3. Desarrollo de servicios con Node.js

Parte I. Introducción a Node.js

Programación de Servicios y Procesos

Nacho Iborra
Álvaro Pérez
Javier Carrasco

Tabla de contenido

1. Introducción

En esta unidad se aprenderá a desarrollar servicios web básicos usando el *framework* **Node.js** y **MongoDB**. Pero en esta primera parte de la unidad se verán algunos conceptos básicos sobre qué es Node.js, cómo instalarlo y usarlo, y qué otros recursos se necesitarán para implementar los servicios.

1.1. ¿Qué es Node.js?

Node.js es un entorno de ejecución para el lado del servidor web creado con el motor Javascript de Google Chrome, llamado V8. Piensa un poco en lo que acabas de leer… Hace unos años, Javascript era solo un lenguaje de programación del lado del cliente o *front-end*, que permitía definir páginas web dinámicas basadas en AJAX con *frameworks* como jQuery, o algunos años después, Angular. Pero con Node.js también se puede usar Javascript para desarrollar el servidor o el *back-end*, reemplazando algunos lenguajes de programación de servidor tradicionales, como PHP, ASP.NET o JSP. De hecho, algunas empresas importantes confían total o parcialmente en Node.js: Netflix, Paypal o Uber son algunos ejemplos.

Para hacer esto, el motor V8 se basa en un núcleo C++, que proporciona a Javascript algunas características completamente nuevas, como acceder al sistema de archivos local o conectarse a bases de datos, que son dos características esenciales para un lenguaje de programación del lado del servidor. Pero no nos vamos a centrar en estos detalles de bajo nivel. Solo necesitamos saber que podemos desarrollar aplicaciones *back-end* con Javascript usando Node.js, y que este entorno es multiplataforma y de código abierto, por lo que se puede usar en muchos sistemas operativos, y se pueden aprovechar módulos de terceros, e incluso editar estos módulos, para mejorar las capacidades de nuestras aplicaciones.

1.2. Características principales de Node.js

Entre todas las características de Node.js, se pueden destacar las siguientes:

- Node.js ofrece una API asincrónica basada en eventos. En otras palabras, el uso de métodos Node.js no bloquea la aplicación principal: se llama y devuelve un valor tan pronto como termine, mientras el hilo principal está en marcha. Además, se pueden definir algunos eventos que lanzan automáticamente algunas funciones cuando se emiten, por lo que no hay que preocuparse de cuándo llamar a estas funciones.
- Es realmente rápido (recuerda, el núcleo V8 está escrito en C++).
- Es de un hilo simple. Solo hay un hilo que se ocupa de todas las solicitudes, pero debido a la API asincrónica, no necesita otro hilo para aumentar el rendimiento. Por tanto, las aplicaciones de Node.js suelen necesitar menos recursos que otras aplicaciones de *back-end*.

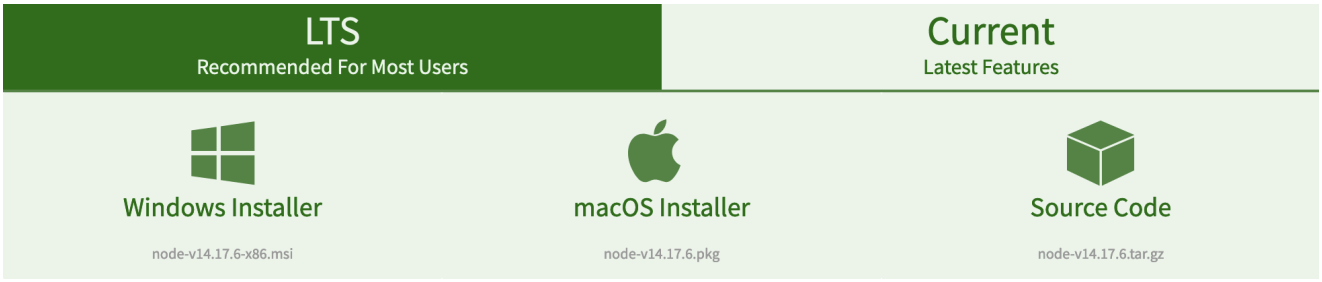
2. Descarga, instalación y prueba

Antes de comenzar con las primeras pruebas (básicas) con Node.js, es necesario descargarlo e instalarlo en el sistema. Observa cómo hacer esto.

2.1. Descarga e instalación

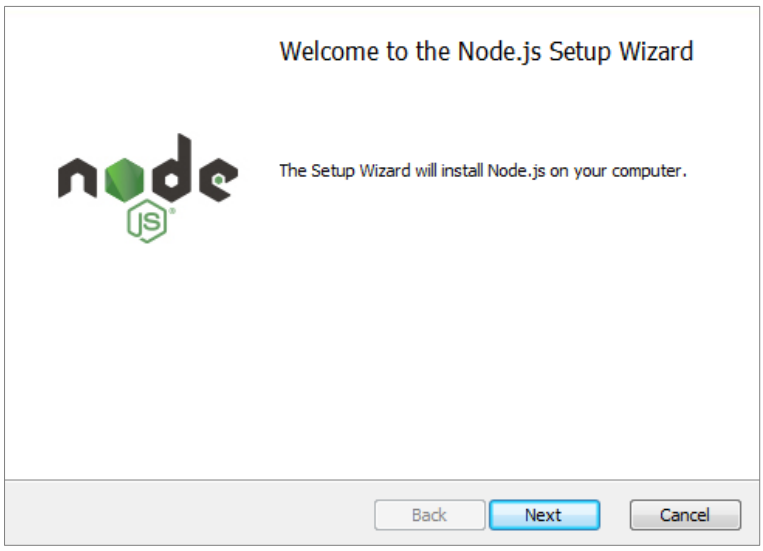
2.1.1. Windows y Mac OS X

Para descargar Node.js para Windows o Mac OS X, debes ir a su sitio web oficial (nodejs.org), y hacer clic en el enlace de descarga que podrás encontrar en la página principal. Puedes elegir entre dos opciones:



La versión LTS (enlace izquierdo) es la opción recomendada para la mayoría de los usuarios. No es la última versión, pero ofrece soporte a largo plazo (LTS). Pero si deseas probar algunas de las últimas funciones de Node.js, es posible que debas instalar la versión actual (enlace derecho). Para el propósito de esta unidad, se instalará la versión LTS, ya que no se van a utilizar las últimas funciones que podrían haber sido incluidas. Por tanto, haz clic en el enlace de la izquierda y descarga el instalador.

Este instalador es un archivo *.msi* para sistemas Windows o un archivo *.pkg* para sistemas Mac OS X. Solo necesitarás seguir los pasos del asistente para completar la instalación.



2.1.2. Linux

En cuanto a los usuarios de Linux, la mejor opción para instalar Node.js es a través de un repositorio. Si vas al sitio web oficial, obtendrás un archivo *.tar.xz* que necesitarás descomprimir y procesar, pero desde un repositorio, solo necesitas escribir algunos comandos (como super usuario o *root*):

Este comando solo es necesario si no tienes el comando *curl* ya instalado:

```
sudo apt-get install curl
```

Si ya lo tienes, puedes descargar e instalar Node.js con estos dos comandos:

```
curl -fsSL https://deb.nodesource.com/setup_16.x | bash -
apt-get install -y nodejs
```

Una vez que hayas terminado, podrás utilizar el comando *node* (para ejecutar las aplicaciones) o el comando *npm* (para descargar módulos adicionales, como se verá más adelante).

2.1.3. Actualización de versiones anteriores

Si deseas actualizar una versión anterior de Node.js, solo necesitas descargar e instalar la nueva versión desde el sitio web oficial si utilizas Windows o Mac OS X, o usa el mismo conjunto de comandos en Linux, reemplazando la versión número con el apropiado.

También se puede usar un administrador de versiones llamado *nvm*, que te ayudará a descargar la versión deseada y elegir entre todas las versiones instaladas en el sistema. Esta última opción no se explicará en esta unidad.

2.2. La primera aplicación Node.js

Una vez instalado Node.js, se puede comprobar que todo está bien escribiendo algunos comandos básicos en la línea de comandos. Así que abre una terminal y escribe este comando:

```
node -v
```

También se puede usar *node --versión*. En ambos casos se obtendrá la versión de Node.js que esté instalada, por ejemplo:

```
v14.17.6
```

> **IMPORTANTE**: asegúrate de que este comando le muestre el resultado apropiado. De lo contrario, es posible que no puedas implementar los siguientes proyectos de esta unidad.

2.2.1. “Hola mundo” en Node.js

A continuación, se creará la primera aplicación Node. Crea un archivo de texto llamado “helloworld.js” en su directorio preferido (inicio, Documentos... donde sea), con este contenido. Puedes utilizar el bloc de notas o cualquier otro editor de texto básico:

```
console.log("Hello world");
```

Después de guardar los cambios, puedes ejecutar la aplicación desde la terminal yendo a la misma carpeta del archivo y escribiendo este comando en la terminal:

```
node helloworld.js
```

La salida en este caso será “Hola mundo”.

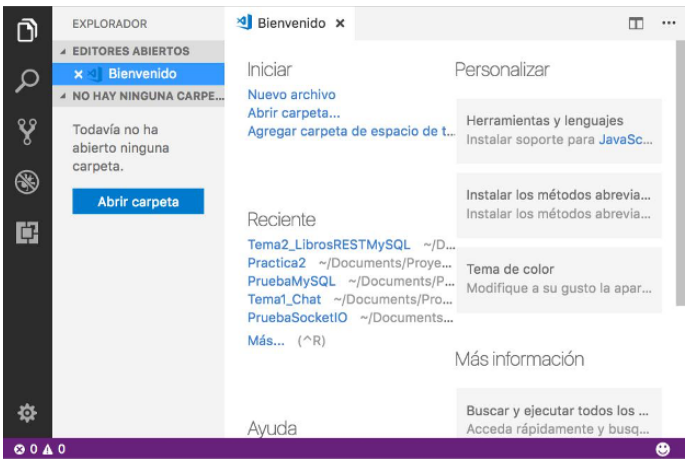
3. Elegir un IDE para aplicaciones Node

Además de Node.js, es necesrio instalar algunos otros recursos externos para implementar las aplicaciones para esta unidad. En particular, será necesario tener un IDE apropiado para desarrollar proyectos Node.js y un sistema de administración de base de datos para almacenar los datos de la aplicación.

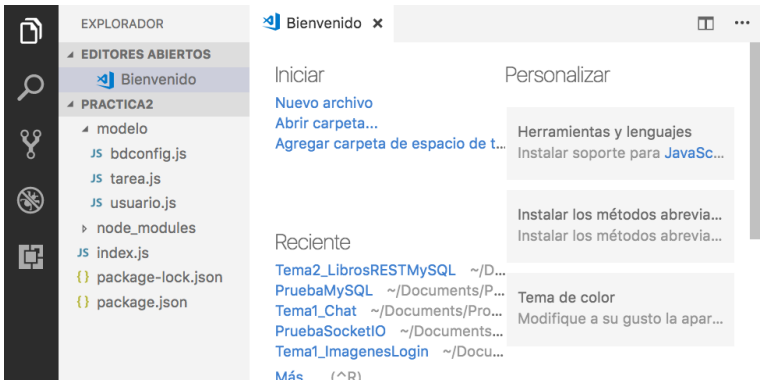
Para crear e implementar una aplicación Node.js se pueden elegir entre algunos IDE interesantes, como Atom, Sublime Text, Visual Studio Code, etc. e incluso se puede usar NetBeans o Eclipse como editores. Entre todas estas opciones, se ha elegido **Visual Studio Code**, ya que es un editor realmente sencillo con un terminal integrado que permite ejecutar aplicaciones Node fácilmente.

3.1. Descarga e instalación de VS Code

Para descargar e instalar Visual Studio Code, simplemente ve a su página web oficial, y elije el instalador según tu sistema operativo. Después de instalarlo, puedes elegirlo de la lista de aplicaciones. Para los usuarios de Linux, si no puedes encontrarlo en la lista de aplicaciones después de ejecutar el instalador *.deb* o *.rpm*, simplemente escribe *code* en un terminal y se iniciará el IDE.



Para abrir un proyecto, simplemente utiliza al menú ***File > Open…*** y elije la carpeta del proyecto. Luego, todo el contenido del proyecto se mostrará en la parte izquierda.



3.2. Integrar Node.js con Visual Studio Code

Visual Studio Code ofrece una integración muy interesante con Node.js, para que se pueda editar, ejecutar e incluso depurar aplicaciones Node desde este IDE. Observa cómo trabajar con proyectos de Node.

3.2.1. Configurar el espacio de trabajo

Cada proyecto de Node está autocontenido en su propia carpeta, por lo que solo es necesario abrir esta carpeta desde VS Code. Una vez que se abre, se pueden añadir carpetas y archivos desde la barra izquierda.

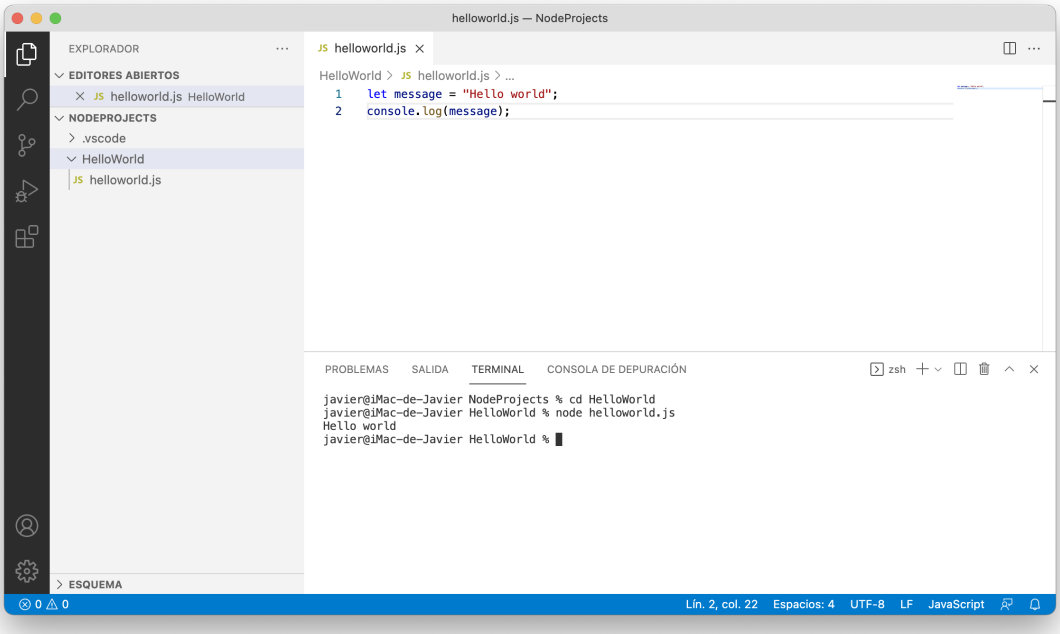
Para organizar todos los proyectos de esta unidad, se creará una carpeta general llamada *"NodeProjects"*. Luego, cada proyecto o ejercicio de prueba que se implemente se colocará dentro de esta carpeta.

Empieza con un proyecto realmente simple. Crea una carpeta de proyecto *"HelloWorld"* con un archivo llamado *"helloworld.js"* dentro y escribe este código en el archivo fuente:

Crea nuevos archivos y carpetas en el proyecto actual.

3.2.2. Ejecutar un archivo de Node desde VS Code

Si quieres ejecutar el programa anterior, podrás abrir una terminal e ir a la carpeta del proyecto para ejecutar el comando *node*, o simplemente abre el terminal integrado que viene con Visual Studio Code. Para hacer esto, ve al menú *View* > *Terminal*. Se mostrará en la parte inferior de la ventana, y el *prompt* se ubicará automáticamente en la carpeta del proyecto, así que simplemente escribe *node helloworld.js* para ejecutar el ejemplo anterior.



4.Bibliotecas o módulos

Node.js es un *framework* muy modular, es decir, su funcionalidad está completamente dividida en múltiples módulos o bibliotecas (a lo largo de esta unidad se utilizarán ambos términos por igual). De esta forma, solo se necesita añadir al proyecto los módulos que se necesiten.

Dentro del núcleo de Node puedes encontrar varios módulos relevantes, como:

- **http** y **https**, para que la aplicación actúe como servidor web.
- **fs** para acceder al sistema de archivos y leer/crear/escribir/etc archivos y directorios.
- **utils**, con algunas funciones útiles, como el formato de texto.
- … etc. Si quieres ver una lista completa de los módulos disponibles dentro del núcleo de Node, puedes verlo en https://nodejs.org/api/. Es una API de cada módulo integrado de Node, con documentación sobre cómo usarlos y sus propiedades y métodos públicos. Por ejemplo, aquí puede ver la descripción del método *readdirSync*, del módulo *fs*:

fs.readdirSync(path[, options])

▸ History

- `path` `<string>` | `<Buffer>` | `<URL>`
- `options` `<string>` | `<Object>`
  - `encoding` `<string>` **Default:** `'utf8'`
  - `withFileTypes` `<boolean>` **Default:** `false`
- Returns: `<string[]>` | `<Buffer[]>` | `<fs.Dirent[]>`

Reads the contents of the directory.

See the POSIX `readdir(3)` documentation for more details.

The optional `options` argument can be a string specifying an encoding, or an object with an `encoding` property specifying the character encoding to use for the filenames returned. If the `encoding` is set to `'buffer'`, the filenames returned will be passed as `<Buffer>` objects.

Además, hay una gran cantidad de módulos de terceros que se pueden descargar e instalar en las aplicaciones, como *mongoose* para utilizar bases de datos MongoDB, o *express* para incorporar el *framework Express.js* a un servidor web. Se instalarán algunos de estos módulos más adelante.

4.1. Uso de *"require"* para incluir módulos

Si deseas utilizar cualquier módulo en tu aplicación (ya sean tus propios módulos, módulos del núcleo de Node o módulos de terceros), debes incluirlo en tu código a través de la función *require*. Necesita un solo parámetro: el nombre del módulo que se va a incluir. Observa cómo utilizarlo.

4.1.1. Un ejemplo simple: usar *"fs"* para listar archivos y directorios

Se creará otra carpeta en la carpeta *"NodeProjects"*, llamada *"FSTest"*. Crea un archivo fuente llamado *"file_list.js"* dentro de este proyecto desde Visual Studio Code. En este ejemplo, se imprimirá cada archivo y carpeta del directorio de inicio, usando la biblioteca *fs*. El código final podría quedar más o menos así:

```
const path = '/Users/javier';
const fs = require('fs');
fs.readdirSync(path).forEach(file => {console.log(file);});
```

Si ejecutas la aplicación (recuerda que puedes usar el terminal integrado de VS Code), obtendrás la lista de archivos/carpetas dentro del directorio de inicio. Recuerda cambiar el valor de la constante *path* a una carpeta apropiada en tu sistema:

```
javier@iMac-de-Javier FSTest % node file_list.js
.Trash
.android
.config
…
```

Ten en cuenta que se utilizan constantes (*const*) en lugar de variables para almacenar el contenido del módulo requerido. Esto se debe a que no se necesita cambiar el valor o las propiedades del módulo incluido, por lo que no es necesario que se almacene en una variable.

Ejercicio 1

Crea una carpeta llamada *"Exercise_HelloUser"* en tu carpeta *"NodeProjects"*. Añade un archivo fuente llamado *"greeting.js"* y echa un vistazo al módulo llamado *os* de la API del núcleo de Node. Para ser más precisos, presta atención al método *userInfo*. Úsala para saludar al usuario del inicio de sesión que entró al sistema. Por ejemplo, si el inicio de sesión del usuario es "javier", el programa debería decir *"Hello javier"*.

AYUDA: El método *userInfo* devuelve un objeto con algunas propiedades. Para obtener el nombre de usuario, deberás obtener propiedad *username*.

AYUDA: El método *console.log* permite especificar un número variable de parámetros que se unen automáticamente mediante espacios en blanco, por lo que puedes unir texto estático y variable de diferentes formas, como:

```
console.log("Hello " + name);
console.log("Hello", name);
```

4.2. Usar *"require"* para incluir tus propios módulos

También se puede (se debe) dividir el código de la aplicación en varios archivos fuente, e incluir algunos de ellos en otros, usando la misma función *require*. En este caso, se deberá especificar la ruta relativa al archivo fuente que se incluirá. Por ejemplo, si quieres incluir un archivo llamado *"my_module.js"* que está dentro de una subcarpeta llamada *"myFolder"*, se podría incluir así:

```
const myModule = require('./myFolder/my_module.js');
```

Con respecto a los archivos fuente de Javascript (archivos .js), no es necesario especificar la extensión del archivo, por lo que la línea anterior podría escribirse de esta manera:

```
const myModule = require('./myFolder/my_module');
```

4.2.1. Exportar contenido de un archivo fuente

El contenido de los archivos fuente requeridos debe tener una estructura determinada para poder compartir todo (o parte) de su código. Si tienes un módulo básico dentro del archivo *"my_module.js"* mencionado anteriormente, con estas funciones básicas:

```
function add(num1, num2) {
    return num1 + num2;
}

function substract(num1, num2) {
    return num1 – num2;
}
```

No se podrá acceder a estos métodos si solo se necesita el módulo. Para acceder a estos métodos, será necesario exportarlos desde el archivo fuente del módulo, a través del objeto ***module.exports***. Este objeto permite definir qué partes del código fuente del módulo son visibles y cuáles no. Por ejemplo, con respecto al ejemplo anterior, se pueden definir dos propiedades dentro del objeto *module.exports* que apunte a las funciones correspondientes que se quieren exportar:

```
function add(num1, num2) {
    return num1 + num2;
}

function substract(num1, num2) {
    return num1 – num2;
}

module.exports = {
    add: add,
    substract: substract
};
```

De esta manera, se puede requerir este módulo en un archivo fuente principal en la carpeta raíz del proyecto (por ejemplo) y usar las funciones:

```
const myModule = require('./myFolder/my_module');
console.log(myModule.add(3, 2));
```

4.2.2. Uso de funciones flecha

Javascript también ha incluido expresiones lambda (también conocidas como “funciones de flecha”) de EcmaScript2015 (ES6). Es una notación alternativa para declarar funciones, muy similar a la que se usa en Java (aunque la flecha es => en lugar de Java ->). Las funciones del ejemplo anterior podrían definirse de la siguiente manera usando funciones de flecha:

```
let add = (num1, num2) => {
    return num1 + num2;
};

let substract = (num1, num2) => {
    return num1 – num2;
}
```

Este código puede ser aún más simple si la función solo devuelve un valor. En este caso, se pueden omitir las llaves y el *return*, y se conseguirá esto:

```
let add = (num1, num2) => num1 + num2;
let substract = (num1, num2) => num1 – num2;
```

Esta notación es válida a menos que se necesite acceder a algunas propiedades determinadas, como *this* o elementos *arguments*. Por lo general, se utilizará así en esta unidad.

5. Uso de npm

npm (*Node Package Manager*) es un administrador de paquetes Javascript que se instala al instalar Node.js. Puedes comprobarlo escribiendo el siguiente comando en una terminal:

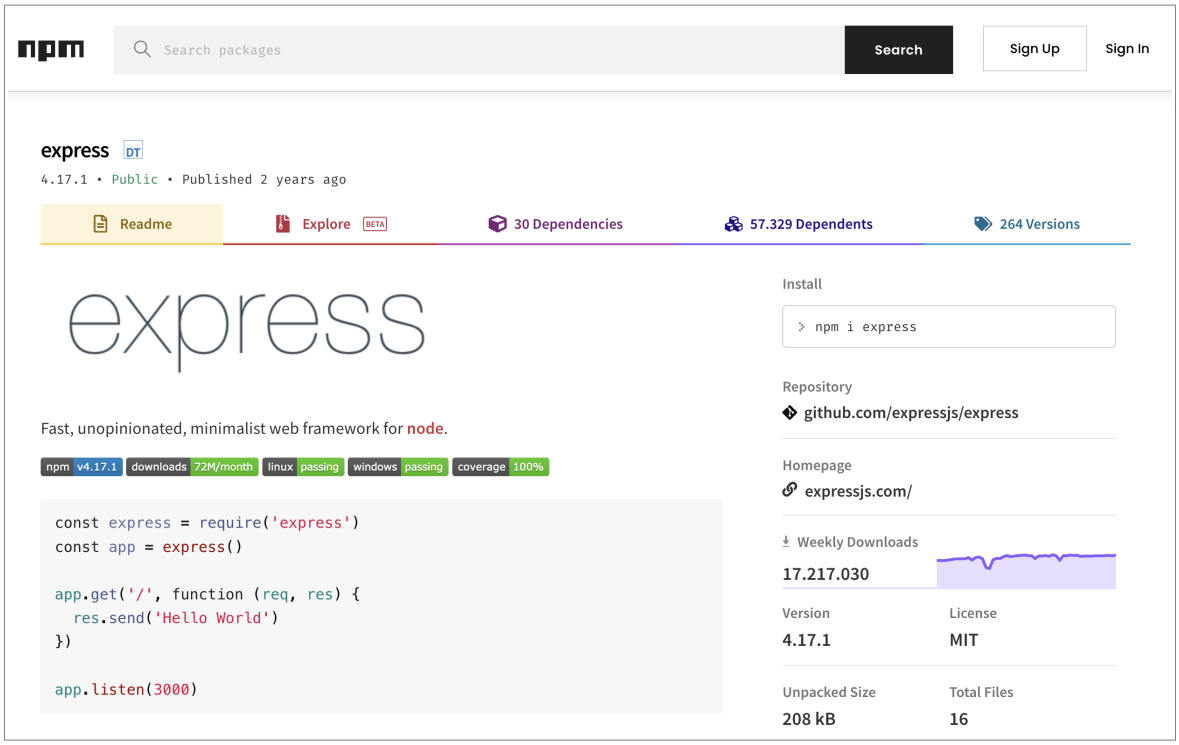
```
npm –v
```

aunque también puedes usar *npm --version*. De esta forma, se obtendrá la versión actual de *npm*, como 6.14.15 o algo similar.

En un principio, *npm* fue desarrollado para instalar algunos módulos en aplicaciones Node, pero se ha vuelto tan popular que hoy en día es mucho más que eso, y también se puede utilizar para instalar otros módulos en aplicaciones cliente o servidor que no estén relacionadas con Node, como jQuery.

Hoy en día, *npm* es (uno de) el mayor ecosistema de bibliotecas de código abierto del mundo. Debido a la gran comunidad de desarrolladores detrás del ecosistema *npm*, te puedes centrar en las necesidades específicas de tu aplicación, sin preocuparte por toda la infraestructura que necesitarías implementar de otra manera.

Puedes consultar todos los módulos disponibles para *npm* en su repositorio oficial, npmjs.com. En este sitio podrás buscar cualquier módulo en particular, consultar sus estadísticas (descargas por día, semana y mes, ediciones pendientes, documentación, etc), y decidir si se ajusta a tus necesidades o no. Por ejemplo, con respecto al módulo *express*, puedes obtener una página web similar a esta:



Se puede usar *npm* para instalar algunos módulos específicos en un proyecto Node específico (la opción más habitual), o para instalar algunos módulos globalmente en el sistema. Observa cómo hacer ambas cosas.

5.1. Instalar módulos en un proyecto local

En esta sección aprenderás cómo instalar módulos en un proyecto. Para practicar con esta función, crea un proyecto/carpeta llamado *"NPMTest"* en tu espacio de trabajo.

5.1.1. Fichero "package.json"

La configuración básica de los proyectos Node se almacena en un archivo JSON llamado *"package.json"*, en la carpeta raíz del proyecto. Este archivo se puede crear automáticamente desde la línea de comando, escribiendo una de estas dos opciones (desde la carpeta raíz del proyecto):

- *npm init --yes*, creará un archivo con valores predeterminados. Luego se puede editar el archivo y cambiar los valores si quieres. Aquí puedes ver una muestra de lo que se crea:

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

- *npm init*, iniciará un asistente para especificar cada atributo del archivo de configuración. Si solo presionar *Enter* para cada opción, se le asignará el valor predeterminado, por lo que no habría que preocuparse por ellos, a menos que quieras especificar algunos valores concretos.

```
Press ^C at any time to quit.
package name: (npmtest)
version: (1.0.0)
description:
git repository:
author:
```

Ten en cuenta que el nombre del proyecto se obtiene automáticamente del nombre de la carpeta raíz del proyecto, y el archivo fuente principal predeterminado de la aplicación es *"index.js"* (pero también puede cambiarse este valor).

Cuando termines este proceso, obtendrás el archivo *"package.json"* en la carpeta raíz del proyecto. Todos los módulos que se instalen se añadirán automáticamente a este archivo más adelante.

5.1.2. Añadir módulos al proyecto

Si quieres instalar un módulo de terceros desde el repositorio *npm* en tu proyecto, deberás llegar a la carpeta raíz del proyecto y escribir el siguiente comando:

```
npm install --save module_name
```

donde *module_name* es el nombre del módulo que se desea instalar. También se pueden especificar varios módulos, separados por espacios en blanco. También se puede instalar una versión específica del módulo añadiéndola al nombre del módulo, separada por '@'. Por ejemplo:

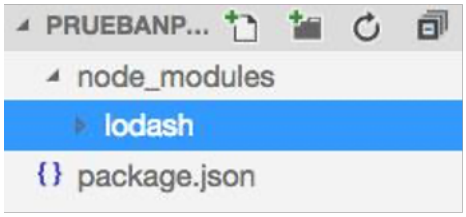
```
npm install --save module_name@1.1.0
```

Prueba con un módulo de terceros fácil y muy popular, con muchas funciones útiles para manejar cadenas, colecciones, etc. Este módulo se llama *"lodash"* (puedes leer sobre él aquí). Para instalarlo, simplemente escribe la siguiente línea desde la carpeta raíz del proyecto:

```
npm install --save lodash
```

Presta atención a algunos problemas antes de continuar:

- Tras ejecutar el comando anterior, tendrás una nueva carpeta llamada *"node_modules"* en tu proyecto Node:



- El nuevo módulo instalado se ha añadido al archivo *"package.json"*, dentro de una sección llamada *"dependencies"*:

```
{
  "name": "NPMTest",
  …
  "dependencies": {
    "lodash": "^4.17.21"
  }
}
```

- Después de instalar cualquier módulo de terceros, verás un nuevo archivo llamado *"package-lock.json"*. Es una copia de seguridad de todos los cambios o instalaciones de módulos, de modo que puedes volver a cualquier estado anterior cuando desees, y desinstalar los módulos a partir de ese momento.

Ahora que está instalado *lodash*, puedes usarlo en tu proyecto. Edita un archivo llamado *"index.js"* en la carpeta raíz del proyecto y escribe lo siguiente:

```
const lodash = require('lodash');
console.log(lodash.difference([1, 2, 3], [1]));
```

NOTA: si busca documentación en la red sobre el módulo *lodash*, encontrarás que la constante/variable para almacenarlo no se suele llamar *"lodash"*, sino un guión bajo ("_"). Por tanto, el ejemplo anterior se vería así:

```
const _ = require('lodash');
console.log(_.difference([1, 2, 3], [1]));
```

El programa en sí simplemente elimina un valor de una colección determinada. Si ejecutas el archivo, obtendrás este resultado:

```
node index.js
[ 2, 3 ]
```

Ejercicio 2

Crea una carpeta llamada ***"Exercise_JoinList"*** en tu espacio de trabajo. Define un archivo *"package.json"* ejecutando el comando *npm init*, con valores predeterminados para todos los atributos. Después, instala el módulo *"lodash"* como se explicó anteriormente, y utilízalo para crear un programa en el archivo "index.js" que una los nombres en una colección determinada, separados por comas. Por ejemplo, si utilizas el array ["Arturo", "Nacho", "Javier"], la salida debería ser:

```
Arturo,Nacho,Javier
```

Puedes consultar la documentación del módulo *lodash* aquí. Echa un vistazo al método *join* y cómo usarlo para obtener el resultado deseado.

5.1.2.1. Añadir módulos a "package.json" manualmente

También puedes añadir módulos manualmente en el archivo *"package.json"*. Por ejemplo, puedes añadir el módulo *"express"* de esta manera:

```
{
  …
  "dependencies": {
    "lodash": "^4.17.4",
    "express": "*"
  }
}
```

Después, para instalar todos los módulos dentro de este archivo que faltan en la carpeta *"node_modules"*, simplemente escribe el siguiente comando (desde la carpeta raíz del proyecto):

```
npm install
```

De hecho, esta es una estrategia común cuando se comparte un proyecto: se elimina el paquete *"node_modules"* y se comparte todo lo demás. Después, siempre que quieras ejecutar o reconstruir la aplicación, escribirás este comando para instalar todos los módulos que falten del archivo *"package.json"*.

5.1.3. Desinstalar un módulo

Si quieres desinstalar un módulo de un proyecto, simplemente escribe este comando desde la carpeta raíz del proyecto:

```
npm uninstall --save nombre_módulo
```

El módulo también se eliminará automáticamente del archivo *"package.json"*.

5.2. Instalación de módulos a nivel global

Existen algunos tipos de módulos que deben instalarse globalmente. En general, son módulos que se pueden ejecutar desde una terminal, como *Grunt* (un administrador de tareas de Javascript), *JSHint* (un verificador de sintaxis de Javascript) o *Nodemon* (un monitor de Node.js).

Para instalar un módulo globalmente, escribe este comando desde la línea de comandos (no necesitas estar dentro de ninguna carpeta en particular, ni será necesario ningún archivo *"package.json"* esta vez):

```
npm install -g module_name
```

El parámetro *-g* hace se refiere a una instalación *"global"*.

> Debes tener en cuenta que no se puede añadir ningún módulo global a un archivo fuente a través de *require* en una aplicación concreta (para hacer esto, deberás instalarlo localmente en el proyecto).

Si quieres desinstalar un módulo global, entonces el comando es este:

```
npm uninstall -g module_name
```

5.3. Organizar los módulos en el código

Aunque no existe un estándar específico sobre cómo ordenar los módulos requeridos en las aplicaciones de Node, deberás seguir un patrón. Lo más habitual es poner primero todos los módulos *core* y de terceros, y luego, separados por una línea vacía, todos los módulos propios locales. Por ejemplo:

```
const fs = require('fs');
const _ = require('lodash');

const my_utils = require('./my_utils');
```