el archivo con el que se iniciará tu programa, por defecto index.js), comando para testing, repositorio de git, palabras clave para ayudar en la búsqueda del registro de **npm**, autor, tipo de licencia, y por último, confirmar los datos ingresados. Por ahora no hay problema con que dejes los campos vacíos, solo debes presionar la tecla enter hasta que terminen las preguntas. Pronto trataremos más en detalle todos estos campos.



```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

Microsoft Windows [Versión 10.0.18363.1679]
(c) 2019 Microsoft Corporation. Todos los derechos reservados.

C:\Users\paquetes>npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See 'npm help init' for definitive documentation on these fields
and exactly what they do.

Use 'npm install <pkg>' afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (paquetes)
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
keywords:
license: (ISC)
About to write to C:\Users\paquetes\package.json:

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

Is this OK? (yes)
C:\Users\paquetes>
```

Ahora, para instalar nuestro primer paquete, ingresamos el comando: **npm install nodemon -g**.

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

C:\Users\paquetes>npm install nodemon -g
C:\Users\Administrador\AppData\Roaming\npm\nodemon -> C:\Users\Administrador\AppData\Roaming\npm\node_modules\nodemon\bin\nodemon.js

> nodemon@2.0.12 postinstall C:\Users\Administrador\AppData\Roaming\npm\node_modules\nodemon
> node bin/postinstall || exit 0

Love nodemon? You can now support the project via the open collective:
> https://opencollective.com/nodemon/donate

npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@2.3.2 (node_modules\nodemon\node_modules\chokidar\node_modules\fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@2.3.2: wanted {"os":"darwin","arch":"any"} (current: {"os":"win32","arch":"x64"})

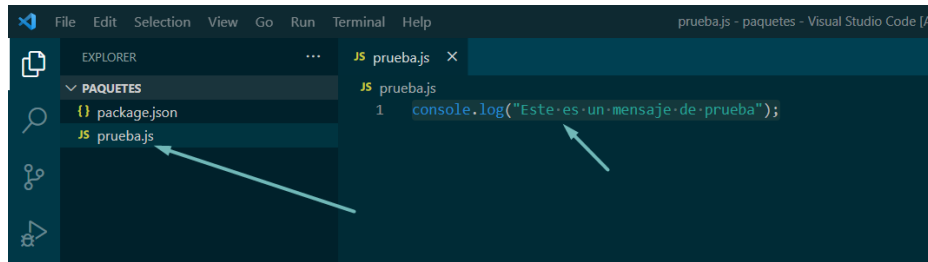
+ nodemon@2.0.12
added 119 packages from 53 contributors in 23.673s

C:\Users\paquetes>
```

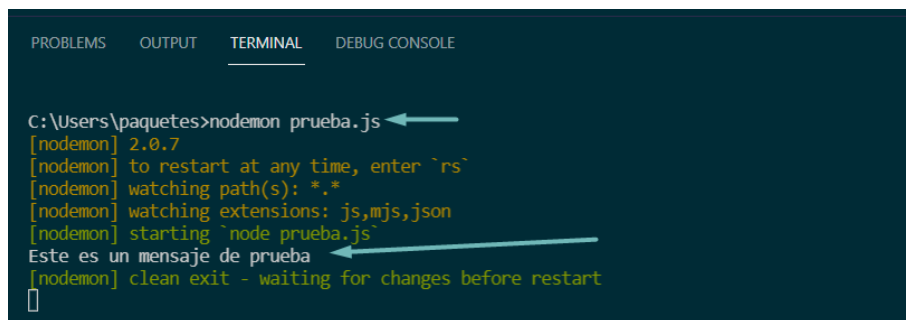
Como puedes observar, hemos agregado **“-g”** al final de nuestro comando. Esto es conocido como **“flag”**, y es muy común que los comandos utilizados en tu terminal acepten su uso. Se puede entender su concepto como opciones de configuración que un comando puede aceptar, en este caso, **-g** viene de **“global”**, y significa que la instalación de **nodemon** funcionará para todos tus proyectos de la misma manera.



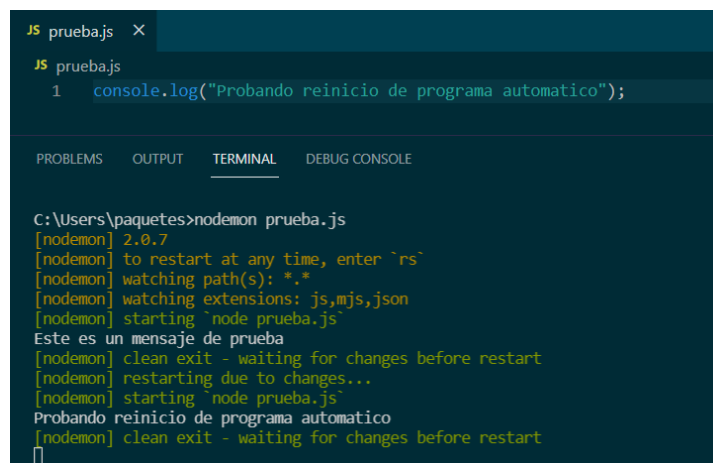
El uso de **nodemon** es muy simple. Primero, crearemos un archivo de prueba, y escribiremos un **console.log()**.



Vamos a nuestro terminal, y escribimos el siguiente comando: **nodemon** prueba.js.



Realizaremos un cambio a nuestro archivo. Presionamos **ctrl + s** para guardar, y veremos que en nuestra terminal el programa se reinicia.

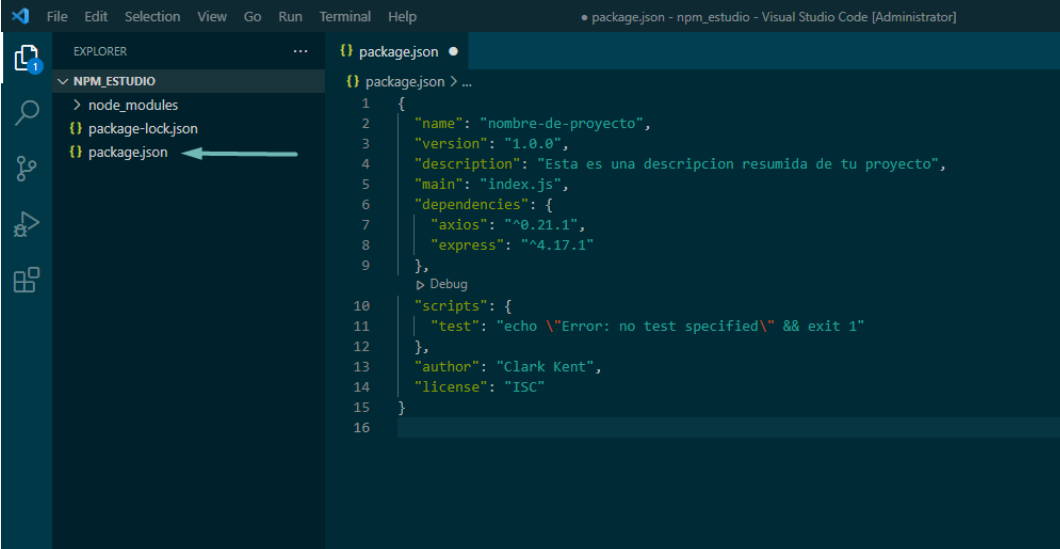


Si revisas la carpeta contenedora de tu proyecto, notarás que existe una carpeta llamada `node_modules`. Ésta agrupa todo el código de los paquetes que has instalado; se ignora al momento de subirla a un repositorio, ya que, junto con la instalación de paquetes, hay un archivo especial que se genera y que guarda las características de tu proyecto, incluyendo la lista de paquetes instalados. En el próximo bloque trataremos más información sobre este archivo en particular.

PACKAGE.JSON Y GESTIÓN DE DEPENDENCIAS


Hasta este momento, solo hemos mencionado un archivo muy particular, llamado `package.json`. Éste aparece cuando utilizamos el comando `npm init`, y es aquel que contiene todos los datos que representan a tu proyecto en formato `json` (tal como lo indica su extensión). Podríamos considerarlo como el plano para construir y hacer correr tu programa o proyecto.

En el ejercicio anterior, creamos una carpeta llamada “npm_estudio”, e inicializamos un proyecto utilizando `npm init`. Además, se instalaron unos paquetes con el comando `npm install`. Revisemos el contenido del archivo `package.json`.



```
1 {
2   "name": "nombre-de-proyecto",
3   "version": "1.0.0",
4   "description": "Esta es una descripción resumida de tu proyecto",
5   "main": "index.js",
6   "dependencies": {
7     "axios": "^0.21.1",
8     "express": "^4.17.1"
9   },
10  "scripts": {
11    "test": "echo \\\"Error: no test specified\\\" && exit 1"
12  },
13  "author": "Clark Kent",
14  "license": "ISC"
15 }
```

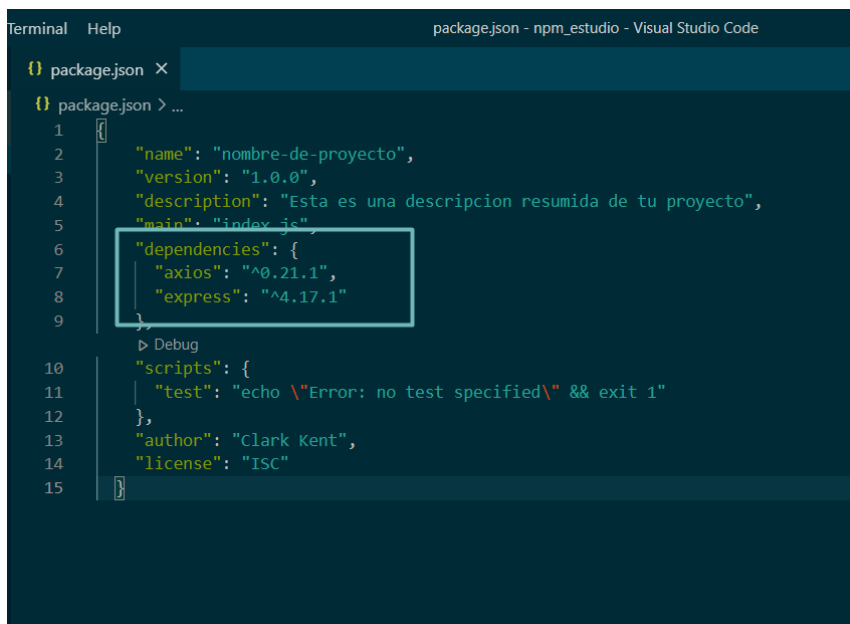
Podemos observar todos los datos que fueron agregados al momento de ejecutar el comando `npm init`.



```
terminal  Help  package.json - npm_estudio - Visual Studio Code

{} package.json ×
{} package.json > ...
1  {
2    "name": "nombre-de-proyecto",
3    "version": "1.0.0",
4    "description": "Esta es una descripcion resumida de tu proyecto",
5    "main": "index.js",
6    "dependencies": {
7      "axios": "^0.21.1",
8      "express": "^4.17.1"
9    },
10   > Debug
11   "scripts": {
12     "test": "echo \\\"Error: no test specified\\\" && exit 1"
13   },
14   "author": "Clark Kent",
15   "license": "ISC"
}
```

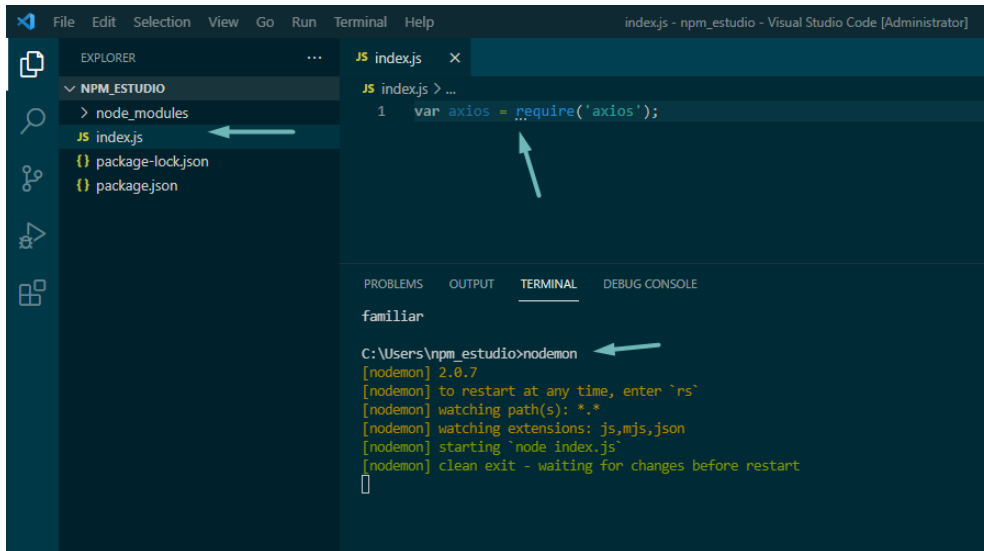
También podemos ver todas las dependencias de tu proyecto, que son los paquetes que quedaron instalados para su uso posterior (recuerda que se desinstalaron dos, de los cuatro instalados inicialmente).



```
terminal  Help  package.json - npm_estudio - Visual Studio Code

{} package.json ×
{} package.json > ...
1  {
2    "name": "nombre-de-proyecto",
3    "version": "1.0.0",
4    "description": "Esta es una descripcion resumida de tu proyecto",
5    "main": "index.js",
6    "dependencies": {
7      "axios": "^0.21.1",
8      "express": "^4.17.1"
9    },
10   > Debug
11   "scripts": {
12     "test": "echo \\\"Error: no test specified\\\" && exit 1"
13   },
14   "author": "Clark Kent",
15   "license": "ISC"
}
```


Vamos a la siguiente prueba. Crea un archivo `index.js`, escribe el siguiente código, y ejecuta tu programa con `nodemon` o `node index.js`.



```
File Edit Selection View Go Run Terminal Help
index.js - npm_estudio - Visual Studio Code [Administrator]

EXPLORER
NPM_ESTUDIO
  > node_modules
  JS index.js
  {} package-lock.json
  {} package.json

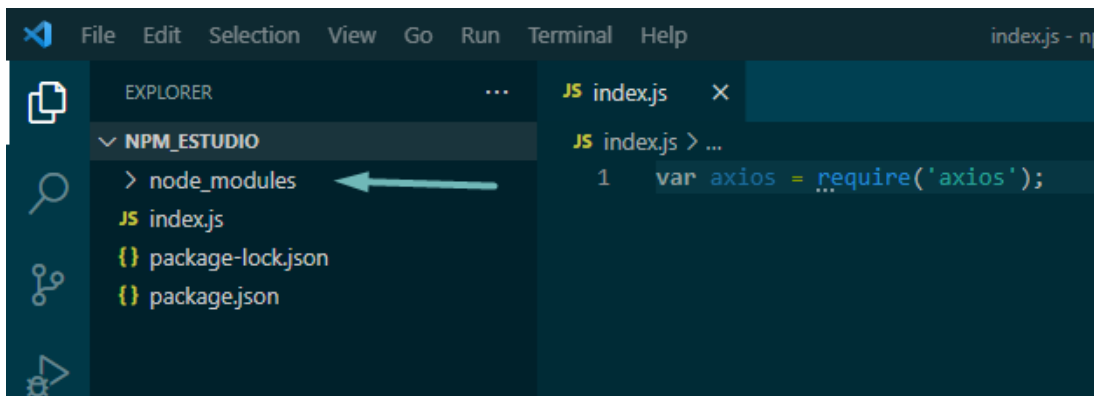
JS index.js
1 var axios = require('axios');

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE
familiar

C:\Users\npm_estudio>nodemon
[nodemon] 2.0.7
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index.js`
[nodemon] clean exit - waiting for changes before restart
[]
```

Este código está requiriendo el paquete `axios` para poder usarlo dentro de nuestro proyecto. Pronto entraremos en detalle respecto a esto, por ahora, veamos el siguiente ejemplo.

Elimina la carpeta `node_modules`.



```
File Edit Selection View Go Run Terminal Help
index.js - npm_estudio - Visual Studio Code [Administrator]

EXPLORER
NPM_ESTUDIO
  > node_modules
  JS index.js
  {} package-lock.json
  {} package.json

JS index.js
1 var axios = require('axios');
```

Ejecuta tu programa con **nodemon**, o **node index.js**.

```

index.js - npm_estudio - Visual Studio Code [Administrator]

EXPLORER
  NPM_ESTUDIO
    index.js
    package-lock.json
    package.json

index.js
1  var axios = require('axios');

TERMINAL
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting 'node index.js'
internal/modules/cjs/loader.js:883
  throw err;
  ^

Error: Cannot find module 'axios'
Require stack:
- C:\Users\npm_estudio\index.js
    at Function.Module._resolveFilename (internal/modules/cjs/loader.js:880:15)
    at Function.Module._load (internal/modules/cjs/loader.js:725:27)
    at Module.require (internal/modules/cjs/loader.js:952:19)
    at require (internal/modules/cjs/helpers.js:88:18)
    at Object.<anonymous> (C:\Users\npm_estudio\index.js:1:13)
    at Module._compile (internal/modules/cjs/loader.js:1063:30)
    at Object.Module._extensions..js (internal/modules/cjs/loader.js:1092:10)
    at Module.load (internal/modules/cjs/loader.js:928:32)
    at Function.Module._load (internal/modules/cjs/loader.js:769:14)
    at Function.executeUserEntryPoint [as runMain] (internal/modules/run_main.js:72:12) {
  code: 'MODULE_NOT_FOUND',
  requireStack: [ 'C:\Users\npm_estudio\index.js' ]
}
[nodemon] app crashed - waiting for file changes before starting...
  
```

Tenemos un error. Nuestro programa está tratando de requerir un paquete, pero no ha sido capaz de encontrar las dependencias necesarias.

¿Recuerdas que cuando subimos nuestro proyecto a git, generalmente ignoramos la carpeta **node_modules** para no subir código innecesario?

Aquí es donde entra una de las características más importantes del archivo **package.json**.

Ejecuta el comando **npm install**, sin ningún argumento.

```

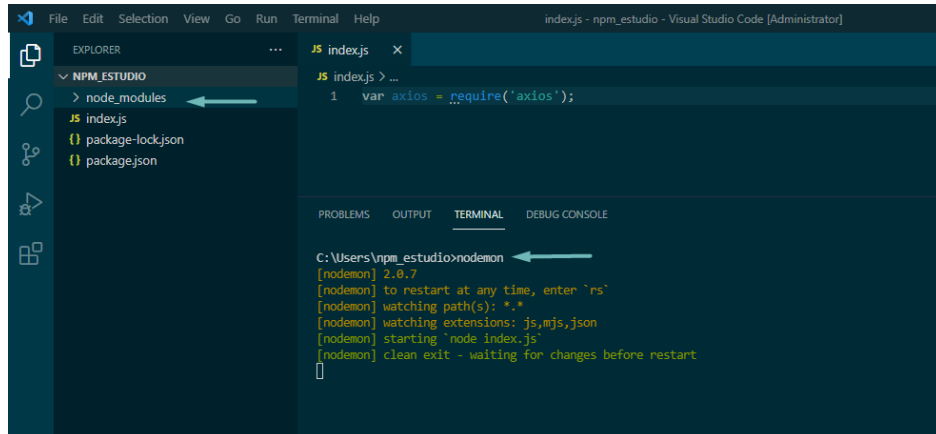
index.js - npm_estudio - Visual Studio Code [Administrator]

EXPLORER
  NPM_ESTUDIO
    index.js
    package-lock.json
    package.json

index.js
1  var axios = require('axios');

TERMINAL
C:\Users\npm_estudio>npm install
[.....] - extract:moment: verb lock using C:\Users\Administrador\AppData\Roaming\npm-cache\
  
```

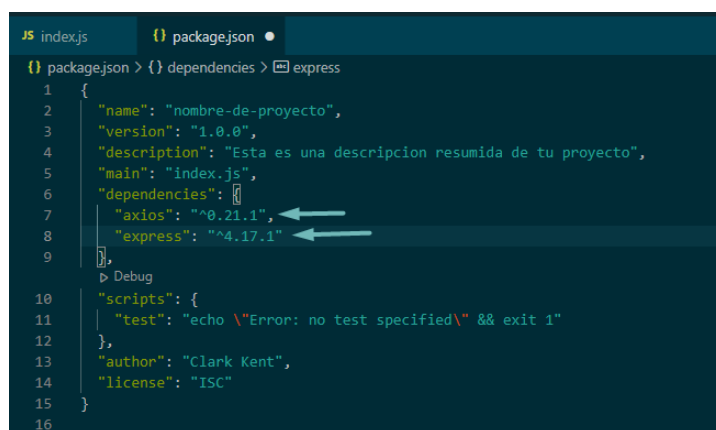
Revisa la carpeta **node_modules**, y ejecuta tu programa nuevamente.



Gracias al archivo **package.json**, siempre podremos contar con las dependencias que nuestro programa necesita. Si estamos trabajando con otros desarrolladores, no es necesario que ellos descarguen y agreguen manualmente todos los paquetes para que tu programa funcione, pues solo necesitan de tu código, y del archivo **package.json**. Esto asegura que no existan inconsistencias al momento de ejecutarlo en distintos ambientes.

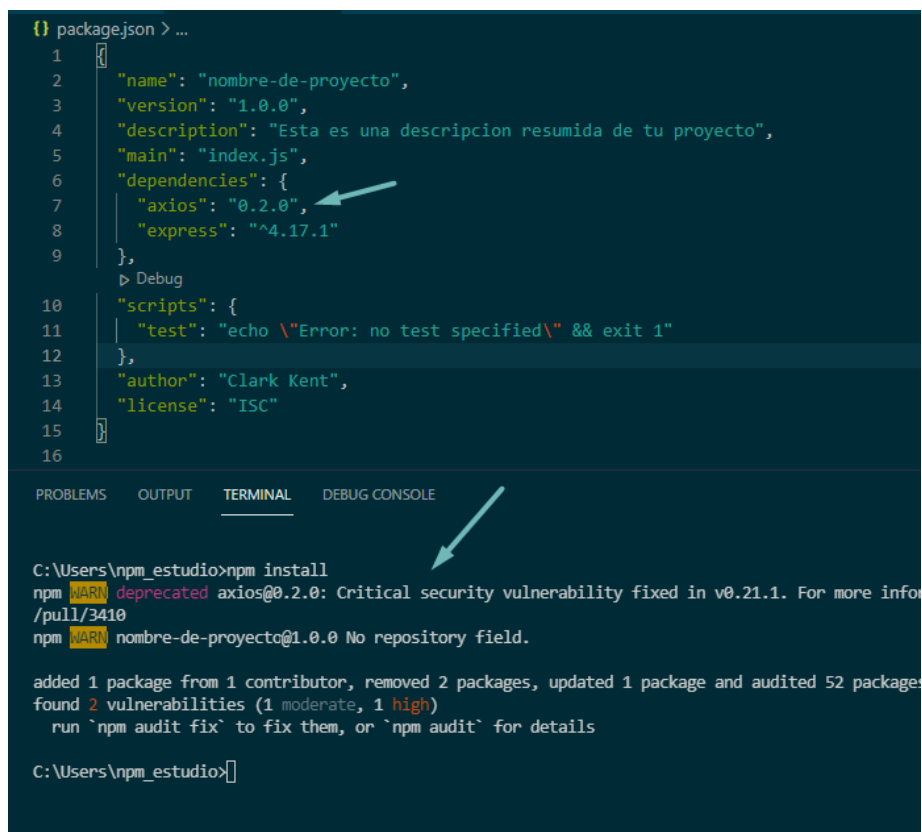
Como puedes ver, las dependencias de tu proyecto también tienen su versión.

Es posible que si existe algún cambio de versión en alguna de las dependencias de tu proyecto, se genere un bug o problema inesperado. Si alguien más quisiera usarlo, y descargara e instalara manualmente las dependencias, es muy posible que se instale una versión que sea distinta a la que utiliza tu programa, lo cual también puede causar inconsistencias. Es por ello que el archivo **package.json** también guarda las versiones de tus dependencias.



Puedes modificar dichas directamente en tu archivo `package.json`, y luego ejecutar el comando `npm install` sin argumentos, para así instalar la versión recién editada. Es importante revisar en la documentación de los paquetes que estás usando, para saber cuáles son las versiones exactas a las que puedes acceder.

También es importante mencionar que el símbolo “**caret**” (^), al inicio de la versión, significa que cada vez que se realice la instalación de dependencias utilizando el `package.json`, usará todas las actualizaciones desde esa versión, hasta antes del siguiente cambio a una más actualizada. Por ejemplo: la versión de `axios` en `package.json` es “^0.21.1”, por lo tanto, solo serán considerados los cambios hasta antes de la versión 1.0.0; si quieres instalar la versión exacta de un paquete, solo debes remover el símbolo “caret” (Puedes leer más sobre esto en la documentación de npm: <https://docs.npmjs.com/cli/v7/using-npm/semver#caret-ranges-123-025-004>).



```
{
  "name": "nombre-de-proyecto",
  "version": "1.0.0",
  "description": "Esta es una descripcion resumida de tu proyecto",
  "main": "index.js",
  "dependencies": {
    "axios": "0.2.0",
    "express": "^4.17.1"
  },
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Clark Kent",
  "license": "ISC"
}
```

TERMINAL

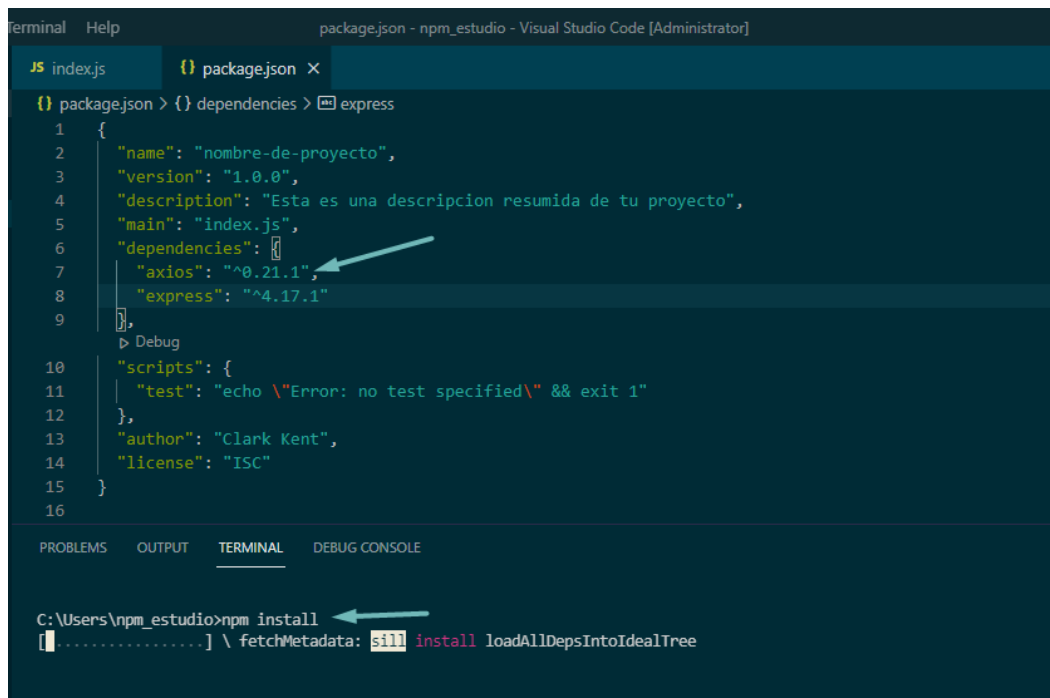
```
C:\Users\npm_estudio>npm install
npm WARN deprecated axios@0.2.0: Critical security vulnerability fixed in v0.21.1. For more info
/pull/3410
npm WARN nombre-de-proyecto@1.0.0 No repository field.

added 1 package from 1 contributor, removed 2 packages, updated 1 package and audited 52 packages
found 2 vulnerabilities (1 moderate, 1 high)
  run `npm audit fix` to fix them, or `npm audit` for details

C:\Users\npm_estudio>
```

Una de las razones para mantener siempre tus paquetes actualizados a las últimas versiones, es la existencia de vulnerabilidades de seguridad o bugs que se van encontrando, tal como lo muestra el mensaje de **npm** por consola.

Luego, puedes hacer lo mismo para volver a la versión más reciente.

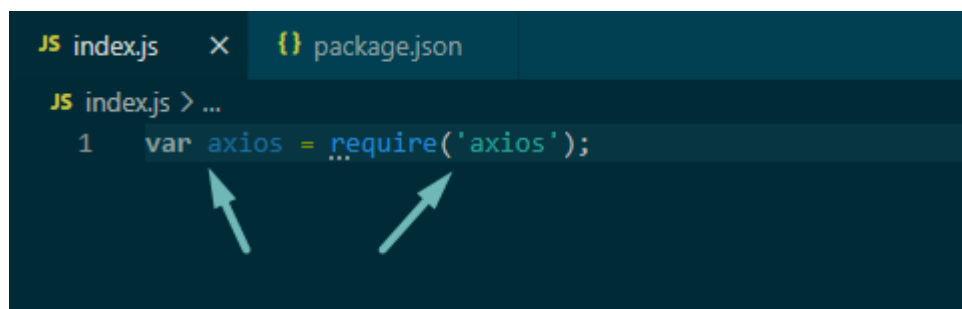


```

Terminal Help package.json - npm_estudio - Visual Studio Code [Administrator]
JS index.js {} package.json X
{} package.json > {} dependencies > express
1 {
2   "name": "nombre-de-proyecto",
3   "version": "1.0.0",
4   "description": "Esta es una descripcion resumida de tu proyecto",
5   "main": "index.js",
6   "dependencies": {
7     "axios": "^0.21.1",
8     "express": "^4.17.1"
9   },
10  "scripts": {
11    "test": "echo \\\"Error: no test specified\\\" && exit 1"
12  },
13  "author": "Clark Kent",
14  "license": "ISC"
15 }
16
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE
C:\Users\npm_estudio>npm install
[.....] \ fetchMetadata: sill install loadAllDepsIntoIdealTree
  
```

En este punto, desde el archivo **index.js** ya hemos requerido un paquete en nuestro programa, pero no hemos hecho uso de éste.

Para requerir un paquete, generalmente se utiliza la forma `var nombre = require(nombre)`.



```

JS index.js X {} package.json
JS index.js > ...
1 var axios = require('axios');
  
```

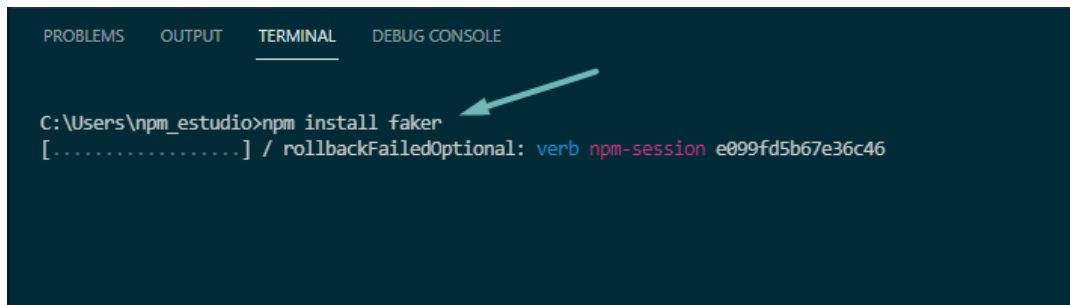


Pero, cada paquete tiene su forma particular, por lo tanto, siempre es recomendable leer su documentación para saber cómo hacer uso de él.

Utilizaremos un paquete de prueba, para que puedas familiarizarte con la forma de llamarlos y utilizarlos dentro de tu programa.

El paquete en cuestión se llama **faker.js**, y como su nombre lo indica, éste genera fake data, o datos falsos, para realizar pruebas en tus aplicaciones. Además, seguiremos utilizando el paquete **nodemon** para evitar tener que reiniciar manualmente nuestro programa.

Ejecutamos el comando **npm install faker**.



```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

C:\Users\npm_estudio>npm install faker
[.....] / rollbackFailedOptional: verb npm-session e099fd5b67e36c46
```

Y como es un paquete nuevo que no conocemos, lo más recomendable es leer su documentación: <https://www.npmjs.com/package/faker>

Usage

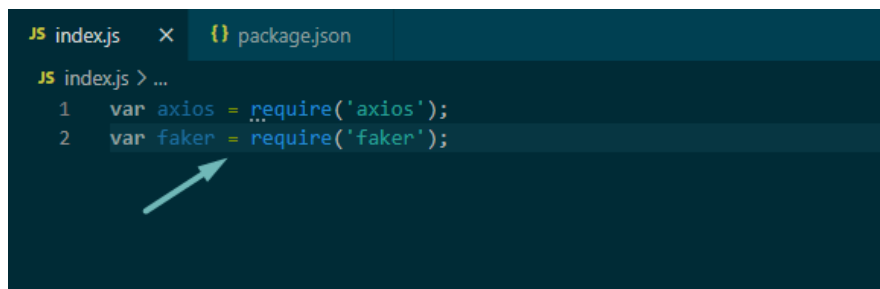
Browser

```
<script src = "faker.js" type = "text/javascript"></script>
<script>
  var randomName = faker.name.findName(); // Caitlyn Kerluke
  var randomEmail = faker.internet.email(); // Rusty@arne.info
  var randomCard = faker.helpers.createCard(); // random contact card contain:
</script>
```

Node.js

```
var faker = require('faker');
var randomName = faker.name.findName(); // Rowan Nikolaus
var randomEmail = faker.internet.email(); // Cassandra.Haley@erich.biz
var randomCard = faker.helpers.createCard(); // random contact card containing many
```

Usamos la forma de importación, tal como se muestra en la documentación de este paquete en nuestro código. En este punto, ésta también nos muestra cómo usar el paquete para empezar a obtener data falsa.



```
JS index.js  X  {} package.json
JS index.js > ...
1  var axios = require('axios');
2  var faker = require('faker');
```

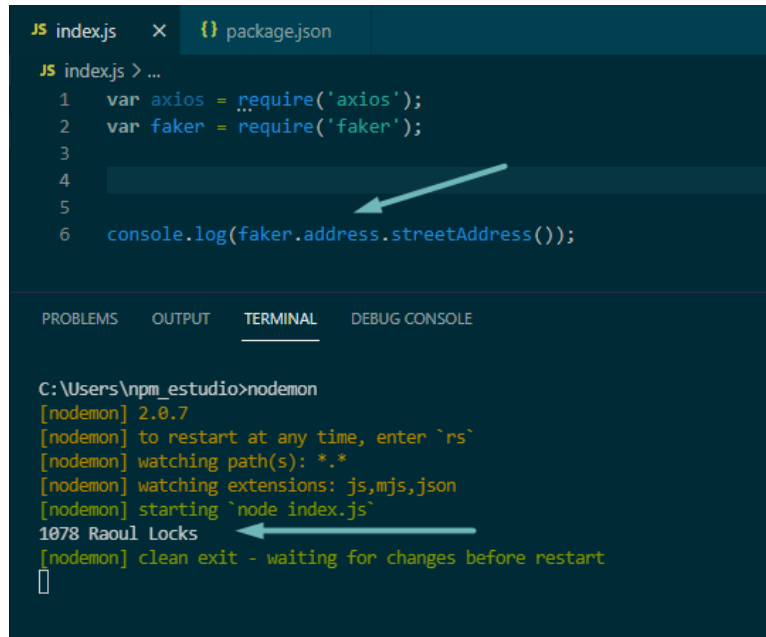
Si revisamos más abajo en la documentación, tenemos una larga lista de todos los posibles métodos que podemos utilizar para generar data de prueba.



API Methods

- address
 - zipCode
 - zipCodeByState
 - city
 - cityPrefix
 - citySuffix
 - cityName
 - streetName
 - streetAddress
 - streetSuffix
 - streetPrefix
 - secondaryAddress
 - county
 - country
 - countryCode
 - state
 - stateAbbr
 - latitude
 - longitude
 - direction
 - cardinalDirection
 - ordinalDirection
 - nearbyGPSCoordinate
 - timeZone
- animal
 - dog
 - cat
 - snake
 - bear

Hagamos uso del siguiente código, y veamos el resultado en consola.



The image shows a VS Code editor with two tabs: `index.js` and `package.json`. The `index.js` file contains the following code:

```
1 var axios = require('axios');
2 var faker = require('faker');
3
4
5
6 console.log(faker.address.streetAddress());
```

Below the editor, the **TERMINAL** tab is active, showing the output of running `nodemon` in the directory `C:\Users\npm_estudio`:

```
C:\Users\npm_estudio>nodemon
[nodemon] 2.0.7
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index.js`
1078 Raoul Locks
[nodemon] clean exit - waiting for changes before restart
```

Two green arrows point from the text in the terminal to the corresponding lines in the code: one from `1078 Raoul Locks` to line 4, and another from `[nodemon] starting `node index.js`` to line 6.

Ahora, busquemos en la documentación cómo podemos configurar el lenguaje del paquete para obtener datos en español.

Localization

As of version v2.0.0 faker.js has support for multiple localities.

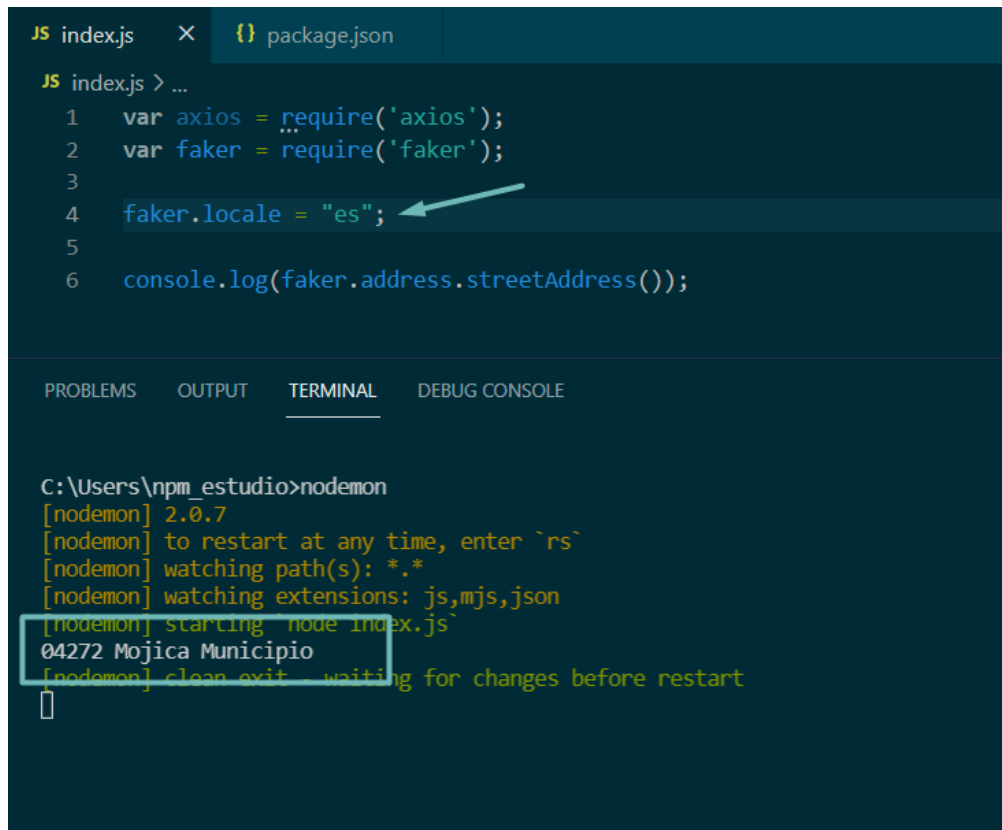
The default language locale is set to English.

Setting a new locale is simple:

```
// sets locale to de
faker.locale = "de";
```

- az
- ar
- cz
- de
- de_AT
- de_CH
- en
- en_AU
- en_AU_ocker
- en_BORK
- en_CA
- en_GB
- en_IE
- en_IND
- en_US
- en_ZA
- es
- es_MX

Ejecutamos el siguiente código.



```
JS index.js X {} package.json

JS index.js > ...
1 var axios = require('axios');
2 var faker = require('faker');
3
4 faker.locale = "es";
5
6 console.log(faker.address.streetAddress());

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

C:\Users\npm_estudio>nodemon
[nodemon] 2.0.7
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting node index.js
04272 Mojica Municipio
[nodemon] clean exit - waiting for changes before restart
█
```

Bien. Acabas de hacer uso del primer paquete en tu programa de Node.js, esta es una herramienta muy poderosa y fundamental, que permite trabajar de manera más eficiente; incluso, puedes usar **frameworks** completos como **express**, los cuales ya son parte del estándar de la industria. Pronto construirás una aplicación web haciendo uso de **express**.

¿QUÉ ES EXPRESS?

Es un **framework** web transigente, escrito en JavaScript, y alojado dentro del entorno de ejecución **NodeJS**. **Permite que se creen aplicaciones de manera rápida y sencilla**. Se trata de un marco de aplicación web para back-end.

Proporciona un amplio conjunto de funciones para crear aplicaciones web (de una sola página, de varias páginas, e híbridas). Y con él puedes estructurar una aplicación web que sea capaz de manejar múltiples solicitudes **HTTP** en una determinada URL.

La flexibilidad es visible en numerosos componentes accesibles en un administrador de paquetes. Dichos componentes perseveran automáticamente en Express.js.

La razón por la que Express es el marco web más popular, es que facilita el desarrollo de aplicaciones web, sitios web, y **API**. También ofrece una colección subyacente de topografías.

Con Express.js, podrás perfeccionar diferentes aspectos de la aplicación web. Puede determinar configuraciones, como la ubicación de las plantillas que se usarán para la respuesta, o el puerto para establecer una conexión.

¿Por qué usar Express?

Permite que se desarrollen funcionalidades; tales como el enrutamiento, que se utiliza para almacenar información sobre redes que se encuentren conectadas, y ayuda a que el tráfico de información se transmita de una forma sencilla.

Características principales de **Express.js**:

- Un marco de trabajo de código abierto.
- Se centra en el alto rendimiento.
- Cobertura de prueba súper alta.
- Admite múltiples motores de plantillas (lo que simplifica la generación de HTML).
- Permite escribir respuestas a URL específicas.
- Un mecanismo simple que localiza errores en las aplicaciones rápidamente.

En resumen, si trabajamos con Node.js, encontraremos un excelente aliado en Express.

En la **API** de **Express**, encontraremos una gran variedad de métodos útiles para nuestros proyectos, ya que son de uso cotidiano, y de fácil implementación para los desarrolladores.

Casos de uso de Express, en conjunto con **Node.js**:

- Manejo de sesión.
- Parseo de datos.
- Trabajo con diferentes engines de templates.
- Manejo de rutas.

Express v/s Node

Las diferencias clave entre **Express.js** y **Node.js** son:

- **Categorización**
 - Node.js se incluye principalmente en la categoría Frameworks (Full Stack).
 - Express.js se clasifica en la categoría Microframeworks (Backend).
- **Bloque de construcción**



- Node.js se basa en el motor **V8** de Google.
- Express.js se basa en **Node.js**.
- **Características**
 - Node.js tiene menos funciones que Express.js.
 - Express.js incorpora más funciones, ya que es una adición a las funcionalidades de Node.js.
- **Uso**
 - Node.js se utiliza para crear aplicaciones controladas por eventos **E/S** del lado del servidor.
 - Express.js utiliza enfoques de Node.js para crear aplicaciones web y **API**.
- **Dependencia**
 - Node.js se puede utilizar independientemente de Express.js.
 - Express.js requiere Node.js.
- **Tiempo**
 - Node.js demanda más tiempo, debido a tareas multifacéticas que requieren más líneas de código y, por lo tanto, dedicación.
 - Express.js requiere menos tiempo, pues se puede escribir en menos líneas y en pocos minutos.
- **Modelo de vista**
 - Node.js no es compatible con el modelo de vista.
 - Express.js es compatible con el modelo de vista.
- **Lenguaje de programación**
 - Node.js está escrito en C, C++, JavaScript.
 - Express.js está escrito en JavaScript.
- **Controladores**
 - En Node.js, no se proporcionan controladores.
 - En Express.js, se proporcionan controladores.
- **Enrutamiento**
 - En Node.js, no se proporciona el enrutamiento.
 - En Express.js, se proporciona enrutamiento.
- **Middleware**
 - Node.js no usa la provisión.
 - Express.js usa middleware para organizar funciones de manera sistemática.



Ventajas

Express proporciona mecanismos para:

- Escritura de manejadores de peticiones con diferentes verbos **HTTP** en diferentes caminos URL (rutas).
- Integración con motores de renderización de "vistas", para generar respuestas mediante la introducción de datos en plantillas.
- Establecer ajustes de aplicaciones web, como qué puerto usar para conectar, y la localización de las plantillas que se utilizan para renderizar la respuesta.
- Añadir procesamiento de peticiones "middleware" adicional, en cualquier punto dentro de la tubería de manejo de la petición.
- Análisis de las cookies.
- Determinación de las cabeceras apropiadas para las respuestas.
- Cabeceras automáticas en las respuestas.
- Enrutado y mejor organización del código.

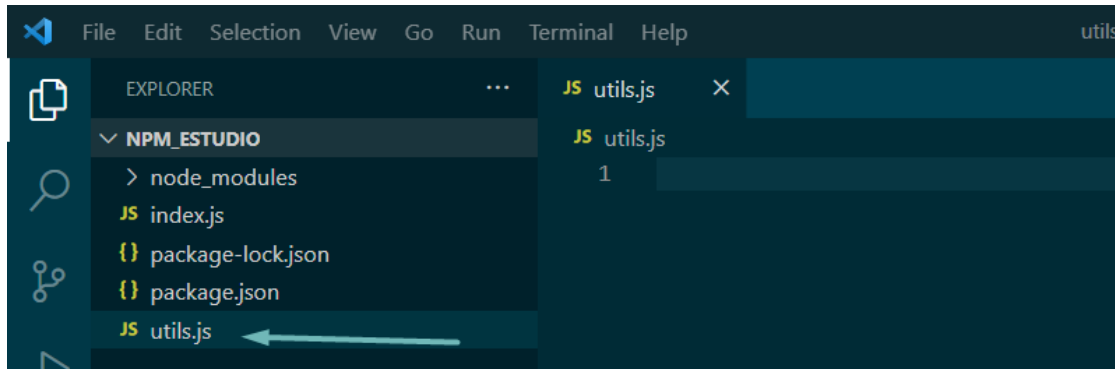
DIVIDIENDO NUESTRO CÓDIGO

Una práctica muy común y, sobre todo, recomendada, es dividir tu código en distintos archivos. Esta acción trae muchos beneficios consigo.

Tu programa fácilmente puede crecer, tanto en complejidad, como en la cantidad de tareas que necesita realizar, acumulando gran cantidad de líneas de código, por lo que dividirlo permite facilitar la lectura de éste. Por otra parte, también hace más fácil encontrar errores, que pueden ser mucho más complicados de rastrear cuando tienes tu programa en una sola pieza. Además, al separar tu código, logras crear uno modular, que permite aislar los componentes que conforman tu programa, haciéndolo más efectivo.

Escribiremos dos funciones utilitarias: una que nos permitirá sumar dos enteros, y otra que nos permitirá multiplicar dos enteros.

Primero crearemos un nuevo archivo llamado **utils.js**.



Luego, escribiremos nuestras funciones.

```
JS utils.js > multiplicaDosEnteros
1 function sumaDosEnteros(enteroUno, enteroDos) {
2   |   return enteroUno + enteroDos
3 }
4
5 function multiplicaDosEnteros(enteroUno, enteroDos) {
6   |   return enteroUno * enteroDos
7 }
```

Para hacer uso de estas funciones en nuestro archivo principal, debemos indicarle donde encontrarlas; esto es muy similar a requerir los paquetes instalados con **npm**.

Esta es una tarea de dos pasos.

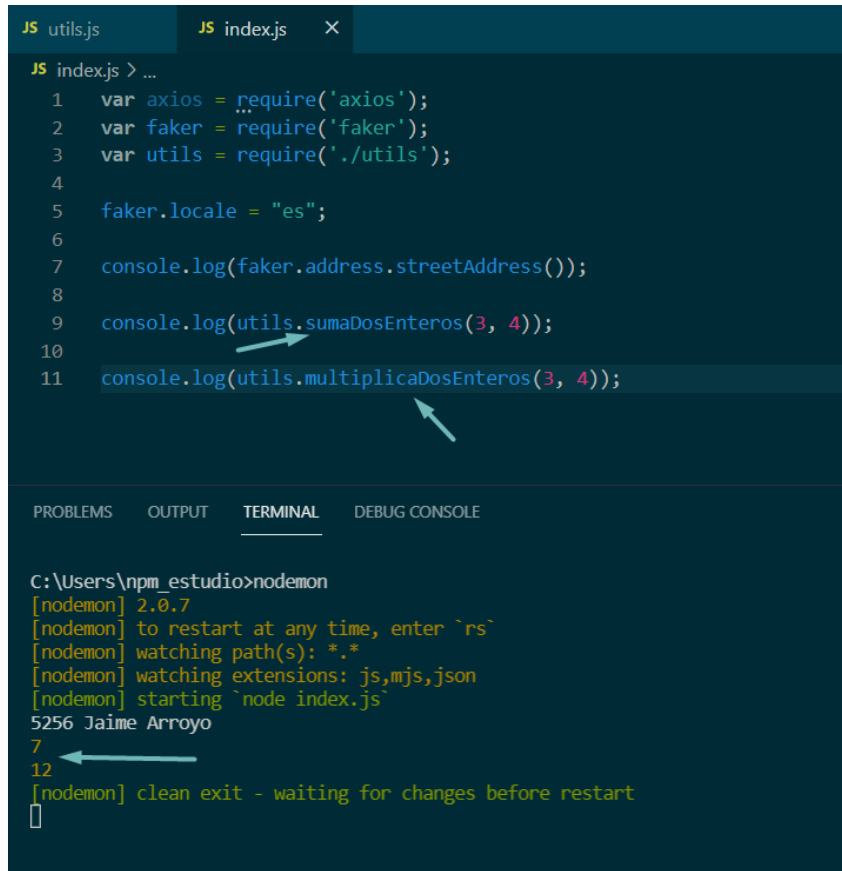
Primero, en nuestro archivo **utils**, debemos especificar que queremos exportar estas funciones. Para ello, usamos la sintaxis **"module.exports"**, e ingresamos el nombre de nuestras funciones como parte de un objeto, haciendo uso de pares llave valor. En este caso, hemos decidido darle el mismo nombre de la función a la llave, pero puedes ponerle el nombre que quieras, solo recuerda que ese será con el que deberás llamarlo en donde sea que vayas a requerirlo.

```
JS utils.js  X  JS index.js
JS utils.js > [?] <unknown>
1  function sumaDosEnteros(enteroUno, enteroDos) {
2    |   return enteroUno + enteroDos
3  }
4
5  function multiplicaDosEnteros(enteroUno, enteroDos) {
6    |   return enteroUno * enteroDos
7  }
8
9  module.exports = { sumaDosEnteros: sumaDosEnteros, multiplicaDosEnteros: multiplicaDosEnteros }
```

Y en nuestro archivo **index.js**, debemos indicarle que queremos importar el contenido exportado del archivo **utils**, de la misma manera en que requerimos paquetes instalados con **npm**. Es importante destacar que el formato **“./nombreDeArchivo”**, es para indicarle a **node** que el archivo a importar se encuentra en la misma ruta que el archivo que lo importa.

```
JS utils.js  JS index.js  X
JS index.js > ...
1  var axios = require('axios');
2  var faker = require('faker');
3  var utils = require('./utils'); ←
4
```

Una vez importado, podemos hacer uso de nuestras funciones en el archivo **index.js**.



```
JS utils.js JS index.js X
JS index.js > ...
1 var axios = require('axios');
2 var faker = require('faker');
3 var utils = require('./utils');
4
5 faker.locale = "es";
6
7 console.log(faker.address.streetAddress());
8
9 console.log(utils.sumaDosEnteros(3, 4));
10
11 console.log(utils.multiplicaDosEnteros(3, 4));

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

C:\Users\npm_estudio>nodemon
[nodemon] 2.0.7
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index.js`
5256 Jaime Arroyo
7
12
[nodemon] clean exit - waiting for changes before restart
```

El hecho de que tengamos estas funciones en nuestro archivo **utils.js**, nos permite utilizarlas en cualquier otro que tengamos, y nos evita tener que declarar la función cada vez que la usemos, optimizando así tu código.