**Node**

**Introduction**

A Node.js app is run in a single process, without creating a new thread for every request. Node.js provides a set of asynchronous I/O primitives in its standard library that prevent JavaScript code from blocking and generally, libraries in Node.js are written using non-blocking paradigms, making blocking behavior the exception rather than the norm.

When Node.js performs an I/O operation, like reading from the network, accessing a database or the filesystem, instead of blocking the thread and wasting CPU cycles waiting, Node.js will resume the operations 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.

**A Vast Number of Libraries**

npm with its simple structure helped the ecosystem of Node.js proliferate, and now the npm registry hosts over 1.3 million open source packages you can freely use.

**An Example Node.js Application**

const http = require('http')

const hostname = '127.0.0.1'

const port = process.env.PORT

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

res.statusCode = 200

res.setHeader('Content-Type', 'text/plain')

res.end('Hello World!\n')

})

server.listen(port, hostname, () => {

console.log(`Server running at http://${hostname}:${port}/`)

})

Whenever a new request is received, the [request event](https://nodejs.org/api/http.html#http_event_request) is called, providing two objects: a request (an [http.IncomingMessage](https://nodejs.org/api/http.html#http_class_http_incomingmessage) object) and a response (an [http.ServerResponse](https://nodejs.org/api/http.html#http_class_http_serverresponse) object).

**A little bit of history**

JavaScript is a programming language that was created at Netscape as a scripting tool to manipulate web pages inside their browser.

Part of the business model of Netscape was to sell Web Servers, which included an environment called *Netscape LiveWire* that could create dynamic pages using server-side JavaScript. Unfortunately, *Netscape LiveWire* wasn't very successful and server-side JavaScript wasn't popularized until recently, by the introduction of Node.js.

**Differences between Node.js and the Browser**

Another difference is that Node.js uses the CommonJS module system, while in the browser we are starting to see the ES Modules standard being implemented.

In practice, this means that for the time being you use require() in Node.js and import in the browser.

**What is CommonJS?** CommonJS is a module formatting system. It is a standard for structuring and organizing JavaScript code. CJS assists in the server-side development of apps and it’s format has heavily influenced NodeJS’s module management.

**Ok… So, what is a module?** A module is just a bit of code encapsulated in a file, and exported to another file. Modules focus on a single part of functionality and remain loosely coupled with other filed in an application. This is because there are no global or shared variables between modules, as they only communicate via the module.exports object. Any code that you want to be accessible in another file can be a module!

***How can I use CommonJS?*** CommonJS wraps each module in a function called ‘require’, and includes an object called ‘module.exports’, which exports code for availability to be required by other modules.

**The V8 JavaScript Engine**

V8 provides the runtime environment in which JavaScript executes. The DOM, and the other Web Platform APIs are provided by the browser.

**Other JS engines**

Other browsers have their own JavaScript engine:

* Firefox has [**SpiderMonkey**](https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey)
* Safari has [**JavaScriptCore**](https://developer.apple.com/documentation/javascriptcore) (also called Nitro)
* Edge was originally based on [**Chakra**](https://github.com/Microsoft/ChakraCore) but has more recently been [rebuilt using Chromium](https://support.microsoft.com/en-us/help/4501095/download-the-new-microsoft-edge-based-on-chromium) and the V8 engine.

**The quest for performance**

V8 is written in C++, and it's continuously improved. It is portable and runs on Mac, Windows, Linux and several other systems.

V8 is always evolving, just like the other JavaScript engines around, to speed up the Web and the Node.js ecosystem.

**Compilation**

JavaScript is generally considered an interpreted language, but modern JavaScript engines no longer just interpret JavaScript, they compile it.

This has been happening since 2009, when the SpiderMonkey JavaScript compiler was added to Firefox 3.5, and everyone followed this idea.

JavaScript is internally compiled by V8 with **just-in-time** (JIT) **compilation** to speed up the execution.

This might seem counter-intuitive, but since the introduction of Google Maps in 2004, JavaScript has evolved from a language that was generally executing a few dozens of lines of code to complete applications with thousands to hundreds of thousands of lines running in the browser.

Our applications now can run for hours inside a browser, rather than being just a few form validation rules or simple scripts.

In this new world, compiling JavaScript makes perfect sense because while it might take a little bit more to have the JavaScript ready, once done it's going to be much more performant than purely interpreted code.

# How to exit from a Node.js program

Let's start with the most drastic one, and see why you're better off not using it.

The process core module provides a handy method that allows you to programmatically exit from a Node.js program: process.exit().

When Node.js runs this line, the process is immediately forced to terminate.

This means that any callback that's pending, any network request still being sent, any filesystem access, or processes writing to stdout or stderr - all is going to be ungracefully terminated right away.

If this is fine for you, you can pass an integer that signals the operating system the exit code:

process.exit(1)

By default the exit code is 0, which means success. Different exit codes have different meaning, which you might want to use in your own system to have the program communicate to other programs.

Basically if you want to exit with success use 0 if you want to exit with failure use 1.

You can also set the process.exitCode property:

process.exitCode = 1

and when the program will later end, Node.js will return that exit code.

A program will gracefully exit when all the processing is done.

Many times with Node.js we start servers, like this HTTP server

This program is never going to end. If you call process.exit(), any currently pending or running request is going to be aborted. This is *not nice*.

In this case you need to send the command a SIGTERM signal, and handle that with the process signal handler:

process.on('SIGTERM', () => {

server.close(() => {

console.log('Process terminated')

})

})

*What are signals? Signals are a POSIX intercommunication system: a notification sent to a process in order to notify it of an event that occurred.*

SIGKILL is the signal that tells a process to immediately terminate, and would ideally act like process.exit().

SIGTERM is the signal that tells a process to gracefully terminate. It is the signal that's sent from process managers like upstart or supervisord and many others.

You can send this signal from inside the program, in another function:

process.kill(process.pid, 'SIGTERM')

Or from another Node.js running program, or any other app running in your system that knows the PID of the process you want to terminate.

# How to read environment variables from Node.js

The process core module of Node.js provides the env property which hosts all the environment variables that were set at the moment the process was started.

Here is an example that accesses the NODE\_ENV environment variable, which is set to development by default.

process.env.NODE\_ENV // "development"

Setting it to "production" before the script runs will tell Node.js that this is a production environment.

In the same way you can access any custom environment variable you set.

# How to use the Node.js REPL

If we omit the filename, we use it in REPL mode:

node

*Note: REPL also known as Read Evaluate Print Loop is a programming language environment(Basically a console window) that takes single expression as user input and returns the result back to the console after execution.*

The REPL is waiting for us to enter some JavaScript code, to be more precise.

Start simple and enter

> console.log('test')

test

undefined

>

The first value, test, is the output we told the console to print, then we get undefined which is the return value of running console.log().

## Use the tab to autocomplete

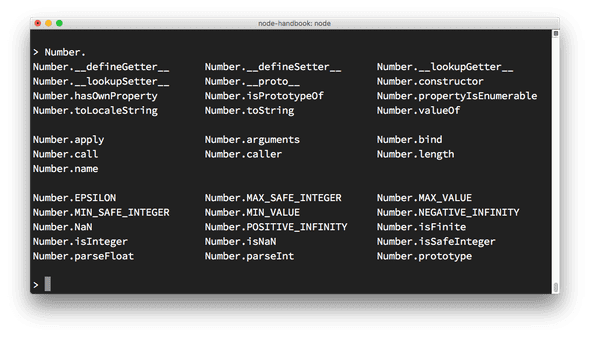
The cool thing about the REPL is that it's interactive.

As you write your code, if you press the tab key the REPL will try to autocomplete what you wrote to match a variable you already defined or a predefined one.

## Exploring JavaScript objects

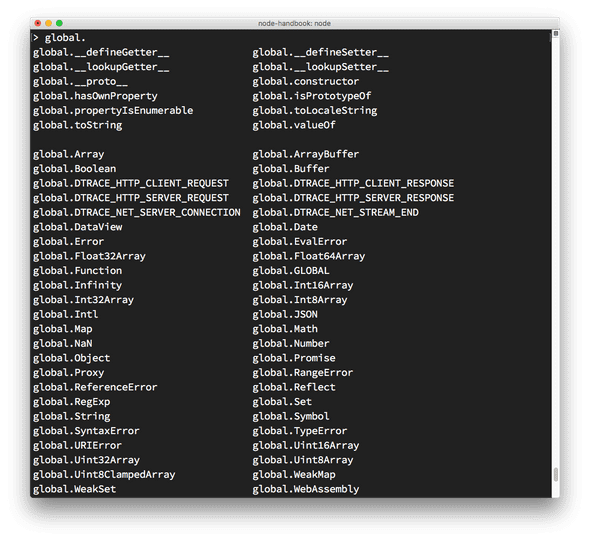
Try entering the name of a JavaScript class, like Number, add a dot and press tab.

The REPL will print all the properties and methods you can access on that class:

[](https://nodejs.dev/static/2b60eb9487f93b672da38e391d2e5e56/6937a/tab.png)

## Explore global objects

You can inspect the globals you have access to by typing global. and pressing tab:

[](https://nodejs.dev/static/c2bef52ca393ecb33846c54af34927a1/6937a/globals.png)

**The \_ special variable**

If after some code you type \_, that is going to print the result of the last operation.

## Dot commands

The REPL has some special commands, all starting with a dot .. They are

* .help: shows the dot commands help
* .editor: enables editor mode, to write multiline JavaScript code with ease. Once you are in this mode, enter ctrl-D to run the code you wrote.
* .break: when inputting a multi-line expression, entering the .break command will abort further input. Same as pressing ctrl-C.
* .clear: resets the REPL context to an empty object and clears any multi-line expression currently being input.
* .load: loads a JavaScript file, relative to the current working directory
* .save: saves all you entered in the REPL session to a file (specify the filename)
* .exit: exits the repl (same as pressing ctrl-C two times)

The REPL knows when you are typing a multi-line statement without the need to invoke .editor.

For example if you start typing an iteration like this:

[1, 2, 3].forEach(num => {

and you press enter, the REPL will go to a new line that starts with 3 dots, indicating you can now continue to work on that block.

... console.log(num)

... })

If you type .break at the end of a line, the multiline mode will stop and the statement will not be executed.

# Node.js, accept arguments from the command line

You can pass any number of arguments when invoking a Node.js application using

node app.js

Arguments can be standalone or have a key and a value.

For example:

node app.js joe

or

node app.js name=joe

This changes how you will retrieve this value in the Node.js code.

The way you retrieve it is using the process object built into Node.js.

It exposes an argv property, which is an array that contains all the command line invocation arguments.

The first element is the full path of the node command.

The second element is the full path of the file being executed.

All the additional arguments are present from the third position going forward.

You can iterate over all the arguments (including the node path and the file path) using a loop:

process.argv.forEach((val, index) => {

console.log(`${index}: ${val}`)

})

You can get only the additional arguments by creating a new array that excludes the first 2 params:

const args = process.argv.slice(2)

If you have one argument without an index name, like this:

node app.js joe

you can access it using

const args = process.argv.slice(2)

args[0]

In this case:

node app.js name=joe

args[0] is name=joe, and you need to parse it. The best way to do so is by using the [minimist](https://www.npmjs.com/package/minimist) library, which helps dealing with arguments:

const args = require('minimist')(process.argv.slice(2))

args['name'] //joe

This time you need to use double dashes before each argument name:

node app.js --name=joe

# Output to the command line using Node.js

We can also format pretty phrases by passing variables and a format specifier.

For example:

console.log('My %s has %d years', 'cat', 2)

* %s format a variable as a string
* %d format a variable as a number
* %i format a variable as its integer part only
* %o format a variable as an object

## Clear the console

console.clear() clears the console (the behavior might depend on the console used)

## Counting elements

console.count() is a handy method.

Graphical user interface, text, application

Description automatically generated

## Print the stack trace

There might be cases where it's useful to print the call stack trace of a function, maybe to answer the question how did you reach that part of the code?

You can do so using console.trace():

const function2 = () => console.trace()

const function1 = () => function2()

function1()

## Calculate the time spent

You can easily calculate how much time a function takes to run, using time() and timeEnd()

const doSomething = () => console.log('test')

const measureDoingSomething = () => {

console.time('doSomething()')

//do something, and measure the time it takes

doSomething()

console.timeEnd('doSomething()')

}

measureDoingSomething()

## stdout and stderr

As we saw console.log is great for printing messages in the Console. This is what's called the standard output, or stdout.

console.error prints to the stderr stream.

It will not appear in the console, but it will appear in the error log.

## Color the output

You can color the output of your text in the console by using [escape sequences](https://gist.github.com/iamnewton/8754917). An escape sequence is a set of characters that identifies a color.

Example:

console.log('\x1b[33m%s\x1b[0m', 'hi!')

You can try that in the Node.js REPL, and it will print hi! in yellow.

However, this is the low-level way to do this. The simplest way to go about coloring the console output is by using a library. [Chalk](https://github.com/chalk/chalk) is such a library, and in addition to coloring it also helps with other styling facilities, like making text bold, italic or underlined.

You install it with npm install chalk, then you can use it:

const chalk = require('chalk')

console.log(chalk.yellow('hi!'))

Using chalk.yellow is much more convenient than trying to remember the escape codes, and the code is much more readable.

## Create a progress bar

[Progress](https://www.npmjs.com/package/progress) is an awesome package to create a progress bar in the console. Install it using npm install progress

This snippet creates a 10-step progress bar, and every 100ms one step is completed. When the bar completes we clear the interval:

const ProgressBar = require('progress')

const bar = new ProgressBar(':bar', { total: 10 })

const timer = setInterval(() => {

bar.tick()

if (bar.complete) {

clearInterval(timer)

}

}, 100)

# Accept input from the command line in Node.js

How to make a Node.js CLI program interactive?

Node.js since version 7 provides the [readline module](https://nodejs.org/api/readline.html) to perform exactly this: get input from a readable stream such as the process.stdin stream, which during the execution of a Node.js program is the terminal input, one line at a time.

const readline = require('readline').createInterface({

input: process.stdin,

output: process.stdout

})

readline.question(`What's your name?`, name => {

console.log(`Hi ${name}!`)

readline.close()

})

This piece of code asks the username, and once the text is entered and the user presses enter, we send a greeting.

The question() method shows the first parameter (a question) and waits for the user input. It calls the callback function once enter is pressed.

In this callback function, we close the readline interface.

readline offers several other methods, and I'll let you check them out on the package documentation linked above.

If you need to require a password, it's best not to echo it back, but instead show a \* symbol.

The simplest way is to use the [readline-sync package](https://www.npmjs.com/package/readline-sync) which is very similar in terms of the API and handles this out of the box.

A more complete and abstract solution is provided by the [Inquirer.js package](https://github.com/SBoudrias/Inquirer.js).

You can install it using npm install inquirer, and then you can replicate the above code like this:

const inquirer = require('inquirer')

var questions = [

{

type: 'input',

name: 'name',

message: "What's your name?"

}

]

inquirer.prompt(questions).then(answers => {

console.log(`Hi ${answers['name']}!`)

})

Inquirer.js lets you do many things like asking multiple choices, having radio buttons, confirmations, and more.

It's worth knowing all the alternatives, especially the built-in ones provided by Node.js, but if you plan to take CLI input to the next level, Inquirer.js is an optimal choice.

# Expose functionality from a Node.js file using exports

Node.js has a built-in module system.

A Node.js file can import functionality exposed by other Node.js files.

When you want to import something you use

const library = require('./library')

to import the functionality exposed in the library.js file that resides in the current file folder.

In this file, functionality must be exposed before it can be imported by other files.

Any other object or variable defined in the file by default is private and not exposed to the outer world.

This is what the module.exports API offered by the [module system](https://nodejs.org/api/modules.html) allows us to do.

When you assign an object or a function as a new exports property, that is the thing that's being exposed, and as such, it can be imported in other parts of your app, or in other apps as well.

You can do so in 2 ways.

The first is to assign an object to module.exports, which is an object provided out of the box by the module system, and this will make your file export just that object:

const car = {

brand: 'Ford',

model: 'Fiesta'

}

module.exports = car

//..in the other file

const car = require('./car')

The second way is to add the exported object as a property of exports. This way allows you to export multiple objects, functions or data:

const car = {

brand: 'Ford',

model: 'Fiesta'

}

exports.car = car

or directly

exports.car = {

brand: 'Ford',

model: 'Fiesta'

}

And in the other file, you'll use it by referencing a property of your import:

const items = require('./items')

items.car

or

const car = require('./items').car

What's the difference between module.exports and exports?

The first exposes the object it points to. The latter exposes the properties of the object it points to.

# An introduction to the npm package manager

## Introduction to npm

npm is the standard package manager for Node.js.

There are many things that npm does.

[***Yarn***](https://yarnpkg.com/en/)*is an alternative to npm. Make sure you check it out as well.*

## Downloads

npm manages downloads of dependencies of your project.

### Installing all dependencies

If a project has a package.json file, by running

npm install

it will install everything the project needs, in the node\_modules folder, creating it if it's not existing already.

### Installing a single package

You can also install a specific package by running

npm install <package-name>

Often you'll see more flags added to this command:

* --save installs and adds the entry to the package.json file dependencies
* --save-dev installs and adds the entry to the package.json file devDependencies

The difference is mainly that devDependencies are usually development tools, like a testing library, while dependencies are bundled with the app in production.

### Updating packages

Updating is also made easy, by running

npm update

npm will check all packages for a newer version that satisfies your versioning constraints.

You can specify a single package to update as well:

npm update <package-name>

## Running Tasks

The package.json file supports a format for specifying command line tasks that can be run by using

npm run <task-name>

For example:

{

"scripts": {

"start-dev": "node lib/server-development",

"start": "node lib/server-production"

},

}

It's very common to use this feature to run Webpack:

{

"scripts": {

"watch": "webpack --watch --progress --colors --config webpack.conf.js",

"dev": "webpack --progress --colors --config webpack.conf.js",

"prod": "NODE\_ENV=production webpack -p --config webpack.conf.js",

},

}

So instead of typing those long commands, which are easy to forget or mistype, you can run

$ npm run watch

$ npm run dev

$ npm run prod

# Where does npm install the packages?

A global installation is performed using the -g flag:

npm install -g lodash

When this happens, npm won't install the package under the local folder, but instead, it will use a global location.

Where, exactly?

The npm root -g command will tell you where that exact location is on your machine.

On macOS or Linux this location could be /usr/local/lib/node\_modules. On Windows it could be C:\Users\YOU\AppData\Roaming\npm\node\_modules

If you use nvm to manage Node.js versions, however, that location would differ.

I for example use nvm and my packages location was shown as /Users/joe/.nvm/versions/node/v8.9.0/lib/node\_modules.

# How to use or execute a package installed using npm

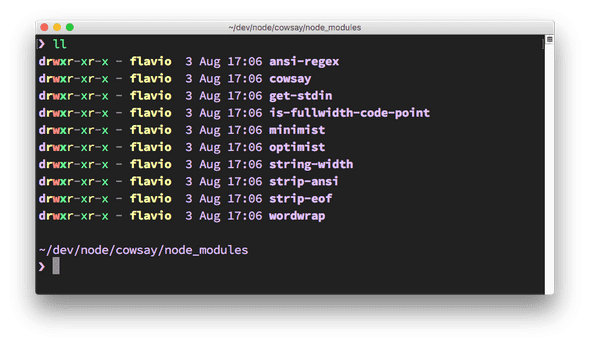
To use it in your code, you just need to import it into your program using require:

const \_ = require('lodash')

What if your package is an executable?

In this case, it will put the executable file under the node\_modules/.bin/ folder.

When you install the package using npm install cowsay, it will install itself and a few dependencies in the node\_modules folder:

[](https://nodejs.dev/static/b245c50f5080dae16a2525fae0ba2c91/d2c2a/node_modules-content.png)

There is a hidden .bin folder, which contains symbolic links to the cowsay binaries:

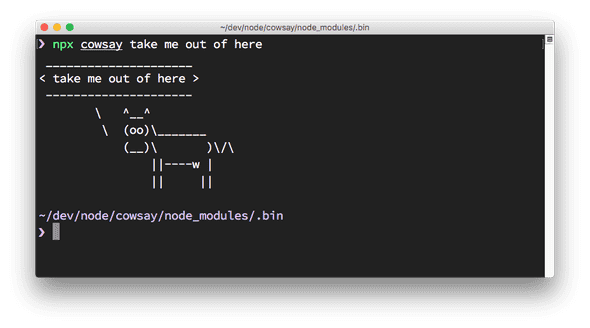
[](https://nodejs.dev/static/99830aefa055e247397de544ad7b7744/d2c2a/binary-files.png)

How do you execute those?

You can of course type ./node\_modules/.bin/cowsay to run it, and it works, but npx, included in the recent versions of npm (since 5.2), is a much better option. You just run:

npx cowsay

and npx will find the package location.

[](https://nodejs.dev/static/ad4f3d3a7464bb0f8a2845fe8e6588c2/d2c2a/cow-say.png)

# The package.json guide

If you work with JavaScript, or you've ever interacted with a JavaScript project, Node.js or a frontend project, you surely met the package.json file.

What's that for? What should you know about it, and what are some of the cool things you can do with it?

The package.json file is kind of a manifest for your project. It can do a lot of things, completely unrelated. It's a central repository of configuration for tools, for example. It's also where npm and yarn store the names and versions for all the installed packages.

## The file structure

Here's an example package.json file:

{}

It's empty! There are no fixed requirements of what should be in a package.json file, for an application. The only requirement is that it respects the JSON format, otherwise it cannot be read by programs that try to access its properties programmatically.

If you're building a Node.js package that you want to distribute over npm things change radically, and you must have a set of properties that will help other people use it. We'll see more about this later on.

This is another package.json:

{

"name": "test-project"

}

It defines a name property, which tells the name of the app, or package, that's contained in the same folder where this file lives.

Here's a much more complex example, which was extracted from a sample Vue.js application:

{

"name": "test-project",

"version": "1.0.0",

"description": "A Vue.js project",

"main": "src/main.js",

"private": true,

"scripts": {

"dev": "webpack-dev-server --inline --progress --config build/webpack.dev.conf.js",

"start": "npm run dev",

"unit": "jest --config test/unit/jest.conf.js --coverage",

"test": "npm run unit",

"lint": "eslint --ext .js,.vue src test/unit",

"build": "node build/build.js"

},

"dependencies": {

"vue": "^2.5.2"

},

"devDependencies": {

},

"engines": {

"node": ">= 6.0.0",

"npm": ">= 3.0.0"

},

"browserslist": ["> 1%", "last 2 versions", "not ie <= 8"]

}

there are *lots* of things going on here:

* version indicates the current version
* name sets the application/package name
* description is a brief description of the app/package
* main set the entry point for the application
* private if set to true prevents the app/package to be accidentally published on npm
* scripts defines a set of node scripts you can run
* dependencies sets a list of npm packages installed as dependencies
* devDependencies sets a list of npm packages installed as development dependencies
* engines sets which versions of Node.js this package/app works on
* browserslist is used to tell which browsers (and their versions) you want to support

All those properties are used by either npm or other tools that we can use.

## Properties breakdown

### name

Sets the package name.

Example:

"name": "test-project"

The name must be less than 214 characters, must not have spaces, it can only contain lowercase letters, hyphens (-) or underscores (\_).

This is because when a package is published on npm, it gets its own URL based on this property.

If you published this package publicly on GitHub, a good value for this property is the GitHub repository name.

### author

Lists the package author name

Example:

{

"author": "Joe <joe@whatever.com> (https://whatever.com)"

}

Can also be used with this format:

{

"author": {

"name": "Joe",

"email": "joe@whatever.com",

"url": "https://whatever.com"

}

}

### contributors

As well as the author, the project can have one or more contributors. This property is an array that lists them.

Example:

{

"contributors": ["Joe <joe@whatever.com> (https://whatever.com)"]

}

Can also be used with this format:

{

"contributors": [

{

"name": "Joe",

"email": "joe@whatever.com",

"url": "https://whatever.com"

}

]

}

### bugs

Links to the package issue tracker, most likely a GitHub issues page

Example:

{

"bugs": "https://github.com/whatever/package/issues"

}

### homepage

Sets the package homepage

Example:

{

"homepage": "https://whatever.com/package"

}

### version

Indicates the current version of the package.

Example:

"version": "1.0.0"

This property follows the semantic versioning (semver) notation for versions, which means the version is always expressed with 3 numbers: x.x.x.

The first number is the major version, the second the minor version and the third is the patch version.

There is a meaning in these numbers: a release that only fixes bugs is a patch release, a release that introduces backward-compatible changes is a minor release, a major release can have breaking changes.

### license

Indicates the license of the package.

Example:

"license": "MIT"

### keywords

This property contains an array of keywords that associate with what your package does.

Example:

"keywords": [

"email",

"machine learning",

"ai"

]

This helps people find your package when navigating similar packages, or when browsing the <https://www.npmjs.com/> website.

### description

This property contains a brief description of the package

Example:

"description": "A package to work with strings"

This is especially useful if you decide to publish your package to npm so that people can find out what the package is about.

### repository

This property specifies where this package repository is located.

Example:

"repository": "github:whatever/testing",

Notice the github prefix. There are other popular services baked in:

"repository": "gitlab:whatever/testing",

"repository": "bitbucket:whatever/testing",

You can explicitly set the version control system:

"repository": {

"type": "git",

"url": "https://github.com/whatever/testing.git"

}

You can use different version control systems:

"repository": {

"type": "svn",

"url": "..."

}

### main

Sets the entry point for the package.

When you import this package in an application, that's where the application will search for the module exports.

Example:

"main": "src/main.js"

### private

if set to true prevents the app/package to be accidentally published on npm

Example:

"private": true

### scripts

Defines a set of node scripts you can run

Example:

"scripts": {

"dev": "webpack-dev-server --inline --progress --config build/webpack.dev.conf.js"

}

These scripts are command line applications. You can run them by calling npm run XXXX or yarn XXXX, where XXXX is the command name. Example: npm run dev.

You can use any name you want for a command, and scripts can do literally anything you want.

### dependencies

Sets a list of npm packages installed as dependencies.

When you install a package using npm or yarn:

npm install <PACKAGENAME>

yarn add <PACKAGENAME>

### devDependencies

Sets a list of npm packages installed as development dependencies.

They differ from dependencies because they are meant to be installed only on a development machine, not needed to run the code in production.

When you install a package using npm or yarn:

npm install --save-dev <PACKAGENAME>

yarn add --dev <PACKAGENAME>

### engines

Sets which versions of Node.js and other commands this package/app work on

Example:

"engines": {

"node": ">= 6.0.0",

"npm": ">= 3.0.0",

"yarn": "^0.13.0"

}

**Difference between process.cwd() and \_\_dirname?**

Knowing the **scope** of each can make things easier to remember.

**process** is node's global object, and **.cwd()** returns where node is running.

**\_\_dirname** is module's property, and represents the file path of the module. In node, one module resides in one file.

Similarly, **\_\_filename** is another module's property, which holds the file name of the module.

suppose we have another file script.js files inside a sub directory of project ie C:/Project/lib/script.js and running node main.js which require script.js

main.js

require('./lib/script.js')

console.log(process.cwd())

// C:\Project

console.log(\_\_dirname)

// C:\Project

console.log(\_\_dirname===process.cwd())

// true

script.js

console.log(process.cwd())

// C:\Project

console.log(\_\_dirname)

// C:\Project\lib

console.log(\_\_dirname===process.cwd())

// false