


# Table of Contents

---

- [Table of Contents](#)
- [What Is Node.js](#)
- [Node.js Introduction](#)
- [JS in Browser](#)  [Server](#)
- [Download & Install Node.js](#)
- [Node.js Repl](#)
- [Node.js Globals](#)
- [Node.js Modules](#)
- [Node.js User-defined Modules](#)
- [Built-in Modules](#)
  - [os](#) Module
  - [path](#) Module
  - [fs](#) Module
- [Sync Vs Async](#)
  - [HTTP Module](#)
- [NPM](#)
- [Package.json](#)
- [npm scripts](#)
- [Event Loop](#)
- [Summary](#)

## What Is Node.js

---

Node.js is an open-source, cross-platform JavaScript runtime environment that executes JavaScript code outside of a web browser. It allows developers to run JavaScript on the server side, and build fast, scalable, and high-performance network applications.

## Node.js Introduction

---

Node.js was created by Ryan Dahl in 2009 and has since become one of the most popular platforms for building server-side web applications. It is built on the V8 JavaScript engine, which is the same engine that powers Google Chrome, and uses an event-driven, non-blocking I/O model. This makes it well-suited for real-time, data-intensive applications that run across distributed devices.

Node.js comes with a built-in package manager, called npm (Node Package Manager), which makes it easy to download and use packages of pre-written JavaScript code. This allows developers to easily add functionality to their applications without having to write everything from scratch. There are more than 1 million packages available on npm, making it one of the largest package ecosystems in the world.

Node.js is used by a wide variety of companies, from small startups to large enterprises, and is well-suited for building web applications, real-time applications, and microservices. Some of the well-known companies that use Node.js in their tech stacks are Netflix, Uber, Trello, LinkedIn, PayPal and many more.

It's worth noting that Node.js is not suitable for all types of applications. It excels at handling a high number of simultaneous connections and I/O-bound or high-level file system tasks. But it may not perform well in heavy CPU-bound tasks, for that reason it's important to evaluate the task and select the appropriate technology.

<https://nodejs.dev/en/learn/>

# JS in Browser Server

---

JavaScript is a versatile programming language that can be used both in web browsers and on servers.

When it comes to the browser, JavaScript is used to create interactive and dynamic user interfaces. JavaScript code is typically embedded in web pages using the script tag, and is executed by the web browser when the page is loaded. JavaScript code in the browser can interact with the Document Object Model (DOM) and the Browser Object Model (BOM) to dynamically update web pages, handle user input, and make web pages more interactive.

On the other hand, JavaScript can also be used on servers through the use of Node.js. Node.js is a JavaScript runtime that allows developers to run JavaScript code on the server side. With Node.js, JavaScript code can be used to build server-side web applications, real-time applications, and microservices. Node.js uses an event-driven, non-blocking I/O model, which makes it well-suited for real-time, data-intensive applications that run across distributed devices.

JavaScript can be used for full-stack web development, which means, both the front-end and back-end of an application can be developed using JavaScript. Due to its unifying character, it makes it easier to have developers work on both parts of the application, knowing they are using the same language and general tooling.

The popularity of JavaScript has made it a go-to language for web development, and its use in both the browser and on the server makes it a versatile and powerful choice for building modern web applications.

Browser	Server
DOM	<b>No</b> Dom
Frontend	Backend
<b>No</b> File system	File system
Window Object	No Window Object

[Read More - https://nodejs.dev/en/learn/differences-between-nodejs-and-the-browser/](https://nodejs.dev/en/learn/differences-between-nodejs-and-the-browser/)

## Download & Install Node.js

---

You can download and install Node.js on your computer by following these steps:

Go to the Node.js website (<https://nodejs.org/>) and select the version of Node.js that you want to download. The latest stable version is recommended, but you can also choose to download an older version if necessary.

Click on the "Download" button for your operating system (Windows, Mac, or Linux) to begin the download.

Once the download is complete, open the installer package and follow the prompts to install Node.js on your computer. The installer will guide you through the process, and you can choose the location where Node.js will be installed.

After the installation is complete, you can open the command prompt or terminal and type 'node -v' to check the version of Node.js you have installed. This will also let you know if the installation was successful

Similarly, You can check the version of npm (Node Package Manager) by typing 'npm -v'

You're all set, you can start using Node.js to run JavaScript on your computer.

You can also use other methods for installing node, such as using package manager for specific distros (e.g. apt, dnf, brew, etc) or version managers like nvm (Node Version Manager) which can be a great choice if you want to switch between different Node.js versions easily.

It's worth noting that the instructions and steps may vary slightly depending on your operating system, but the basic process should be the same.

[Check this link for detailed instructions:](#)

## Node.js Repl

---

REPL stands for Read-Eval-Print-Loop and is a simple interactive language shell that can be used to run Node.js commands. It allows developers to enter JavaScript code and see the result immediately, making it a useful tool for testing and development.

You can start the Node.js REPL by opening a terminal or command prompt and typing the command `node`. This will start the REPL and you will see the `>` prompt, indicating that you can enter JavaScript code.

For example, you can enter a simple mathematical expression and the REPL will return the result:

```
> 1 + 1
2
```

You can also use the REPL to run JavaScript functions and work with variables. Here's an example:

```
> let x = 5
undefined
> x * 2
10
> function greet(name) { return `Hello, ${name}!`; }
[Function: greet]
> greet('John')
'Hello, John!'
```

The REPL also includes some useful commands such as:

- `.help`: display help
- `.clear`: clear the context
- `.save`: save to a file
- `.load`: load a file
- `.exit`: Exit

The REPL is a powerful tool that allows you to experiment with and test your JavaScript code quickly and easily. It can be especially useful for learning the basics of Node.js and JavaScript, as well as for debugging and troubleshooting your code.

`Ctrl + C` to exit the repl.

## Node.js Globals

---

In Node.js, there are several global objects and variables that are available to all modules without the need to import them. These global objects and variables are called "globals" and can be used to perform various tasks and access various properties in a Node.js application.

Here are some of the most commonly used Node.js globals:

- `global`: The global object is the top-level object in the Node.js runtime environment. It is similar to the window object in a web browser. You can use the global object to define global variables and functions.
- `process`: The process object is a global object that provides information about the current process and allows you to interact with it. You can use the process object to access environment variables, process arguments, and other information about the current process.
- `console`: The console object provides methods for logging information to the console, such as `console.log()`, `console.info()`, `console.error()`, etc. It is widely used for debugging and inspecting the state of the application.
- `Buffer`: The Buffer class is a global object that allows you to work with binary data in Node.js. You can use it to create, manipulate, and convert binary data.
- `setTimeout` and `setInterval`: These are functions that you can use to schedule function calls after a certain period of time, it allows you to add delay and to execute something periodically. In addition to these globals, Node.js also provides several other global objects for working with the file system, streams, and other features of the runtime environment. You can access the full list of global objects by typing `global` in the Node.js REPL or by reading the Node.js documentation.

It's worth mentioning that some global variables have been marked as deprecated or have been moved to different modules, Therefore it's important to check the current version of Node.js you are using and refer to the documentation to ensure you are using up-to-date Global variables.

Check [global variables](#) for more documentation. As an exercise, check the following list of mostly used global variables:

- `__dirname` Path to current directory
- `__filename` File name of current file
- `exports` A reference to the module.exports that is shorter to type
- `module` A reference to the current module

- `require()` Function to Used to import modules

# Node.js Modules

---

In Node.js, a module is a self-contained piece of JavaScript code that exports one or more properties or functions. Modules are used to organize and reuse code in a Node.js application. Node.js includes a built-in module system, which allows you to easily organize your application into small, reusable units.

Node.js modules are created using the `exports` or `module.exports` objects. You can add properties and functions to these objects, and then import them in other parts of your application.

For example, let's say you have a module called `math.js` that exports a single function called `add`:

```
// math.js
exports.add = function(a, b) {
  return a + b;
};
```

You can then use this function in another module by using the `require` function to import the `math` module:

```
// app.js
const math = require('./math');
console.log(math.add(2, 3)); // 5
```

The `require` function is used to import modules in Node.js. It takes a single argument, which is the name or path of the module to be imported. The name of the module is a relative path of the file. If the module is built-in Node.js, you don't need to provide the path, it will automatically pick it up.

Additionally, Node.js also support the import statement, with the `import` statement you can use destructuring to import only what you need, the syntax is similar to the `require` statement, but instead of using the `const` keyword you use the `import` keyword and it provides a cleaner way to import your modules.

```
import { add } from './math'
console.log(add(3,3)) // 6
```

Node.js modules are singleton, that means they are only instantiated once, regardless of how many times they are imported. This allows modules to maintain state across multiple imports.

Node.js has a large and active ecosystem with thousands of modules that are available on npm (Node Package Manager) repository, which makes it easy for developers to find and reuse existing modules. This also means that it's easy to add new functionality to your application without having to write everything from scratch.

In summary, modules are the building blocks for structuring and organizing your Node.js application. They help to keep your code organized, maintainable, and reusable. Modules make it easy to share and reuse code, and also make it easy to test individual parts of your application.

## Node.js User-defined Modules

---

To create a user-defined module in Node.js, you will need to create a JavaScript file that exports the properties and functions that you want to make available to other parts of your application. Here is an example of a simple module that exports a single function:

```
// math.js
exports.add = function(a, b) {
  return a + b;
};
```

This module exports a single function called `add` that takes two arguments and returns their sum.

To use this module in another part of your application, you will need to use the `require` function to import it. The `require` function takes a single argument, which is the name or path of the module to be imported.

Here is an example of how you can import and use the `math` module in another file:

```
// app.js
const math = require('./math');
console.log(math.add(2, 3)); // 5
```

In this example, the `math` module is imported using the `require` function and stored in a variable called `math`. The `add` function from the `math` module is then called using the `math.add` syntax.

You can also use the `import` statement to import a module. The syntax is similar to the `require` function, but instead of using the `const` keyword you use the `import` keyword. Here is an example:

```
import { add } from './math';
console.log(add(2, 3)); // 5
```

In this example, only the `add` function is imported from the `math` module using destructuring.

You can also use the `module.exports` object to export multiple properties and functions at once. Here is an example:

```
// math.js
module.exports = {
```

```
add: function(a, b) {  
  return a + b;  
},  
subtract: function(a, b) {  
  return a - b;  
}  
};
```

Now, you can import the module and call the add and subtract function.

```
const math = require('./math')  
console.log(math.add(3,3)) // 6  
console.log(math.subtract(3,3)) //
```

Read More - <https://nodejs.dev/en/learn/expose-functionality-from-a-nodejs-file-using-exports/>

## Built-in Modules

---

Node.js includes a number of built-in modules, which are modules that are included with the Node.js runtime and are available to all Node.js applications. These built-in modules provide a wide range of functionality, including file system access, networking, and cryptography.

Here are some examples of commonly used built-in modules in Node.js:

- **http**: The **http** module provides a way to create HTTP servers and clients. You can use this module to create web servers, make HTTP requests, and handle incoming HTTP requests.
- **fs**: The **fs** module provides an API for interacting with the file system. You can use this module to read and write files, create and delete directories, and perform other file system operations.
- **path**: The **path** module provides utilities for working with file and directory paths. You can use this module to manipulate file paths and extract information about files and directories.
- **stream**: The **stream** module provides a way to work with streams of data. Streams are a way to handle large amounts of data, such as files and network connections, in a memory-efficient way.
- **crypto**: The **crypto** module provides cryptographic functionality, such as hashing, signing and encryption/decryption.
- **os**: The **os** module provides information about the underlying operating system. You can use this module to get information about the system architecture, uptime, and available memory.
- **util**: The **util** module provides utility functions for formatting and debugging.

You can use the **require** function to import a built-in module, just like you would with any other module. Once a module is imported, you can access its properties and methods to perform various tasks.

Check the this [list of built-in modules](#) that you can use without further installation. Next, we will delve into the functionality provided by the `os`, `path`, `fs`, and `http` modules in Node.js.

## os Module

The `os` module is a built-in Node.js module that provides a way to access various operating system-related utility methods. You can use this module to get information about the system's platform, hostname, architecture, and various other details. Here are a few examples of how you might use this module:

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

console.log(os.platform()); // 'linux'
console.log(os.arch()); // 'x64'
console.log(os.hostname()); // 'my-machine'
console.log(os.type()); // 'Windows_NT'
console.log(os.release()); // '10.0.17134'
console.log(os.uptime()); // 123597
console.log(os.loadavg()); // [2.59375, 2.7109375, 2.83984375]
console.log(os.totalmem()); // 17179869184
console.log(os.freemem()); // 8357457920
console.log(os.cpus()); // [{...}, {...}, ...]
console.log(os.networkInterfaces()); // {'eth0': [{...}, {...}], 'lo': [{...}]}
```

The `os.cpus()` method returns an array of objects containing information about each logical CPU core, including the model name, clock speed, and times (in user, nice, system, idle, and irq mode).

The `os.networkInterfaces()` method returns an object containing network interfaces, such as 'eth0' and 'lo', and a list of their associated addresses.

This is just a subset of methods provided by `os` module. You can see the full list in the official documentation, including methods to work with the file system and user info.

<https://nodejs.org/api/os.html>

## path Module

The `path` module is another built-in Node.js module that provides a way to work with file and directory paths. This module has several methods that can help you manipulate file paths in a platform-independent manner. Here are a few examples of how you might use this module:

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

console.log(path.join('/foo', 'bar', 'baz/asdf', 'quux', '..'));
// '/foo/bar/baz/asdf'

console.log(path.resolve('/foo/bar', './baz'));
// '/foo/bar/baz'
```



```
console.log(path.normalize('/foo/bar//baz/asdf/quux/..'));  
// '/foo/bar/baz/asdf'  
  
console.log(path.extname('index.html'));  
// '.html'  
  
console.log(path.dirname('/foo/bar/baz/asdf/quux'));  
// '/foo/bar/baz/asdf'  
  
console.log(path.basename('/foo/bar/baz/asdf/quux.html', '.html'));  
// 'quux'
```

The `path.join()` method joins all arguments together and normalizes the resulting path. The resulting path will use the appropriate separator for the operating system.

The `path.resolve()` method resolves a sequence of paths or path segments into an absolute path.

The `path.normalize()` method normalizes the given path, resolving '..' and '.' segments.

The `path.extname()` method returns the file extension of a path, like '.html'.

The `path.dirname()` method returns the directory name of a path, like '/foo/bar/baz/asdf'.

The `path.basename()` method returns the last portion of a path, like 'quux'.

This is just a subset of the available methods provided by path module. You can see the full list in the official documentation for more details

<https://nodejs.org/api/path.html>

## fs Module

The `fs` (File System) module is another built-in Node.js module that provides a way to interact with the file system. It allows you to perform various operations on files and directories, such as reading, writing, and deleting files, as well as creating and deleting directories. Here are a few examples of how you might use this module:

```
const fs = require('fs');  
  
// Read the contents of a file  
fs.readFile('file.txt', 'utf8', (err, data) => {  
  if (err) throw err;  
  console.log(data);  
});  
  
// Write to a file  
fs.writeFile('file.txt', 'Hello World!', (err) => {  
  if (err) throw err;  
  console.log('File written.');
```

```
});

// Check if a file exists
fs.exists('file.txt', (exists) => {
  console.log(exists); // true or false
});

// Delete a file
fs.unlink('file.txt', (err) => {
  if (err) throw err;
  console.log('File deleted.');
```

```
});

// Create a directory
fs.mkdir('new_directory', (err) => {
  if (err) throw err;
  console.log('Directory created.');
```

```
});

// Read a directory
fs.readdir('path/to/directory', (err, files) => {
  if (err) throw err;
  console.log(files);
});
```

```


// Delete a directory
fs.rmdir('path/to/directory', (err) => {
  if (err) throw err;
  console.log('Directory deleted.');
```

```
});
```

These are just a subset of the available methods provided by fs module. You can see the full list in the official documentation for more details

<https://nodejs.org/api/fs.html> The fs module also provide a way to do operations asynchronously and synchronously and also stream the reading and writing operations with in the file system.

### Example

Let us try to read two files and write them to another:

```
const { readFileSync, writeFileSync } = require('fs')
const x = readFileSync('./t1.txt', 'utf-8')
const y = readFileSync('./t2.txt', 'utf-8')

writeFileSync('./t3.txt', `${x}${y}`)
```

./t3.txt content:

```
t1t2
```

./t1.txt content:

```
t1
```

./t2.txt content:

```
t2
```

## Sync Vs Async

---

In Node.js, the `fs` module provides both synchronous and asynchronous versions of many of its methods. The difference between synchronous and asynchronous methods is that synchronous methods block execution of the program until they complete, while asynchronous methods do not.

When a synchronous method is called, the program will not continue to execute any further code until the method has completed. This can cause the program to become unresponsive, especially if the method is performing a long-running operation, such as reading a large file or accessing a remote resource over the network.

On the other hand, asynchronous methods use a callback function, which is executed once the operation is complete. The program can continue to execute other code while the asynchronous method is performing its operation in the background. This allows the program to remain responsive, even when performing long-running operations.

Here's an example of how the same operation would be performed using the synchronous and asynchronous versions of a method:

```
// Synchronous version
const data = fs.readFileSync('file.txt', 'utf8');
console.log(data);

// Asynchronous version
fs.readFile('file.txt', 'utf8', (err, data) => {
  if (err) throw err;
  console.log(data);
});
```

In general, it's a good idea to use asynchronous methods in Node.js, because they allow your program to remain responsive and handle many concurrent operations. However, there are some cases where

synchronous methods may be more appropriate, such as in simple scripts or during initialization of an application.

It is also important to note that, depending on the operation

Let us repeat the previous example but using *asynchronous* functions this time.

```
const { readFile, writeFile } = require('fs')

var x = ""
var y = ""

readFile('./t1.txt', 'utf-8', (err, result) => {
  if (err) {
    console.log(err);
    return
  }
  x = result

  readFile('./t2.txt', 'utf-8', (err, result) => {
    if (err) {
      console.log(err);
      return
    }
    y = result

    writeFile('./t3.txt', `${x}${y}`, (err) => {
      if (err) {
        console.log(err);
        return
      }
    })
  })
})
```

Notice here, how we start to have the *call back hell* which make the code less readable.

Here how to rewrite the previous code using *promises*:

A promise is commonly defined as a proxy for a value that will eventually become available.

```
const { readFile, writeFile } = require('fs')

const getText = (fName) => {
  return new Promise((resolve, reject) => {
    readFile(fName, 'utf-8', (err, result) => {
      if (err) {
        reject(err)
      }
    })
  })
}
```

```

    }
    else {
      resolve(result)
    }
  })
})
}

var x = ""
var y = ""

getText('./t1.txt')
  .then((result) => {
    x = result
    getText('./t2.txt')
      .then((result) => {
        y = result
        // console.log(result);
        writeFile('./t3.txt', `${x}${y}`, (err) => {if (err) console.log("write" +
err); })
      })
  })
  .catch((err) => { console.log("catch" + err); })

```

But the ~~callback~~ promise hell is also there. The workaround here is to return a promise from `getText` and have all the `thens` in one level:

```

const { readFile, writeFile } = require('fs')

const getText = (fName) => {
  return new Promise((resolve, reject) => {
    readFile(fName, 'utf-8', (err, result) => {
      if (err) {
        reject(err)
      }
      else {
        resolve(result)
      }
    })
  })
}

var x = ""
var y = ""

getText('./t1.txt')
  .then((result) => {
    x = result

```

```
    return getText('./t2.txt')
  })
  .then((result) => {
    y = result
    // console.log(result);
    writeFile('./t3.txt', `${x}${y}`, (err) => { if (err) console.log("write" +
err); })
  })
  .catch((err) => { console.log("catch" + err); })
```

Read More - <https://nodejs.dev/en/learn/understanding-javascript-promises/>

Let us move to the next approach using *async/await*:

```
const { readFile, writeFile, write } = require('fs')

const getText = (fName) => {
  return new Promise((resolve, reject) => {
    readFile(fName, 'utf-8', (err, result) => {
      if (err) {
        reject(err)
      }
      else {
        resolve(result)
      }
    })
  })
}

const start = async () => {
  var x = "rubbish"
  var y = "rubbish"
  try {
    x = await getText('./t1.txt')
    console.log(x);
    y = await getText('./t2.txt')
    console.log(y);
  }
  catch (err) {
    console.log(err);
  }
  writeFile('./t3.txt', `${x}${y}`, (err) => { if (err) console.log(err); })
}

start()
```

Or we can use the `promisify` function from the `util` library:

```
const { readFile, writeFile, write } = require('fs')
const util = require('util')
const readFilePromise = util.promisify(readFile)
const writeFilePromise = util.promisify(writeFile)

var x = "rubbish"
var y = "rubbish"

const start = async () => {
  try {
    x = await readFilePromise('./t1.txt', 'utf-8')
    y = await readFilePromise('./t2.txt', 'utf-8')
  }
  catch (err) {
    console.log(err);
  }
  await writeFilePromise('./t3.txt', `${x}${y}`, (err) => { if (err)
console.log(err); })
}

start()
```

Read more - <https://nodejs.dev/en/learn/differences-between-nodejs-and-the-browser/>

## HTTP Module

The `http` module is a built-in module in Node.js that provides a way to create HTTP servers and clients. It allows you to create server applications that can handle HTTP requests and serve HTTP responses, as well as client applications that can make HTTP requests to other servers. Here are a few examples of how you might use this module to create an HTTP server and client:

```
// Creating an HTTP server
const http = require('http');
const server = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello World!');
});
server.listen(3000);

// Creating an HTTP client
const http = require('http');
const options = {
```

```
    host: 'www.example.com',  
    path: '/'  
  };  
  const req = http.get(options, (res) => {  
    console.log(`statusCode: ${res.statusCode}`);  
    res.on('data', (d) => {  
      process.stdout.write(d);  
    });  
  });  
  req.on('error', (error) => {  
    console.error(error);  
  });
```

The `http.createServer()` method is used to create an HTTP server and the server listens on a specified port, in this example 3000, and the callback function passed to `createServer` will be called each time a client makes a request. The request and response objects passed to the callback function provide information about the client's request and allow you to send a response back to the client.

The `http.get()` method is used to create an HTTP client, and it makes a GET request to the specified host and path, and the callback function passed to `get` will be called when a response is received from the server. The response object passed to the callback function provides information about the server's response, such as the status code and the data received.

This is just a basic example, you can use http module to create a more complex web server and http client by using the request, response and other objects provided by the module. Also, you can use the https version of the module (https) to create a secure web server, which uses the SSL/TLS protocol to encrypt the data.

<https://nodejs.org/api/http.html>

## NPM

---

npm (short for Node Package Manager) is a package manager for Node.js that allows you to easily install and manage third-party libraries (also called packages) in your Node.js projects. It is the default package manager for Node.js, and it is included with the Node.js installation.

With npm, you can install packages that provide useful functionality for your Node.js projects, such as parsing JSON, connecting to databases, or handling HTTP requests. You can install packages globally (which makes them available for all Node.js projects) or locally (which makes them available only for the current project).

To use npm, you can run commands in the command line. Some common npm commands include:

- `npm install <package-name>` - Installs the specified package and its dependencies in the current project.
- `npm install <package-name> -g` - Installs the specified package globally on your system.
- `npm uninstall <package-name>` - Uninstalls the specified package from the current project.
- `npm update <package-name>` - Updates the specified package to the latest version.
- `npm list` - Lists all the packages installed in the current project.



When you install a package with npm, it will also download and install any other packages that the package depends on (these are called "dependencies"). Additionally, when you install a package, npm will also add the package to a file called `package.json` in your project, which keeps track of all the packages that your project depends on. This makes it easy for other developers to set up the same environment on their own machines and also to deploy your application.

In addition to being a package manager, npm also provides a way for developers to share their own packages with the community. This makes it easy for developers to reuse code and collaborate on projects.

Overall, npm plays a vital role in making the development with Node.js easy and efficient by allowing developers to easily share and reuse code, and handle dependencies.

## Package.json

---

`package.json` is a file that is used by npm (Node Package Manager) to manage the dependencies and other metadata for a Node.js project. It is typically located in the root directory of a project and contains information such as the project's name, version, and the packages that it depends on.

When you create a new Node.js project and initialize it with npm, a `package.json` file is automatically generated. You can also create one manually.

The `package.json` file contains a set of key-value pairs that describe your project and its dependencies. Some of the most important fields include:

- **name**: The name of the package, which needs to be unique among all npm packages.
- **version**: The version of the package, in the format `x.y.z` where x, y, and z are integers.
- **dependencies**: A map of package names to version numbers that lists the packages that the project depends on.
- **devDependencies**: Same as above, but these are packages that are needed only for development, not for production.
- **scripts**: A map of command-line scripts that can be run with npm (`npm run <script>`)
- **main**: The entry point of the package when it is loaded.
- **author**, **license** and **description**: metadata fields that gives an information about the package and the author

An example package.json file may look like this:

```
{
  "name": "my-project",
  "version": "1.0.0",
  "dependencies": {
    "request": "^2.88.0",
    "lodash": "^4.17.15"
  },
  "devDependencies": {
    "jest": "^24.9.0"
  },
  "scripts": {
    "start": "node index.js",
```

```
"test": "jest"
},
"main": "index.js",
"author": "Your Name",
"license": "MIT",
"description": "An example Node.js project"
}
```

When you run `npm install` in your project, npm will look at the `dependencies` and `devDependencies` fields in `package.json` and download the specified packages, as well as any packages that they depend on, into a folder called `node_modules`. And also when you run `npm install <package-name>` it will automatically add the package to the `dependencies` field.

It also includes other fields that can be added to provide additional information about the package. When you want to share your package with the community or install packages created by others, you would use the `package.json` file, it acts as a blueprint that lists all dependencies and metadata of your package.

## npm scripts

---

In `package.json`, the `scripts` field is used to define command-line scripts that can be run with npm. These scripts can be used to automate common tasks in your project, such as building, testing, and deploying.

The `scripts` field is a map of command names to commands. For example, you could have a script called "start" that starts your application, like this:

```
{
  "scripts": {
    "start": "node index.js"
  }
}
```

You can then run this script by running the command `npm run start` in the command line.

You can also chain multiple scripts together, and run multiple scripts at once using `&&` or `&` (depending on your operating system)

```
{
  "scripts": {
    "start": "npm run lint && node index.js"
  }
}
```

This script will first run `npm run lint` and after it completes, it will run `node index.js`.

You can also define scripts that run test, build, linting, deployment and many more.

The `npm run` command is a shorthand for `npx run-script` command, so you can also use `npx run-script <script-name>` to run the scripts defined in your package.json. You can also pass arguments to scripts by appending them after the script name like `npm run script-name -- --arg1 --arg2`. The arguments passed in this way can be accessed within the script using `process.argv`

By using npm scripts, you can easily automate repetitive tasks, and also make your development process more efficient. And it also allows other developers to understand how to run the project by looking at the scripts field in the package.json.

## Event Loop

---

In Node.js, the event loop is the mechanism that allows the program to handle asynchronous I/O operations. The event loop is a single-threaded loop that repeatedly checks the message queue for new messages (or events) and processes them one at a time.

When a Node.js program starts, it initializes the event loop and begins processing any messages that are already in the queue. While the event loop is running, it can receive new messages in one of two ways:

- Through the execution of a callback function, which is passed to an asynchronous function as an argument. The callback is added to the message queue once the asynchronous operation is completed.
- Through the use of `process.nextTick()` or `setImmediate()` which also adds the callbacks to the message queue. Once the event loop is empty (no more messages to process), the Node.js process will exit.

Node.js's event loop works in a single thread, that means it can process one task at a time. But it makes use of an OS feature called Asynchronous I/O or Non-blocking I/O which allows other operations to be performed while IO operations are in progress.

This helps Node.js to handle many connections at the same time with a single thread and also makes it efficient to handle a large number of concurrent connections and perform heavy I/O operations.

It is important to note that long-running computations can block the event loop and prevent it from processing other messages, so it's best to use a worker thread or process to perform heavy computations. And this is where the cluster module can come in handy by distributing the work to multiple cores of a system.

## Summary

---

Node.js is a JavaScript runtime that allows you to run JavaScript code outside of a web browser. It allows you to use JavaScript to write command-line tools, scripts, and other types of server-side applications.

Node.js uses the V8 JavaScript engine, which is also used by Google Chrome, to execute JavaScript code. It also includes a built-in library that provides functionality for things like working with the file system, making HTTP requests, and managing network connections.

One of the key features of Node.js is its support for asynchronous I/O (input/output). This allows Node.js to handle many connections at the same time with a single thread and makes it efficient to handle a large

number of concurrent connections and perform heavy I/O operations. It uses the Event Loop to handle the flow of execution and callbacks.

Node.js also has a package manager called npm (short for Node Package Manager) that allows you to easily install and manage third-party libraries (also called packages) in your Node.js projects. This makes it easy to share and reuse code among different projects, and allows developers to collaborate on projects more easily.

In simple terms, Node.js is a JavaScript runtime that allows you to run JavaScript code on a server (or computer), to make web server, build command line tools and perform other non-browser related tasks.