Server-side
Web Development

Unit 14. Node.js. Introduction

Fidel Oltra, Ricardo Sánchez

.sf-sub-indicator
.cart-menu .cart-icon-wr
-outer.transparent header#top
.sf-menu > li.current_page
.sf-menu > li.current-menu-
> ul > li > a:hover > .sf-sub
ul #search-btn a:hover span,
.sf-menu > li.current-menu-
hover .icon-salient-cart,.ascend
!important;color:#ffffff!impo
header#top nav>ul>li.butt
widget-area-toggle a i
transparent

IES Jaume II El Just
Tavernes de la Valldigna
Departament d'Informàtica
Curs 2023-24

Index

1 Node.js **2**

1.1 Main features . . . 2

1.2 Installation . . . 2

1.3 A Hello world app . . . 4

2 Node libraries **5**

2.1 Using Node libraries . . . 5

2.1.1 Guided practice . . . 7

2.1.2 Practice . . . 8

2.2 Creating our own libraries . . . 9

2.2.1 Including an entire folder of modules . . . 10

2.2.2 Sharing information between modules with Json . . . 12

3 Third party libraries **13**

3.1 Install a module locally . . . 13

3.1.1 Example . . . 14

3.2 Uninstall a module . . . 15

3.3 Managing versions . . . 15

3.4 Editing package.json . . . 16

3.5 Install a module globally . . . 16

3.6 Exercise . . . 17

1 Node.js

Node.js is a server-side runtime environment built using the Google Chrome JavaScript engine, called V8. Node adds to Javascript the ability to work with local files, access databases, handle client requests/responses and execute a lot of server-side tasks.

The main advantage of **node.js** is that, since it is based on JavaScript, which is a client language, we don't need to learn two languages to develop an entire application in a Client/Server environment.

1.1 Main features

Among the main features of node.js, we can highlight these ones:

- ***Asynchronism***. Node.js offers an asynchronous API, so the main program doesn't block when we call its methods and the application is waiting for a response.
- ***Event driven***. The application is able to collect a response when it happens, without blocking the program while waiting for that response.
- ***Fast execution and high efficiency*** because is implemented using C++
- ***Single processing*** A Node.js app runs in a single process, without creating a new thread for every request. When an operation interrupts another (for instance, when we launch a database query), instead of blocking the thread and wasting CPU cycles waiting, Node.js will resume the main operation when the response comes back.

  This allows Node.js to handle thousands of concurrent connections with a single server without introducing the burden of managing thread concurrency, which could be a significant source of bugs.
- ***Compatibility*** The client code is compatible with any browser.

1.2 Installation

We can install Node.js in different ways. The most common is by using `nvm` (Node Version Manager), a tool that allows to install Node automatically and manage different versions at the same time.

Installing nvm with curl:

```
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.5/install.sh |
↪  bash
```

With wget:

```
wget -qO- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.5/install.sh |
↪   bash
```

Close the terminal and open again. Nom we can try:

```
nvm ls-remote
```

to check the versions available. We will install the LTS latest. At this moment, it's v20.10.0. We can just ask for the latest version by doing

```
nvm install 20.10.0
```

If we have any trouble with nvm, there's an alternative:

```
apt-get update
```

```
sudo apt-get install -y ca-certificates curl gnupg

sudo mkdir -p /etc/apt/keyrings
```

```
curl -fsSL https://deb.nodesource.com/gpgkey/nodesource-repo.gpg.key | sudo
↪   gpg --dearmor -o /etc/apt/keyrings/nodesource.gpg
```

```
echo "deb [signed-by=/etc/apt/keyrings/nodesource.gpg]
↪   https://deb.nodesource.com/node_20.x nodistro main" | sudo tee
↪   /etc/apt/sources.list.d/nodesource.list
```

```
sudo apt-get update
sudo apt-get install nodejs -y
```

We can check if node is installed, and which version:

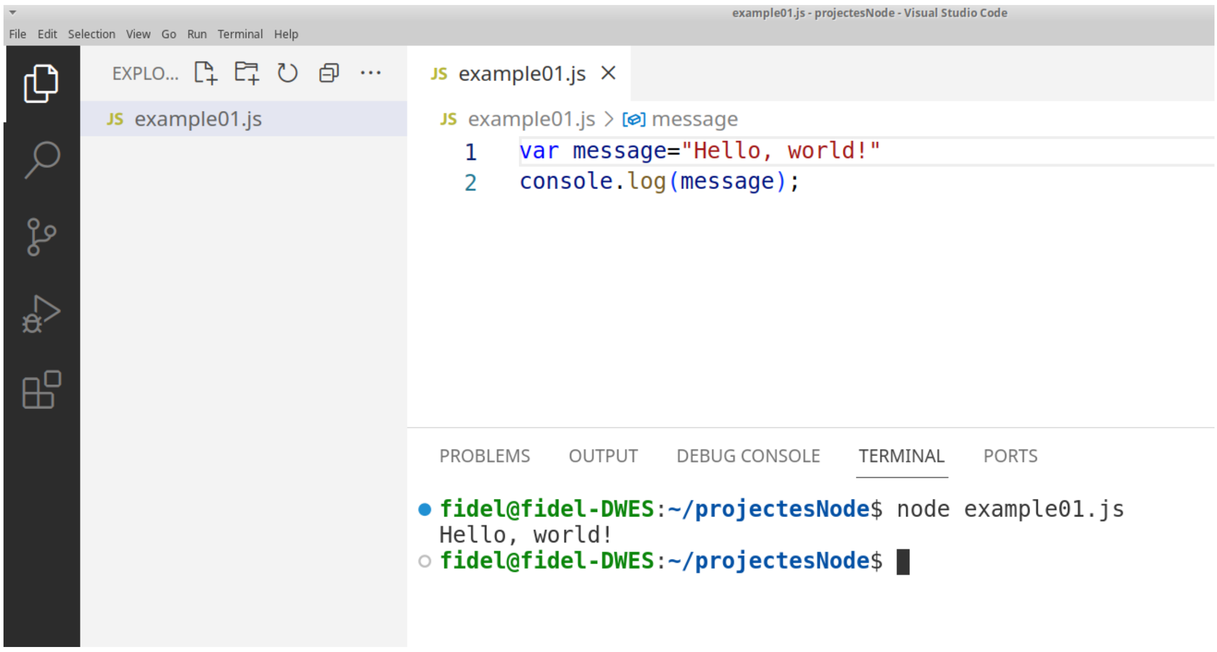
```
node -v
```

1.3 A Hello world app

First, create a folder to save our Node projects. For instance, To execute a node script, let's create a file `example01.js` with Visual Studio Code with this content:

```
var message="Hello, world!";
console.log(message);
```

Now we can open a new terminal in VSCode and run the script. We should get the message "Hello, world!".


example01.js - projectesNode - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLO...
example01.js
example01.js
example01.js > message
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS

```
var message="Hello, world!"
console.log(message);
```

```
fidel@fidel-DWES:~/projectesNode$ node example01.js
Hello, world!
fidel@fidel-DWES:~/projectesNode$
```

Figure 1: Our first Node script

At this point, you probably know enough the fundamentals of JavaScript. It's important to know about functions, arrow and anonymous functions, and promises. Just in case, here's a link to a guide.

JavaScript guide

2 Node libraries

Node.js is very modular. We can use a lot of different components named **libraries** or **packages**. Some libraries are incorporated to Node by default, while others (**third party libraries/packages**) might be installed when we need them. The user can also create his of her own libraries.

2.1 Using Node libraries

To use a library, we must require it.

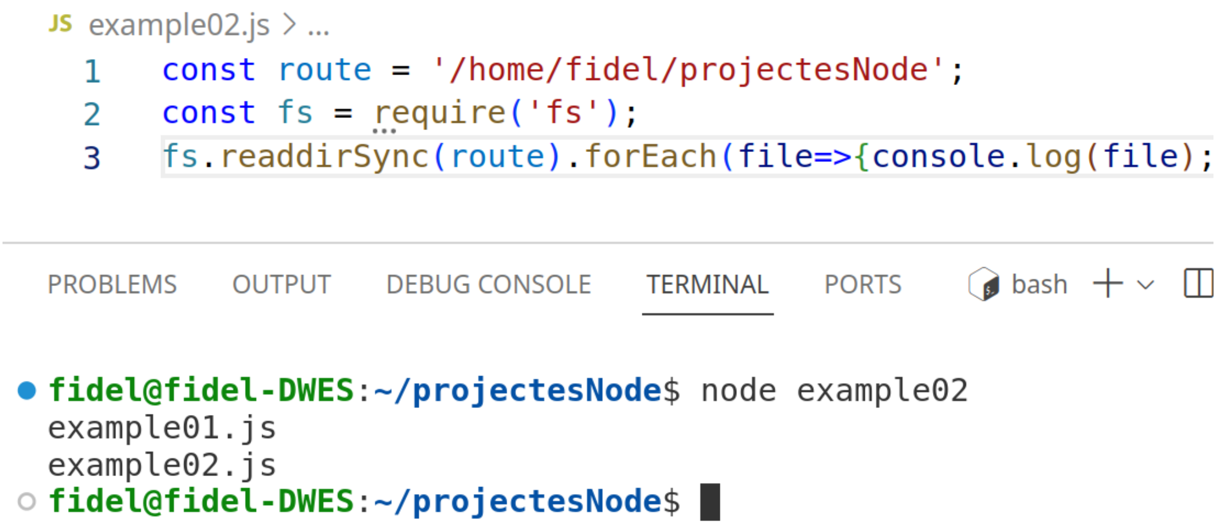
```
const fs = require('fs');
```

In the example, we are creating an object `fs` to work with the file system. We need the library `fs`, which is native to Node but we have to require it anyway.

Now we can use the file handler to list the files in a certain folder. For instance:

```
const route = '/home/fidel/projectesNode';
const fs = require('fs');
fs.readdirSync(route).forEach(file=>{console.log(file);});
```

If we run the script in a VSCode terminal, we will get:



```
const route = '/home/fidel/projectesNode';
const fs = require('fs');
fs.readdirSync(route).forEach(file=>{console.log(file);
```

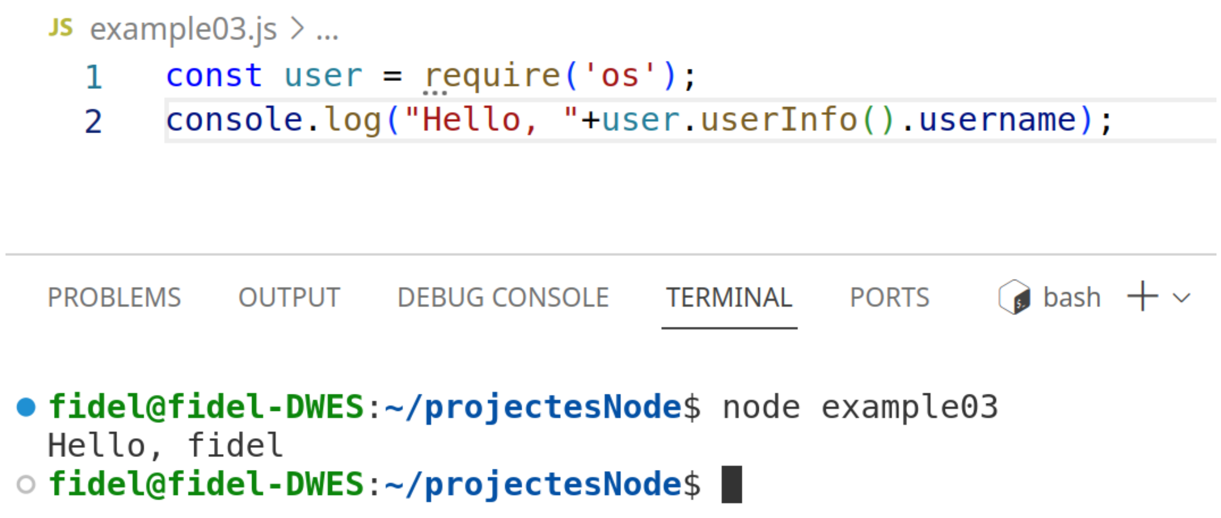
```
fidel@fidel-DWES:~/projectesNode$ node example02
example01.js
example02.js
fidel@fidel-DWES:~/projectesNode$ 
```

Figure 2: example02.js

> Sometimes you will probably find the notation `import fs from 'fs'`, which is equivalent to `const fs = require('fs')`

Another example:

```
const user = require('os');
console.log("Hello, "+user.userInfo().username);
```


example03.js > ...

```
const user = require('os');
console.log("Hello, "+user.userInfo().username);
```

```
fidel@fidel-DWES:~/projectesNode$ node example03
Hello, fidel
fidel@fidel-DWES:~/projectesNode$
```

Figure 3: example03.js

You can find info about most of Node libraries in the following link:

Node libraries documentation

2.1.1 Guided practice

Let's create a little server in node using the library `http`.

- Create the script `nodepractice01.js`
- Import the library `http`

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

- Now we create the server

```
const server = http.createServer(req,res) => {

}
```

- We must create a header to our response (parameter `res`) using the method `writeHead` of the `server` object. In this example, we will return a json object.

```
const server = http.createServer((req,res) => {
    res.writeHead(200, { 'Content-Type': 'application/json' });
});
```

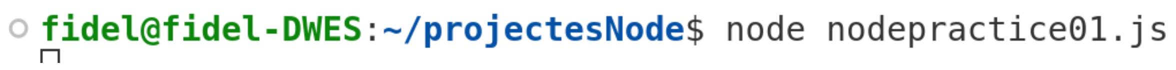
- Finally, we fill the body of our response with a message "Hello world!"

```
const server = http.createServer((req,res) => {
    res.writeHead(200, { 'Content-Type': 'application/json' });
    res.end(JSON.stringify({message:"Hello world!"}));
});
```

- The server must listen through a port

```
server.listen(8001);
```

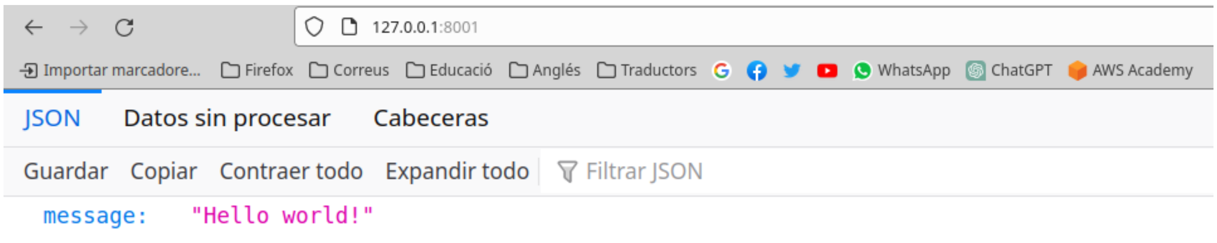
Now, if we run the script from the VSCode terminal, seems as if nothing happens…



```
fidel@fidel-DWES:~/projectesNode$ node nodepractice01.js
```

Figure 4: nodepractice01.js

But if we open our browser and we go to 127.0.0.1:8001 we will get the json message



127.0.0.1:8001
Importar marcadore... Firefox Correus Educació Anglés Traductors WhatsApp ChatGPT AWS Academy
JSON Datos sin procesar Cabeceras
Guardar Copiar Contraer todo Expandir todo Filtrar JSON

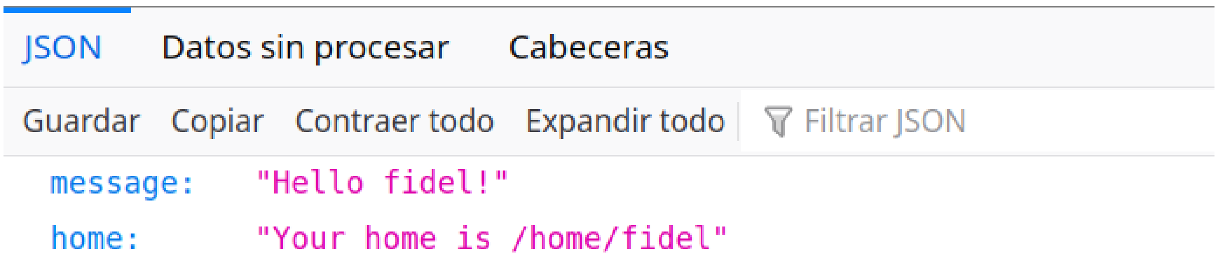
```
message:    "Hello world!"
```

Figure 5: nodepractice01.js

2.1.2 Practice

Copy the script `nodepractice01.js` to `nodepractice02.js` and made changes so the returned json includes the name of your system user and the name of your homedir.

Upload the `nodepractice02.js` to Aules when it's finished.



```
message:   "Hello fidel!"
home:      "Your home is /home/fidel"
```

Figure 6: nodepractice02.js

2.2 Creating our own libraries

We can use `require` to include a file in another file, just as `include` did for php scripts. We must specify the relative path of the file we want to be included.

Let's create a script named `operations.js` with this content:

```
let add_op = (num1, num2) => num1 + num2;
let sub_op = (num1, num2) => num1 - num2;
let mul_op = (num1, num2) => num1 * num2;
let div_op = (num1, num2) => num1 / num2;
```

Now, to include this operations in other script using `require`, we must export them

```
module.exports = {
    sumar: add_op,
    restar: sub_op,
    multiplicar: mul_op,
    dividir: div_op
};
```

Now, in a new script named `example04.js` we can import the `operations.js` library, and then call the functions defined on it.

```
const operations = require('./operations.js');
let a = 10;
let b = 5;
console.log(operations.sumar(a,b));
```

If we run the script in the VSCode terminal, we should get a 15.

Now, try this:

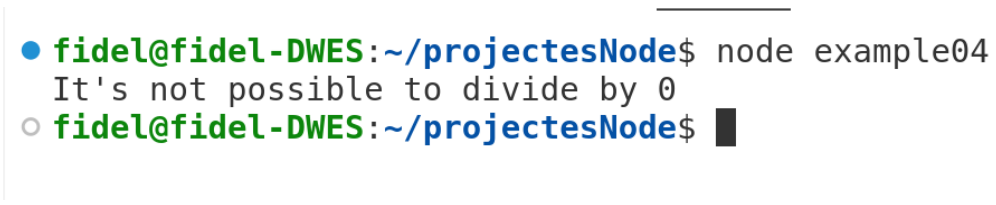
```
const operations = require('./operations.js');
let a = 10;
let b = 0;
console.log(operations.dividir(a,b));
```

We will get *Infinity* as a result, because it's not possible to divide by zero. How can we display a message if the second parameter is zero? We must change the original function in `operations.js`.

```
let div_op = (num1, num2) => num2 === 0
            ? "It's not possible to divide by 0"
            : num1 / num2;
```

If num2 is equal to zero, then we will return a message. If not, we will return the division.

Try now and see what happens.



```
● fidel@fidel-DWES:~/projectesNode$ node example04
  It's not possible to divide by 0
○ fidel@fidel-DWES:~/projectesNode$ █
```

Figure 7: example04.js

2.2.1 Including an entire folder of modules

It is possible to include an entire folder of modules, following a specific nomenclature. To include a whole folder do these steps:

- Add all the modules we want to export (.js files) inside a folder
- Create a file called index.js in the same folder
- Inside this index.js file, include (with require) all the other modules in the folder that we want to export

For instance, if we create a folder `languages` with these files:

`es.js`

```
module.exports = {
    saludo : "Hola"
};
```

`en.js`

```
module.exports = {
    saludo : "Hello"
};
```

Now we can create an `index.js` file with this content:

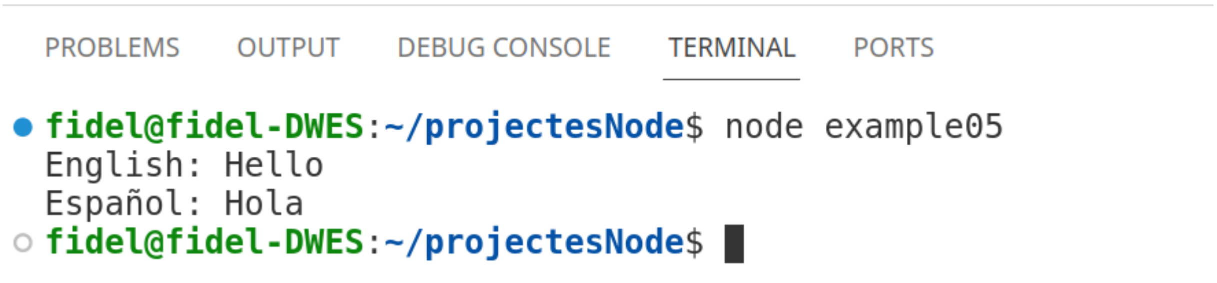
`index.js`

```
const en = require('./en');
const es = require('./es');
module.exports = {
    es : es,
    en : en
};
```

Now, in the root folder of the project, we can create a file, for instance `example05.js`, with this content:

`example05.js`

```
const languages = require('./languages');
console.log("English:", languages.en.saludo);
console.log("Español:", languages.es.saludo);
```


PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
● fidel@fidel-DWES:~/projectesNode$ node example05
  English: Hello
  Español: Hola
○ fidel@fidel-DWES:~/projectesNode$ █
```

Figure 8: example05.js

2.2.2 Sharing information between modules with Json

In the previous example, we saved the shared information in a json file.

`languages.json`

```
{
    "es" : "Hola",
    "en" : "Hello"
}
```

Now, our file `es.js`:

```
const texts = require('./languages.json');
module.exports = {
    saludo : texts.es
};
```

And `en.js`:

```
const texts = require( __dirname + '/operations.js');
module.exports = {
saludo : texts.en
};
```

The files `index.js` and `example05.js` remain the same.

> If we execute a script from a different folder than the one that contains the script, we will get an error in the require. It's a good practice to do
>
> `require(__dirname + '/module_name')`
>
> in order to access to the module from every place in our project. The `__dirname` variable makes reference to the folder containing the invoked script, wherever we are.

3 Third party libraries

By now we are using only node libraries, but we can download and install on our JavaScript applications other modules or libraries that have nothing to do with Node, such as for example ***Bootstrap*** or ***jQuery***. To do this we will use `npm`.

`npm` (Node Package Manager) is a package manager for Javascript, and is installed automatically when we install Node.js. We can check that we have it installed, and what specific version we have, using the command `npm -v` or `npm --version`. The registry of libraries or modules managed by NPM is on the npmjs.com website.

The package manager can be used to install some packages in a certain project, or globally.

3.1 Install a module locally

The information about the configuration of a project is saved in the file `package.json`. The file must be created with `npm` inside the folder of our project:

```
npm init
```

Then `npm` will ask for the package name (by default, the name of the folder), the version (by default, 1.0.0), a description, the entry point (by default index.js), the test command (a command to be executed if we run `npm test`), the git repository, the keywords, the author and the license. If we confirm the data, we will get the `package.json` file. It should look like this:

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

Now we can add to the `package.json` file the modules we need to be installed and used in our project. We don't do this editing the json file, but using `npm install`.

```
npm install module_name
```

If we install an external module, the information about that module will be included in the ***dependencies*** section of the `package.json` file.

We can load the installed modules using `require`, as if they were native modules.

3.1.1 Example

Let's install the module `lodash` in our project.

```
npm install lodash
```

Let's see how our `package.json` has changed:

```
{
  "name": "testnpm",
  "version": "1.0.0",
  "description": "Testing npm",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Fidel Oltra",
  "license": "ISC",
  "dependencies": {
    "lodash": "^4.17.21"
  }
}
```

The `lodash` module has a method called `join` to convert the elements in an array into a string with a separator. Let's load the module and use it in a example:

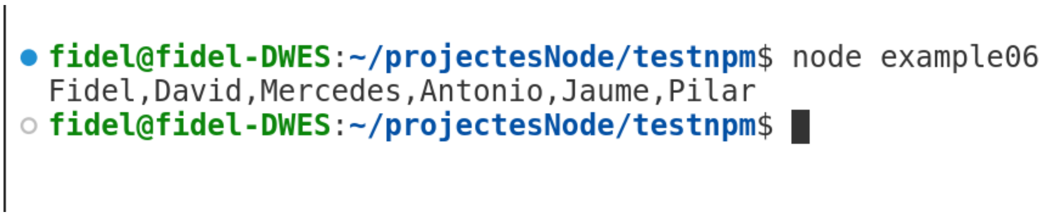
`example06.js`

```
const lds = require('lodash');
const names = ["Fidel","David","Mercedes","Antonio","Jaume","Pilar"];

var namesString = lds.join(names);

console.log(namesString);
```

The result:



```
● fidel@fidel-DWES:~/projectesNode/testnpm$ node example06
  Fidel,David,Mercedes,Antonio,Jaume,Pilar
○ fidel@fidel-DWES:~/projectesNode/testnpm$ █
```

Figure 9: example06.js

3.2 Uninstall a module

We can uninstall a module from our project by doing

```
npm uninstall module_name
```

3.3 Managing versions

We can edit the ***dependencies*** section of our `package.json` file in order to declare specific versions for each module. This way, we can be sure that our application will work with compatible versions of the external modules. Let's see some examples using the `lodash` module:

- `"lodash": "1.0.0"` our application only works with the 1.0.0 version
- `"lodash": "1.0.x"` our application works with any version beginning with 1.0
- `"lodash": "*"` our application will get always the latest version of the module
- `"lodash": ">= 1.0.2"` the application works with any version from 1.0.2.
- `"lodash": "< 1.0.9"` the application only works with versions below 1.0.9
- `"lodash": ">= 1.0.2 < 1.0.9"` versions from 1.0.2 to 1.0.9 (not included)

3.4 Editing package.json

We can edit package.json manually to add or remove dependencies.

Once we have added a new dependency, o remove and old one, we must execute `npm install`.

We can also use `npm install` to deploy the application in another computer without copying the `node_modules` folder. In fact is a good practice not to copy this folder to the final user and install the modules using `npm install` and the `package.json` file.

3.5 Install a module globally

For some modules that are called from the console may be interesting install them globally in order to use them within any project.

When we install a module globally, the `package.json` file is not modified. The command is:

```
npm install -g module_name
```

These modules can't be loaded with `require` in a specific project.

Let's see how the installation of globally scoped modules works with a really useful one: the **nodemon** module. This module operates through the terminal and is used to monitor the execution of a Node application. In the event of any changes to the application, it automatically restarts and runs it again for us, eliminating the need to re-enter the 'node' command in the terminal. You can find information about nodemon here.

To install nodemon globally, we use the following command (with administrator permissions):

```
npm install -g nodemon
```

Once installed globally, the `nodemon` command will be added to the same folder where the 'node' or 'npm' commands reside. To use it, simply navigate to the project folder you want to test and use this command instead of 'node' to launch the application:

```
nodemon index.js
```

Several information messages will automatically appear on the screen, along with the result of running our program. With every change we make, this process will restart, executing the program again.

To end the execution of nodemon (and therefore the application being monitored), simply press Control+C in the terminal.

To uninstall a globally installed module, use the following command:

```
npm uninstall -g module_name
```

3.6 Exercise

- Create a new folder `exercice14_01` and then create the `package.json` with npm.
- Install the package `moment` using npm.
- Create an `index.js` file and load the `moment` package with `require`.
- Create a variable with the current date and hour.

```
let now = moment();
```

- Create a variable with a date of 2022, for instance

```
let longago = moment("01/09/2022","DD/MM/YYYY");
```

- Create a variable with a date of 2024, for instance

```
let faraway = moment("31/07/2024","DD/MM/YYYY");
```

- Find information about the method `duration` of `moment`. Use the method to display the difference, in years, months and days, between the dates `longago` and `faraway`.
- Find information about the method add of `moment` to add 1 year to the current date (now).
- Find information about the method `format` of `moment` to display the new date (now + 1 year) on the screen with the format “DD/MM/YYYY”.