What is Node.js?

- Node.js is an open source server environment
- Node.js is free
- Node.js runs on various platforms (Windows, Linux, Unix, Mac OS X, etc.)
- Node.js uses JavaScript on the server

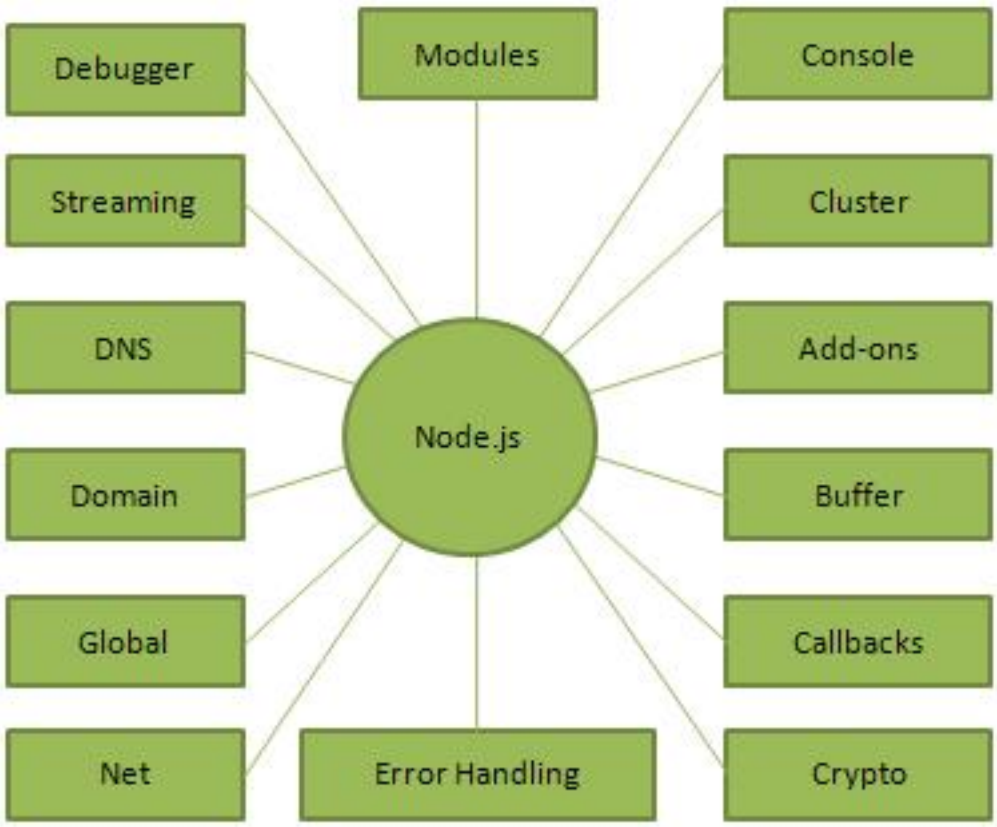
Features of Node.js

Following are some of the important features that make Node.js the first choice of software architects.

- **Asynchronous and Event Driven** – All APIs of Node.js library are asynchronous, that is, non-blocking. It essentially means a Node.js based server never waits for an API to return data. The server moves to the next API after calling it and a notification mechanism of Events of Node.js helps the server to get a response from the previous API call.
- **Very Fast** – Being built on Google Chrome's V8 JavaScript Engine, Node.js library is very fast in code execution.
- **Single Threaded but Highly Scalable** – Node.js uses a single threaded model with event looping. Event mechanism helps the server to respond in a non-blocking way and makes the server highly scalable as opposed to traditional servers which create limited threads to handle requests. Node.js uses a single threaded program and the same program can provide service to a much larger number of requests than traditional servers like Apache HTTP Server.
- **No Buffering** – Node.js applications never buffer any data. These applications simply output the data in chunks.
- **License** – Node.js is released under the MIT license.

Concepts

The following diagram depicts some important parts of Node.js which we will discuss in detail in the subsequent chapters.



Where to Use Node.js?

Following are the areas where Node.js is proving itself as a perfect technology partner.

- I/O bound Applications
- Data Streaming Applications
- Data Intensive Real-time Applications (DIRT)
- JSON APIs based Applications
- Single Page Applications

Where Not to Use Node.js?

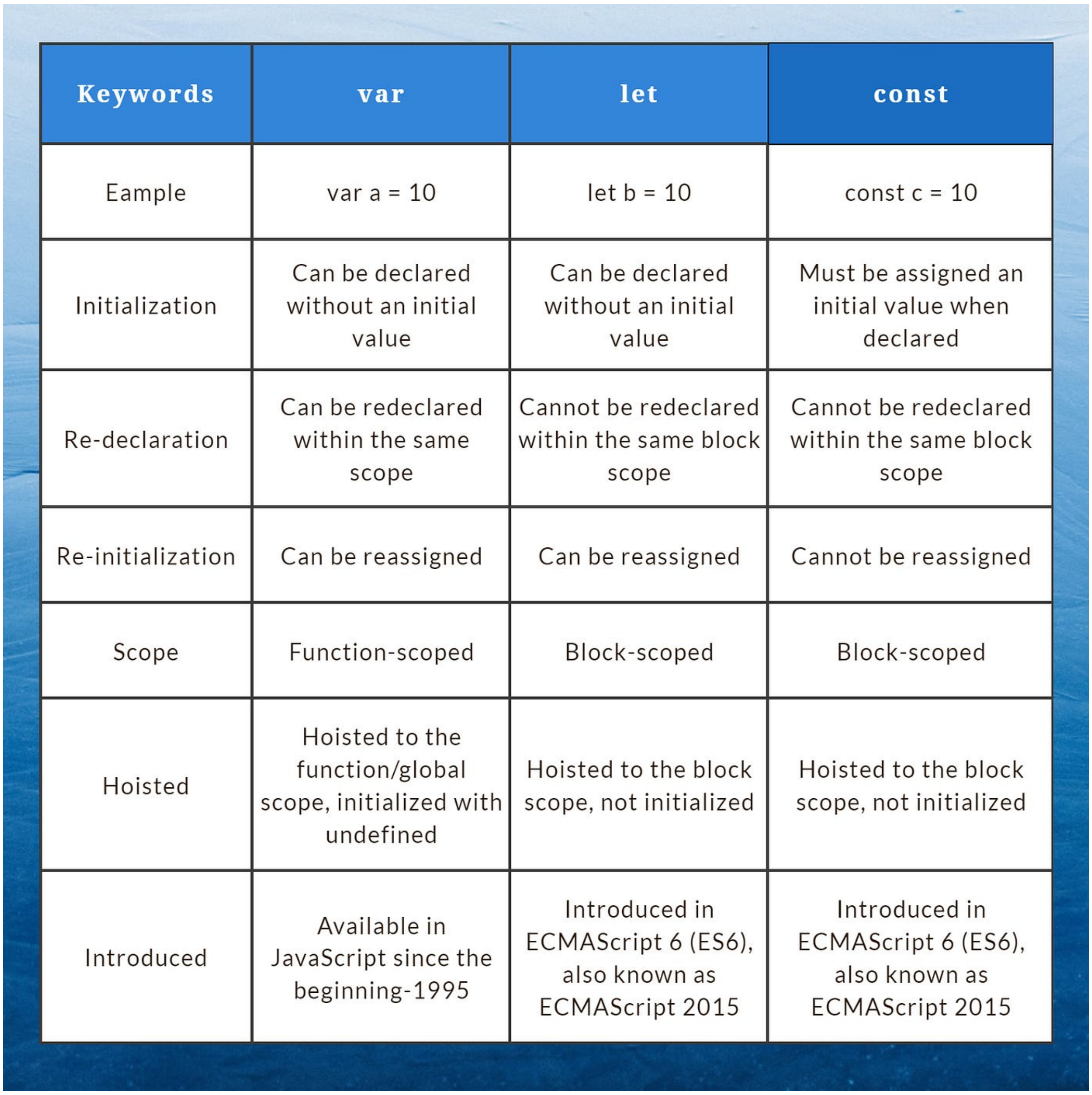
It is not advisable to use Node.js for CPU intensive applications.

How to Install NodeJS

Installing NodeJS is straightforward. If you already have Node installed in your machine, you can skip this section. If not, then follow along.

Here are the steps to download NodeJS on your machine:

1. Navigate to https://nodejs.org/
2. Download the LTS Version of NodeJS for your operating system
3. Run the installer and follow the installation wizard. Simply answer Yes to all the questions.
4. Once the installation is complete, open a new terminal or command prompt window and run the following command to verify that NodeJS is installed correctly: node -v. If you see the version of NodeJS printed in your terminal, Congratulations! You have now successfully installed NodeJS on your machine.

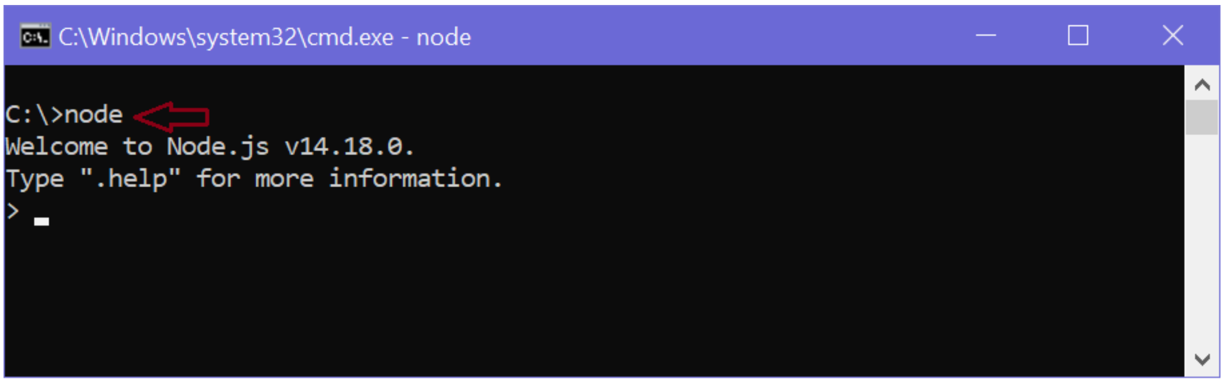


Keywords	var	let	const
Eample	var a = 10	let b = 10	const c = 10
Initialization	Can be declared without an initial value	Can be declared without an initial value	Must be assigned an initial value when declared
Re-declaration	Can be redeclared within the same scope	Cannot be redeclared within the same block scope	Cannot be redeclared within the same block scope
Re-initialization	Can be reassigned	Can be reassigned	Cannot be reassigned
Scope	Function-scoped	Block-scoped	Block-scoped
Hoisted	Hoisted to the function/global scope, initialized with undefined	Hoisted to the block scope, not initialized	Hoisted to the block scope, not initialized
Introduced	Available in JavaScript since the beginning-1995	Introduced in ECMAScript 6 (ES6), also known as ECMAScript 2015	Introduced in ECMAScript 6 (ES6), also known as ECMAScript 2015

Node.js Console/REPL

Node.js comes with virtual environment called REPL (aka Node shell). REPL stands for Read-Eval-Print-Loop. It is a quick and easy way to test simple Node.js/JavaScript code.

To launch the REPL (Node shell), open command prompt (in Windows) or terminal (in Mac or UNIX/Linux) and type *node* as shown below. It will change the prompt to > in Windows and MAC.

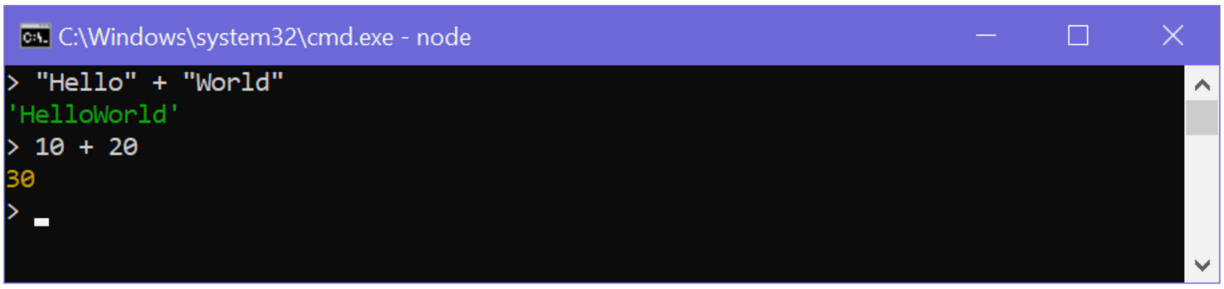


```
C:\>node
Welcome to Node.js v14.18.0.
Type ".help" for more information.
>
```

Launch Node.js REPL

You can now test pretty much any Node.js/JavaScript expression in REPL. 10 + 20 will display 30 immediately in new line.

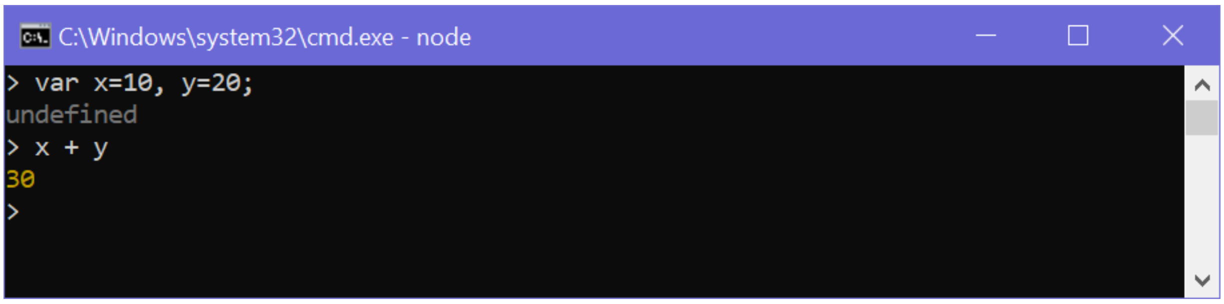
The + operator also concatenates strings as in browser's JavaScript.



```
> "Hello" + "World"
'HelloWorld'
> 10 + 20
30
>
```

Node.js Examples

You can also define variables and perform some operation on them.

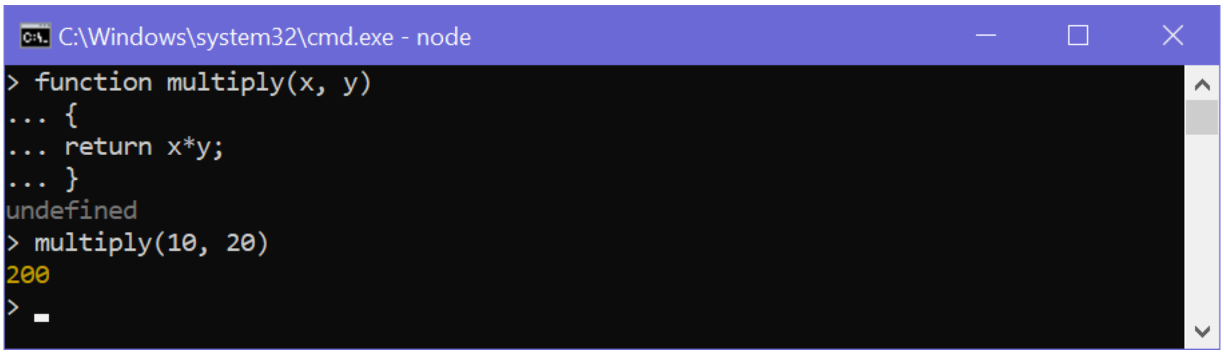


```
> var x=10, y=20;
undefined
> x + y
30
>
```

Define Variables on REPL

If you need to write multi line JavaScript expression or function then just press **Enter** whenever you want to write something in the next line as a continuation of your code. The REPL terminal will display three dots (...), it means you can continue on next line. Write .break to get out of continuity mode.

For example, you can define a function and execute it as shown below.



C:\Windows\system32\cmd.exe - node

```
> function multiply(x, y)
... {
... return x*y;
... }
undefined
> multiply(10, 20)
200
> 
```

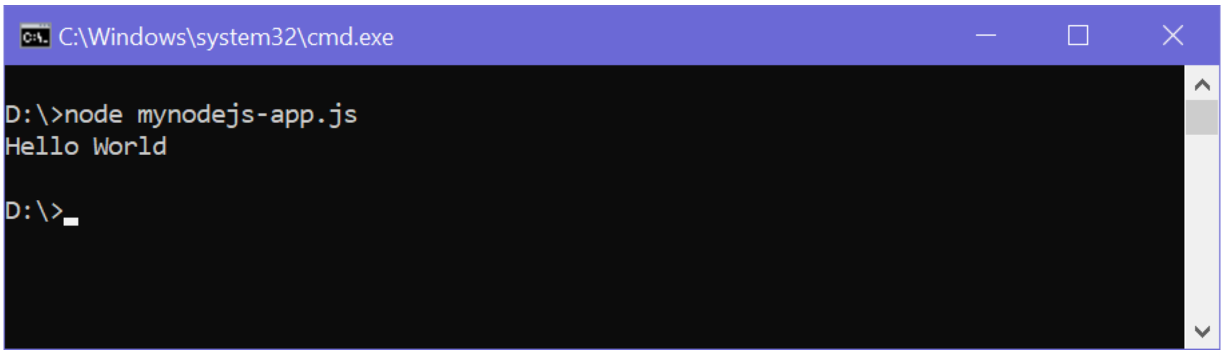
Node.js Example in REPL

You can execute an external JavaScript file by executing the node fileName command. For example, the following runs mynodejs-app.js on the command prompt/terminal and displays the result.

mynodejs-app.js Copy

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

Now, you can execute mynodejs-app from command prompt as shown below.

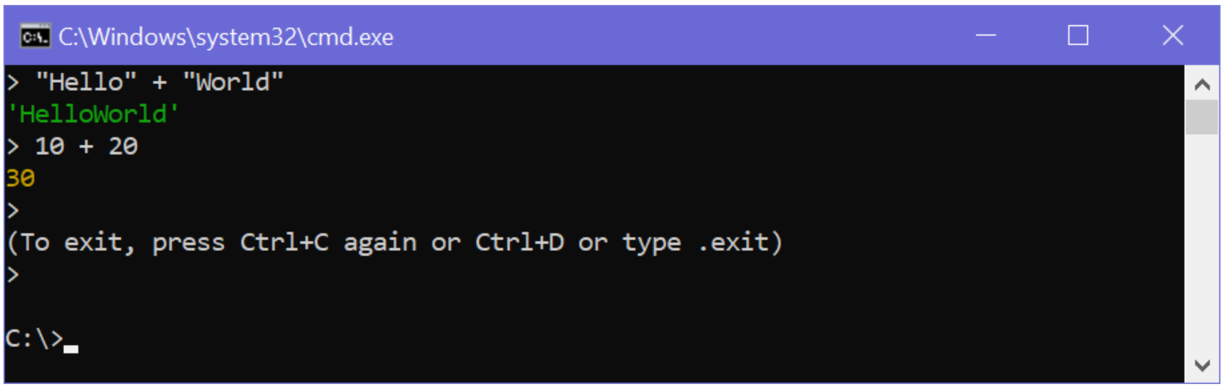


C:\Windows\system32\cmd.exe

```
D:\>node mynodejs-app.js
Hello World

D:\>
```

Run External JavaScript file

To exit from the REPL terminal, press Ctrl + C twice or write .exit and press Enter.



```
> "Hello" + "World"
'HelloWorld'
> 10 + 20
30
>
(To exit, press Ctrl+C again or Ctrl+D or type .exit)
>

C:\>_
```

Quit from REPL

Thus, you can execute any Node.js/JavaScript code in the node shell (REPL). This will give you a result which is similar to the one you will get in the console of Google Chrome browser.

The following table lists important REPL commands.

REPL Command	Description
.help	Display help on all the commands
tab Keys	Display the list of all commands.
Up/Down Keys	See previous commands applied in REPL.
.save filename	Save current Node REPL session to a file.
.load filename	Load the specified file in the current Node REPL session.
ctrl + c	Terminate the current command.
ctrl + c (twice)	Exit from the REPL.
ctrl + d	Exit from the REPL.
.break	Exit from multiline expression.
.clear	Exit from multiline expression.

Global Variables

Let's start this article by learning about some variables present in NodeJS called Global Variables. These are basically variables which store some data and can be accessed from anywhere in your code – doesn't matter how deeply nested the code is.

You should know about these commonly used Global variables:

- __dirname: This variable stores the path to the current working directory.
- __filename: This variable stores the path to the current working file.

Create a new file called app.js and open up a new integrated VS Code Terminal.

Paste the following code in the app.js file and save it:

```
// __dirname Global Variable
console.log(__dirname);

// __filename Global Variable
console.log(__filename);
```

To run this code using Node, type in the following command in the terminal and press Enter: node app.js. You will see the absolute path to the present working directory and the path to the current file is printed in the terminal. This is what the output looks like in my case:

```
C:\Desktop\NodeJSTut
C:\Desktop\NodeJSTut\app.js
```

You can go ahead and create your own global variables which can be accessed from anywhere in your code. You can do so, like this:

```
// Define a global variable in NodeJS
global.myVariable = 'Hello World';

// Access the global variable
console.log(myVariable); // Output: Hello World
```

Primitive Types

Node.js includes following primitive types:

- String
- Number
- Boolean
- Undefined
- Null
- RegExp

Object Literal

Object literal syntax is same as browser's JavaScript.

```
var obj = {
   authorName: 'Ryan Dahl',
   language: 'Node.js'
}
```

Functions

Functions are first class citizens in Node's JavaScript, similar to the browser's JavaScript. A function can have attributes and properties also. It can be treated like a class in JavaScript.

```
function Display(x) {
   console.log(x);
}

Display(100);
```

Node.js Module

Module in Node.js is a simple or complex functionality organized in single or multiple JavaScript files which can be reused throughout the Node.js application.

Each module in Node.js has its own context, so it cannot interfere with other modules or pollute global scope. Also, each module can be placed in a separate .js file under a separate folder.

Node.js Module Types

Node.js includes three types of modules:

1. Core Modules
2. Local Modules
3. Third Party Modules

Node.js Core Modules

Node.js is a light weight framework. The core modules include bare minimum functionalities of Node.js. These core modules are compiled into its binary distribution and load automatically when Node.js process starts. However, you need to import the core module first in order to use it in your application.

The following table lists some of the important core modules in Node.js.

Core Module	Description
http	http module includes classes, methods and events to create Node.js http server.
url	url module includes methods for URL resolution and parsing.
querystring	querystring module includes methods to deal with query string.
path	path module includes methods to deal with file paths.
fs	fs module includes classes, methods, and events to work with file I/O.
util	util module includes utility functions useful for programmers.

Loading Core Modules

In order to use Node.js core or NPM modules, you first need to import it using require() function as shown below.

```
var module = require('module_name');
```

As per above syntax, specify the module name in the require() function. The require() function will return an object, function, property or any other JavaScript type, depending on what the specified module returns.

The following example demonstrates how to use Node.js http module to create a web server.

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

var server = http.createServer(function(req, res){

  //write code here

});

server.listen(5000);
```

In the above example, require() function returns an object because http module returns its functionality as an object, you can then use its properties and methods using dot notation e.g. http.createServer().

Different Servers in Node.js

Creating Server using 'http' Module:

Import http module: Import http module and store returned HTTP instance into a variable.

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

Creating and Binding Server: Create a server instance using the createServer() method and bind it to some port using listen() method.

```
const server = http.createServer().listen(port)
```

port <Number>: Ports are in the range 1024 to 65535 containing both registered and Dynamic ports.

Another Way

Creating Server using 'http' Module:

```
const http = require(“http");
```

Creating and Binding Server: Create a server instance using the createServer() method and bind it to some port using listen() method.

```
const server = https.createServer(options,
        onResponseCallback).listen(port)
```

Parameter: This method accepts three parameters as mentioned above and described below:

options <key, certi>**:** It includes the key and certificate passed.
onResponseCallback <Callback>: It is a callback function that is called in response of createServer.
port <Number>**:** Ports are in the range 1024 to 65535 containing both registered and Dynamic ports.

Node.js Local Module

Local modules are modules created locally in your Node.js application. These modules include different functionalities of your application in separate files and folders. You can also package it and distribute it via NPM, so that Node.js community can use it. For example, if you need to connect to MongoDB and fetch data then you can create a module for it, which can be reused in your application.

Writing Simple Module

Let's write simple logging module which logs the information, warning or error to the console.

In Node.js, module should be placed in a separate JavaScript file. So, create a Log.js file and write the following code in it.

```
var log = {
        info: function (info) {
           console.log('Info: ' + info);
        },
        warning:function (warning) {
           console.log('Warning: ' + warning);
        },
        error:function (error) {
           console.log('Error: ' + error);
        }
    };

module.exports = log
```

In the above example of logging module, we have created an object with three functions - info(), warning() and error(). At the end, we have assigned this object to **module.exports**. The module.exports in the above example exposes a log object as a module.

The *module.exports* is a special object which is included in every JS file in the Node.js application by default. Use **module.exports** or **exports** to expose a function, object or variable as a module in Node.js.

Loading Local Module

To use local modules in your application, you need to load it using require() function in the same way as core module. However, you need to specify the path of JavaScript file of the module.

The following example demonstrates how to use the above logging module contained in Log.js.

app.js

```
var myLogModule = require('./Log.js');

myLogModule.info('Node.js started');
```

Run the above example

```
C:\> node app.js
Info: Node.js started
```

Export Module in Node.js

Here, you will learn how to expose different types as a module using module.exports.

The module.exports is a special object which is included in every JavaScript file in the Node.js application by default. The module is a variable that represents the current module, and exports is an object that will be exposed as a module. So, whatever you assign to module.exports will be exposed as a module.

Export Literals

As mentioned above, exports is an object. So it exposes whatever you assigned to it as a module. For example, if you assign a string literal then it will expose that string literal as a module

Message.js

```
module.exports = 'Hello world';
```

app.js

```
var msg = require('./Message.js');

console.log(msg);
```

Run the above example

```
C:\> node app.js
Hello World
```

Note:
You must specify ./ as a path of root folder to import a local module. However, you do not need to specify the path to import Node.js core modules or NPM modules in the require() function.

What is NPM?

NPM is a package manager for Node.js packages, or modules if you like.

www.npmjs.com hosts thousands of free packages to download and use.

The NPM program is installed on your computer when you install Node.js

What is a Package?

A package in Node.js contains all the files you need for a module.

Modules are JavaScript libraries you can include in your project.

Check the version of NPM in the command terminal –

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

In case you have an older version of NPM, you need to update it to the latest version using the following command.

```
npm install -g npm@latest
```

Install Package Locally

There is a simple syntax to install any Node.js module –

```
npm install <Module Name>
```

For example, following is the command to install a famous Node.js web framework module called express –

```
npm install express
```

Now you can use this module in your js file as following –

```
var express = require('express');
```

The local mode installation of a package refers to the package installation in node_modules directory lying in the folder where Node application is present. Locally deployed packages are accessible via require() method. Use --save at the end of the install command to add dependency entry into package.json of your application.

The package.json file is a JSON file that is used to manage dependencies in Node.js projects. It contains information about the project, such as its name, version, and dependencies. The package.json file is used by the npm package manager to install and manage dependencies.

The package.json file is typically located in the root directory of a Node.js project. It can be created by running the npm init command.

Create a new folder for a new Node.js project, and run npm init command inside it –

```
D:\nodejs\newnodeapp> npm init -y
```

D:\nodejs\NewNodeApp\package.json – will be created

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

```
D:\nodejs\newnodeapp>npm install express –save
```

```
{
  "name": "newnodeapp",
  "version": "1.0.0",
  "description": "Test Node.js App",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "test",
    "nodejs"
  ],
  "author": "mvl",
  "license": "ISC",
  "dependencies": {
    "express": "^4.18.2"
  }
}
```

The express package code will be placed inside the node_modules subfolder of the package folder.

- --save-dev installs and adds the entry to the package.json file devDependencies
- --no-save installs but does not add the entry to the package.json file dependencies
- --save-optional installs and adds the entry to the package.json file optionalDependencies
- --no-optional will prevent optional dependencies from being installed

Shorthands of the flags can also be used –

- -S: --save
- -D: --save-dev
- -O: —save-optional

Install Package Globally

Globally installed packages/dependencies are stored in system directory. Such dependencies can be used in CLI (Command Line Interface) function of any node.js but cannot be imported using require() in Node application directly.

npm install express -g
This will produce a similar result but the module will be installed globally. On Linux, the global packages are placed in /usr/lib/node_modules folder, while for Windows, the path is C:\Users\your-username\AppData\Roaming\npm\node_modules.

Update Package

To update the package installed locally in your Node.js project, open the command prompt or terminal in your project project folder and write the following update command.

npm update <package name>

```
D:\nodejs\newnodeapp> npm update express

up to date, audited 63 packages in 2s
```

Uninstall Packages

To uninstall a package from your project's dependencies, use the following command to remove a local package from your project.

npm uninstall <package name>

```
D:\nodejs\newnodeapp> npm uninstall express
removed 62 packages, and audited 1 package in 603ms
found 0 vulnerabilities
```

The package entry will also be removed from the list of dependencies in the package.json file.

npx - Node Package Expert

In the world of Node.js, many developers use npm (Node Package Manager) to add and manage packages in their projects. However, sometimes you may want to quickly run specific packages or test different versions. That's where npx (Node Package Runner) comes into play.

How Does It Work?

npx uses a temporary folder to run packages without installing them. The process goes like this:

- First, npx looks for the package to be run with the npx command and the optional specified version.
- Then, it downloads the package and its dependencies into a temporary folder. This process is similar to how packages and dependencies are usually installed with npm install.
- The package's script file (e.g., cli.js) is located in the temporary folder and executed. During this process, the package uses its dependencies in the temporary folder and runs in isolation from the rest of the system.
- Once the package's operation is complete, npx deletes the temporary folder and all its contents. As a result, the package and its dependencies don't stay on your system and waste disk space.

Advantages

- Running packages directly without local or global installation
- Flexibility to work with different package versions: With npx, you can run any package version you want. This prevents package version conflicts and allows you to use the latest package version each time.

Usage Examples

- **create-react-app**: Let's consider the create-react-app package, which is used to create React applications. Instead of installing it globally, you can run this tool directly with npx:

```
npx create-react-app my-app
```

This command downloads the create-react-app package and uses it to create a new React application. Once the process is complete, a new folder called my-app is created.

2. **http-server**: The http-server package allows you to quickly serve your static files. Instead of installing it globally, you can run it using npx:

```
npx http-server
```

This command serves the files in the current directory over an HTTP server.

Include Modules

To include a module, use the require() function with the name of the module:

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

// Create a server
http.createServer((request, response) => {

    // Sends a chunk of the response body
    response.write('Hello World!');

// Signals the server that all of the response headers and bod have been sent
    response.end();
}).listen(3000); // Server listening on port 3000

console.log("Server started on port 3000");
```

Step to run this program: Run this **max.js** file using the below command:

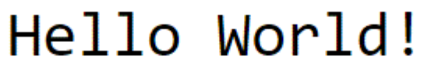
```
node max.js
```

Output

```
Bhuvaneswaris-MacBook-Pro:JQuery bhuvaneswarigokul$ node app.js
Server started on port 3000
```

In the Crome http://localhost:3000/

Browser Output:



```
Hello World!
```

Let's see how we can make our own modules. For this, we are going to write some code where we will be defining a function called sayHello() in a file called hello.js. This function will accept a name as the parameter and simply print a greeting message in the console.

We will then import it in another file called app.js and use it there. How interesting, right 😂? Let's check out the code:

This is the code in hello.js file:

```
function sayHello(name){
    console.log(`Hello ${name}`);
}

module.exports = sayHello
```

This is the code in app.js file:

```
const sayHello = require('./hello.js');

sayHello('John');
sayHello('Peter');
sayHello('Rohit');
```

The app.js file imports the sayHello() function from hello.js and stores it in the sayHello variable. To import something from a module, we use the require() method which accepts the path to the module. Now we can simply invoke the variable and pass a name as a parameter. Running the code in app.js file will produce the following output:

```
Hello John
Hello Peter
Hello Rohit
```

Short Note on module.exports

module.exports is a special object in NodeJS that allows you to export functions, objects, or values from a module, so that other modules can access and use them. Here's an example of how to use module.exports to export a function from a module:

```
// myModule.js

function myFunction() {
  console.log('Hello from myFunction!');
}

module.exports = myFunction;
```

In this example, we define a function myFunction and then export it using module.exports. Other modules can now require this module and use the exported function:

```
// app.js
```

```
const myFunction = require('./myModule');

myFunction(); // logs 'Hello from myFunction!’

// module.js

function myFunction() {
  console.log('Hello from myFunction!');
}

function myFunction2() {
  console.log('Hello from myFunction2!');
}

// First Export
module.exports = myFunction;

// Second Export
module.exports = myFunction2;
```

This problem can be solved if you assign module.exports to an object which contains all the functions you want to export, like this:

```
// myModule.js

function myFunction1() {
  console.log('Hello from myFunction1!');
}

function myFunction2() {
  console.log('Hello from myFunction2!');
}

module.exports = {
  foo: 'bar',
  myFunction1: myFunction1,
  myFunction2: myFunction2
};
```

In this example, we export an object with three properties: foo, myFunction1, and myFunction2. Other modules can require this module and access these properties:

```
// app.js

const myModule = require('./myModule');

console.log(myModule.foo); // logs 'bar'
myModule.myFunction1(); // logs 'Hello from myFunction1!'
myModule.myFunction2(); // logs 'Hello from myFunction2!'
```

What is Callback?

Callback is an asynchronous equivalent for a function. A callback function is called at the completion of a given task. Node makes heavy use of callbacks. All the APIs of Node are written in such a way that they support callbacks.

Blocking Code Example

Create a text file named **input.txt** with the following content –

```
Imarticus Learning is there
to teach the world in simple and easy way!!!!!
```

Create a js file named **main.js** with the following code –

```
var fs = require("fs");
var data = fs.readFileSync('input.txt');

console.log(data.toString());
console.log("Program Ended");
```

Now run the main.js to see the result –

```
$ node main.js
```

Verify the Output.

```
Imarticus Learning is there
to teach the world in simple and easy way!!!!!
Program Ended
```

Non-Blocking Code Example

Create a text file named input.txt with the following content.

```
Imarticus Learning is there
to teach the world in simple and easy way!!!!!
```

Update main.js to have the following code –

```
var fs = require("fs");

fs.readFile('input.txt', function (err, data) {
   if (err) return console.error(err);
   console.log(data.toString());
});

console.log("Program Ended");
```

Now run the main.js to see the result –

```
$ node main.js
```

Verify the Output.

```
Program Ended
Imarticus Learning is there
to teach the world in simple and easy way!!!!!
```

These two examples explain the concept of blocking and non-blocking calls.

- The first example shows that the program blocks until it reads the file and then only it proceeds to end the program.
- The second example shows that the program does not wait for file reading and proceeds to print "Program Ended" and at the same time, the program without blocking continues reading the file.

The FS Module
This module helps you with file handling operations such as:

- Reading a file (sync or async way)
- Writing to a file (sync or async way)
- Deleting a file
- Reading the contents of a director
- Renaming a file
- Watching for changes in a file, and much more

How to create a directory using fs.mkdir()

The fs.mkdir() function in Node.js is used to create a new directory. It takes two arguments: the path of the directory to be created and an optional callback function that gets executed when the operation is complete.

- **path**: Here, path refers to the location where you want to create a new folder. This can be an absolute or a relative path. In my case, the path to the present working directory (the folder I am currently in), is: C:\Desktop\NodeJSTut. So, Let's create a folder in the NodeJSTut directory called myFolder.
- **callback function:** The purpose of the callback function is to notify that the directory creation process has completed. This is necessary because the fs.mkdir() function is asynchronous, meaning that it does not block the execution of the rest of the code while the operation is in progress. Instead, it immediately returns control to the callback function, allowing it to continue executing other tasks.

```
// Import fs module
const fs = require('fs');

// Present Working Directory: C:\Desktop\NodeJSTut
// Making a new directory called ./myFolder:

fs.mkdir('./myFolder', (err) => {
   if(err){
       console.log(err);
   } else{
       console.log('Folder Created Successfully');
   }
})
```

How to create and write to a file asynchronously using fs.writeFile()

After the myFolder directory is created successfully, it's time to create a file and write something to it by using the fs module.

writeFile() is a method provided by the fs (file system) module in Node.js. It is used to write data to a file asynchronously. The method takes three arguments:

1. The **path** of the file to write to (including the file name and extension)

2. The **data** to write to the file (as a string or buffer)
3. An optional **callback function** that is called once the write operation is complete or an error occurs during the write operation.

Below is the code where we create a new file called myFile.txt in the myFolder directory and write this data to it: Hi,this is newFile.txt.

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

const data = "Hi,this is newFile.txt";

fs.writeFile('./myFolder/myFile.txt', data, (err)=> {
   if(err){
      console.log(err);
      return;
   } else {
       console.log('Writen to file successfully!');
   }
})
```

The problem with this code is: when you run the same code multiple times, it erases the previous data that is already present in newFile.txt and writes the data to it.

```
const fs = require('fs');
const data = 'Hi,this is newFile.txt';

fs.writeFile('./myFolder/myFile.txt', data, {flag: 'a'}, (err) => {
   if(err){
      console.log(err);
      return;
   } else {
       console.log('Writen to file successfully!');
   }
})
```

Once you run the above code again and again, you will see that the myFile.txt has the value of the data variable written to it multiple times. This is because the object (3rd parameter): {flag: 'a'} indicates the writeFile() method to append the data at the end of the file instead of erasing the previous data present in it.

How to read the contents of a directory using fs.readdir()

If you have been following along until now, you will see that we currently have 2 files in the myFolder directory, i.e, myFile.txt and myFileSync.txt.

The readdir() function accepts 2 parameters:

- The **path** of the folder whose contents are to be read.
- **Callback function** which gets executed once the operation is completed or if any error occurs during the operation. This function accepts 2 parameters: The first one which accepts the error object (if any error occurs) and the second parameter which accepts an array of the various files and folders present in the directory whose path has been provided.

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

fs.readdir('./myFolder', (err, files) => {
   if(err){
       console.log(err);
      return;
   }
   console.log('Directory read successfully! Here are the files:');
   console.log(files);
})
```

```
Output:
[ 'myFile.txt', 'myFileSync.txt' ]
```

How to rename a file using fs.rename()
Here's the syntax for the fs.rename() method:

```
fs.rename(oldPath, newPath, callback);
```

where:

- oldPath (string) - The current file path
- newPath (string) - The new file path
- callback (function) - A callback function to be executed when the renaming is complete. This function takes an error object as its only parameter.

Let's rename the newFile.txt file to newFileAsync.txt:

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

fs.rename('./newFolder/newFile.txt', './newFolder/newFileAsync.txt', (err)=>{
   if(err){
        console.log(err);
      return;
   }
   console.log('File renamed successfully!')
})
```

Once you run the above code, you will see that the newFile.txt gets renamed to newFileAsync.txt.

How to delete a file using fs.unlink()

Last but not the least, we have the fs.unlink() function which is used to delete a file.
Running the following code deletes the newFileSync.txt file present in the myFolder directory:

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

fs.unlink('./myFolder/myFileSync.txt', (err) => {
   if(err){
      console.log(err);
      return;
   }
   console.log('File Deleted Successfully!')
})
```

Introduction to the node.js OS module

To use the `os` module, you include it as follows:

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

The `os` module provides you with many useful properties and methods for interacting with the operating system and server.

For example, the `os.EOL` property returns the platform-specific end-of-line marker.

The `os.EOL` property returns `\r\n` on Windows and `\n` on Linux or macOS.

Getting the current Operating System information

The `os` module provides you with some useful methods to retrieve the operating system of the server. For example:

```
let currentOS = {
    name: os.type(),
    architecture: os.arch(),
    platform: os.platform(),
    release: os.release(),
    version: os.version()
};

console.log(currentOS);
```

Output:

```
{
    name: 'Windows_NT',
    architecture: 'x64',
    platform: 'win32',
    release: '10.0.18362',
    version: 'Windows 10 Pro'
}
```

Checking server uptime

The `os.uptime()` method returns the system uptime in seconds. For example:

```
console.log(`The server has been up for ${os.uptime()} seconds.`);
```

Output:

```
The server has been up for 44203 seconds.
```

Getting the current user information

The `os.userInfo()` method returns the information about the current user:

```
console.log(os.userInfo());
```

Output:

```
{
  uid: -1,
  gid: -1,
  username: 'john',
  homedir: 'C:\\Users\\john',
  shell: null
}
```

Getting the server hardware information

The `os.totalmem()` method returns the total memory in bytes of the server:

```
let totalMem = os.totalmem();
console.log(totalMem);
```

Output:

```
8464977920
```

To get the amount of free memory in bytes, you use the `os.freemem()` method:

```
let freeMem = os.freemem();
console.log(freeMem);
```

Output:

```
1535258624
```

To get the information of the CPU, you use the `os.cpus()` method:

```
os.cpus();
```

The following example shows the model and speed of the server's CPU:

```
const { model, speed } = os.cpus()[0];

console.log(`Model: ${model}`);
console.log(`Speed (MHz): ${speed}`);
```

Retrieving network interface information

The `os.networkInterfaces()` method returns an object that contains network interface information.

Each key in the returned object identifies a network interface:

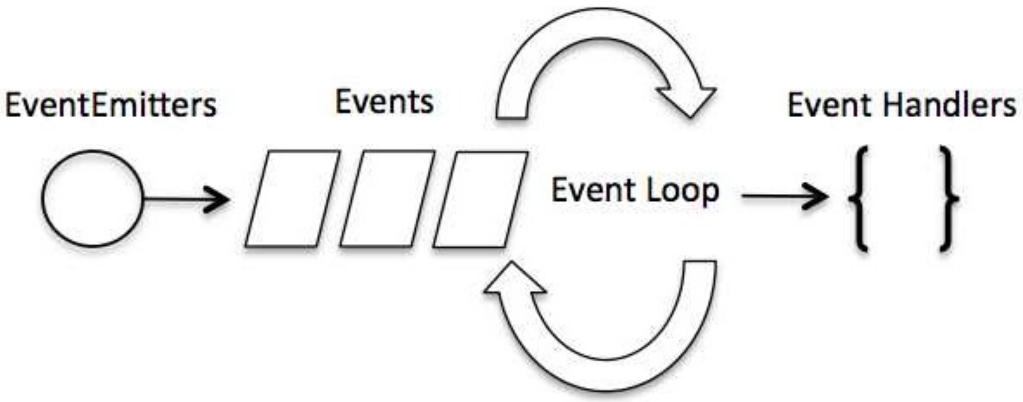
```
console.log(os.networkInterfaces());
```

Output:

```
os.networkInterfaces();
```

Event-Driven Programming

Event-driven programming is a programming paradigm where program flow is largely determined by events or user actions, rather than by the program's logic.



To implement Event Driven Programming in NodeJS, You need to remember 2 things:

- There is a function called emit() which causes an event to occur. For example, emit('myEvent') emits/causes an event called myEvent.
- There is another function called on() which is used to listen for a particular event and when this event occurs, the on() method executes a listener function in response to it. For example, Consider this code: on('myEvent', myFunction): Here we are listening for an event called myEvent and when this event takes place, we run the myFunction listener function in response to it.

In the below code, we are listening for the userJoined event and once this event takes place, we run the welcomeUser() function using the on() method and we emit the userJoined event using the emit() method:

```
// Importing 'events' module and creating an instance of the EventEmitter
Class

const EventEmitter = require('events');
const myEmitter = new EventEmitter();

// Listener Function - welcomeUser()
const welcomeUser = () => {
   console.log('Hi There, Welcome to the server!');
}

// Listening for the userJoined event using the on() method
myEmitter.on('userJoined', welcomeUser);

// Emitting the userJoined event using the emit() method
myEmitter.emit('userJoined');
```

Points to Note:
There are 3 points you should note while working with events in Node. Each point in shown in action in the corresponding code snippets:

- There can be multiple on()'s for a single emit():

```
// Importing `events` module and creating an instance of EventEmitter class
const EventEmitter = require('events');
const myEmitter = new EventEmitter();
```

```
// Listener Function 1: sayHello
const sayHello = () => {
	console.log('Hello User');
}

// Listener Function 2: sayHi
const sayHi = () => {
	console.log('Hi User');
}

// Listener Function 3: greetNewYear
const greetNewYear = () => {
	console.log('Happy New Year!');
}

// Subscribing to `userJoined` event
myEmitter.on('userJoined', sayHello);
myEmitter.on('userJoined', sayHi);
myEmitter.on('userJoined', greetNewYear);

// Emiting the `userJoined` Event
myEmitter.emit(‘userJoined');
```

Output:

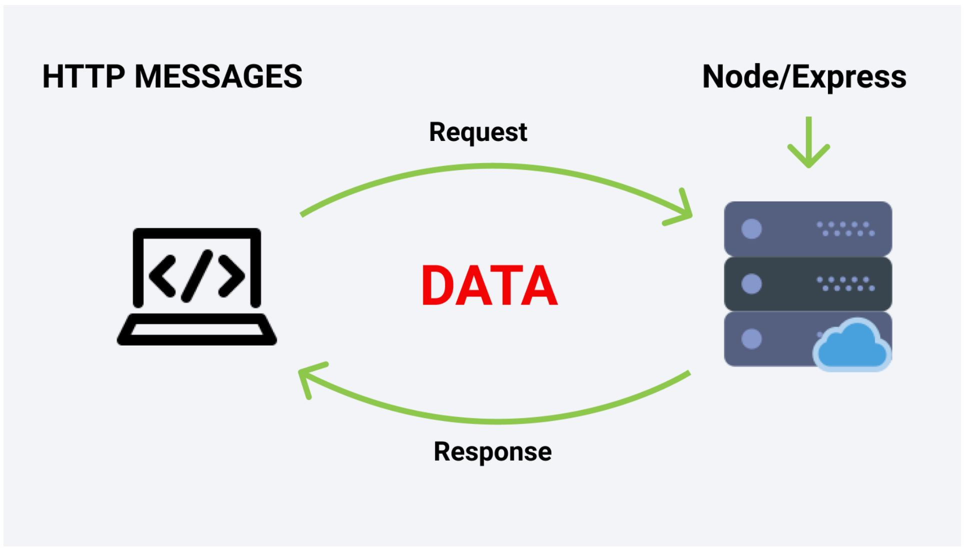
```
Hello User
Hi User
Happy New Year!
```

The extra parameters mentioned in the emit() function, gets passed as parameters to all the listener functions which will run in response to the birthdayEvent. Therefore John and 24 gets passed as parameters to the greetBirthday() function.

```
const EventEmitter = require('events');
const myEmitter = new EventEmitter();

// Listener function
const greetBirthday = (name, newAge) => {
   // name = John
   // newAge = 24
   console.log(`Happy Birthday ${name}. You are now ${newAge}!`);
}
```

```
// Listening for the birthdayEvent
myEmitter.on('birthdayEvent', greetBirthday);

// Emitting the birthdayEvent with some extra parameters
myEmitter.emit('birthdayEvent', 'John', '24');

Output:
Happy Birthday John, You are now 24!.

const EventEmitter = require('events');
const myEmitter = new EventEmitter();

// Listener Function 1 - sayHi
const sayHi = () => {
	console.log('Hi User');
}

// Listener Function 2 - sayHello
const sayHello = () => {
	console.log('Hello User');
}

// Registering sayHi function as listener
myEmitter.on('userJoined', sayHi);

// Emitting the event
myEmitter.emit('userJoined');

// Registering sayHello function as listener
myEmitter.on('userJoined', sayHello);

Output:
Hi User
```

The HTTP Module
HTTP stands for Hypertext Transfer Protocol. It is used to transfer data over the internet which allows communication between clients and servers.
The client sends a request to the server in the form of a URL with some additional information, such as headers and query parameters.
The server processes the request, performs necessary operations, and sends a response back to the client. The response contains a status code, headers, and the response body with the requested data.

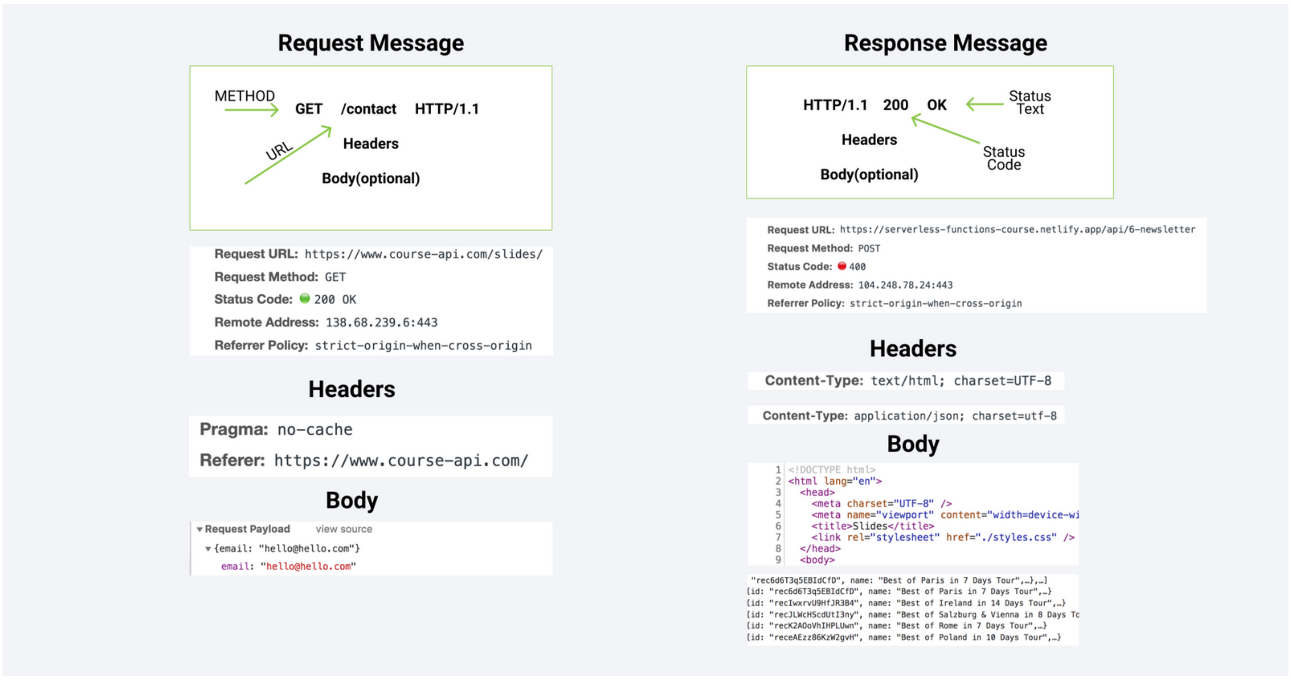


Components Of Request-Response

Both the Request (sent by client to the server) and the Response (sent by server to the client) comprises of 3 parts:

1. **The Status Line**: This is the first line of the request or response. It contains information about the message, such as the method used, URL, protocol version, and so on.
2. **The Header**: This is a collection of key-value pairs, separated by colon.
   The headers include additional information about the message such as the content type, content length, caching information, and so on.
3. **The Body**: The Body contains the actual data being sent or received. In the case of requests, it might contain form data or query parameters. In the case of responses, it could be HTML, JSON, XML, or any other data format.

The 3 components of a Request and Response are described in much more detail in the below image:



```
Request URL: https://www.course-api.com/slides/
Request Method: GET
Status Code: ● 200 OK
Remote Address: 138.68.239.6:443
Referrer Policy: strict-origin-when-cross-origin
```

```
Pragma: no-cache
Referer: https://www.course-api.com/
```

```
▾ {email: "hello@hello.com"}
    email: "hello@hello.com"
```

```
Request URL: https://serverless-functions-course.netlify.app/api/6-newsletter
Request Method: POST
Status Code: ● 400
Remote Address: 104.248.78.24:443
Referrer Policy: strict-origin-when-cross-origin
```

```
Content-Type: text/html; charset=UTF-8
```

```
Content-Type: application/json; charset=utf-8
```

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-wi
    <title>Slides</title>
    <link rel="stylesheet" href="./styles.css" />
  </head>
  <body>
```

```
 "rec6d6T3q5EBIdCfD", name: "Best of Paris in 7 Days Tour",…},…]
{id: "rec6d6T3q5EBIdCfD", name: "Best of Paris in 7 Days Tour",…}
{id: "recIwxrvU9HfJR3B4", name: "Best of Ireland in 14 Days Tour",…}
{id: "recJLWcHScdUtI3ny", name: "Best of Salzburg & Vienna in 8 Days To
{id: "recK2AOoVhIHPLUwn", name: "Best of Rome in 7 Days Tour",…}
{id: "receAEzz86KzW2gvH", name: "Best of Poland in 10 Days Tour",…}
```

What are HTTP Methods?

HTTP methods, also known as HTTP verbs, are actions that a Client can perform on a Server. The 4 HTTP Methods are:

- GET: Retrieves a resource from the server
- POST: Inserts a resource in the server
- PUT: Updates an existing resource in the server
- DELETE: Deletes a resource from the server

1. **GET:** Retrieves a resource from the server
   When you enter http://www.google.com in your web browser's address bar and press enter, your browser sends a HTTP GET request to the Google server asking for the HTML content of the Google homepage. That's then rendered and displayed by your browser.

2. **POST**: Inserts a resource in the server
   Imagine you're filling out a registration form to create an account on Google. When you submit the form, your browser sends a POST request to Google's server with the data you typed in the form fields like: Username, Age, Birthdate, Address, Phone Number, Email, Gender and so on.

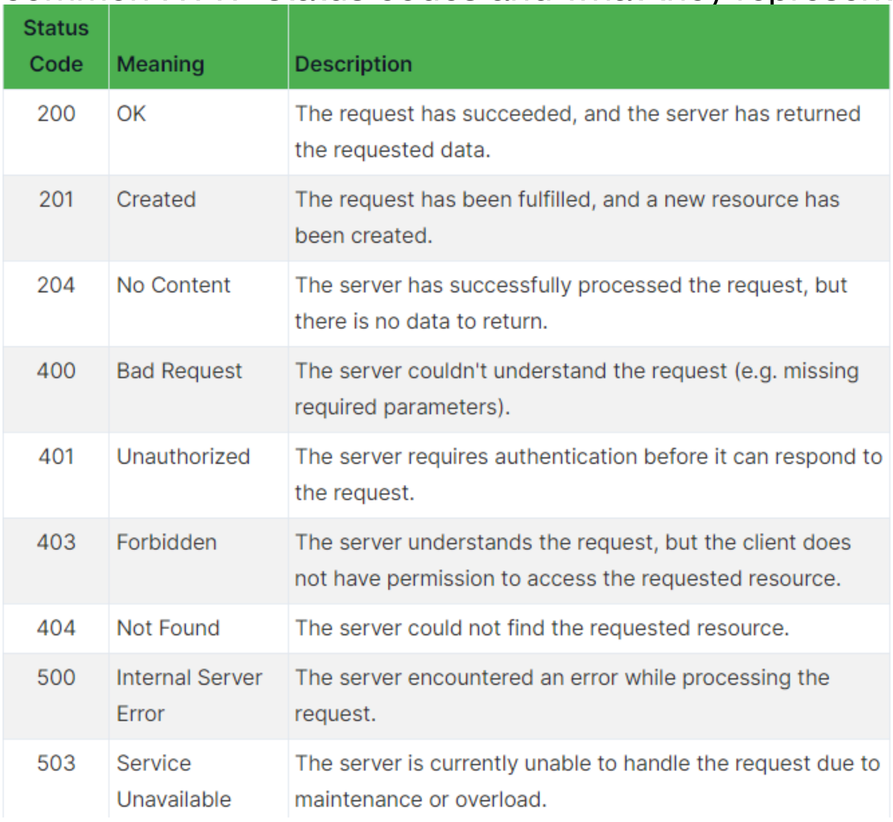
   The server will then create a new user account in its database storing all the information sent to it using the POST Request. So a POST request is used to add/insert a resource in the server.

3. **PUT**: Updates an existing resource in the server
   Now imagine you want to update your Google account's password. You would send a PUT request to the server with the new password. The server would then update your user account in its database with the new password.

4. **DELETE**: Deletes a resource from the server
   Finally, imagine you want to delete your Google user account. You would send a DELETE request to the server indicating that you want your account to be deleted. The server would then delete your user account from its database.

What is a Status Code?

HTTP status codes are three-digit numbers that indicate the status of a HTTP request made to a server. They are server responses that provide information about the request's outcome. Here are some of the most common HTTP status codes and what they represent:



Status Code	Meaning	Description
200	OK	The request has succeeded, and the server has returned the requested data.
201	Created	The request has been fulfilled, and a new resource has been created.
204	No Content	The server has successfully processed the request, but there is no data to return.
400	Bad Request	The server couldn't understand the request (e.g. missing required parameters).
401	Unauthorized	The server requires authentication before it can respond to the request.
403	Forbidden	The server understands the request, but the client does not have permission to access the requested resource.
404	Not Found	The server could not find the requested resource.
500	Internal Server Error	The server encountered an error while processing the request.
503	Service Unavailable	The server is currently unable to handle the request due to maintenance or overload.

Let's Create a Server

Step 1: Import the http module like this:

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

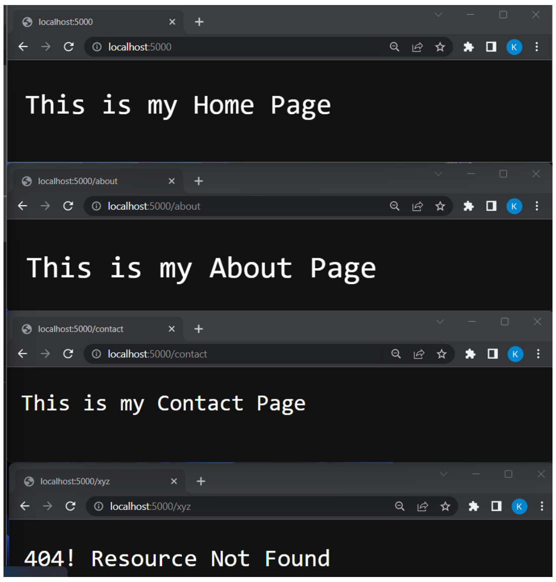
Step 2: The http module provides you with http.createServer() function which helps you create a server. This function accepts a callback function with 2 parameters – req (which stores the incoming request object) and res which stands for the response to be sent by the server. This callback function gets executed every time someone hits the server.

```
const http = require('http');
const server = http.createServer((req, res) => {
        res.end('Hello World');
})
```

Step 3: Listening the server at some port using the listen() method.

The listen() function in Node.js http module is used to start a server that listens for incoming requests. It takes a port number as an argument and binds the server to that port number so that it can receive incoming requests on that port.

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

const server = http.createServer((req, res) => {
        res.end('Hello World');
})

server.listen(5000, () => {
        console.log('Server listening at port 5000');
})
```



```
This is my Home Page
```

```
This is my About Page
```

```
This is my Contact Page
```

```
404! Resource Not Found
```

- res.writeHead() – This method is used to send the response headers to the client. The status code and headers like content-type can be set using this method.

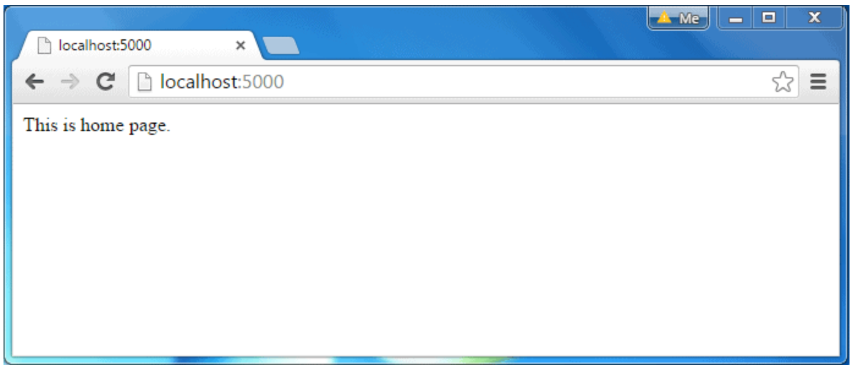
- res.write() – This method is used to send the response body to the client.
- res.end() – This method is used to end the response process.

Example

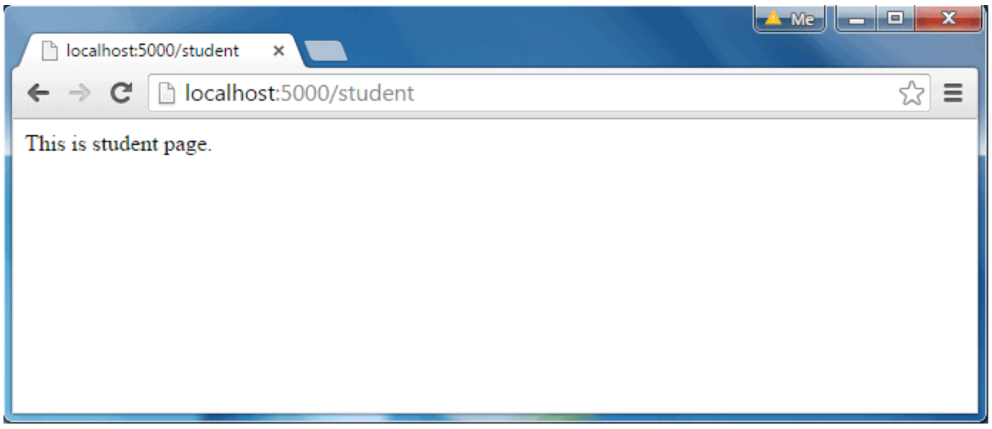
Server.js

```
var http = require('http');
// Import Node.js core module

var server = http.createServer(function (req, res) {   //create web server
    if (req.url == '/') {
//check the URL of the current request

        // set response header
        res.writeHead(200, { 'Content-Type': 'text/html' });

        // set response content
        res.write('<html><body><p>This is home Page.</p></body></html>');
        res.end();

    }
    else if (req.url == "/student") {

        res.writeHead(200, { 'Content-Type': 'text/html' });
                res.write('<html><body><p>This is student Page.</p></body></
html>');
        res.end();

    }
    else if (req.url == "/admin") {

        res.writeHead(200, { 'Content-Type': 'text/html' });
        res.write('<html><body><p>This is admin Page.</p></body></html>');
        res.end();

    }
    else
        res.end('Invalid Request!');

});

server.listen(5000);
//6 - listen for any incoming requests
```

```
console.log('Node.js web server at port 5000 is running..')
```



This is home page.



This is student page.

It will display "Invalid Request" for all requests other than the above URLs.

Sending JSON Response

The following example demonstrates how to serve JSON response from the Node.js web server.

server.js

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

var server = http.createServer(function (req, res) {

   if (req.url == '/data') {
//check the URL of the current request
         res.writeHead(200, { 'Content-Type': 'application/json' });
         res.write(JSON.stringify({ message: "Hello World"}));
         res.end();
   }
});

server.listen(5000);

console.log('Node.js web server at port 5000 is running..')
```

Let's Serve Something Interesting

1. Set up the navbar-app folder locally
2. Use the fs module to read the contents of the HTML, CSS, JS file and the Logo
3. Using the http Module to render the files when someone tries to access the / route or the home page. So Let's Get Started:

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

// Get the contents of the HTML, CSS, JS and Logo files
const homePage = fs.readFileSync('./navbar-app/index.html');
const homeStyles = fs.readFileSync('./navbar-app/style.css');
const homeLogo = fs.readFileSync('./navbar-app/logo.svg');
const homeLogic = fs.readFileSync('./navbar-app/browser-app.js');

// Creating the Server
const server = http.createServer((req, res) => {
  const url = req.url;
  if(url === '/'){
     res.writeHead(200, {'content-type': 'text/html'});
    res.write(homePage);
    res.end();
  } else if(url === '/about'){
     res.writeHead(200, {'content-type': 'text/html'});
    res.write(<h1>About Page</h1>);
    res.end();
  } else{
        res.writeHead(200, {'content-type': 'text/html'});
       res.write(<h1>404, Resource Not Found</h1>);
       res.end();
    }
})

server.listen(5000, () => {
        console.log('Server listening at port 5000');
})
```

Once we refresh the page, we see that initially the browser asks for the home page and makes a GET request with the / URL. Afterward it makes 3 more requests:

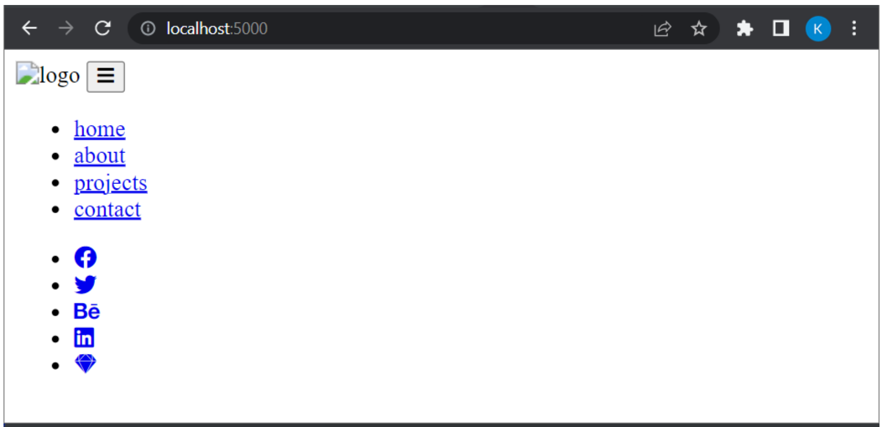
- /style.css – asking for the CSS file
- /browser-app.js – asking for the JS file
- /logo.svg – asking for the logo

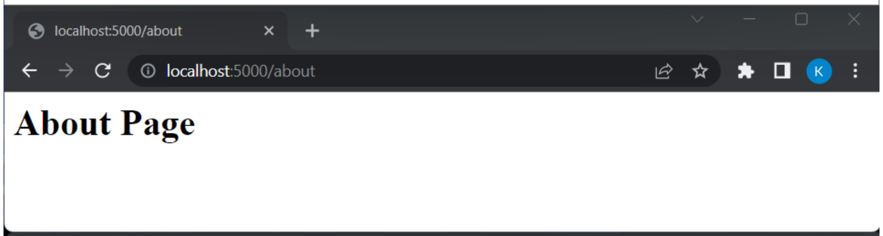
index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-
scale=1.0" />
    <title>Navbar</title>
    <!-- font-awesome -->
    <link
      rel="stylesheet"
      href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/5.14.0/css/
all.min.css"
    />

    <!-- styles -->
    <link rel="stylesheet" href="./styles.css" />
  </head>
  <body>
```

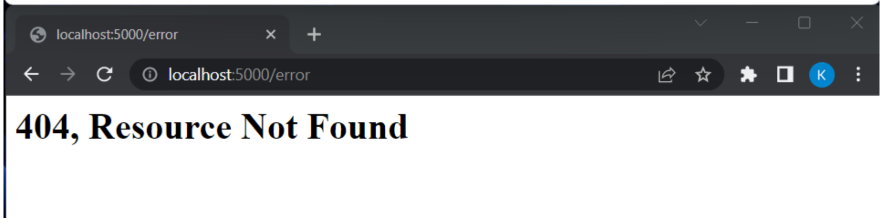
```
<nav>
  <div class="nav-center">
    <!-- nav header -->
    <div class="nav-header">
      <img src="./logo.svg" class="logo" alt="logo" />
      <button class="nav-toggle">
        <i class="fas fa-bars"></i>
      </button>
    </div>
    <!-- links -->
    <ul class="links">
      <li>
        <a href="index.html">home</a>
      </li>
      <li>
        <a href="about.html">about</a>
      </li>
      <li>
        <a href="projects.html">projects</a>
      </li>
      <li>
        <a href="contact.html">contact</a>
      </li>
    </ul>
    <!-- social media -->
    <ul class="social-icons">
      <li>
        <a href="https://www.twitter.com">
          <i class="fab fa-facebook"></i>
        </a>
      </li>
      <li>
        <a href="https://www.twitter.com">
          <i class="fab fa-twitter"></i>
        </a>
      </li>
      <li>
        <a href="https://www.twitter.com">
```

```
              <i class="fab fa-behance"></i>
            </a>
          </li>
          <li>
            <a href="https://www.twitter.com">
              <i class="fab fa-linkedin"></i>
            </a>
          </li>
          <li>
            <a href="https://www.twitter.com">
              <i class="fab fa-sketch"></i>
            </a>
          </li>
        </ul>
      </div>
    </nav>
    <!-- javascript -->
    <script src="./browser-app.js"></script>
  </body>
</html>
```





About Page



404, Resource Not Found

browser.app.js

```
const navToggle = document.querySelector('.nav-toggle')
const links = document.querySelector('.links')

navToggle.addEventListener('click', function () {
  links.classList.toggle('show-links')
})
```

styles.css

```
/*
===============
Fonts
===============
*/
@import url(“https://fonts.googleapis.com/css?family=Open+Sans|
Roboto:400,700&display=swap");

/*
===============
Variables
===============
*/

:root {
  /* dark shades of primary color*/
  --clr-primary-1: hsl(205, 86%, 17%);
  --clr-primary-2: hsl(205, 77%, 27%);
  --clr-primary-3: hsl(205, 72%, 37%);
  --clr-primary-4: hsl(205, 63%, 48%);
  /* primary/main color */
  --clr-primary-5: hsl(205, 78%, 60%);
  /* lighter shades of primary color */
  --clr-primary-6: hsl(205, 89%, 70%);
  --clr-primary-7: hsl(205, 90%, 76%);
  --clr-primary-8: hsl(205, 86%, 81%);
  --clr-primary-9: hsl(205, 90%, 88%);
  --clr-primary-10: hsl(205, 100%, 96%);
  /* darkest grey - used for headings */
  --clr-grey-1: hsl(209, 61%, 16%);
  --clr-grey-2: hsl(211, 39%, 23%);
  --clr-grey-3: hsl(209, 34%, 30%);
  --clr-grey-4: hsl(209, 28%, 39%);
  /* grey used for paragraphs */
  --clr-grey-5: hsl(210, 22%, 49%);
  --clr-grey-6: hsl(209, 23%, 60%);
  --clr-grey-7: hsl(211, 27%, 70%);
  --clr-grey-8: hsl(210, 31%, 80%);
  --clr-grey-9: hsl(212, 33%, 89%);
  --clr-grey-10: hsl(210, 36%, 96%);
  --clr-white: #fff;
  --clr-red-dark: hsl(360, 67%, 44%);
  --clr-red-light: hsl(360, 71%, 66%);
  --clr-green-dark: hsl(125, 67%, 44%);
  --clr-green-light: hsl(125, 71%, 66%);
  --clr-black: #222;
  --ff-primary: "Roboto", sans-serif;
  --ff-secondary: "Open Sans", sans-serif;
  --transition: all 0.3s linear;
  --spacing: 0.1rem;
  --radius: 0.25rem;
  --light-shadow: 0 5px 15px rgba(0, 0, 0, 0.1);
  --dark-shadow: 0 5px 15px rgba(0, 0, 0, 0.2);
  --max-width: 1170px;
  --fixed-width: 620px;
}
/*
===============
Global Styles
===============
*/

*,
::after,
::before {
  margin: 0;
  padding: 0;
```

```css
  box-sizing: border-box;
}
body {
  font-family: var(--ff-secondary);
  background: var(--clr-grey-10);
  color: var(--clr-grey-1);
  line-height: 1.5;
  font-size: 0.875rem;
}
ul {
  list-style-type: none;
}
a {
  text-decoration: none;
}
h1,
h2,
h3,
h4 {
  letter-spacing: var(--spacing);
  text-transform: capitalize;
  line-height: 1.25;
  margin-bottom: 0.75rem;
  font-family: var(--ff-primary);
}
h1 {
  font-size: 3rem;
}
h2 {
  font-size: 2rem;
}
h3 {
  font-size: 1.25rem;
}
h4 {
  font-size: 0.875rem;
}
p {
  margin-bottom: 1.25rem;
  color: var(--clr-grey-5);
}
@media screen and (min-width: 800px) {
  h1 {
    font-size: 4rem;
  }
  h2 {
    font-size: 2.5rem;
  }
  h3 {
    font-size: 1.75rem;
  }
  h4 {
    font-size: 1rem;
  }
  body {
    font-size: 1rem;
  }
  h1,
  h2,
  h3,
  h4 {
    line-height: 1;
  }
}
/*  global classes */

/* section */
.section {
  padding: 5rem 0;
}

.section-center {
  width: 90vw;
  margin: 0 auto;
  max-width: 1170px;
}
```

```
@media screen and (min-width: 992px) {
  .section-center {
    width: 95vw;
  }
}
main {
  min-height: 100vh;
  display: grid;
  place-items: center;
}

/*
===============
Navbar
===============
*/
nav {
  background: var(--clr-white);
  box-shadow: var(--light-shadow);
}
.nav-header {
  display: flex;
  align-items: center;
  justify-content: space-between;
  padding: 1rem;
}
.nav-toggle {
  font-size: 1.5rem;
  color: var(--clr-primary-5);
  background: transparent;
  border-color: transparent;
  transition: var(--transition);
  cursor: pointer;
}
.nav-toggle:hover {
  color: var(--clr-primary-1);
  transform: rotate(90deg);
}
```

```
.logo {
  height: 40px;
}
.links a {
  color: var(--clr-grey-3);
  font-size: 1rem;
  text-transform: capitalize;
  letter-spacing: var(--spacing);
  display: block;
  padding: 0.5rem 1rem;
  transition: var(--transition);
}
.links a:hover {
  background: var(--clr-primary-8);
  color: var(--clr-primary-5);
  padding-left: 1.5rem;
}
.social-icons {
  display: none;
}
.links {
  height: 0;
  overflow: hidden;
  transition: var(--transition);
}
.show-links {
  height: 10rem;
}
@media screen and (min-width: 800px) {
  .nav-center {
    max-width: 1170px;
    margin: 0 auto;
    display: flex;
    align-items: center;
    justify-content: space-between;
    padding: 1rem;
  }
  .nav-header {
    padding: 0;
  }
  .nav-toggle {
    display: none;
  }
  .links {
    height: auto;
    display: flex;
  }
  .links a {
    padding: 0;
    margin: 0 0.5rem;
  }
  .links a:hover {
    padding: 0;
    background: transparent;
  }
  .social-icons {
    display: flex;
  }
  .social-icons a {
    margin: 0 0.5rem;
    color: var(--clr-primary-5);
    transition: var(--transition);
  }
  .social-icons a:hover {
    color: var(--clr-primary-7);
  }
}
```

Understanding Clustering

Clustering in Node.js involves creating multiple worker processes that share the incoming workload. Each worker process runs in its own event loop, utilizing the available CPU cores. The master process manages the worker processes, distributes incoming requests, and handles process failures.

Benefits of Clustering:

- **Improved Performance**: Clustering enables parallel processing of requests across multiple cores, leading to improved performance and responsiveness of the application. It allows for better utilization of available system resources, especially on machines with multiple CPU cores.
- **Scalability**: Clustering enhances the scalability of Node.js applications by handling concurrent requests in parallel. As the workload increases, additional worker processes can be created dynamically to distribute the load effectively.
- **Fault Tolerance**: If a worker process crashes or becomes unresponsive, the master process can detect the failure and restart the worker process automatically. This fault tolerance ensures that the application remains available even in the presence of process failures.

Step 1 - Setting Up the Project Directory

In this step, you will create the directory for the project and download dependencies for the application you will build later in this tutorial. In Step 2, you'll build the application using Express. You'll then scale it in Step 3 to multiple CPUs with the built-in `node-cluster` module, which you'll measure with the `loadtest` package in Step 4. From there, you'll scale it with the `pm2` package and measure it again in Step 5.

$ mkdir cluster_demo

Next, move into the directory:

$ cd cluster_demo

Then, initialize the project, which will also create a `package.json` file:

$ npm init -y

Note these properties that are aligned with your specific project:

- `name`: the name of the npm package.
- `version`: your package’s version number.
- `main`: the entry point of your project.

Next, open the `package.json` file with your preferred editor

```
$ nano package.json
```

Next, you will download the following packages:

- `express`: a framework for building web applications in Node.js.
- `loadtest`: a load testing tool, useful for generating traffic to an app to measure its performance.
- `pm2`: a tool that automatically scales an app to multiple CPUs.

```
$ npm install express
```

Next, run the command to download the `loadtest` and `pm2` packages globally:

```
$ npm install -g loadtest pm2
```

Step 2 — Creating an Application Without Using a Cluster

In this step, you will create a sample program containing a single route that will start a CPU-intensive task upon each user’s visit. The program will not use the `cluster` module so that you can access the performance implications of running a single instance of an app on one CPU.

In your `index.js` file, add the following lines to import and instantiate Express:

```
const express =  require(‘express’)
const port = 3000;
const app = express();
```

```
console.log(`worker pid=${process.pid}`);
```

In the first line, you import the `express` package. In the second line, you set the `port` variable to port `3000`, which the application's server will listen on. Next, you set the `app` variable to an instance of Express. After that, you log the process ID of the application's process in the console using the built-in `process` module.

Next, add these lines to define the route `/heavy`, which will contain a CPU-bound loop:

```
...
app.get("/heavy", (req, res) => {
  let total = 0;
  for (let i = 0; i < 5_000_000; i++) {
    total++;
  }
  res.send(`The result of the CPU intensive task is ${total}\n`);
});
```

In the `/heavy` route, you define a loop that increments the `total` variable 5 million times. You then send a response containing the value in the `total` variable using the `res.send()` method. While the example of the CPU-bound task is arbitrary, it demonstrates CPU-bound tasks without adding complexity. You can also use other names for the route, but this tutorial uses `/heavy` to indicate a heavy performance task.

Next, call the `listen()` method of the Express module to have the server listening on port `3000` stored in the `port` variable:

```
...
app.listen(port, () => {
  console.log(`App listening on port ${port}`);
});
```

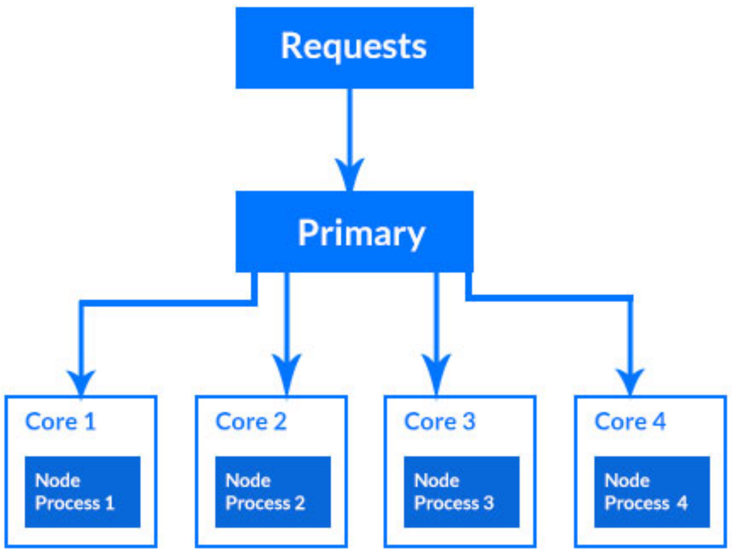
The complete file will match the following:

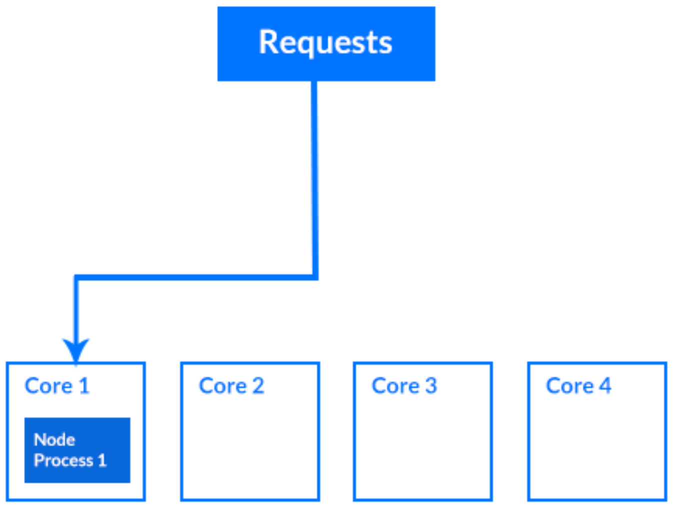
```
const express =  require(‘express’)
const port = 3000;
const app = express();

console.log(`worker pid=${process.pid}`);

app.get("/heavy", (req, res) => {
  let total = 0;
  for (let i = 0; i < 5_000_000; i++) {
    total++;
  }
  res.send(`The result of the CPU intensive task is ${total}\n`);
});

app.listen(port, () => {
  console.log(`App listening on port ${port}`);
});
```

When you’ve finished adding your code, save and exit your file. Then run the file using the `node` command:

```
$ node index.js
```

```
Output
worker pid=11023
App listening on port 3000
```





The output states the process ID of the process running and a message confirming that the server is listening on port `3000`.

To test if the application is working, open another terminal and run the following command:

$ curl http://localhost:3000/heavy

The output will match the following:

```
Output
The result of the CPU intensive task is 5000000
```

Now that you have created an app without using the `cluster` module, you will use the `cluster` module to scale the application to use multiple CPUs next.

Step 3 — Clustering the Application

In this step, you will add the `cluster` module to create multiple instances of the same program to handle more load and improve performance. When you run processes with the `cluster` module, you can have multiple processes running on each CPU on your machine:

In this diagram, the requests go through the load balancer in the primary process, which then uses the round-robin algorithm to distribute the requests among the processes.

You’ll now add the `cluster` module. In your terminal, create the `primary.js` file:

In your `primary.js` file, add the following lines to import dependencies:

```
const cluster = require(‘cluster’)
const os = require(‘os’)
```

Next, add the following code to reference the `index.js` file:

```
...
const cpuCount = os.cpus().length;
console.log(`The total number of CPUs is ${cpuCount}`);
console.log(`Primary pid=${process.pid}`);
cluster.setupPrimary({
  exec: __dirname + "/index.js",
});
```

First, you set the `cpuCount` variable to the number of CPUs in your machine, which should be four or higher. Next, you log the number of CPUs in the console. Then after, you log the process ID of the primary process, which is the one that will receive all the requests, and use a load balancer to distribute them among worker processes.

Following that, you reference the `index.js` file using the `setupPrimary()` method of the `cluster` module so that it will be executed in each worker process spawned.

Next, add the following code to create the processes:

```
...
for (let i = 0; i < cpuCount; i++) {
  cluster.fork();
}
cluster.on("exit", (worker, code, signal) => {
  console.log(`worker ${worker.process.pid} has been killed`);
  console.log("Starting another worker");
  cluster.fork();
});
```

The loop iterates as many times as the value in the `cpuCount` and calls the `fork()` method of the `cluster` module during each iteration. You attach the `exit` event using the `on()` method of the `cluster` module to listen when a process has emitted the `exit` event, which is usually when the process dies. When the `exit` event is triggered, you log the process ID of the worker that has died and then invoke the `fork()` method to create a new worker process to replace the dead process.

Your complete code will now match the following:

```
const cluster = require('cluster')
const os =  require('os')
const cpuCount = os.cpus().length;

console.log(`The total number of CPUs is ${cpuCount}`);
console.log(`Primary pid=${process.pid}`);
cluster.setupPrimary({
  exec: __dirname + "/index.js",
});

for (let i = 0; i < cpuCount; i++) {
  cluster.fork();
}
cluster.on("exit", (worker, code, signal) => {
  console.log(`worker ${worker.process.pid} has been killed`);
  console.log("Starting another worker");
  cluster.fork();
});
```

Once you have finished adding the lines, save and exit your file.

Next, run the file:

```
$ node primary.js
```

The output will closely match the following (your process IDs and order of information may differ):

```
Output
The total number of CPUs is 4
Primary pid=7341
worker pid=7353
worker pid=7354
worker pid=7360
App listening on port 3000
App listening on port 3000
App listening on port 3000
worker pid=7352
App listening on port 3000
```

The output will indicate four CPUs, one primary process that includes a load balancer, and four worker processes listening on port `3000`.

Next, return to the second terminal, then send a request to the `/heavy` route:

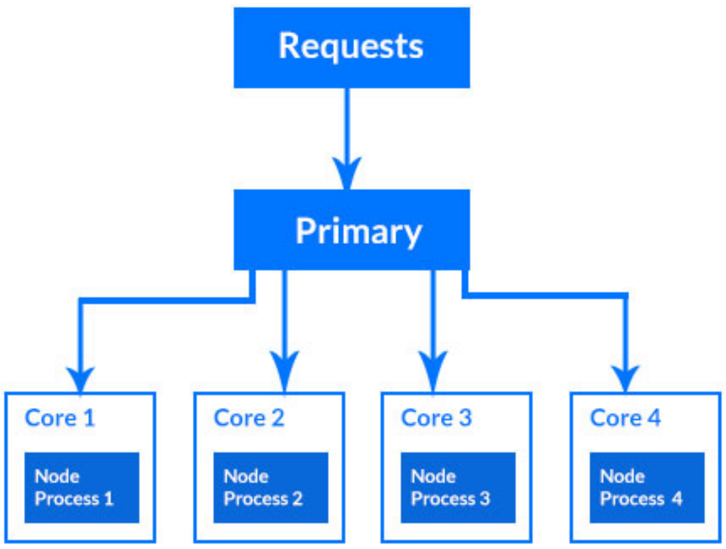
```
$ curl http://localhost:3000/heavy
```

The output confirms the program is working:

```
Output
The result of the CPU intensive task is 5000000
```

You can stop the server now.

At this point, you will have four processes running on all the CPUs on your machine:



With clustering added to the application, you can compare the program performances for the one using the `cluster` module and the one without the `cluster` module.

Step 4 — Comparing Performance Using a Load Testing Tool

In this step, you will use the `loadtest` package to generate traffic against the two programs you've built. You'll compare the performance of the `primary.js` program which uses the `cluster` module with that of the `index.js` program which does not use clustering. You will notice that the program using the `cluster` module performs faster and can handle more requests within a specific time than the program that doesn't use clustering.

First, you will measure the performance of the `index.js` file, which doesn't use the `cluster` module and only runs on a single instance.

In your first terminal, run the `index.js` file to start the server:

```
$ node index.js
```

You'll receive an output that the app is running:

```
Output
worker pid=7731
App listening on port 3000
```

Next, return to your second terminal to use the `loadtest` package to send requests to the server:

```
$ loadtest -n 1200 -c 200 -k http://localhost:3000/heavy
```

The `-n` option accepts the number of requests the package should send, which is `1200` requests here. The `-c` option accepts the number of requests that should be sent simultaneously to the server.

Once the requests have been sent, the package will return output similar to the following:

```
Output
Requests: 0 (0%), requests per second: 0, mean latency: 0
ms
Requests: 430 (36%), requests per second: 87, mean
latency: 1815.1 ms
Requests: 879 (73%), requests per second: 90, mean
latency: 2230.5 ms
```

```
Target URL:          http://localhost:3000/heavy
Max requests:        1200
Concurrency level:   200
Agent:               keepalive

Completed requests:  1200
Total errors:        0
Total time:          13.712728601 s
Requests per second: 88
Mean latency:        2085.1 ms

Percentage of the requests served within a certain time
  50%      2234 ms
  90%      2340 ms
  95%      2385 ms
  99%      2406 ms
 100%      2413 ms (longest request)
```

From this output, take note of the following metrics:

- `Total time` measures how long it took for all the requests to be served. In this output, it took just over 13 seconds to serve all `1200` requests.
- `Requests per second` measures the number of requests the server can handle per second. In this output, the server handles `88` requests per second.
- `Mean latency` measures the time it took to send a request and get a response, which is `2085.1 ms` in the sample output.

These metrics will vary depending on your network or processor speed, but they will be close to these examples.

Now that you have measured the performance of the `index.js` file, you can stop the server.

Next, you will measure the performance of the `primary.js` file, which uses the `cluster` module.

To do that, return to the first terminal and rerun the `primary.js` file:

```
$ node primary.js
```

You'll receive a response with the same information as earlier:

```
Output
The total number of CPUs is 4
Primary pid=7841
worker pid=7852
App listening on port 3000
worker pid=7854
App listening on port 3000
worker pid=7853
worker pid=7860
App listening on port 3000
App listening on port 3000
```

In the second terminal, run the `loadtest` command again:

$ loadtest -n 1200 -c 200 -k http://localhost:3000/heavy

When it finishes, you'll receive a similar output (it can differ based on the number of CPUs on your system):

```
Output
Requests: 0 (0%), requests per second: 0, mean latency: 0
ms

Target URL:          http://localhost:3000/heavy
Max requests:        1200
Concurrency level:   200
Agent:               keepalive

Completed requests:  1200
Total errors:        0
Total time:          3.412741962 s
Requests per second: 352
Mean latency:        514.2 ms

Percentage of the requests served within a certain time
  50%      194 ms
  90%      2521 ms
  95%      2699 ms
  99%      2710 ms
 100%      2759 ms (longest request)
```

The output for the `primary.js` app, which is running with the `cluster` module, indicates that the total time is down to 3 seconds from 13 seconds in the program that doesn't use clustering. The number of requests the server can handle per second has tripled to

`352` from the previous `88`, which means that your server can take a huge load. Another important metric is the mean latency, which has significantly dropped from `2085.1 ms` to `514.2 ms`.

This response confirms that the scaling has worked and that your application can handle more requests in a short time without delays. If you upgrade your machine to have more CPUs, the app will automatically scale to the number of CPUs and improve performance further.

As a reminder, the metrics in your terminal output will differ because of your network and processor speed. The total time and the mean latency will drop significantly, and the total time will increase rapidly.

Now that you have made the comparison and noted that the app performs better with the `cluster` module, you can stop the server. In the next step, you will use `pm2` in place of the `cluster` module.

Step 5 — Using `pm2` for Clustering

So far, you have used the `cluster` module to create worker processes according to the number of CPUs on your machine. You have also added the ability to restart a worker process when it dies. In this step, you will set up an alternative to automate scaling your app by using the `pm2` process manager, which is built upon the `cluster` module. This process manager contains a load balancer and can automatically create as many worker processes as CPUs on your machine. It also allows you to monitor the processes and can spawn a new worker process automatically if one dies.

To use it, you need to run the `pm2` package with the file you need to scale, which is the `index.js` file in this tutorial.

In your initial terminal, start the `pm2` cluster with the following command:

```
$ pm2 start index.js -i 0
```

The `-i` option accepts the number of worker processes you want `pm2` to create. If you pass the argument `0`, `pm2` will automatically create as many worker processes as there are CPUs on your machine.

Upon running the command, `pm2` will show you more details about the worker processes:

The table contains each worker’s process ID, status, CPU utilization, and memory consumption, which you can use to understand how the processes behave.

When starting the cluster with `pm2`, the package runs in the background and will automatically restart even when you reboot your system.

If you want to read the logs from the worker processes, you can use the following command:

$ pm2 logs

Command	Description
`pm2 start app_name`	Starts the cluster
`pm2 restart app_name`	Kills the cluster and starts it again
`pm2 reload app_name`	Restarts the cluster without downtime
`pm2 stop app_name`	Stops the cluster

`pm2 delete app_name`	Deletes the cluster