DAVID LANDUP

MARCUS SANATAN

HOW TO CODE IN NODE.JS

ISBN 978-1-7358317-2-5

How To Code in Node.js

David Landup and Marcus Sanatan

Editors: Timothy Nolan and Brian MacDonald

DigitalOcean, New York City, New York, USA

2020-12

How To Code in Node.js

About DigitalOcean

DigitalOcean is a cloud services platform delivering the simplicity developers love and businesses trust to run production applications at scale. It provides highly available, secure, and scalable compute, storage, and networking solutions that help developers build great software faster. Founded in 2012 with offices in New York City and Cambridge, MA, DigitalOcean offers transparent and affordable pricing, an elegant user interface, and one of the largest libraries of open source resources available. For more information, please visit https://www.digitalocean.com or follow @digitalocean on Twitter.

Introduction

Preface — Getting Started with this Book

We recommend that you begin with a clean, new server to start learning how to program with Node. You can also use a local computer like a laptop or desktop. If you are unfamiliar with Node, or do not have a development environment set up, Chapter 1 of this book links to a tutorial that explains how to install Node for development on macOS, or an Ubuntu 20.04 system.

Programming using Node requires some familiarity with JavaScript, so if you would like to learn more about the language itself before exploring this book, visit the DigitalOcean Community's JavaScript section to explore tutorials that focus on using JavaScript in a browser.

Once you are set up with a local or remote Node development environment, you will be able to follow along with each chapter at your own pace, and in the order that you choose.

About this Book

Node.js is a popular open-source runtime environment that can execute JavaScript outside of the browser. The Node runtime is commonly used for back-end web development, leveraging its asynchronous capabilities to create networking applications and web servers. Node is also a popular choice for building command line tools.

In this book, you will go through exercises to learn the basics of how to code in Node.js, gaining skills that apply equally to back-end and full stack

development in the process. Each chapter is written by members of the Stack Abuse team.

Learning Goals and Outcomes

The chapters in this book cover a broad range of Node topics, from using and packaging your own modules, to writing complete web servers and clients. While there is a general progression that starts with installing Node locally and running small Node programs on the command line, each chapter in this book can be read independently of the others. If there is a particular topic or chapter that catches your attention feel free to jump ahead and work through it.

By the end of this book you will be able to write programs that leverage Node's asynchronous code execution capabilities, complete with event emitters and listeners that will respond to user actions. Along the way you will learn how to debug Node applications using the built-in debugging utilities, as well as the Chrome browser's DevTools utilities. You will also learn how to write automated tests for your programs to ensure that any features that you add or change function as you expect.

If you would like to learn more about Node development after you have finished reading this book, be sure to visit the DigitalOcean Community's Node.js section.

How To Write and Run Your First Program in Node.js

Written by Stack Abuse

The author selected the Open Internet/Free Speech Fund to receive a donation as part of the Write for DOnations program.

Node.js is a popular open-source runtime environment that can execute JavaScript outside of the browser using the V8 JavaScript engine, which is the same engine used to power the Google Chrome web browser's JavaScript execution. The Node runtime is commonly used to create command line tools and web servers.

Learning Node.js will allow you to write your front-end code and your back-end code in the same language. Using JavaScript throughout your entire stack can help reduce time for context switching, and libraries are more easily shared between your back-end server and front-end projects.

Also, thanks to its support for asynchronous execution, Node.js excels at I/O-intensive tasks, which is what makes it so suitable for the web. Real-time applications, like video streaming, or applications that continuously send and receive data, can run more efficiently when written in Node.js.

In this tutorial you'll create your first program with the Node.js runtime. You'll be introduced to a few Node-specific concepts and build your way up to create a program that helps users inspect environment variables on their system. To do this, you'll learn how to output strings to the console, receive input from the user, and access environment variables.

Prerequisites

To complete this tutorial, you will need:

- Node.js installed on your development machine. This tutorial uses Node.js version 10.16.0. To install this on macOS or Ubuntu 18.04, follow the steps in How to Install Node.js and Create a Local Development Environment on macOS or the "Installing Using a PPA" section of How To Install Node.js on Ubuntu 18.04.
- A basic knowledge of JavaScript, which you can find here: How To Code in JavaScript.

Step 1 — Outputting to the Console

To write a "Hello, World!" program, open up a command line text editor such as `nano` and create a new file:

```
nano hello.js
```

With the text editor opened, enter the following code:

hello.js

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

The `console` object in Node.js provides simple methods to write to `stdout`, `stderr`, or to any other Node.js stream, which in most cases is the command line. The `log` method prints to the `stdout` stream, so you can see it in your console.

In the context of Node.js, streams are objects that can either receive data, like the `stdout` stream, or objects that can output data, like a network socket or a file. In the case of the `stdout` and `stderr` streams, any data sent to them will then be shown in the console. One of the great things about streams is that they're easily redirected, in which case you can redirect the output of your program to a file, for example.

Save and exit `nano` by pressing `CTRL+X`, when prompted to save the file, press `Y`. Now your program is ready to run.

Step 2 — Running the Program

To run this program, use the `node` command as follows:

```
node hello.js
```

The `hello.js` program will execute and display the following output:

```
Output
Hello World
```

The Node.js interpreter read the file and executed `console.log("Hello World");` by calling the `log` method of the global `console` object. The string `"Hello World"` was passed as an argument to the `log` function.

Although quotation marks are necessary in the code to indicate that the text is a string, they are not printed to the screen.

Having confirmed that the program works, let's make it more interactive.

Step 3 — Receiving User Input via Command Line Arguments

Every time you run the Node.js "Hello, World!" program, it produces the same output. In order to make the program more dynamic, let's get input from the user and display it on the screen.

Command line tools often accept various arguments that modify their behavior. For example, running `node` with the `--version` argument prints the installed version instead of running the interpreter. In this step, you will make your code accept user input via command line arguments.

Create a new file `arguments.js` with nano:

```
nano arguments.js
```

Enter the following code:

arguments.js

```
console.log(process.argv);
```

The `process` object is a global Node.js object that contains functions and data all related to the currently running Node.js process. The `argv` property is an array of strings containing all the command line arguments given to a program.

Save and exit `nano` by typing `CTRL+X`, when prompted to save the file, press `Y`.

Now when you run this program, you provide a command line argument like this:

```
node arguments.js hello world
```

The output looks like the following:

```
Output
[ '/usr/bin/node',
  '/home/sammy/first-program/arguments.js',
  'hello',
  'world' ]
```

The first argument in the `process.argv` array is always the location of the Node.js binary that is running the program. The second argument is always the location of the file being run. The remaining arguments are what the user entered, in this case: `hello` and `world`.

We are mostly interested in the arguments that the user entered, not the default ones that Node.js provides. Open the `arguments.js` file for editing:

```
nano arguments.js
```

Change `console.log(process.arg);` to the following:

arguments.js

```
console.log(process.argv.slice(2));
```

Because `argv` is an array, you can use JavaScript's built-in `slice` method that returns a selection of elements. When you provide the `slice` function with `2` as its argument, you get all the elements of `argv` that comes after its second element; that is, the arguments the user entered.

Re-run the program with the `node` command and the same arguments as last time:

```
node arguments.js hello world
```

Now, the output looks like this:

```
Output
[ 'hello', 'world' ]
```

Now that you can collect input from the user, let's collect input from the program's environment.

Step 4 — Accessing Environment Variables

Environment variables are key-value data stored outside of a program and provided by the OS. They are typically set by the system or user and are available to all running processes for configuration or state purposes. You can use Node's `process` object to access them.

Use `nano` to create a new file `environment.js`:

```
nano environment.js
```

Add the following code:

environment.js

```
console.log(process.env);
```

The `env` object stores all the environment variables that are available when Node.js is running the program.

Save and exit like before, and run the `environment.js` file with the `node` command.

```
node environment.js
```

Upon running the program, you should see output similar to the following:

Output

```
{ SHELL: '/bin/bash',
  SESSION_MANAGER:
   'local/digitalocean:@/tmp/.ICE-unix/1003,unix/digitalocea
n:/tmp/.ICE-unix/1003',
  COLORTERM: 'truecolor',
  SSH_AUTH_SOCK: '/run/user/1000/keyring/ssh',
  XMODIFIERS: '@im=ibus',
  DESKTOP_SESSION: 'ubuntu',
  SSH_AGENT_PID: '1150',
  PWD: '/home/sammy/first-program',
  LOGNAME: 'sammy',
  GPG_AGENT_INFO: '/run/user/1000/gnupg/S.gpg-agent:0:1',
  GJS_DEBUG_TOPICS: 'JS ERROR;JS LOG',
  WINDOWPATH: '2',
  HOME: '/home/sammy',
  USERNAME: 'sammy',
  IM_CONFIG_PHASE: '2',
  LANG: 'en_US.UTF-8',
  VTE_VERSION: '5601',
  CLUTTER_IM_MODULE: 'xim',
  GJS_DEBUG_OUTPUT: 'stderr',
  LESSCLOSE: '/usr/bin/lesspipe %s %s',
  TERM: 'xterm-256color',
  LESSOPEN: '| /usr/bin/lesspipe %s',
  USER: 'sammy',
```

```
  DISPLAY: ':0',
  SHLVL: '1',
  PATH:
   '/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/b
in:/usr/games:/usr/local/games:/snap/bin',
  DBUS_SESSION_BUS_ADDRESS: 'unix:path=/run/user/1000/bus',
  _: '/usr/bin/node',
  OLDPWD: '/home/sammy' }
```

Keep in mind that many of the environment variables you see are dependent on the configuration and settings of your system, and your output may look substantially different than what you see here. Rather than viewing a long list of environment variables, you might want to retrieve a specific one.

Step 5 — Accessing a Specified Environment Variable

In this step you'll view environment variables and their values using the global `process.env` object and print their values to the console.

The `process.env` object is a simple mapping between environment variable names and their values stored as strings. Like all objects in JavaScript, you access an individual property by referencing its name in square brackets.

Open the `environment.js` file for editing:

```
nano environment.js
```

Change `console.log(process.env);` to:

environment.js

```
console.log(process.env["HOME"]);
```

Save the file and exit. Now run the `environment.js` program:

```
node environment.js
```

The output now looks like this:

```
Output
/home/sammy
```

Instead of printing the entire object, you now only print the `HOME` property of `process.env`, which stores the value of the `$HOME` environment variable.

Again, keep in mind that the output from this code will likely be different than what you see here because it is specific to your system. Now that you can specify the environment variable to retrieve, you can enhance your program by asking the user for the variable they want to see.

Step 6 — Retrieving An Argument in Response to User Input

Next, you'll use the ability to read command line arguments and environment variables to create a command line utility that prints the value of an environment variable to the screen.

Use `nano` to create a new file `echo.js`:

```
nano echo.js
```

Add the following code:

echo.js

```
const args = process.argv.slice(2);
console.log(process.env[args[0]]);
```

The first line of `echo.js` stores all the command line arguments that the user provided into a constant variable called `args`. The second line prints the environment variable stored in the first element of `args`; that is, the first command line argument the user provided.

Save and exit `nano`, then run the program as follows:

```
node echo.js HOME
```

Now, the output would be:

```
Output
/home/sammy
```

The argument `HOME` was saved to the `args` array, which was then used to find its value in the environment via the `process.env` object.

At this point you can now access the value of any environment variable on your system. To verify this, try viewing the following variables: `PWD`, `USER`, `PATH`.

Retrieving single variables is good, but letting the user specify how many variables they want would be better.

Step 7 — Viewing Multiple Environment Variables

Currently, the application can only inspect one environment variable at a time. It would be useful if we could accept multiple command line arguments and get their corresponding value in the environment. Use `nano` to edit `echo.js`:

```
nano echo.js
```

Edit the file so that it has the following code instead:

echo.js

```
const args = process.argv.slice(2);

args.forEach(arg => {
  console.log(process.env[arg]);
});
```

The `forEach` method is a standard JavaScript method on all array objects. It accepts a callback function that is used as it iterates over every element of

the array. You use `forEach` on the `args` array, providing it a callback function that prints the current argument's value in the environment.

Save and exit the file. Now re-run the program with two arguments:

```
node echo.js HOME PWD
```

You would see the following output:

```
Output
/home/sammy
/home/sammy/first-program
```

The `forEach` function ensures that every command line argument in the `args` array is printed.

Now you have a way to retrieve the variables the user asks for, but we still need to handle the case where the user enters bad data.

Step 8 — Handling Undefined Input

To see what happens if you give the program an argument that is not a valid environment variable, run the following:

```
node echo.js HOME PWD NOT_DEFINED
```

The output will look similar to the following:

```
Output
/home/sammy
/home/sammy/first-program
undefined
```

The first two lines print as expected, and the last line only has `undefined`. In JavaScript, an `undefined` value means that a variable or property has not been assigned a value. Because `NOT_DEFINED` is not a valid environment variable, it is shown as `undefined`.

It would be more helpful to a user to see an error message if their command line argument was not found in the environment.

Open `echo.js` for editing:

```
nano echo.js
```

Edit `echo.js` so that it has the following code:

echo.js

```
const args = process.argv.slice(2);

args.forEach(arg => {
  let envVar = process.env[arg];
  if (envVar === undefined) {
    console.error(`Could not find "${arg}" in environment`);
  } else {
    console.log(envVar);
  }
});
```

Here, you have modified the callback function provided to `forEach` to do the following things:

1. Get the command line argument's value in the environment and store it in a variable `envVar`.
2. Check if the value of `envVar` is `undefined`.
3. If the `envVar` is `undefined`, then we print a helpful message indicating that it could not be found.
4. If an environment variable was found, we print its value.

Note: The `console.error` function prints a message to the screen via the `stderr` stream, whereas `console.log` prints to the screen via the `stdout` stream. When you run this program via the command line, you won't notice

the difference between the `stdout` and `stderr` streams, but it is good practice to print errors via the `stderr` stream so that they can be easier identified and processed by other programs, which can tell the difference.

Now run the following command once more:

```
node echo.js HOME PWD NOT_DEFINED
```

This time the output will be:

```
Output
/home/sammy
/home/sammy/first-program
Could not find "NOT_DEFINED" in environment
```

Now when you provide a command line argument that's not an environment variable, you get a clear error message stating so.

Conclusion

Your first program displayed "Hello World" to the screen, and now you have written a Node.js command line utility that reads user arguments to display environment variables.

If you want to take this further, you can change the behavior of this program even more. For example, you may want to validate the command line arguments before you print. If an argument is undefined, you can return an error, and the user will only get output if all arguments are valid environment variables.

If you'd like to continue learning Node.js, you can return to the How To Code in Node.js series, or browse programming projects and setups on our Node topic page.

How To Use the Node.js REPL

Written by Stack Abuse

The author selected the Open Internet/Free Speech Fund to receive a donation as part of the Write for DOnations program.

The Node.js Read-Eval-Print-Loop (REPL) is an interactive shell that processes Node.js expressions. The shell **reads** JavaScript code the user enters, **eval**uates the result of interpreting the line of code, **prints** the result to the user, and **loops** until the user signals to quit.

The REPL is bundled with with every Node.js installation and allows you to quickly test and explore JavaScript code within the Node environment without having to store it in a file.

Prerequisites

To complete this tutorial, you will need:

- Node.js installed on your development machine. This tutorial uses version 10.16.0. To install this on macOS or Ubuntu 18.04, follow the steps in How to Install Node.js and Create a Local Development Environment on macOS or the "Installing Using a PPA" section of How To Install Node.js on Ubuntu 18.04.
- A basic knowledge of JavaScript, which you can find here: How To Code in JavaScript

Step 1 — Starting and Stopping the REPL

If you have `node` installed, then you also have the Node.js REPL. To start it, simply enter `node` in your command line shell:

```
node
```

This results in the REPL prompt:

```
>
```

The `>` symbol lets you know that you can enter JavaScript code to be immediately evaluated.

For an example, try adding two numbers in the REPL by typing this:

```
> 2 + 2
```

When you press `ENTER`, the REPL will evaluate the expression and return:

```
4
```

To exit the REPL, you can type `.exit`, or press `CTRL+D` once, or press `CTRL+C` twice, which will return you to the shell prompt.

With starting and stopping out of the way, let's take a look at how you can use the REPL to execute simple JavaScript code.

Step 2 — Executing Code in the Node.js REPL

The REPL is a quick way to test JavaScript code without having to create a file. Almost every valid JavaScript or Node.js expression can be executed in the REPL.

In the previous step you already tried out addition of two numbers, now let's try division. To do so, start a new REPL:

```
node
```

In the prompt type:

```
> 10 / 5
```

Press `ENTER` , and the output will be `2`, as expected:

```
2
```

The REPL can also process operations on strings. Concatenate the following strings in your REPL by typing:

```
> "Hello " + "World"
```

Again, press `ENTER`, and the string expression is evaluated:

```
'Hello World'
```

Note: You may have noticed that the output used single quotes instead of double quotes. In JavaScript, the quotes used for a string do not affect its value. If the string you entered used a single quote, the REPL is smart enough to use double quotes in the output.

Calling Functions

When writing Node.js code, it's common to print messages via the global `console.log` method or a similar function. Type the following at the prompt:

```
> console.log("Hi")
```

Pressing `ENTER` yields the following output:

```
Hi
undefined
```

The first result is the output from `console.log`, which prints a message to the `stdout` stream (the screen). Because `console.log` prints a string instead of returning a string, the message is seen without quotes. The `undefined` is the return value of the function.

Creating Variables

Rarely do you just work with literals in JavaScript. Creating a variable in the REPL works in the same fashion as working with `.js` files. Type the following at the prompt:

```
> let age = 30
```

Pressing `ENTER` results in:

```
undefined
```

Like before, with `console.log`, the return value of this command is `undefined`. The `age` variable will be available until you exit the REPL session. For example, you can multiply `age` by two. Type the following at the prompt and press `ENTER`:

```
> age * 2
```

The result is:

```
60
```

Because the REPL returns values, you don't need to use `console.log` or similar functions to see the output on the screen. By default, any returned value will appear on the screen.

Multi-line Blocks

Multi-line blocks of code are supported as well. For example, you can create a function that adds 3 to a given number. Start the function by typing the following:

```
> const add3 = (num) => {
```

Then, pressing `ENTER` will change the prompt to:

```
...
```

The REPL noticed an open curly bracket and therefore assumes you're writing more than one line of code, which needs to be indented. To make it easier to read, the REPL adds 3 dots and a space on the next line, so the following code appears to be indented.

Enter the second and third lines of the function, one at a time, pressing `ENTER` after each:

```
... return num + 3;
... }
```

Pressing `ENTER` after the closing curly bracket will display an `undefined`, which is the "return value" of the function assignment to a variable. The `...` prompt is now gone and the `>` prompt returns:

```
undefined
>
```

Now, call `add3()` on a value:

```
> add3(10)
```

As expected, the output is:

```
13
```

You can use the REPL to try out bits of JavaScript code before including them into your programs. The REPL also includes some handy shortcuts to make that process easier.

Step 3 — Mastering REPL Shortcuts

The REPL provides shortcuts to decrease coding time when possible. It keeps a history of all the entered commands and allows us to cycle through them and repeat a command if necessary.

For an example, enter the following string:

```
"The answer to life the universe and everything is 32"
```

This results in:

```
'The answer to life the universe and everything is 32'
```

If we'd like to edit the string and change the "32" to "42", at the prompt, use the `UP` arrow key to return to the previous command:

```
> "The answer to life the universe and everything is 32"
```

Move the cursor to the left, delete `3`, enter `4`, and press `ENTER` again:

```
'The answer to life the universe and everything is 42'
```

Continue to press the `UP` arrow key, and you'll go further back through your history until the first used command in the current REPL session. In contrast, pressing `DOWN` will iterate towards the more recent commands in the history.

When you are done maneuvering through your command history, press `DOWN` repeatedly until you have exhausted your recent command history and are once again seeing the prompt.

To quickly get the last evaluated value, use the underscore character. At the prompt, type `_` and press `ENTER`:

```
> _
```

The previously entered string will appear again:

```
'The answer to life the universe and everything is 42'
```

The REPL also has an autocompletion for functions, variables, and keywords. If you wanted to find the square root of a number using the `Math.sqrt` function, enter the first few letters, like so:

```
> Math.sq
```

Then press the `TAB` key and the REPL will autocomplete the function:

```
> Math.sqrt
```

When there are multiple possibilities for autocompletion, you're prompted with all the available options. For an example, enter just:

```
> Math.
```

And press `TAB` twice. You're greeted with the possible autocompletions:

```
> Math.
Math.__defineGetter__      Math.__defineSetter__      Math.__l
ookupGetter__
Math.__lookupSetter__      Math.__proto__             Math.con
structor
Math.hasOwnProperty        Math.isPrototypeOf         Math.pro
pertyIsEnumerable
Math.toLocaleString        Math.toString              Math.val
ueOf

Math.E                     Math.LN10                  Math.LN2
Math.LOG10E                Math.LOG2E                 Math.PI
Math.SQRT1_2               Math.SQRT2                 Math.abs
Math.acos                  Math.acosh                 Math.asi
n
Math.asinh                 Math.atan                  Math.ata
n2
Math.atanh                 Math.cbrt                  Math.cei
l
Math.clz32                 Math.cos                   Math.cos
h
Math.exp                   Math.expm1                 Math.flo
or
Math.fround                Math.hypot                 Math.imu
l
Math.log                   Math.log10                 Math.log
```

```
1p
Math.log2                   Math.max                    Math.min
Math.pow                    Math.random                 Math.rou
nd
Math.sign                   Math.sin                    Math.sin
h
Math.sqrt                   Math.tan                    Math.tan
h
Math.trunc
```

Depending on the screen size of your shell, the output may be displayed with a different number of rows and columns. This is a list of all the functions and properties that are available in the `Math` module.

Press `CTRL+C` to get to a new line in the prompt without executing what is in the current line.

Knowing the REPL shortcuts makes you more efficient when using it. Though, there's another thing REPL provides for increased productivity—The REPL commands.

Step 4 — Using REPL Commands

The REPL has specific keywords to help control its behavior. Each command begins with a dot `.`.

.help

To list all the available commands, use the `.help` command:

```
> .help
```

There aren't many, but they're useful for getting things done in the REPL:

```
.break    Sometimes you get stuck, this gets you out
.clear    Alias for .break
.editor   Enter editor mode
.exit     Exit the repl
.help     Print this help message
.load     Load JS from a file into the REPL session
.save     Save all evaluated commands in this REPL session to
 a file

Press ^C to abort current expression, ^D to exit the repl
```

If ever you forget a command, you can always refer to `.help` to see what it does.

.break/.clear

Using `.break` or `.clear`, it's easy to exit a multi-line expression. For example, begin a `for loop` as follows:

```
> for (let i = 0; i < 100000000; i++) {
```

To exit from entering any more lines, instead of entering the next one, use the `.break` or `.clear` command to break out:

```
... .break
```

You'll see a new prompt:

```
>
```

The REPL will move on to a new line without executing any code, similar to pressing `CTRL+C`.

.save and .load

The `.save` command stores all the code you ran since starting the REPL, into a file. The `.load` command runs all the JavaScript code from a file inside the REPL.

Quit the session using the `.exit` command or with the `CTRL+D` shortcut. Now start a new REPL with `node`. Now only the code you are about to write will be saved.

Create an array with fruits:

```
> fruits = ['banana', 'apple', 'mango']
```

In the next line, the REPL will display:

```
[ 'banana', 'apple', 'mango' ]
```

Save this variable to a new file, `fruits.js`:

```
> .save fruits.js
```

We're greeted with the confirmation:

```
Session saved to: fruits.js
```

The file is saved in the same directory where you opened the Node.js REPL. For example, if you opened the Node.js REPL in your home directory, then your file will be saved in your home directory.

Exit the session and start a new REPL with `node`. At the prompt, load the `fruits.js` file by entering:

```
  > .load fruits.js
```

This results in:

```
fruits = ['banana', 'apple', 'mango']


[ 'banana', 'apple', 'mango' ]
```

The `.load` command reads each line of code and executes it, as expected of a JavaScript interpreter. You can now use the `fruits` variable as if it was available in the current session all the time.

Type the following command and press `ENTER`:

```
> fruits[1]
```

The REPL will output:

```
'apple'
```

You can load any JavaScript file with the `.load` command, not only items you saved. Let's quickly demonstrate by opening your preferred code editor or `nano`, a command line editor, and create a new file called `peanuts.js`:

```
nano peanuts.js
```

Now that the file is open, type the following:

peanuts.js

```
console.log('I love peanuts!');
```

Save and exit nano by pressing `CTRL+X`.

In the same directory where you saved `peanuts.js`, start the Node.js REPL with `node`. Load `peanuts.js` in your session:

```
> .load peanuts.js
```

The `.load` command will execute the single `console` statement and display the following output:

```
console.log('I love peanuts!');


I love peanuts!
undefined
>
```

When your REPL usage goes longer than expected, or you believe you have an interesting code snippet worth sharing or explore in more depth, you can use the `.save` and `.load` commands to make both those goals possible.

Conclusion

The REPL is an interactive environment that allows you to execute JavaScript code without first having to write it to a file.

You can use the REPL to try out JavaScript code from other tutorials:

- How To Define Functions in JavaScript
- How To Use the Switch Statement in JavaScript
- How To Use Object Methods in JavaScript
- How To Index, Split, and Manipulate Strings in JavaScript

How To Use Node.js Modules with npm and package.json

Written by Stack Abuse

The author selected the Open Internet/Free Speech Fund to receive a donation as part of the Write for DOnations program.

Because of such features as its speedy Input/Output (I/O) performance and its well-known JavaScript syntax, Node.js has quickly become a popular runtime environment for back-end web development. But as interest grows, larger applications are built, and managing the complexity of the codebase and its dependencies becomes more difficult. Node.js organizes this complexity using modules, which are any single JavaScript files containing functions or objects that can be used by other programs or modules. A collection of one or more modules is commonly referred to as a package, and these packages are themselves organized by package managers.

The Node.js Package Manager (npm) is the default and most popular package manager in the Node.js ecosystem, and is primarily used to install and manage external modules in a Node.js project. It is also commonly used to install a wide range of CLI tools and run project scripts. npm tracks the modules installed in a project with the `package.json` file, which resides in a project's directory and contains:

- All the modules needed for a project and their installed versions
- All the metadata for a project, such as the author, the license, etc.
- Scripts that can be run to automate tasks within the project

As you create more complex Node.js projects, managing your metadata and dependencies with the `package.json` file will provide you with more predictable builds, since all external dependencies are kept the same. The file will keep track of this information automatically; while you may change the file directly to update your project's metadata, you will seldom need to interact with it directly to manage modules.

In this tutorial, you will manage packages with npm. The first step will be to create and understand the `package.json` file. You will then use it to keep track of all the modules you install in your project. Finally, you will list your package dependencies, update your packages, uninstall your packages, and perform an audit to find security flaws in your packages.

Prerequisites

To complete this tutorial, you will need:

- Node.js installed on your development machine. This tutorial uses version 10.17.0. To install this on macOS or Ubuntu 18.04, follow the steps in How to Install Node.js and Create a Local Development Environment on macOS or the **Installing Using a PPA** section of How To Install Node.js on Ubuntu 18.04. By having Node.js installed you will also have npm installed; this tutorial uses version 6.11.3.

Step 1 — Creating a `package.json` File

We begin this tutorial by setting up the example project—a fictional Node.js `locator` module that gets the user's IP address and returns the country of origin. You will not be coding the module in this tutorial. However, the packages you manage would be relevant if you were developing it.

First, you will create a `package.json` file to store useful metadata about the project and help you manage the project's dependent Node.js modules. As the suffix suggests, this is a JSON (JavaScript Object Notation) file. JSON is a standard format used for sharing, based on JavaScript objects and consisting of data stored as key-value pairs. If you would like to learn more about JSON, read our Introduction to JSON article.

Since a `package.json` file contains numerous properties, it can be cumbersome to create manually, without copy and pasting a template from somewhere else. To make things easier, npm provides the `init` command. This is an interactive command that asks you a series of questions and creates a `package.json` file based on your answers.

Using the `init` Command

First, set up a project so you can practice managing modules. In your shell, create a new folder called `locator`:

```
mkdir locator
```

Then move into the new folder:

```
cd locator
```

Now, initialize the interactive prompt by entering:

```
npm init
```

Note: If your code will use Git for version control, create the Git repository first and then run `npm init`. The command automatically

understands that it is in a Git-enabled folder. If a Git remote is set, it automatically fills out the `repository`, `bugs`, and `homepage` fields for your `package.json` file. If you initialized the repo after creating the `package.json` file, you will have to add this information in yourself. For more on Git version control, see our Introduction to Git: Installation, Usage, and Branches series.

You will receive the following output:

```
Output
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (locator)
```

You will first be prompted for the `name` of your new project. By default, the command assumes it's the name of the folder you're in. Default values

for each property are shown in parentheses `()`. Since the default value for `name` will work for this tutorial, press `ENTER` to accept it.

The next value to enter is `version`. Along with the `name`, this field is required if your project will be shared with others in the npm package repository.

Note: Node.js packages are expected to follow the Semantic Versioning (semver) guide. Therefore, the first number will be the `MAJOR` version number that only changes when the API changes. The second number will be the `MINOR` version that changes when features are added. The last number will be the `PATCH` version that changes when bugs are fixed.

Press `ENTER` so the default version is accepted.

The next field is `description`—a useful string to explain what your Node.js module does. Our fictional `locator` project would get the user's IP address and return the country of origin. A fitting `description` would be `Finds the country of origin of the incoming request`, so type in something like this and press `ENTER`. The `description` is very useful when people are searching for your module.

The following prompt will ask you for the `entry point`. If someone installs and `requires` your module, what you set in the `entry point` will be the first part of your program that is loaded. The value needs to be the relative location of a JavaScript file, and will be added to the `main` property of the `package.json`. Press `ENTER` to keep the default value.

Note: Most modules have an `index.js` file as the main point of entry. This is the default value for a `package.json`'s `main` property, which is the

point of entry for npm modules. If there is no `package.json`, Node.js will try to load `index.js` by default.

Next, you'll be asked for a `test command`, an executable script or command to run your project tests. In many popular Node.js modules, tests are written and executed with Mocha, Jest, Jasmine, or other test frameworks. Since testing is beyond the scope of this article, leave this option empty for now, and press `ENTER` to move on.

The `init` command will then ask for the project's GitHub Repository. You won't use this in this example, so leave it empty as well.

After the repository prompt, the command asks for `keywords`. This property is an array of strings with useful terms that people can use to find your repository. It's best to have a small set of words that are really relevant to your project, so that searching can be more targeted. List these keywords as a string with commas separating each value. For this sample project, type `ip,geo,country` at the prompt. The finished `package.json` will have three items in the array for `keywords`.

The next field in the prompt is `author`. This is useful for users of your module who want to get in contact with you. For example, if someone discovers an exploit in your module, they can use this to report the problem so that you can fix it. The `author` field is a string in the following format: `"Name \<Email\> (Website)"`. For example, `"Sammy \<sammy@your_domain\> (https://your_domain)"` is a valid author. The email and website data are optional—a valid author could just be a name. Add your contact details as an author and confirm with `ENTER`.

Finally, you'll be prompted for the `license`. This determines the legal permissions and limitations users will have while using your module. Many

Node.js modules are open source, so npm sets the default to ISC.

At this point, you would review your licensing options and decide what's best for your project. For more information on different types of open source licenses, see this license list from the Open Source Initiative. If you do not want to provide a license for a private repository, you can type `UNLICENSED` at the prompt. For this sample, use the default ISC license, and press `ENTER` to finish this process.

The `init` command will now display the `package.json` file it's going to create. It will look similar to this:

```
Output
About to write to /home/sammy/locator/package.json:


{
  "name": "locator",
  "version": "1.0.0",
  "description": "Finds the country of origin of the incoming
 request",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "ip",
    "geo",
    "country"
  ],
  "author": "Sammy <sammy@your_domain> (https://your_domain)",
  "license": "ISC"
}



Is this OK? (yes)
```

Once the information matches what you see here, press `ENTER` to complete this process and create the `package.json` file. With this file, you

can keep a record of modules you install for your project.

Now that you have your `package.json` file, you can test out installing modules in the next step.

Step 2 — Installing Modules

It is common in software development to use external libraries to perform ancillary tasks in projects. This allows the developer to focus on the business logic and create the application more quickly and efficiently.

For example, if our sample `locator` module has to make an external API request to get geographical data, we could use an HTTP library to make that task easier. Since our main goal is to return pertinent geographical data to the user, we could install a package that makes HTTP requests easier for us instead of rewriting this code for ourselves, a task that is beyond the scope of our project.

Let's run through this example. In your `locator` application, you will use the axios library, which will help you make HTTP requests. Install it by entering the following in your shell:

```
npm install axios --save
```

You begin this command with `npm install`, which will install the package (for brevity you can use `npm i`). You then list the packages that you want installed, separated by a space. In this case, this is `axios`. Finally, you end the command with the optional `--save` parameter, which specifies that `axios` will be saved as a project dependency.

When the library is installed, you will see output similar to the following:

```
Output
...
+ axios@0.19.0
added 5 packages from 8 contributors and audited 5 packages in
0.764s
found 0 vulnerabilities
```

Now, open the `package.json` file, using a text editor of your choice. This tutorial will use `nano`:

```
nano package.json
```

You'll see a new property, as highlighted in the following:

locator/package.json

```
{
  "name": "locator",
  "version": "1.0.0",
  "description": "Finds the country of origin of the incoming r
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "ip",
    "geo",
    "country"
  ],
  "author": "Sammy sammy@your_domain (https://your_domain)",
  "license": "ISC",
  "dependencies": {
    "axios": "^0.19.0"
  }
}
```

The `--save` option told npm to update the `package.json` with the module and version that was just installed. This is great, as other developers

working on your projects can easily see what external dependencies are needed.

Note: You may have noticed the `^` before the version number for the `axios` dependency. Recall that semantic versioning consists of three digits: **MAJOR**, **MINOR**, and **PATCH**. The `^` symbol signifies that any higher MINOR or PATCH version would satisfy this version constraint. If you see `~` at the beginning of a version number, then only higher PATCH versions satisfy the constraint.

When you are finished reviewing `package.json`, exit the file.

Development Dependencies

Packages that are used for the development of a project but not for building or running it in production are called development dependencies. They are not necessary for your module or application to work in production, but may be helpful while writing the code.

For example, it's common for developers to use code linters to ensure their code follows best practices and to keep the style consistent. While this is useful for development, this only adds to the size of the distributable without providing a tangible benefit when deployed in production.

Install a linter as a development dependency for your project. Try this out in your shell:

```
npm i eslint@6.0.0 --save-dev
```

In this command, you used the `--save-dev` flag. This will save `eslint` as a dependency that is only needed for development. Notice also that you added `@6.0.0` to your dependency name. When modules are updated, they

are tagged with a version. The `@` tells npm to look for a specific tag of the module you are installing. Without a specified tag, npm installs the latest tagged version. Open `package.json` again:

```
nano package.json
```

This will show the following:

locator/package.json

```
{
  "name": "locator",
  "version": "1.0.0",
  "description": "Finds the country of origin of the incoming r
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "ip",
    "geo",
    "country"
  ],
  "author": "Sammy sammy@your_domain (https://your_domain)",
  "license": "ISC",
  "dependencies": {
    "axios": "^0.19.0"
  },
  "devDependencies": {
    "eslint": "^6.0.0"
  }
}
```

`eslint` has been saved as a `devDependencies`, along with the version number you specified earlier. Exit `package.json`.

Automatically Generated Files: `node_modules` and `package-lock.json`

When you first install a package to a Node.js project, npm automatically creates the `node_modules` folder to store the modules needed for your project and the `package-lock.json` file that you examined earlier.

Confirm these are in your working directory. In your shell, type `ls` and press `ENTER`. You will observe the following output:

```
Output
node_modules    package.json    package-lock.json
```

The `node_modules` folder contains every installed dependency for your project. In most cases, you should **not** commit this folder into your version controlled repository. As you install more dependencies, the size of this folder will quickly grow. Furthermore, the `package-lock.json` file keeps a record of the exact versions installed in a more succinct way, so including `node_modules` is not necessary.

While the `package.json` file lists dependencies that tell us the suitable versions that should be installed for the project, the `package-lock.json` file keeps track of all changes in `package.json` or `node_modules` and tells us the exact version of the package installed. You usually commit this to your version controlled repository instead of `node_modules`, as it's a cleaner representation of all your dependencies.

Installing from package.json

With your `package.json` and `package-lock.json` files, you can quickly set up the same project dependencies before you start development on a new project. To demonstrate this, move up a level in your directory tree and create a new folder named `cloned_locator` in the same directory level as `locator`:

```
cd ..
mkdir cloned_locator
```

Move into your new directory:

```
cd cloned_locator
```

Now copy the `package.json` and `package-lock.json` files from `locator` to `cloned_locator`:

```
cp ../locator/package.json ../locator/package-lock.json .
```

To install the required modules for this project, type:

```
npm i
```

npm will check for a `package-lock.json` file to install the modules. If no lock file is available, it would read from the `package.json` file to determine the installations. It is usually quicker to install from `package-lock.json`, since the lock file contains the exact version of modules and their

dependencies, meaning npm does not have to spend time figuring out a suitable version to install.

When deploying to production, you may want to skip the development dependencies. Recall that development dependencies are stored in the `devDependencies` section of `package.json`, and have no impact on the running of your app. When installing modules as part of the CI/CD process to deploy your application, omit the dev dependencies by running:

```
npm i --production
```

The `--production` flag ignores the `devDependencies` section during installation. For now, stick with your development build.

Before moving to the next section, return to the `locator` folder:

```
cd ../locator
```

Global Installations

So far, you have been installing npm modules for the `locator` project. npm also allows you to install packages globally. This means that the package is available to your user in the wider system, like any other shell command. This ability is useful for the many Node.js modules that are CLI tools.

For example, you may want to blog about the `locator` project that you're currently working on. To do so, you can use a library like Hexo to create and manage your static website blog. Install the Hexo CLI globally like this:

```
npm i hexo-cli -g
```

To install a package globally, you append the `-g` flag to the command.

Note: If you get a permission error trying to install this package globally, your system may require super user privileges to run the command. Try again with `sudo npm i hexo-cli -g`.

Test that the package was successfully installed by typing:

```
hexo --version
```

You will see output similar to:

```
Output
hexo-cli: 2.0.0
os: Linux 4.15.0-64-generic linux x64
http_parser: 2.7.1
node: 10.14.0
v8: 7.6.303.29-node.16
uv: 1.31.0
zlib: 1.2.11
ares: 1.15.0
modules: 72
nghttp2: 1.39.2
openssl: 1.1.1c
brotli: 1.0.7
napi: 4
llhttp: 1.1.4
icu: 64.2
unicode: 12.1
cldr: 35.1
tz: 2019a
```

So far, you have learned how to install modules with npm. You can install packages to a project locally, either as a production or development dependency. You can also install packages based on pre-existing `package.json` or `package-lock.json` files, allowing you to develop with the same dependencies as your peers. Finally, you can use the `-g` flag to install

packages globally, so you can access them regardless of whether you're in a Node.js project or not.

Now that you can install modules, in the next section you will practice techniques to administer your dependencies.

Step 3 — Managing Modules

A complete package manager can do a lot more than install modules. npm has over 20 commands relating to dependency management available. In this step, you will:

- List modules you have installed.
- Update modules to a more recent version.
- Uninstall modules you no longer need.
- Perform a security audit on your modules to find and fix security flaws.

While these examples will be done in your `locator` folder, all of these commands can be run globally by appending the `-g` flag at the end of them, exactly like you did when installing globally.

Listing Modules

If you would like to know which modules are installed in a project, it would be easier to use the `list` or `ls` command instead of reading the `package.json` directly. To do this, enter:

```
npm ls
```

You will see output like this:

```
Output
├─┬ axios@0.19.0
│ ├─┬ follow-redirects@1.5.10
│ │ └─┬ debug@3.1.0
│ │   └── ms@2.0.0
│ └── is-buffer@2.0.3
└─┬ eslint@6.0.0
  ├─┬ @babel/code-frame@7.5.5
  │ └─┬ @babel/highlight@7.5.0
  │   ├── chalk@2.4.2 deduped
  │   ├── esutils@2.0.3 deduped
  │   └── js-tokens@4.0.0
  ├─┬ ajv@6.10.2
  │ ├── fast-deep-equal@2.0.1
  │ ├── fast-json-stable-stringify@2.0.0
  │ ├── json-schema-traverse@0.4.1
  │ └─┬ uri-js@4.2.2
...
```

By default, `ls` shows the entire dependency tree—the modules your project depends on and the modules that your dependencies depend on. This can be a bit unwieldy if you want a high-level overview of what's installed.

To only print the modules you installed without their dependencies, enter the following in your shell:

```
npm ls --depth 0
```

Your output will be:

```
Output
├── axios@0.19.0
└── eslint@6.0.0
```

The `--depth` option allows you to specify what level of the dependency tree you want to see. When it's `0`, you only see your top level dependencies.

Updating Modules

It is a good practice to keep your npm modules up to date. This improves your likelihood of getting the latest security fixes for a module. Use the `outdated` command to check if any modules can be updated:

```
npm outdated
```

You will get output like the following:

```
Output
Package  Current  Wanted  Latest  Location
eslint     6.0.0   6.7.1   6.7.1  locator
```

This command first lists the `Package` that's installed and the `Current` version. The `Wanted` column shows which version satisfies your version

requirement in `package.json`. The `Latest` column shows the most recent version of the module that was published.

The `Location` column states where in the dependency tree the package is located. The `outdated` command has the `--depth` flag like `ls`. By default, the depth is 0.

It seems that you can update `eslint` to a more recent version. Use the `update` or `up` command like this:

```
npm up eslint
```

The output of the command will contain the version installed:

```
Output
npm WARN locator@1.0.0 No repository field.

+ eslint@6.7.1
added 7 packages from 3 contributors, removed 5 packages, upda
ted 19 packages, moved 1 package and audited 184 packages in
 5.818s
found 0 vulnerabilities
```

If you wanted to update all modules at once, then you would enter:

```
npm up
```

Uninstalling Modules

The npm `uninstall` command can remove modules from your projects. This means the module will no longer be installed in the `node_modules` folder, nor will it be seen in your `package.json` and `package-lock.json` files.

Removing dependencies from a project is a normal activity in the software development lifecycle. A dependency may not solve the problem as advertised, or may not provide a satisfactory development experience. In these cases, it may better to uninstall the dependency and build your own module.

Imagine that `axios` does not provide the development experience you would have liked for making HTTP requests. Uninstall `axios` with the `uninstall` or `un` command by entering:

```
npm un axios
```

Your output will be similar to:

```
Output
npm WARN locator@1.0.0 No repository field.

removed 5 packages and audited 176 packages in 1.488s
found 0 vulnerabilities
```

It doesn't explicitly say that `axios` was removed. To verify that it was uninstalled, list the dependencies once again:

```
npm ls --depth 0
```

Now, we only see that `eslint` is installed:

```
Output
└── eslint@6.7.1
```

This shows that you have successfully uninstalled the `axios` package.

Auditing Modules

npm provides an `audit` command to highlight potential security risks in your dependencies. To see the audit in action, install an outdated version of the request module by running the following:

```
npm i request@2.60.0
```

When you install this outdated version of `request`, you'll notice output similar to the following:

```
Output
+ request@2.60.0
added 54 packages from 49 contributors and audited 243 package
s in 7.26s
found 6 moderate severity vulnerabilities
  run `npm audit fix` to fix them, or `npm audit` for details
```

npm is telling you that you have vulnerabilities in your dependencies. To get more details, audit your entire project with:

```
npm audit
```

The `audit` command shows tables of output highlighting security flaws:

Output

```
                       === npm audit security report ===


# Run  npm install request@2.88.0  to resolve 1 vulnerability
┌───────────────┬──────────────────────────────────────────────────────────────┐
└───────────────┘
│ Moderate      │ Memory Exposure
│
├───────────────┼──────────────────────────────────────────────────────────────┤
└───────────────┘
│ Package       │ tunnel-agent
│
├───────────────┼──────────────────────────────────────────────────────────────┤
└───────────────┘
│ Dependency of │ request
│
├───────────────┼──────────────────────────────────────────────────────────────┤
└───────────────┘
│ Path          │ request > tunnel-agent
│
├───────────────┼──────────────────────────────────────────────────────────────┤
└───────────────┘
│ More info     │ https://npmjs.com/advisories/598
│
└───────────────┴──────────────────────────────────────────────────────────────┘
└───────────────┘
```

```
# Run  npm update request --depth 1  to resolve 1 vulnerabilit
y
┌───────────────┬──────────────────────────────────────────────
──────────────┐
│ Moderate      │ Remote Memory Exposure
│
├───────────────┼──────────────────────────────────────────────
──────────────┤
│ Package       │ request
│
├───────────────┼──────────────────────────────────────────────
──────────────┤
│ Dependency of │ request
│
├───────────────┼──────────────────────────────────────────────
──────────────┤
│ Path          │ request
│
├───────────────┼──────────────────────────────────────────────
──────────────┤
│ More info     │ https://npmjs.com/advisories/309
│
└───────────────┴──────────────────────────────────────────────
──────────────┘
...
```

You can see the path of the vulnerability, and sometimes npm offers ways for you to fix it. You can run the update command as suggested, or you can run the `fix` subcommand of `audit`. In your shell, enter:

```
npm audit fix
```

You will see similar output to:

```
Output
+ request@2.88.0
added 19 packages from 24 contributors, removed 32 packages an
d updated 12 packages in 6.223s
fixed 2 of 6 vulnerabilities in 243 scanned packages
  4 vulnerabilities required manual review and could not be up
dated
```

npm was able to safely update two of the packages, decreasing your vulnerabilities by the same amount. However, you still have four vulnerabilities in your dependencies. The `audit fix` command does not always fix every problem. Although a version of a module may have a security vulnerability, if you update it to a version with a different API then it could break code higher up in the dependency tree.

You can use the `--force` parameter to ensure the vulnerabilities are gone, like this:

```
npm audit fix --force
```

As mentioned before, this is not recommended unless you are sure that it won't break functionality.

Conclusion

In this tutorial, you went through various exercises to demonstrate how Node.js modules are organized into packages, and how these packages are managed by npm. In a Node.js project, you used npm packages as dependencies by creating and maintaining a `package.json` file—a record of your project's metadata, including what modules you installed. You also used the npm CLI tool to install, update, and remove modules, in addition to listing the dependency tree for your projects and checking and updating modules that are outdated.

In the future, leveraging existing code by using modules will speed up development time, as you don't have to repeat functionality. You will also be able to create your own npm modules, and these will in turn will be managed by others via npm commands. As for next steps, experiment with what you learned in this tutorial by installing and testing the variety of packages out there. See what the ecosystem provides to make problem solving easier. For example, you could try out TypeScript, a superset of JavaScript, or turn your website into mobile apps with Cordova. If you'd like to learn more about Node.js, see our other Node.js tutorials.

How To Create a Node.js Module

Written by Stack Abuse

The author selected the Open Internet/Free Speech Fund to receive a donation as part of the Write for DOnations program.

In Node.js, a module is a collection of JavaScript functions and objects that can be used by external applications. Describing a piece of code as a module refers less to what the code is and more to what it does—any Node.js file or collection of files can be considered a module if its functions and data are made usable to external programs.

Because modules provide units of functionality that can be reused in many larger programs, they enable you to create loosely coupled applications that scale with complexity, and open the door for you to share your code with other developers. Being able to write modules that export useful functions and data will allow you to contribute to the wider Node.js community—in fact, all packages that you use on npm were bundled and shared as modules. This makes creating modules an essential skill for a Node.js developer.

In this tutorial, you will create a Node.js module that suggests what color web developers should use in their designs. You will develop the module by storing the colors as an array, and providing a function to retrieve one randomly. Afterwards, you will run through various ways of importing a module into a Node.js application.

Prerequisites

- You will need Node.js and npm installed on your development environment. This tutorial uses version 10.17.0. To install this on macOS or Ubuntu 18.04, follow the steps in How To Install Node.js and Create a Local Development Environment on macOS or the **Installing Using a PPA** section of How To Install Node.js on Ubuntu 18.04. By having Node.js installed you will also have npm installed; this tutorial uses version 6.11.3.
- You should also be familiar with the `package.json` file, and experience with npm commands would be useful as well. To gain this experience, follow How To Use Node.js Modules with npm and package.json, particularly the **Step 1 — Creating a `package.json` File**.
- It will also help to be comfortable with the Node.js REPL (Read-Evaluate-Print-Loop). You will use this to test your module. If you need more information on this, read our guide on How To Use the Node.js REPL.

Step 1 — Creating a Module

This step will guide you through creating your first Node.js module. Your module will contain a collection of colors in an array and provide a function to get one at random. You will use the Node.js built-in `exports` property to make the function and array available to external programs.

First, you'll begin by deciding what data about colors you will store in your module. Every color will be an object that contains a `name` property that humans can easily identify, and a `code` property that is a string containing an HTML color code. HTML color codes are six-digit hexadecimal numbers that allow you to change the color of elements on a

web page. You can learn more about HTML color codes by reading this HTML Color Codes and Names article.

You will then decide what colors you want to support in your module. Your module will contain an array called `allColors` that will contain six colors. Your module will also include a function called `getRandomColor()` that will randomly select a color from your array and return it.

In your terminal, make a new folder called `colors` and move into it:

```
mkdir colors
cd colors
```

Initialize npm so other programs can import this module later in the tutorial:

```
npm init -y
```

You used the `-y` flag to skip the usual prompts to customize your `packag e.json`. If this were a module you wished to publish to npm, you would answer all these prompts with relevant data, as explained in How To Use Node.js Modules with npm and package.json.

In this case, your output will be:

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

Now, open up a command-line text editor such as `nano` and create a new file to serve as the entry point for your module:

```
nano index.js
```

Your module will do a few things. First, you'll define a `Color` class. Your `Color` class will be instantiated with its name and HTML code. Add the following lines to create the class:

~/colors/index.js

```
class Color {
  constructor(name, code) {
    this.name = name;
    this.code = code;
  }
}
```

Now that you have your data structure for `Color`, add some instances into your module. Write the following highlighted array to your file:

~/colors/index.js

```
class Color {
  constructor(name, code) {
    this.name = name;
    this.code = code;
  }
}

const allColors = [
  new Color('brightred', '#E74C3C'),
  new Color('soothingpurple', '#9B59B6'),
  new Color('skyblue', '#5DADE2'),
  new Color('leafygreen', '#48C9B0'),
  new Color('sunkissedyellow', '#F4D03F'),
  new Color('groovygray', '#D7DBDD'),
];
```

Finally, enter a function that randomly selects an item from the `allColors` array you just created:

~/colors/index.js

```
class Color {
  constructor(name, code) {
    this.name = name;
    this.code = code;
  }
}

const allColors = [
  new Color('brightred', '#E74C3C'),
  new Color('soothingpurple', '#9B59B6'),
  new Color('skyblue', '#5DADE2'),
  new Color('leafygreen', '#48C9B0'),
  new Color('sunkissedyellow', '#F4D03F'),
  new Color('groovygray', '#D7DBDD'),
];

exports.getRandomColor = () => {
  return allColors[Math.floor(Math.random() * allColors.length)
}

exports.allColors = allColors;
```

The `exports` keyword references a global object available in every Node.js module. All functions and objects stored in a module's `exports` object are exposed when other Node.js modules import it. The `getRandomColor()` function was created directly on the `exports` object, for example. You then added an `allColors` property to the `exports` object that references the local constant `allColors` array created earlier in the script.

When other modules import this module, both `allColors` and `getRandomColor()` will be exposed and available for usage.

Save and exit the file.

So far, you have created a module that contains an array of colors and a function that returns one randomly. You have also exported the array and function, so that external programs can use them. In the next step, you will use your module in other applications to demonstrate the effects of `export`.

Step 2 — Testing your Module with the REPL

Before you build a complete application, take a moment to confirm that your module is working. In this step, you will use the REPL to load the `colors` module. While in the REPL, you will call the `getRandomColor()` function to see if it behaves as you expect it to.

Start the Node.js REPL in the same folder as the `index.js` file:

```
node
```

When the REPL has started, you will see the `>` prompt. This means you can enter JavaScript code that will be immediately evaluated. If you would like to read more about this, follow our guide on using the REPL.

First, enter the following:

```
colors = require('./index');
```

In this command, `require()` loads the `colors` module at its entry point. When you press `ENTER` you will get:

```
Output
{
  getRandomColor: [Function],
  allColors: [
    Color { name: 'brightred', code: '#E74C3C' },
    Color { name: 'soothingpurple', code: '#9B59B6' },
    Color { name: 'skyblue', code: '#5DADE2' },
    Color { name: 'leafygreen', code: '#48C9B0' },
    Color { name: 'sunkissedyellow', code: '#F4D03F' },
    Color { name: 'groovygray', code: '#D7DBDD' }
  ]
}
```

The REPL shows us the value of `colors`, which are all the functions and objects imported from the `index.js` file. When you use the `require` keyword, Node.js returns all the contents within the `exports` object of a module.

Recall that you added `getRandomColor()` and `allColors` to `exports` in the `colors` module. For that reason, you see them both in the REPL when

they are imported.

At the prompt, test the `getRandomColor()` function:

```
colors.getRandomColor();
```

You'll be prompted with a random color:

```
Output
Color { name: 'groovygray', code: '#D7DBDD' }
```

As the index is random, your output may vary. Now that you confirmed that the `colors` module is working, exit the Node.js REPL:

```
.exit
```

This will return you to your terminal command line.

You have just confirmed that your module works as expected using the REPL. Next, you will apply these same concepts and load your module into an application, as you would do in a real project.

Step 3 — Saving your Local Module as a Dependency

While testing your module in the REPL, you imported it with a relative path. This means you used the location of the `index.js` file in relation to the working directory to get its contents. While this works, it is usually a better programming experience to import modules by their names so that the import is not broken when the context is changed. In this step, you will install the `colors` module with npm's local module `install` feature.

Set up a new Node.js module outside the `colors` folder. First, go to the previous directory and create a new folder:

```
cd ..
mkdir really-large-application
```

Now move into your new project:

```
cd really-large-application
```

Like with the `colors` module, initialize your folder with npm:

```
npm init -y
```

The following `package.json` will be generated:

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

Now, install your `colors` module and use the `--save` flag so it will be recorded in your `package.json` file:

```
npm install --save ../colors
```

You just installed your `colors` module in the new project. Open the `package.json` file to see the new local dependency:

```
nano package.json
```

You will find that the following highlighted lines have been added:

~/really-large-application/package.json

```
{
  "name": "really-large-application",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "colors": "file:../colors"
  }
}
```

Exit the file.

The `colors` module was copied to your `node_modules` directory. Verify it's there with the following command:

```
ls node_modules
```

This will give the following output:

```
Output
colors
```

Use your installed local module in this new program. Re-open your text editor and create another JavaScript file:

```
nano index.js
```

Your program will first import the `colors` module. It will then choose a color at random using the `getRandomColor()` function provided by the module. Finally, it will print a message to the console that tells the user what color to use.

Enter the following code in `index.js`:

~/really-large-application/index.js

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


const chosenColor = colors.getRandomColor();
console.log(`You should use ${chosenColor.name} on your website
```

Save and exit this file.

Your application will now tell the user a random color option for a website component.

Run this script with:

```
node index.js
```

Your output will be similar to:

```
Output
You should use leafygreen on your website. It's HTML code is #
48C9B0
```

You've now successfully installed the `colors` module and can manage it like any other npm package used in your project. However, if you added more colors and functions to your local `colors` module, you would have to run `npm update` in your applications to be able to use the new options. In the next step, you will use the local module `colors` in another way and get automatic updates when the module code changes.

Step 4 — Linking a Local Module

If your local module is in heavy development, continually updating packages can be tedious. An alternative would be to link the modules. Linking a module ensures that any updates to the module are immediately reflected in the applications using it.

In this step, you will link the `colors` module to your application. You will also modify the `colors` module and confirm that its most recent changes work in the application without having to reinstall or upgrade.

First, uninstall your local module:

```
npm un colors
```

npm links modules by using symbolic links (or symlinks), which are references that point to files or directories in your computer. Linking a module is done in two steps:

1. Creating a global link to the module. npm creates a symlink between your global `node_modules` directory and the directory of your module. The global `node_modules` directory is the location in which all your system-wide npm packages are installed (any package you install with the `-g` flag).
2. Create a local link. npm creates a symlink between your local project that's using the module and the global link of the module.

First, create the global link by returning to the `colors` folder and using the `link` command:

```
cd ../colors
sudo npm link
```

Once complete, your shell will output:

```
Output
/usr/local/lib/node_modules/colors -> /home/sammy/colors
```

You just created a symlink in your `node_modules` folder to your `colors` directory.

Return to the `really-large-application` folder and link the package:

```
cd ../really-large-application
sudo npm link colors
```

You will receive output similar to the following:

```
Output
/home/sammy/really-large-application/node_modules/colors -> /u
sr/local/lib/node_modules/colors -> /home/sammy/colors
```

Note: If you would like to type a bit less, you can use `ln` instead of `link`. For example, `npm ln colors` would have worked the exact same way.

As the output shows, you just created a symlink from your `really-large-application`'s local `node_modules` directory to the `colors` symlink in your global `node_modules`, which points to the actual directory with the `colors` module.

The linking process is complete. Run your file to ensure it still works:

```
node index.js
```

Your output will be similar to:

```
Output
You should use sunkissedyellow on your website. It's HTML code
is #F4D03F
```

Your program functionality is intact. Next, test that updates are immediately applied. In your text editor, re-open the `index.js` file in the `colors` module:

```
cd ../colors
nano index.js
```

Now add a function that selects the very best shade of blue that exists. It takes no arguments, and always returns the third item of the `allColors` array. Add these lines to the end of the file:

~/colors/index.js

```
class Color {
  constructor(name, code) {
    this.name = name;
    this.code = code;
  }
}

const allColors = [
  new Color('brightred', '#E74C3C'),
  new Color('soothingpurple', '#9B59B6'),
  new Color('skyblue', '#5DADE2'),
  new Color('leafygreen', '#48C9B0'),
  new Color('sunkissedyellow', '#F4D03F'),
  new Color('groovygray', '#D7DBDD'),
];

exports.getRandomColor = () => {
        return allColors[Math.floor(Math.random() * allColors.l
        }

exports.allColors = allColors;

exports.getBlue = () => {
  return allColors[2];
}
```

Save and exit the file, then re-open the `index.js` file in the `really-large-application` folder:

```
cd ../really-large-application
nano index.js
```

Make a call to the newly created `getBlue()` function, and print a sentence with the color's properties. Add these statements to the end of the file:

~/really-large-application/index.js

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

const chosenColor = colors.getRandomColor();
console.log(`You should use ${chosenColor.name} on your website

const favoriteColor = colors.getBlue();
console.log(`My favorite color is ${favoriteColor.name}/${favor
```

Save and exit the file.

The code now uses the newly create `getBlue()` function. Execute the file as before:

```
node index.js
```

You will get output like:

```
Output
You should use brightred on your website. It's HTML code is #E
74C3C
My favorite color is skyblue/#5DADE2, btw
```

Your script was able to use the latest function in your `colors` module, without having to run `npm update`. This will make it easier to make changes to this application in development.

As you write larger and more complex applications, think about how related code can be grouped into modules, and how you want these modules to be set up. If your module is only going to be used by one program, it can stay within the same project and be referenced by a relative path. If your module will later be shared separately or exists in a very different location from the project you are working on now, installing or linking might be more viable. Modules in active development also benefit from the automatic updates of linking. If the module is not under active development, using `npm install` may be the easier option.

Conclusion

In this tutorial, you learned that a Node.js module is a JavaScript file with functions and objects that can be used by other programs. You then created a module and attached your functions and objects to the global `exports`

object to make them available to external programs. Finally, you imported that module into a program, demonstrating how modules come together into larger applications.

Now that you know how to create modules, think about the type of program you want to write and break it down into various components, keeping each unique set of activities and data in their own modules. The more practice you get writing modules, the better your ability to write quality Node.js programs on your learning journey. To work through an example of a Node.js application that uses modules, see our How To Set Up a Node.js Application for Production on Ubuntu 18.04 tutorial.

How To Write Asynchronous Code in Node.js

Written by Stack Abuse

The author selected the Open Internet/Free Speech Fund to receive a donation as part of the Write for DOnations program.

For many programs in JavaScript, code is executed as the developer writes it—line by line. This is called synchronous execution, because the lines are executed one after the other, in the order they were written. However, not every instruction you give to the computer needs to be attended to immediately. For example, if you send a network request, the process executing your code will have to wait for the data to return before it can work on it. In this case, time would be wasted if it did not execute other code while waiting for the network request to be completed. To solve this problem, developers use asynchronous programming, in which lines of code are executed in a different order than the one in which they were written. With asynchronous programming, we can execute other code while we wait for long activities like network requests to finish.

JavaScript code is executed on a single thread within a computer process. Its code is processed synchronously on this thread, with only one instruction run at a time. Therefore, if we were to do a long-running task on this thread, all of the remaining code is blocked until the task is complete. By leveraging JavaScript's asynchronous programming features, we can offload long-running tasks to a background thread to avoid this problem. When the task is complete, the code we need to process the task's data is put back on the main single thread.

In this tutorial, you will learn how JavaScript manages asynchronous tasks with help from the Event Loop, which is a JavaScript construct that completes a new task while waiting for another. You will then create a program that uses asynchronous programming to request a list of movies from a Studio Ghibli API and save the data to a CSV file. The asynchronous code will be written in three ways: callbacks, promises, and with the `async`/`await` keywords.

Note: As of this writing, asynchronous programming is no longer done using only callbacks, but learning this obsolete method can provide great context as to why the

JavaScript community now uses promises. The `async`/`await` keywords enable us to use promises in a less verbose way, and are thus the standard way to do asynchronous programming in JavaScript at the time of writing this article.

Prerequisites

- Node.js installed on your development machine. This tutorial uses version 10.17.0. To install this on macOS or Ubuntu 18.04, follow the steps in How to Install Node.js and Create a Local Development Environment on macOS or the **Installing Using a PPA** section of How To Install Node.js on Ubuntu 18.04.
- You will also need to be familiar with installing packages in your project. Get up to speed by reading our guide on How To Use Node.js Modules with npm and package.json.
- It is important that you're comfortable creating and executing functions in JavaScript before learning how to use them asynchronously. If you need an introduction or refresher, you can read our guide on How To Define Functions in JavaScript

The Event Loop

Let's begin by studying the internal workings of JavaScript function execution. Understanding how this behaves will allow you to write asynchronous code more deliberately, and will help you with troubleshooting code in the future.

As the JavaScript interpreter executes the code, every function that is called is added to JavaScript's call stack. The call stack is a stack—a list-like data structure where items can only be added to the top, and removed from the top. Stacks follow the "Last in, first out" or LIFO principle. If you add two items on the stack, the most recently added item is removed first.

Let's illustrate with an example using the call stack. If JavaScript encounters a function `functionA()` being called, it is added to the call stack. If that function `functionA()` calls another function `functionB()`, then `functionB()` is added to the top of the call stack. As JavaScript completes the execution of a function, it is removed from the call stack. Therefore, JavaScript will execute `functionB()` first, remove it from the stack when

complete, and then finish the execution of `functionA()` and remove it from the call stack. This is why inner functions are always executed before their outer functions.

When JavaScript encounters an asynchronous operation, like writing to a file, it adds it to a table in its memory. This table stores the operation, the condition for it to be completed, and the function to be called when it's completed. As the operation completes, JavaScript adds the associated function to the message queue. A queue is another list-like data structure where items can only be added to the bottom but removed from the top. In the message queue, if two or more asynchronous operations are ready for their functions to be executed, the asynchronous operation that was completed first will have its function marked for execution first.

Functions in the message queue are waiting to be added to the call stack. The event loop is a perpetual process that checks if the call stack is empty. If it is, then the first item in the message queue is moved to the call stack. JavaScript prioritizes functions in the message queue over function calls it interprets in the code. The combined effect of the call stack, message queue, and event loop allows JavaScript code to be processed while managing asynchronous activities.

Now that you have a high-level understanding of the event loop, you know how the asynchronous code you write will be executed. With this knowledge, you can now create asynchronous code with three different approaches: callbacks, promises, and `async`/`await`.

Asynchronous Programming with Callbacks

A callback function is one that is passed as an argument to another function, and then executed when the other function is finished. We use callbacks to ensure that code is executed only after an asynchronous operation is completed.

For a long time, callbacks were the most common mechanism for writing asynchronous code, but now they have largely become obsolete because they can make code confusing to read. In this step, you'll write an example of asynchronous code using callbacks so that you can use it as a baseline to see the increased efficiency of other strategies.

There are many ways to use callback functions in another function. Generally, they take this structure:

```
function asynchronousFunction([ Function Arguments ], [ Callback Function ]) {
    [ Action ]
}
```

While it is not syntactically required by JavaScript or Node.js to have the callback function as the last argument of the outer function, it is a common practice that makes callbacks easier to identify. It's also common for JavaScript developers to use an anonymous function as a callback. Anonymous functions are those created without a name. It's usually much more readable when a function is defined at the end of the argument list.

To demonstrate callbacks, let's create a Node.js module that writes a list of Studio Ghibli movies to a file. First, create a folder that will store our JavaScript file and its output:

```
mkdir ghibliMovies
```

Then enter that folder:

```
cd ghibliMovies
```

We will start by making an HTTP request to the Studio Ghibli API, which our callback function will log the results of. To do this, we will install a library that allows us to access the data of an HTTP response in a callback.

In your terminal, initialize npm so we can have a reference for our packages later:

```
npm init -y
```

Then, install the `request` library:

```
npm i request --save
```

Now open a new file called `callbackMovies.js` in a text editor like `nano`:

```
nano callbackMovies.js
```

In your text editor, enter the following code. Let's begin by sending an HTTP request with the `request` module:

callbackMovies.js

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


request('https://ghibliapi.herokuapp.com/films');
```

In the first line, we load the `request` module that was installed via npm. The module returns a function that can make HTTP requests; we then save that function in the `request` constant.

We then make the HTTP request using the `request()` function. Let's now print the data from the HTTP request to the console by adding the highlighted changes:

callbackMovies.js

```
const request = require('request');
request('https://ghibliapi.herokuapp.com/films', (error, response, body) =>
    if (error) {
        console.error(`Could not send request to API: ${error.message}`);
        return;
    }

    if (response.statusCode != 200) {        console.error(`Expected status
        return;
    }

    console.log('Processing our list of movies');
    movies = JSON.parse(body);
    movies.forEach(movie => {
        console.log(`${movie['title']}, ${movie['release_date']}`);
    });
});
```

When we use the `request()` function, we give it two parameters:

- The URL of the website we are trying to request
- A callback function that handles any errors or successful responses after the request is complete

Our callback function has three arguments: `error`, `response`, and `body`. When the HTTP request is complete, the arguments are automatically given values depending on the outcome. If the request failed to send, then `error` would contain an object, but `respo nse` and `body` would be `null`. If it made the request successfully, then the HTTP

response is stored in `response`. If our HTTP response returns data (in this example we get JSON) then the data is set in `body`.

Our callback function first checks to see if we received an error. It's best practice to check for errors in a callback first so the execution of the callback won't continue with missing data. In this case, we log the error and the function's execution. We then check the status code of the response. Our server may not always be available, and APIs can change causing once sensible requests to become incorrect. By checking that the status code is `200`, which means the request was "OK", we can have confidence that our response is what we expect it to be.

Finally, we parse the response body to an `Array` and loop through each movie to log its name and release year.

After saving and quitting the file, run this script with:

```
node callbackMovies.js
```

You will get the following output:

```
Output
Castle in the Sky, 1986
Grave of the Fireflies, 1988
My Neighbor Totoro, 1988
Kiki's Delivery Service, 1989
Only Yesterday, 1991
Porco Rosso, 1992
Pom Poko, 1994
Whisper of the Heart, 1995
Princess Mononoke, 1997
My Neighbors the Yamadas, 1999
Spirited Away, 2001
The Cat Returns, 2002
Howl's Moving Castle, 2004
Tales from Earthsea, 2006
Ponyo, 2008
Arrietty, 2010
From Up on Poppy Hill, 2011
The Wind Rises, 2013
The Tale of the Princess Kaguya, 2013
When Marnie Was There, 2014
```

We successfully received a list of Studio Ghibli movies with the year they were released. Now let's complete this program by writing the movie list we are currently logging into a file.

Update the `callbackMovies.js` file in your text editor to include the following highlighted code, which creates a CSV file with our movie data:

callbackMovies.js

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

request('https://ghibliapi.herokuapp.com/films', (error, response, body) =>
    if (error) {
        console.error(`Could not send request to API: ${error.message}`);
        return;
    }

    if (response.statusCode != 200) {        console.error(`Expected status
        return;
    }

    console.log('Processing our list of movies');
    movies = JSON.parse(body);
    let movieList = '';
    movies.forEach(movie => {
        movieList += `${movie['title']}, ${movie['release_date']}\n`;
    });

    fs.writeFile('callbackMovies.csv', movieList, (error) => {
        if (error) {
            console.error(`Could not save the Ghibli movies to a file: ${er
            return;
        }

        console.log('Saved our list of movies to callbackMovies.csv');;
    });
});
```

Noting the highlighted changes, we see that we import the `fs` module. This module is standard in all Node.js installations, and it contains a `writeFile()` method that can asynchronously write to a file.

Instead of logging the data to the console, we now add it to a string variable `movieList`. We then use `writeFile()` to save the contents of `movieList` to a new file—`callbackMovies.csv`. Finally, we provide a callback to the `writeFile()` function, which has one argument: `error`. This allows us to handle cases where we are not able to write to a file, for example when the user we are running the `node` process on does not have those permissions.

Save the file and run this Node.js program once again with:

```
node callbackMovies.js
```

In your `ghibliMovies` folder, you will see `callbackMovies.csv`, which has the following content:

callbackMovies.csv

```
Castle in the Sky, 1986
Grave of the Fireflies, 1988
My Neighbor Totoro, 1988
Kiki's Delivery Service, 1989
Only Yesterday, 1991
Porco Rosso, 1992
Pom Poko, 1994
Whisper of the Heart, 1995
Princess Mononoke, 1997
My Neighbors the Yamadas, 1999
Spirited Away, 2001
The Cat Returns, 2002
Howl's Moving Castle, 2004
Tales from Earthsea, 2006
Ponyo, 2008
Arrietty, 2010
From Up on Poppy Hill, 2011
The Wind Rises, 2013
The Tale of the Princess Kaguya, 2013
When Marnie Was There, 2014
```

It's important to note that we write to our CSV file in the callback of the HTTP request. Once the code is in the callback function, it will only write to the file after the HTTP request was completed. If we wanted to communicate to a database after we wrote our CSV file, we would make another asynchronous function that would be called in the callback of `writeFile()`. The more asynchronous code we have, the more callback functions have to be nested.

Let's imagine that we want to execute five asynchronous operations, each one only able to run when another is complete. If we were to code this, we would have something like this:

```
doSomething1(() => {
    doSomething2(() => {
        doSomething3(() => {
            doSomething4(() => {
                doSomething5(() => {
                    // final action
                });
            });
        });
    });
});
```

When nested callbacks have many lines of code to execute, they become substantially more complex and unreadable. As your JavaScript project grows in size and complexity, this effect will become more pronounced, until it is eventually unmanageable. Because of this, developers no longer use callbacks to handle asynchronous operations. To improve the syntax of our asynchronous code, we can use promises instead.

Using Promises for Concise Asynchronous Programming

A promise is a JavaScript object that will return a value at some point in the future. Asynchronous functions can return promise objects instead of concrete values. If we get a value in the future, we say that the promise was fulfilled. If we get an error in the future, we say that the promise was rejected. Otherwise, the promise is still being worked on in a pending state.

Promises generally take the following form:

```
promiseFunction()
    .then([ Callback Function for Fulfilled Promise ])
    .catch([ Callback Function for Rejected Promise ])
```

As shown in this template, promises also use callback functions. We have a callback function for the `then()` method, which is executed when a promise is fulfilled. We also have a callback function for the `catch()` method to handle any errors that come up while the promise is being executed.

Let's get firsthand experience with promises by rewriting our Studio Ghibli program to use promises instead.

Axios is a promise-based HTTP client for JavaScript, so let's go ahead and install it:

```
npm i axios --save
```

Now, with your text editor of choice, create a new file `promiseMovies.js`:

```
nano promiseMovies.js
```

Our program will make an HTTP request with `axios` and then use a special promised-based version of `fs` to save to a new CSV file.

Type this code in `promiseMovies.js` so we can load Axios and send an HTTP request to the movie API:

promiseMovies.js

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


axios.get('https://ghibliapi.herokuapp.com/films');
```

In the first line we load the `axios` module, storing the returned function in a constant called `axios`. We then use the `axios.get()` method to send an HTTP request to the API.

The `axios.get()` method returns a promise. Let's chain that promise so we can print the list of Ghibli movies to the console:

promiseMovies.js

```
const axios = require('axios');
const fs = require('fs').promises;



axios.get('https://ghibliapi.herokuapp.com/films')
    .then((response) => {
        console.log('Successfully retrieved our list of movies');
        response.data.forEach(movie => {
            console.log(`${movie['title']}, ${movie['release_date']}`);
        });
    })
```

Let's break down what's happening. After making an HTTP GET request with `axios.get()`, we use the `then()` function, which is only executed when the promise is fulfilled. In this case, we print the movies to the screen like we did in the callbacks example.

To improve this program, add the highlighted code to write the HTTP data to a file:

promiseMovies.js

```
const axios = require('axios');
const fs = require('fs').promises;



axios.get('https://ghibliapi.herokuapp.com/films')
    .then((response) => {
        console.log('Successfully retrieved our list of movies');
        let movieList = '';
        response.data.forEach(movie => {
            movieList += `${movie['title']}, ${movie['release_date']}\n`;
        });

        return fs.writeFile('promiseMovies.csv', movieList);
    })
    .then(() => {
        console.log('Saved our list of movies to promiseMovies.csv');
    })
```

We additionally import the `fs` module once again. Note how after the `fs` import we have `.promises`. Node.js includes a promised-based version of the callback-based `fs` library, so backward compatibility is not broken in legacy projects.

The first `then()` function that processes the HTTP request now calls `fs.writeFile()` instead of printing to the console. Since we imported the promise-based version of `fs`, our `writeFile()` function returns another promise. As such, we append another `then()` function for when the `writeFile()` promise is fulfilled.

A promise can return a new promise, allowing us to execute promises one after the other. This paves the way for us to perform multiple asynchronous operations. This is

called promise chaining, and it is analogous to nesting callbacks. The second `then()` is only called after we successfully write to the file.

Note: In this example, we did not check for the HTTP status code like we did in the callback example. By default, `axios` does not fulfil its promise if it gets a status code indicating an error. As such, we no longer need to validate it.

To complete this program, chain the promise with a `catch()` function as it is highlighted in the following:

promiseMovies.js

```
const axios = require('axios');
const fs = require('fs').promises;



axios.get('https://ghibliapi.herokuapp.com/films')
    .then((response) => {
        console.log('Successfully retrieved our list of movies');
        let movieList = '';
        response.data.forEach(movie => {
            movieList += `${movie['title']}, ${movie['release_date']}\n`;
        });

        return fs.writeFile('promiseMovies.csv', movieList);
    })
    .then(() => {
        console.log('Saved our list of movies to promiseMovies.csv');
    })
    .catch((error) => {
        console.error(`Could not save the Ghibli movies to a file: ${error}
    });
```

If any promise is not fulfilled in the chain of promises, JavaScript automatically goes to the `catch()` function if it was defined. That's why we only have one `catch()` clause even though we have two asynchronous operations.

Let's confirm that our program produces the same output by running:

```
node promiseMovies.js
```

In your `ghibliMovies` folder, you will see the `promiseMovies.csv` file containing:

promiseMovies.csv

```
Castle in the Sky, 1986
Grave of the Fireflies, 1988
My Neighbor Totoro, 1988
Kiki's Delivery Service, 1989
Only Yesterday, 1991
Porco Rosso, 1992
Pom Poko, 1994
Whisper of the Heart, 1995
Princess Mononoke, 1997
My Neighbors the Yamadas, 1999
Spirited Away, 2001
The Cat Returns, 2002
Howl's Moving Castle, 2004
Tales from Earthsea, 2006
Ponyo, 2008
Arrietty, 2010
From Up on Poppy Hill, 2011
The Wind Rises, 2013
The Tale of the Princess Kaguya, 2013
When Marnie Was There, 2014
```

With promises, we can write much more concise code than using only callbacks. The promise chain of callbacks is a cleaner option than nesting callbacks. However, as we make more asynchronous calls, our promise chain becomes longer and harder to maintain.

The verbosity of callbacks and promises come from the need to create functions when we have the result of an asynchronous task. A better experience would be to wait for an asynchronous result and put it in a variable outside the function. That way, we can use the results in the variables without having to make a function. We can achieve this with the `async` and `await` keywords.

Writing JavaScript with `async`/`await`

The `async`/`await` keywords provide an alternative syntax when working with promises. Instead of having the result of a promise available in the `then()` method, the result is returned as a value like in any other function. We define a function with the `async` keyword to tell JavaScript that it's an asynchronous function that returns a promise. We use the `await` keyword to tell JavaScript to return the results of the promise instead of returning the promise itself when it's fulfilled.

In general, `async`/`await` usage looks like this:

```
async function() {
    await [Asynchronous Action]
}
```

Let's see how using `async`/`await` can improve our Studio Ghibli program. Use your text editor to create and open a new file `asyncAwaitMovies.js`:

```
nano asyncAwaitMovies.js
```

In your newly opened JavaScript file, let's start by importing the same modules we used in our promise example:

asyncAwaitMovies.js

```
const axios = require('axios');
const fs = require('fs').promises;
```

The imports are the same as `promiseMovies.js` because `async`/`await` uses promises. Now we use the `async` keyword to create a function with our asynchronous code:

asyncAwaitMovies.js

```
const axios = require('axios');
const fs = require('fs').promises;

async function saveMovies() {}
```

We create a new function called `saveMovies()` but we include `async` at the beginning of its definition. This is important as we can only use the `await` keyword in an asynchronous function.

Use the `await` keyword to make an HTTP request that gets the list of movies from the Ghibli API:

asyncAwaitMovies.js

```
const axios = require('axios');
const fs = require('fs').promises;

async function saveMovies() {
    let response = await axios.get('https://ghibliapi.herokuapp.com/films')
    let movieList = '';
    response.data.forEach(movie => {
        movieList += `${movie['title']}, ${movie['release_date']}\n`;
    });
}
```

In our `saveMovies()` function, we make an HTTP request with `axios.get()` like before. This time, we don't chain it with a `then()` function. Instead, we add `await` before it is called. When JavaScript sees `await`, it will only execute the remaining code of the function after `axios.get()` finishes execution and sets the `response` variable. The other code saves the movie data so we can write to a file.

Let's write the movie data to a file:

asyncAwaitMovies.js

```
const axios = require('axios');
const fs = require('fs').promises;

async function saveMovies() {
    let response = await axios.get('https://ghibliapi.herokuapp.com/films')
    let movieList = '';
    response.data.forEach(movie => {
        movieList += `${movie['title']}, ${movie['release_date']}\n`;
    });
    await fs.writeFile('asyncAwaitMovies.csv', movieList);
}
```

We also use the `await` keyword when we write to the file with `fs.writeFile()`.

To complete this function, we need to catch errors our promises can throw. Let's do this by encapsulating our code in a `try`/`catch` block:

asyncAwaitMovies.js

```
const axios = require('axios');
const fs = require('fs').promises;

async function saveMovies() {
    try {
        let response = await axios.get('https://ghibliapi.herokuapp.com/fil
        let movieList = '';
        response.data.forEach(movie => {
            movieList += `${movie['title']}, ${movie['release_date']}\n`;
        });
        await fs.writeFile('asyncAwaitMovies.csv', movieList);
    } catch (error) {
        console.error(`Could not save the Ghibli movies to a file: ${error}
    }
}
```

Since promises can fail, we encase our asynchronous code with a `try`/`catch` clause. This will capture any errors that are thrown when either the HTTP request or file writing operations fail.

Finally, let's call our asynchronous function `saveMovies()` so it will be executed when we run the program with `node`

asyncAwaitMovies.js

```
const axios = require('axios');
const fs = require('fs').promises;

async function saveMovies() {
    try {
        let response = await axios.get('https://ghibliapi.herokuapp.com/fil
        let movieList = '';
        response.data.forEach(movie => {
            movieList += `${movie['title']}, ${movie['release_date']}\n`;
        });
        await fs.writeFile('asyncAwaitMovies.csv', movieList);
    } catch (error) {
        console.error(`Could not save the Ghibli movies to a file: ${error}
    }
}

saveMovies();
```

At a glance, this looks like a typical synchronous JavaScript code block. It has fewer functions being passed around, which looks a bit neater. These small tweaks make asynchronous code with `async`/`await` easier to maintain.

Test this iteration of our program by entering this in your terminal:

```
node asyncAwaitMovies.js
```

In your `ghibliMovies` folder, a new `asyncAwaitMovies.csv` file will be created with the following contents:

asyncAwaitMovies.csv

```
Castle in the Sky, 1986
Grave of the Fireflies, 1988
My Neighbor Totoro, 1988
Kiki's Delivery Service, 1989
Only Yesterday, 1991
Porco Rosso, 1992
Pom Poko, 1994
Whisper of the Heart, 1995
Princess Mononoke, 1997
My Neighbors the Yamadas, 1999
Spirited Away, 2001
The Cat Returns, 2002
Howl's Moving Castle, 2004
Tales from Earthsea, 2006
Ponyo, 2008
Arrietty, 2010
From Up on Poppy Hill, 2011
The Wind Rises, 2013
The Tale of the Princess Kaguya, 2013
When Marnie Was There, 2014
```

You have now used the JavaScript features `async`/`await` to manage asynchronous code.

Conclusion

In this tutorial, you learned how JavaScript handles executing functions and managing asynchronous operations with the event loop. You then wrote programs that created a CSV file after making an HTTP request for movie data using various asynchronous programming techniques. First, you used the obsolete callback-based approach. You then used promises, and finally `async`/`await` to make the promise syntax more succinct.

With your understanding of asynchronous code with Node.js, you can now develop programs that benefit from asynchronous programming, like those that rely on API calls. Have a look at this list of public APIs. To use them, you will have to make asynchronous HTTP requests like we did in this tutorial. For further study, try building an app that uses these APIs to practice the techniques you learned here.

How To Test a Node.js Module with Mocha and Assert

Written by Stack Abuse

The author selected the Open Internet/Free Speech Fund to receive a donation as part of the Write for DOnations program.

Testing is an integral part of software development. It's common for programmers to run code that tests their application as they make changes in order to confirm it's behaving as they'd like. With the right test setup, this process can even be automated, saving a lot of time. Running tests consistently after writing new code ensures that new changes don't break pre-existing features. This gives the developer confidence in their code base, especially when it gets deployed to production so users can interact with it.

A test framework structures the way we create test cases. Mocha is a popular JavaScript test framework that organizes our test cases and runs them for us. However, Mocha does not verify our code's behavior. To compare values in a test, we can use the Node.js `assert` module.

In this article, you'll write tests for a Node.js TODO list module. You will set up and use the Mocha test framework to structure your tests. Then you'll use the Node.js `assert` module to create the tests themselves. In this sense, you will be using Mocha as a plan builder, and `assert` to implement the plan.

Prerequisites

- Node.js installed on your development machine. This tutorial uses Node.js version 10.16.0. To install this on macOS or Ubuntu 18.04, follow the steps in How to Install Node.js and Create a Local Development Environment on macOS or the Installing Using a PPA section of How To Install Node.js on Ubuntu 18.04.
- A basic knowledge of JavaScript, which you can find in our How To Code in JavaScript series.

Step 1 — Writing a Node Module

Let's begin this article by writing the Node.js module we'd like to test. This module will manage a list of TODO items. Using this module, we will be able to list all the TODOs that we are keeping track of, add new items, and mark some as complete. Additionally, we'll be able to export a list of TODO items to a CSV file. If you'd like a refresher on writing Node.js modules, you can read our article on How To Create a Node.js Module.

First, we need to set up the coding environment. Create a folder with the name of your project in your terminal. This tutorial will use the name `todos`:

```
mkdir todos
```

Then enter that folder:

```
cd todos
```

Now initialize npm, since we'll be using its CLI functionality to run the tests later:

```
npm init -y
```

We only have one dependency, Mocha, which we will use to organize and run our tests. To download and install Mocha, use the following:

```
npm i request --save-dev mocha
```

We install Mocha as a `dev` dependency, as it's not required by the module in a production setting. If you would like to learn more about Node.js packages or npm, check out our guide on How To Use Node.js Modules with npm and package.json.

Finally, let's create our file that will contain our module's code:

```
touch index.js
```

With that, we're ready to create our module. Open `index.js` in a text editor like `nano`:

```
nano index.js
```

Let's begin by defining the `Todos` class. This class contains all the functions that we need to manage our TODO list. Add the following lines of code to `index.js`:

todos/index.js

```
class Todos {
    constructor() {
        this.todos = [];
    }
}

module.exports = Todos;
```

We begin the file by creating a `Todos` class. Its `constructor()` function takes no arguments, therefore we don't need to provide any values to instantiate an object for this class. All we do when we initialize a `Todos` object is create a `todos` property that's an empty array.

The `modules` line allows other Node.js modules to require our `Todos` class. Without explicitly exporting the class, the test file that we will create later would not be able to use it.

Let's add a function to return the array of `todos` we have stored. Write in the following highlighted lines:

todos/index.js

```
class Todos {
    constructor() {
        this.todos = [];
    }

    list() {
        return [...this.todos];
    }
}

module.exports = Todos;
```

Our `list()` function returns a copy of the array that's used by the class. It makes a copy of the array by using JavaScript's destructuring syntax. We make a copy of the array so that changes the user makes to the array returned by `list()` does not affect the array used by the `Todos` object.

Note: JavaScript arrays are reference types. This means that for any variable assignment to an array or function invocation with an array as a parameter, JavaScript refers to the original array that was created. For example, if we have an array with three items called `x`, and create a new variable `y` such that `y = x`, `y` and `x` both refer to the same thing. Any changes we make to the array with `y` impacts variable `x` and vice versa.

Now let's write the `add()` function, which adds a new TODO item:

todos/index.js

```
class Todos {
    constructor() {
        this.todos = [];
    }

    list() {
        return [...this.todos];
    }

    add(title) {
        let todo = {
            title: title,
            completed: false,
        }

        this.todos.push(todo);
    }
}

module.exports = Todos;
```

Our `add()` function takes a string, and places it in a new JavaScript object's `title` property. The new object also has a `completed` property, which is set to `false` by default. We then add this new object to our array of TODOs.

Important functionality in a TODO manager is to mark items as completed. For this implementation, we will loop through our `todos` array to find the TODO item the user is searching for. If one is found, we'll mark it as completed. If none is found, we'll throw an error.

Add the `complete()` function like this:

todos/index.js

```
class Todos {
    constructor() {
        this.todos = [];
    }

    list() {
        return [...this.todos];
    }

    add(title) {
        let todo = {
            title: title,
            completed: false,
        }

        this.todos.push(todo);
    }

    complete(title) {
        let todoFound = false;
        this.todos.forEach((todo) => {
            if (todo.title === title) {
                todo.completed = true;
                todoFound = true;
                return;
```

```
            }
        });

        if (!todoFound) {
            throw new Error(`No TODO was found with the title:
"${title}"`);
        }
    }
}

module.exports = Todos;
```

Save the file and exit from the text editor.

We now have a basic TODO manager that we can experiment with. Next, let's manually test our code to see if the application is working.

Step 2 — Manually Testing the Code

In this step, we will run our code's functions and observe the output to ensure it matches our expectations. This is called manual testing. It's likely the most common testing methodology programmers apply. Although we will automate our testing later with Mocha, we will first manually test our code to give a better sense of how manual testing differs from testing frameworks.

Let's add two TODO items to our app and mark one as complete. Start the Node.js REPL in the same folder as the `index.js` file:

```
node
```

You will see the `>` prompt in the REPL that tells us we can enter JavaScript code. Type the following at the prompt:

```
const Todos = require('./index');
```

With `require()`, we load the TODOs module into a `Todos` variable. Recall that our module returns the `Todos` class by default.

Now, let's instantiate an object for that class. In the REPL, add this line of code:

```
const todos = new Todos();
```

We can use the `todos` object to verify our implementation works. Let's add our first TODO item:

```
todos.add("run code");
```

So far we have not seen any output in our terminal. Let's verify that we've stored our `"run code"` TODO item by getting a list of all our TODOs:

```
todos.list();
```

You will see this output in your REPL:

```
Output
[ { title: 'run code', completed: false } ]
```

This is the expected result: We have one TODO item in our array of TODOs, and it's not completed by default.

Let's add another TODO item:

```
todos.add("test everything");
```

Mark the first TODO item as completed:

```
todos.complete("run code");
```

Our `todos` object will now be managing two items: `"run code"` and `"test everything"`. The `"run code"` TODO will be completed as well. Let's confirm this by calling `list()` once again:

```
todos.list();
```

The REPL will output:

```
Output
[
  { title: 'run code', completed: true },
  { title: 'test everything', completed: false }
]
```

Now, exit the REPL with the following:

```
.exit
```

We've confirmed that our module behaves as we expect it to. While we didn't put our code in a test file or use a testing library, we did test our code manually. Unfortunately, this form of testing becomes time consuming to do every time we make a change. Next, let's use automated testing in Node.js and see if we can solve this problem with the Mocha testing framework.

Step 3 — Writing Your First Test with Mocha and Assert

In the last step, we manually tested our application. This will work for individual use cases, but as our module scales, this method becomes less viable. As we test new features, we must be certain that the added functionality has not created problems in the old functionality. We would like to test each feature over again for every change in the code, but doing this by hand would take a lot of effort and would be prone to error.

A more efficient practice would be to set up automated tests. These are scripted tests written like any other code block. We run our functions with defined inputs and inspect their effects to ensure they behave as we expect. As our codebase grows, so will our automated tests. When we write new tests alongside the features, we can verify the entire module still works—all without having to remember how to use each function every time.

In this tutorial, we're using the Mocha testing framework with the Node.js `assert` module. Let's get some hands-on experience to see how they work together.

To begin, create a new file to store our test code:

```
touch index.test.js
```

Now use your preferred text editor to open the test file. You can use `nano` like before:

```
nano index.test.js
```

In the first line of the text file, we will load the TODOs module like we did in the Node.js shell. We will then load the `assert` module for when we write our tests. Add the following lines:

todos/index.test.js

```
const Todos = require('./index');
const assert = require('assert').strict;
```

The `strict` property of the `assert` module will allow us to use special equality tests that are recommended by Node.js and are good for future-proofing, since they account for more use cases.

Before we go into writing tests, let's discuss how Mocha organizes our code. Tests structured in Mocha usually follow this template:

```
describe([String with Test Group Name], function() {
    it([String with Test Name], function() {
        [Test Code]
    });
});
```

Notice two key functions: `describe()` and `it()`. The `describe()` function is used to group similar tests. It's not required for Mocha to run tests, but grouping tests make our test code easier to maintain. It's recommended that you group your tests in a way that's easy for you to update similar ones together.

The `it()` contains our test code. This is where we would interact with our module's functions and use the `assert` library. Many `it()` functions can be defined in a `describe()` function.

Our goal in this section is to use Mocha and `assert` to automate our manual test. We'll do this step-by-step, beginning with our describe block. Add the following to your file after the module lines:

todos/index.test.js

```
...
describe("integration test", function() {
});
```

With this code block, we've created a grouping for our integrated tests. Unit tests would test one function at a time. Integration tests verify how well functions within or across modules work together. When Mocha runs our test, all the tests within that describe block will run under the `"integration test"` group.

Let's add an `it()` function so we can begin testing our module's code:

todos/index.test.js

```
...
describe("integration test", function() {
    it("should be able to add and complete TODOs", function()
 {
    });
});
```

Notice how descriptive we made the test's name. If anyone runs our test, it will be immediately clear what's passing or failing. A well-tested application is typically a well-documented application, and tests can sometimes be an effective kind of documentation.

For our first test, we will create a new `Todos` object and verify it has no items in it:

todos/index.test.js

```
...
describe("integration test", function() {
    it("should be able to add and complete TODOs", function()
 {
        let todos = new Todos();
        assert.notStrictEqual(todos.list().length, 1);
    });
});
```

The first new line of code instantiated a new `Todos` object as we would do in the Node.js REPL or another module. In the second new line, we use the `assert` module.

From the `assert` module we use the `notStrictEqual()` method. This function takes two parameters: the value that we want to test (called the `actual` value) and the value we expect to get (called the `expected` value). If both arguments are the same, `notStrictEqual()` throws an error to fail the test.

Save and exit from `index.test.js`.

The base case will be true as the length should be `0`, which isn't `1`. Let's confirm this by running Mocha. To do this, we need to modify our `package.json` file. Open your `package.json` file with your text editor:

```
nano package.json
```

Now, in your `scripts` property, change it so it looks like this:

todos/package.json

```
...
"scripts": {
    "test": "mocha index.test.js"
},
...
```

We have just changed the behavior of npm's CLI `test` command. When we run `npm test`, npm will review the command we just entered in `packag e.json`. It will look for the Mocha library in our `node_modules` folder and run the `mocha` command with our test file.

Save and exit `package.json`.

Let's see what happens when we run our test. In your terminal, enter:

```
npm test
```

The command will produce the following output:

```
Output
> todos@1.0.0 test your_file_path/todos
> mocha index.test.js



integrated test
    ✓ should be able to add and complete TODOs



  1 passing (16ms)
```

This output first shows us which group of tests it is about to run. For every individual test within a group, the test case is indented. We see our

test name as we described it in the `it()` function. The tick at the left side of the test case indicates that the test passed.

At the bottom, we get a summary of all our tests. In our case, our one test is passing and was completed in 16ms (the time varies from computer to computer).

Our testing has started with success. However, this current test case can allow for false-positives. A false-positive is a test case that passes when it should fail.

We currently check that the length of the array is not equal to `1`. Let's modify the test so that this condition holds true when it should not. Add the following lines to `index.test.js`:

todos/index.test.js

```
...
describe("integration test", function() {
    it("should be able to add and complete TODOs", function()
 {
        let todos = new Todos();
        todos.add("get up from bed");
        todos.add("make up bed");
        assert.notStrictEqual(todos.list().length, 1);
    });
});
```

Save and exit the file.

We added two TODO items. Let's run the test to see what happens:

```
npm test
```

This will give the following:

```
Output
...
integrated test
    ✓ should be able to add and complete TODOs



  1 passing (8ms)
```

This passes as expected, as the length is greater than 1. However, it defeats the original purpose of having that first test. The first test is meant to confirm that we start on a blank state. A better test will confirm that in all cases.

Let’s change the test so it only passes if we have absolutely no TODOs in store. Make the following changes to `index.test.js`:

todos/index.test.js

```
...
describe("integration test", function() {
    it("should be able to add and complete TODOs", function()
 {
        let todos = new Todos();
        todos.add("get up from bed");
        todos.add("make up bed");
        assert.strictEqual(todos.list().length, 0);
    });
});
```

You changed `notStrictEqual()` to `strictEqual()`, a function that checks for equality between its actual and expected argument. Strict equal will fail if our arguments are not exactly the same.

Save and exit, then run the test so we can see what happens:

```
npm test
```

This time, the output will show an error:

Output

```
...
  integration test
    1) should be able to add and complete TODOs



  0 passing (16ms)
  1 failing

  1) integration test
       should be able to add and complete TODOs:

      AssertionError [ERR_ASSERTION]: Input A expected to stri
ctly equal input B:
+ expected - actual

- 2
+ 0
      + expected - actual

      -2
      +0

      at Context.<anonymous> (index.test.js:9:10)
```

```
npm ERR! Test failed.  See above for more details.
```

This text will be useful for us to debug why the test failed. Notice that since the test failed there was no tick at the beginning of the test case.

Our test summary is no longer at the bottom of the output, but right after our list of test cases were displayed:

```
...
0 passing (29ms)
  1 failing
...
```

The remaining output provides us with data about our failing tests. First, we see what test case has failed:

```
...
1) integrated test
       should be able to add and complete TODOs:
...
```

Then, we see why our test failed:

```
...
      AssertionError [ERR_ASSERTION]: Input A expected to stri
ctly equal input B:
+ expected - actual

- 2
+ 0
      + expected - actual

      -2
      +0

      at Context.<anonymous> (index.test.js:9:10)
...
```

An `AssertionError` is thrown when `strictEqual()` fails. We see that the `expected` value, 0, is different from the `actual` value, 2.

We then see the line in our test file where the code fails. In this case, it's line 10.

Now, we've seen for ourselves that our test will fail if we expect incorrect values. Let's change our test case back to its right value. First, open the file:

```
nano index.test.js
```

Then take out the `todos.add` lines so that your code looks like the following:

todos/index.test.js

```
...
describe("integration test", function () {
    it("should be able to add and complete TODOs", function ()
        let todos = new Todos();
        assert.strictEqual(todos.list().length, 0);
    });
});
```

Save and exit the file.

Run it once more to confirm that it passes without any potential false-positives:

```
npm test
```

The output will be as follows:

```
Output
...
integration test
    ✓ should be able to add and complete TODOs



  1 passing (15ms)
```

We've now improved our test's resiliency quite a bit. Let's move forward with our integration test. The next step is to add a new TODO item to `index.test.js`:

todos/index.test.js

```
...
describe("integration test", function() {
    it("should be able to add and complete TODOs", function()
 {
        let todos = new Todos();
        assert.strictEqual(todos.list().length, 0);

        todos.add("run code");
        assert.strictEqual(todos.list().length, 1);
        assert.deepStrictEqual(todos.list(), [{title: "run cod
e", completed: false}]);
    });
});
```

After using the `add()` function, we confirm that we now have one TODO being managed by our `todos` object with `strictEqual()`. Our next test confirms the data in the `todos` with `deepStrictEqual()`. The `deepStrictEqual()` function recursively tests that our expected and actual objects have the same properties. In this case, it tests that the arrays we expect both have a JavaScript object within them. It then checks that their JavaScript objects have the same properties, that is, that both their `title` properties are `"run code"` and both their `completed` properties are `false`.

We then complete the remaining tests using these two equality checks as needed by adding the following highlighted lines:

todos/index.test.js

```
...
describe("integration test", function() {
    it("should be able to add and complete TODOs", function()
 {
        let todos = new Todos();
        assert.strictEqual(todos.list().length, 0);

        todos.add("run code");
        assert.strictEqual(todos.list().length, 1);
        assert.deepStrictEqual(todos.list(), [{title: "run cod
e", completed: false}]);

        todos.add("test everything");
        assert.strictEqual(todos.list().length, 2);
        assert.deepStrictEqual(todos.list(),
            [
                { title: "run code", completed: false },
                { title: "test everything", completed: false }
            ]
        );

        todos.complete("run code");
        assert.deepStrictEqual(todos.list(),
            [
                { title: "run code", completed: true },
```

```
                { title: "test everything", completed: false }
            ]
    );
  });
});
```

Save and exit the file.

Our test now mimics our manual test. With these programmatic tests, we don't need to check the output continuously if our tests pass when we run them. You typically want to test every aspect of use to make sure the code is tested properly.

Let's run our test with `npm test` once more to get this familiar output:

```
Output
...
integrated test
    ✓ should be able to add and complete TODOs



  1 passing (9ms)
```

You've now set up an integrated test with the Mocha framework and the `assert` library.

Let's consider a situation where we've shared our module with some other developers and they're now giving us feedback. A good portion of our users would like the `complete()` function to return an error if no TODOs

were added as of yet. Let's add this functionality in our `complete()` function.

Open `index.js` in your text editor:

```
nano index.js
```

Add the following to the function:

todos/index.js

```
...
complete(title) {
    if (this.todos.length === 0) {
        throw new Error(
            "You have no TODOs stored. Why don't you add one fi
    }

    let todoFound = false
    this.todos.forEach((todo) => {
        if (todo.title === title) {
            todo.completed = true;
            todoFound = true;
            return;
        }
    });

    if (!todoFound) {
        throw new Error(`No TODO was found with the title: "${t
    }
}
...
```

Save and exit the file.

Now let's add a new test for this new feature. We want to verify that if we call complete on a `Todos` object that has no items, it will return our special error.

Go back into `index.test.js`:

```
nano index.test.js
```

At the end of the file, add the following code:

todos/index.test.js

```
...
describe("complete()", function() {
    it("should fail if there are no TODOs", function() {
        let todos = new Todos();
        const expectedError = new Error("You have no TODOs stor

        assert.throws(() => {
            todos.complete("doesn't exist");
        }, expectedError);
    });
});
```

We use `describe()` and `it()` like before. Our test begins with creating a new `todos` object. We then define the error we are expecting to receive when we call the `complete()` function.

Next, we use the `throws()` function of the `assert` module. This function was created so we can verify the errors that are thrown in our code. Its first argument is a function that contains the code that throws the error. The second argument is the error we are expecting to receive.

In your terminal, run the tests with `npm test` once again and you will now see the following output:

```
Output
...
integrated test
    ✓ should be able to add and complete TODOs


  complete()
    ✓ should fail if there are no TODOs



  2 passing (25ms)
```

This output highlights the benefit of why we do automated testing with Mocha and `assert`. Because our tests are scripted, every time we run `npm test`, we verify that all our tests are passing. We did not need to manually check if the other code is still working; we know that it is because the test we have still passed.

So far, our tests have verified the results of synchronous code. Let's see how we would need to adapt our newfound testing habits to work with asynchronous code.

Step 4 — Testing Asynchronous Code

One of the features we want in our TODO module is a CSV export feature. This will print all the TODOs we have in store along with the completed status to a file. This requires that we use the `fs` module—a built-in Node.js module for working with the file system.

Writing to a file is an asynchronous operation. There are many ways to write to a file in Node.js. We can use callbacks, Promises, or the `async`/`await` keywords. In this section, we'll look at how we write tests for those different methods.

Callbacks

A callback function is one used as an argument to an asynchronous function. It is called when the asynchronous operation is completed.

Let's add a function to our `Todos` class called `saveToFile()`. This function will build a string by looping through all our TODO items and writing that string to a file.

Open your `index.js` file:

```
nano index.js
```

In this file, add the following highlighted code:

todos/index.js

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


class Todos {
    constructor() {
        this.todos = [];
    }

    list() {
        return [...this.todos];
    }

    add(title) {
        let todo = {
            title: title,
            completed: false,
        }
        this.todos.push(todo);
    }

    complete(title) {
        if (this.todos.length === 0) {
            throw new Error("You have no TODOs stored. Why do
n't you add one first?");
        }
```

```
        let todoFound = false
        this.todos.forEach((todo) => {
            if (todo.title === title) {
                todo.completed = true;
                todoFound = true;
                return;
            }
        });

        if (!todoFound) {
            throw new Error(`No TODO was found with the title: "${title}"`);
        }
    }

    saveToFile(callback) {
        let fileContents = 'Title,Completed\n';
        this.todos.forEach((todo) => {
            fileContents += `${todo.title},${todo.completed}\n`
        });

        fs.writeFile('todos.csv', fileContents, callback);
    }
}

module.exports = Todos;
```

We first have to import the `fs` module in our file. Then we added our new `saveToFile()` function. Our function takes a callback function that will be used once the file write operation is complete. In that function, we create a `fileContents` variable that stores the entire string we want to be saved as a file. It's initialized with the CSV's headers. We then loop through each TODO item with the internal array's `forEach()` method. As we iterate, we add the `title` and `completed` properties of the individual `todos` objects.

Finally, we use the `fs` module to write the file with the `writeFile()` function. Our first argument is the file name: `todos.csv`. The second is the contents of the file, in this case, our `fileContents` variable. Our last argument is our callback function, which handles any file writing errors.

Save and exit the file.

Let's now write a test for our `saveToFile()` function. Our test will do two things: confirm that the file exists in the first place, and then verify that it has the right contents.

Open the `index.test.js` file:

```
nano index.test.js
```

let's begin by loading the `fs` module at the top of the file, as we'll use it to help test our results:

todos/index.test.js

```
const Todos = require('./index');

const assert = require('assert').strict;

const fs = require('fs');

...
```

Now, at the end of the file let's add our new test case:

todos/index.test.js

```
...
describe("saveToFile()", function() {
    it("should save a single TODO", function(done) {
        let todos = new Todos();
        todos.add("save a CSV");
        todos.saveToFile((err) => {
            assert.strictEqual(fs.existsSync('todos.csv'), true
            let expectedFileContents = "Title,Completed\nsave a
            let content = fs.readFileSync("todos.csv").toString
            assert.strictEqual(content, expectedFileContents);
            done(err);
        });
    });
});
```

Like before, we use `describe()` to group our test separately from the others as it involves new functionality. The `it()` function is slightly different from our other ones. Usually, the callback function we use has no arguments. This time, we have `done` as an argument. We need this argument when testing functions with callbacks. The `done()` callback function is used by Mocha to tell it when an asynchronous function is completed.

All callback functions being tested in Mocha must call the `done()` callback. If not, Mocha would never know when the function was complete and would be stuck waiting for a signal.

Continuing, we create our `Todos` instance and add a single item to it. We then call the `saveToFile()` function, with a callback that captures a file writing error. Note how our test for this function resides in the callback. If our test code was outside the callback, it would fail as long as the code was called before the file writing completed.

In our callback function, we first check that our file exists:

todos/index.test.js

```
...
assert.strictEqual(fs.existsSync('todos.csv'), true);
...
```

The `fs.existsSync()` function returns `true` if the file path in its argument exists, `false` otherwise.

Note: The `fs` module's functions are asynchronous by default. However, for key functions, they made synchronous counterparts. This test is simpler by using synchronous functions, as we don't have to nest the asynchronous code to ensure it works. In the `fs` module, synchronous functions usually end with `"Sync"` at the end of their names.

We then create a variable to store our expected value:

todos/index.test.js

```
...
let expectedFileContents = "Title,Completed\nsave a CSV,false\n
...
```

We use `readFileSync()` of the `fs` module to read the file synchronously:

todos/index.test.js

```
...
let content = fs.readFileSync("todos.csv").toString();
...
```

We now provide `readFileSync()` with the right path for the file: `todos.csv`. As `readFileSync()` returns a `Buffer` object, which stores binary data,

we use its `toString()` method so we can compare its value with the string we expect to have saved.

Like before, we use the `assert` module's `strictEqual` to do a comparison:

todos/index.test.js

```
...
assert.strictEqual(content, expectedFileContents);
...
```

We end our test by calling the `done()` callback, ensuring that Mocha knows to stop testing that case:

todos/index.test.js

```
...
done(err);
...
```

We provide the `err` object to `done()` so Mocha can fail the test in the case an error occurred.

Save and exit from `index.test.js`.

Let's run this test with `npm test` like before. Your console will display this output:

```
Output
...
integrated test
    ✓ should be able to add and complete TODOs


  complete()
    ✓ should fail if there are no TODOs


  saveToFile()
    ✓ should save a single TODO



  3 passing (15ms)
```

You've now tested your first asynchronous function with Mocha using callbacks. But at the time of writing this tutorial, Promises are more prevalent than callbacks in new Node.js code, as explained in our How To Write Asynchronous Code in Node.js article. Next, let's learn how we can test them with Mocha as well.

Promises

A Promise is a JavaScript object that will eventually return a value. When a Promise is successful, it is resolved. When it encounters an error, it is rejected.

Let's modify the `saveToFile()` function so that it uses Promises instead of callbacks. Open up `index.js`:

```
nano index.js
```

First, we need to change how the `fs` module is loaded. In your `index.js` file, change the `require()` statement at the top of the file to look like this:

todos/index.js

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

We just imported the `fs` module that uses Promises instead of callbacks. Now, we need to make some changes to `saveToFile()` so that it works with Promises instead.

In your text editor, make the following changes to the `saveToFile()` function to remove the callbacks:

todos/index.js

```
...
saveToFile() {
    let fileContents = 'Title,Completed\n';
    this.todos.forEach((todo) => {
        fileContents += `${todo.title},${todo.completed}\n`
    });

    return fs.writeFile('todos.csv', fileContents);
}
...
```

The first difference is that our function no longer accepts any arguments. With Promises we don't need a callback function. The second change concerns how the file is written. We now return the result of the `writeFile()` promise.

Save and close out of `index.js`.

Let's now adapt our test so that it works with Promises. Open up `index.test.js`:

```
nano index.test.js
```

Change the `saveToFile()` test to this:

todos/index.js

```
...
describe("saveToFile()", function() {
    it("should save a single TODO", function() {
        let todos = new Todos();
        todos.add("save a CSV");
        return todos.saveToFile().then(() => {
            assert.strictEqual(fs.existsSync('todos.csv'), tru
e);
            let expectedFileContents = "Title,Completed\nsave
 a CSV,false\n";
            let content = fs.readFileSync("todos.csv").toStrin
g();
            assert.strictEqual(content, expectedFileContents);
        });
    });
});
```

The first change we need to make is to remove the `done()` callback from its arguments. If Mocha passes the `done()` argument, it needs to be called or it will throw an error like this:

```
1) saveToFile()
       should save a single TODO:
     Error: Timeout of 2000ms exceeded. For async tests and ho
oks, ensure "done()" is called; if returning a Promise, ensure
it resolves. (/home/ubuntu/todos/index.test.js)
      at listOnTimeout (internal/timers.js:536:17)
      at processTimers (internal/timers.js:480:7)
```

When testing Promises, do not include the `done()` callback in `it()`.

To test our promise, we need to put our assertion code in the `then()` function. Notice that we return this promise in the test, and we don't have a `catch()` function to catch when the `Promise` is rejected.

We return the promise so that any errors that are thrown in the `then()` function are bubbled up to the `it()` function. If the errors are not bubbled up, Mocha will not fail the test case. When testing Promises, you need to use `return` on the `Promise` being tested. If not, you run the risk of getting a false-positive.

We also omit the `catch()` clause because Mocha can detect when a promise is rejected. If rejected, it automatically fails the test.

Now that we have our test in place, save and exit the file, then run Mocha with `npm test` and to confirm we get a successful result:

```
Output
...
integrated test
    ✓ should be able to add and complete TODOs


  complete()
    ✓ should fail if there are no TODOs


  saveToFile()
    ✓ should save a single TODO



  3 passing (18ms)
```

We've changed our code and test to use Promises, and now we know for sure that it works. But the most recent asynchronous patterns use `async`/`await` keywords so we don't have to create multiple `then()` functions to handle successful results. Let's see how we can test with `async`/`await`.

async/await

The `async`/`await` keywords make working with Promises less verbose. Once we define a function as asynchronous with the `async` keyword, we can get any future results in that function with the `await` keyword. This way we can use Promises without having to use the `then()` or `catch()` functions.

We can simplify our `saveToFile()` test that's promise based with `async`/`await`. In your text editor, make these minor edits to the `saveToFile()` test in `index.test.js`:

todos/index.test.js

```
...
describe("saveToFile()", function() {
    it("should save a single TODO", async function() {
        let todos = new Todos();
        todos.add("save a CSV");
        await todos.saveToFile();

        assert.strictEqual(fs.existsSync('todos.csv'), true);
        let expectedFileContents = "Title,Completed\nsave a CS
V,false\n";
        let content = fs.readFileSync("todos.csv").toString();
        assert.strictEqual(content, expectedFileContents);
    });
});
```

The first change is that the function used by the `it()` function now has the `async` keyword when it's defined. This allows us to the use the `await` keyword inside its body.

The second change is found when we call `saveToFile()`. The `await` keyword is used before it is called. Now Node.js knows to wait until this function is resolved before continuing the test.

Our function code is easier to read now that we moved the code that was in the `then()` function to the `it()` function's body. Running this code with `npm test` produces this output:

```
Output
...
integrated test
    ✓ should be able to add and complete TODOs


  complete()
    ✓ should fail if there are no TODOs


  saveToFile()
    ✓ should save a single TODO



  3 passing (30ms)
```

We can now test asynchronous functions using any of three asynchronous paradigms appropriately.

We have covered a lot of ground with testing synchronous and asynchronous code with Mocha. Next, let's dive in a bit deeper to some other functionality that Mocha offers to improve our testing experience, particularly how hooks can change test environments.

Step 5 — Using Hooks to Improve Test Cases

Hooks are a useful feature of Mocha that allows us to configure the environment before and after a test. We typically add hooks within a `describe()` function block, as they contain setup and teardown logic specific to some test cases.

Mocha provides four hooks that we can use in our tests:

- `before`: This hook is run once before the first test begins.
- `beforeEach`: This hook is run before every test case.
- `after`: This hook is run once after the last test case is complete.
- `afterEach`: This hook is run after every test case.

When we test a function or feature multiple times, hooks come in handy as they allow us to separate the test's setup code (like creating the `todos` object) from the test's assertion code.

To see the value of hooks, let's add more tests to our `saveToFile()` test block.

While we have confirmed that we can save our TODO items to a file, we only saved one item. Furthermore, the item was not marked as completed. Let's add more tests to be sure that the various aspects of our module works.

First, let's add a second test to confirm that our file is saved correctly when we have a completed a TODO item. Open your `index.test.js` file in your text editor:

```
nano index.test.js
```

Change the last test to the following:

todos/index.test.js

```
...
describe("saveToFile()", function () {
    it("should save a single TODO", async function () {
        let todos = new Todos();
        todos.add("save a CSV");
        await todos.saveToFile();

        assert.strictEqual(fs.existsSync('todos.csv'), true);
        let expectedFileContents = "Title,Completed\nsave a CS
V,false\n";
        let content = fs.readFileSync("todos.csv").toString();
        assert.strictEqual(content, expectedFileContents);
    });

    it("should save a single TODO that's completed", async fun
ction () {
        let todos = new Todos();
        todos.add("save a CSV");
        todos.complete("save a CSV");
        await todos.saveToFile();

        assert.strictEqual(fs.existsSync('todos.csv'), true);
        let expectedFileContents = "Title,Completed\nsave a CS
V,true\n";
        let content = fs.readFileSync("todos.csv").toString();
```

```
        assert.strictEqual(content, expectedFileContents);
    });
});
```

The test is similar to what we had before. The key differences are that we call the `complete()` function before we call `saveToFile()`, and that our `expectedFileContents` now have `true` instead of `false` for the `completed` column's value.

Save and exit the file.

Let's run our new test, and all the others, with `npm test`:

```
npm test
```

This will give the following:

```
Output
...
integrated test
    ✓ should be able to add and complete TODOs


  complete()
    ✓ should fail if there are no TODOs


  saveToFile()
    ✓ should save a single TODO
    ✓ should save a single TODO that's completed



  4 passing (26ms)
```

It works as expected. There is, however, room for improvement. They both have to instantiate a `Todos` object at the beginning of the test. As we add more test cases, this quickly becomes repetitive and memory-wasteful. Also, each time we run the test, it creates a file. This can be mistaken for real output by someone less familiar with the module. It would be nice if we cleaned up our output files after testing.

Let's make these improvements using test hooks. We'll use the `beforeEach()` hook to set up our test fixture of TODO items. A test fixture is any consistent state used in a test. In our case, our test fixture is a new `todos` object that has one TODO item added to it already. We will then use `afterEach()` to remove the file created by the test.

In `index.test.js`, make the following changes to your last test for `saveToFile()`:

todos/index.test.js

```
...
describe("saveToFile()", function () {
    beforeEach(function () {
        this.todos = new Todos();
        this.todos.add("save a CSV");
    });

    afterEach(function () {
        if (fs.existsSync("todos.csv")) {
            fs.unlinkSync("todos.csv");
        }
    });

    it("should save a single TODO without error", async functi
on () {
        await this.todos.saveToFile();

        assert.strictEqual(fs.existsSync("todos.csv"), true);
        let expectedFileContents = "Title,Completed\nsave a CS
V,false\n";
        let content = fs.readFileSync("todos.csv").toString();
        assert.strictEqual(content, expectedFileContents);
    });

    it("should save a single TODO that's completed", async fun
```

```
ction () {
        this.todos.complete("save a CSV");
        await this.todos.saveToFile();

        assert.strictEqual(fs.existsSync('todos.csv'), true);
        let expectedFileContents = "Title,Completed\nsave a CS
V,true\n";
        let content = fs.readFileSync("todos.csv").toString();
        assert.strictEqual(content, expectedFileContents);
    });
});
```

Let's break down all the changes we've made. We added a `beforeEach()` block to the test block:

todos/index.test.js

```
...
beforeEach(function () {
    this.todos = new Todos();
    this.todos.add("save a CSV");
});
...
```

These two lines of code create a new `Todos` object that will be available in each of our tests. With Mocha, the `this` object in `beforeEach()` refers to the same `this` object in `it()`. `this` is the same for every code block inside the `describe()` block. For more information on `this`, see our tutorial Understanding This, Bind, Call, and Apply in JavaScript.

This powerful context sharing is why we can quickly create test fixtures that work for both of our tests.

We then clean up our CSV file in the `afterEach()` function:

todos/index.test.js

```
...
afterEach(function () {
    if (fs.existsSync("todos.csv")) {
        fs.unlinkSync("todos.csv");
    }
});
...
```

If our test failed, then it may not have created a file. That's why we check if the file exists before we use the `unlinkSync()` function to delete it.

The remaining changes switch the reference from `todos`, which were previously created in the `it()` function, to `this.todos` which is available in the Mocha context. We also deleted the lines that previously instantiated `todos` in the individual test cases.

Now, let's run this file to confirm our tests still work. Enter `npm test` in your terminal to get:

```
Output
...
integrated test
    ✓ should be able to add and complete TODOs


  complete()
    ✓ should fail if there are no TODOs


  saveToFile()
    ✓ should save a single TODO without error
    ✓ should save a single TODO that's completed



  4 passing (20ms)
```

The results are the same, and as a benefit, we have slightly reduced the setup time for new tests for the `saveToFile()` function and found a solution to the residual CSV file.

Conclusion

In this tutorial, you wrote a Node.js module to manage TODO items and tested the code manually using the Node.js REPL. You then created a test file and used the Mocha framework to run automated tests. With the `assert`

module, you were able to verify that your code works. You also tested synchronous and asynchronous functions with Mocha. Finally, you created hooks with Mocha that make writing multiple related test cases much more readable and maintainable.

Equipped with this understanding, challenge yourself to write tests for new Node.js modules that you are creating. Can you think about the inputs and outputs of your function and write your test before you write your code?

If you would like more information about the Mocha testing framework, check out the official Mocha documentation. If you'd like to continue learning Node.js, you can return to the How To Code in Node.js series page.

How To Create a Web Server in Node.js with the HTTP Module

Written by Stack Abuse

The author selected the COVID-19 Relief Fund to receive a donation as part of the Write for DOnations program.

When you view a webpage in your browser, you are making a request to another computer on the internet, which then provides you the webpage as a response. That computer you are talking to via the internet is a web server. A web server receives HTTP requests from a client, like your browser, and provides an HTTP response, like an HTML page or JSON from an API.

A lot of software is involved for a server to return a webpage. This software generally falls into two categories: frontend and backend. Front-end code is concerned with how the content is presented, such as the color of a navigation bar and the text styling. Back-end code is concerned with how data is exchanged, processed, and stored. Code that handles network requests from your browser or communicates with the database is primarily managed by back-end code.

Node.js allows developers to use JavaScript to write back-end code, even though traditionally it was used in the browser to write front-end code. Having both the frontend and backend together like this reduces the effort it takes to make a web server, which is a major reason why Node.js is a popular choice for writing back-end code.

In this tutorial, you will learn how to build web servers using the `http` module that's included in Node.js. You will build web servers that can

return JSON data, CSV files, and HTML web pages.

Prerequisites

- Ensure that Node.js is installed on your development machine. This tutorial uses Node.js version 10.19.0. To install this on macOS or Ubuntu 18.04, follow the steps in How to Install Node.js and Create a Local Development Environment on macOS or the **Installing Using a PPA** section of How To Install Node.js on Ubuntu 18.04.
- The Node.js platform supports creating web servers out of the box. To get started, be sure you're familiar with the basics of Node.js. You can get started by reviewing our guide on How To Write and Run Your First Program in Node.js.
- We also make use of asynchronous programming for one of our sections. If you're not familiar with asynchronous programming in Node.js or the `fs` module for interacting with files, you can learn more with our article on How To Write Asynchronous Code in Node.js.

Step 1 — Creating a Basic HTTP Server

Let's start by creating a server that returns plain text to the user. This will cover the key concepts required to set up a server, which will provide the foundation necessary to return more complex data formats like JSON.

First, we need to set up an accessible coding environment to do our exercises, as well as the others in the article. In the terminal, create a folder called `first-servers`:

```
mkdir first-servers
```

Then enter that folder:

```
cd first-servers
```

Now, create the file that will house the code:

```
touch hello.js
```

Open the file in a text editor. We will use `nano` as it's available in the terminal:

```
nano hello.js
```

We start by loading the `http` module that's standard with all Node.js installations. Add the following line to `hello.js`:

first-servers/hello.js

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

The `http` module contains the function to create the server, which we will see later on. If you would like to learn more about modules in Node.js, check out our How To Create a Node.js Module article.

Our next step will be to define two constants, the host and port that our server will be bound to:

first-servers/hello.js

```
...
const host = 'localhost';
const port = 8000;
```

As mentioned before, web servers accept requests from browsers and other clients. We may interact with a web server by entering a domain name, which is translated to an IP address by a DNS server. An IP address is a unique sequence of numbers that identify a machine on a network, like the internet. For more information on domain name concepts, take a look at our An Introduction to DNS Terminology, Components, and Concepts article.

The value `localhost` is a special private address that computers use to refer to themselves. It's typically the equivalent of the internal IP address `127.0.0.1` and it's only available to the local computer, not to any local networks we've joined or to the internet.

The port is a number that servers use as an endpoint or "door" to our IP address. In our example, we will use port `8000` for our web server. Ports `80` and `8000` are typically used as default ports in development, and in most cases developers will use them rather than other ports for HTTP servers.

When we bind our server to this host and port, we will be able to reach our server by visiting `http://localhost:8000` in a local browser.

Let's add a special function, which in Node.js we call a request listener. This function is meant to handle an incoming HTTP request and return an HTTP response. This function must have two arguments, a request object and a response object. The request object captures all the data of the HTTP

request that's coming in. The response object is used to return HTTP responses for the server.

We want our first server to return this message whenever someone accesses it: `"My first server!"`.

Let's add that function next:

first-servers/hello.js

```
...

const requestListener = function (req, res) {
    res.writeHead(200);
    res.end("My first server!");
};
```

The function would usually be named based on what it does. For example, if we created a request listener function to return a list of books, we would likely name it `listBooks()`. Since this one is a sample case, we will use the generic name `requestListener`.

All request listener functions in Node.js accept two arguments: `req` and `res` (we can name them differently if we want). The HTTP request the user sends is captured in a Request object, which corresponds to the first argument, `req`. The HTTP response that we return to the user is formed by interacting with the Response object in second argument, `res`.

The first line `res.writeHead(200);` sets the HTTP status code of the response. HTTP status codes indicate how well an HTTP request was handled by the server. In this case, the status code `200` corresponds to `"OK"`. If you are interested in learning about the various HTTP codes that your web servers can return with the meaning they signify, our guide on How To Troubleshoot Common HTTP Error Codes is a good place to start.

The next line of the function, `res.end("My first server!");`, writes the HTTP response back to the client who requested it. This function returns any data the server has to return. In this case, it's returning text data.

Finally, we can now create our server and make use of our request listener:

first-servers/hello.js

```
...

const server = http.createServer(requestListener);
server.listen(port, host, () => {
    console.log(`Server is running on http://${host}:${port}`);
});
```

Save and exit `nano` by pressing `CTRL+X`.

In the first line, we create a new `server` object via the `http` module's `createServer()` function. This server accepts HTTP requests and passes them on to our `requestListener()` function.

After we create our server, we must bind it to a network address. We do that with the `server.listen()` method. It accepts three arguments: `port`, `host`, and a callback function that fires when the server begins to listen.

All of these arguments are optional, but it is a good idea to explicitly state which port and host we want a web server to use. When deploying web servers to different environments, knowing the port and host it is running on is required to set up load balancing or a DNS alias.

The callback function logs a message to our console so we can know when the server began listening to connections.

Note: Even though `requestListener()` does not use the `req` object, it must still be the first argument of the function.

With less than fifteen lines of code, we now have a web server. Let's see it in action and test it end-to-end by running the program:

```
node hello.js
```

In the console, we will see this output:

```
Output
Server is running on http://localhost:8000
```

Notice that the prompt disappears. This is because a Node.js server is a long running process. It only exits if it encounters an error that causes it to crash and quit, or if we stop the Node.js process running the server.

In a separate terminal window, we'll communicate with the server using cURL, a CLI tool to transfer data to and from a network. Enter the command to make an HTTP `GET` request to our running server:

```
curl http://localhost:8000
```

When we press `ENTER`, our terminal will show the following output:

```
Output
My first server!
```

We've now set up a server and got our first server response.

Let's break down what happened when we tested our server. Using cURL, we sent a `GET` request to the server at `http://localhost:8000`. Our Node.js server listened to connections from that address. The server passed that request to the `requestListener()` function. The function returned text data with the status code `200`. The server then sent that response back to cURL, which displayed the message in our terminal.

Before we continue, let's exit our running server by pressing `CTRL+C`. This interrupts our server's execution, bringing us back to the command line prompt.

In most web sites we visit or APIs we use, the server responses are seldom in plain text. We get HTML pages and JSON data as common response formats. In the next step, we will learn how to return HTTP responses in common data formats we encounter in the web.

Step 2 — Returning Different Types of Content

The response we return from a web server can take a variety of formats. JSON and HTML were mentioned before, and we can also return other text

formats like XML and CSV. Finally, web servers can return non-text data like PDFs, zipped files, audio, and video.

In this article, in addition to the plain text we just returned, you'll learn how to return the following types of data:

- JSON
- CSV
- HTML

The three data types are all text-based, and are popular formats for delivering content on the web. Many server-side development languages and tools have support for returning these different data types. In the context of Node.js, we need to do two things:

1. Set the `Content-Type` header in our HTTP responses with the appropriate value.
2. Ensure that `res.end()` gets the data in the right format.

Let's see this in action with some examples. The code we will be writing in this section and later ones have many similarities to the code we wrote previously. Most changes exist within the `requestListener()` function. Let's create files with this "template code" to make future sections easier to follow.

Create a new file called `html.js`. This file will be used later to return HTML text in an HTTP response. We'll put the template code here and copy it to the other servers that return various types.

In the terminal, enter the following:

```
touch html.js
```

Now open this file in a text editor:

```
nano html.js
```

Let's copy the "template code." Enter this in `nano`:

first-servers/html.js

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

const host = 'localhost';
const port = 8000;

const requestListener = function (req, res) {};

const server = http.createServer(requestListener);
server.listen(port, host, () => {
    console.log(`Server is running on http://${host}:${port}`);
});
```

Save and exit `html.js` with `CTRL+X`, then return to the terminal.

Now let's copy this file into two new files. The first file will be to return CSV data in the HTTP response:

```
cp html.js csv.js
```

The second file will return a JSON response in the server:

```
cp html.js json.js
```

The remaining files will be for later exercises:

```
cp html.js htmlFile.js
cp html.js routes.js
```

We're now set up to continue our exercises. Let's begin with returning JSON.

Serving JSON

JavaScript Object Notation, commonly referred to as JSON, is a text-based data exchange format. As its name suggests, it is derived from JavaScript objects, but it is language independent, meaning it can be used by any programming language that can parse its syntax.

JSON is commonly used by APIs to accept and return data. Its popularity is due to lower data transfer size than previous data exchange standards like XML, as well as the tooling that exists that allow programs to parse them without excessive effort. If you'd like to learn more about JSON, you can read our guide on How To Work with JSON in JavaScript.

Open the `json.js` file with `nano`:

```
nano json.js
```

We want to return a JSON response. Let's modify the `requestListener()` function to return the appropriate header all JSON responses have by

changing the highlighted lines like so:

first-servers/json.js

```
...
const requestListener = function (req, res) {
    res.setHeader("Content-Type", "application/json");
};
...
```

The `res.setHeader()` method adds an HTTP header to the response. HTTP headers are additional information that can be attached to a request or a response. The `res.setHeader()` method takes two arguments: the header's name and its value.

The `Content-Type` header is used to indicate the format of the data, also known as media type, that's being sent with the request or response. In this case our `Content-Type` is `application/json`.

Now, let's return JSON content to the user. Modify `json.js` so it looks like this:

first-servers/json.js

```
...
const requestListener = function (req, res) {
    res.setHeader("Content-Type", "application/json");
    res.writeHead(200);
    res.end(`{"message": "This is a JSON response"}`);
};
...
```

Like before, we tell the user that their request was successful by returning a status code of `200`. This time in the `response.end()` call, our string argument contains valid JSON.

Save and exit `json.js` by pressing `CTRL+X`. Now, let's run the server with the `node` command:

```
node json.js
```

In another terminal, let's reach the server by using cURL:

```
curl http://localhost:8000
```

As we press `ENTER`, we will see the following result:

```
Output
{"message": "This is a JSON response"}
```

We now have successfully returned a JSON response, just like many of the popular APIs we create apps with. Be sure to exit the running server with `CTRL+C` so we can return to the standard terminal prompt. Next, let's look at another popular format of returning data: CSV.

Serving CSV

The Comma Separated Values (CSV) file format is a text standard that's commonly used for providing tabular data. In most cases, each row is separated by a newline, and each item in the row is separated by a comma.

In our workspace, open the `csv.js` file with a text editor:

```
nano csv.js
```

Let's add the following lines to our `requestListener()` function:

first-servers/csv.js

```
...
const requestListener = function (req, res) {
    res.setHeader("Content-Type", "text/csv");
    res.setHeader("Content-Disposition", "attachment;filename=o
};
...
```

This time, our `Content-Type` indicates that a CSV file is being returned as the value is `text/csv`. The second header we add is `Content-Dispositio`

`n`. This header tells the browser how to display the data, particularly in the browser or as a separate file.

When we return CSV responses, most modern browsers automatically download the file even if the `Content-Disposition` header is not set. However, when returning a CSV file we should still add this header as it allows us to set the name of the CSV file. In this case, we signal to the browser that this CSV file is an attachment and should be downloaded. We then tell the browser that the file's name is `oceanpals.csv`.

Let's write the CSV data in the HTTP response:

first-servers/csv.js

```
...
const requestListener = function (req, res) {
    res.setHeader("Content-Type", "text/csv");
    res.setHeader("Content-Disposition", "attachment;filename=o
    res.writeHead(200);
    res.end(`id,name,email\n1,Sammy Shark,shark@ocean.com`);
};
...
```

Like before we return a `200`/`OK` status with our response. This time, our call to `res.end()` has a string that's a valid CSV. The comma separates the value in each column and the new line character (`\n`) separates the rows. We have two rows, one for the table header and one for the data.

We'll test this server in the browser. Save `csv.js` and exit the editor with `CTRL+X`.

Run the server with the Node.js command:

```
node csv.js
```

In another Terminal, let's reach the server by using cURL:

```
curl http://localhost:8000
```

The console will show this:

```
Output
id,name,email
1,Sammy Shark,shark@ocean.com
```

If we go to `http://localhost:8000` in our browser, a CSV file will be downloaded. Its file name will be `oceanpals.csv`.

Exit the running server with `CTRL+C` to return to the standard terminal prompt.

Having returned JSON and CSV, we've covered two cases that are popular for APIs. Let's move on to how we return data for websites people view in a browser.

Serving HTML

HTML, HyperText Markup Language, is the most common format to use when we want users to interact with our server via a web browser. It was

created to structure web content. Web browsers are built to display HTML content, as well as any styles we add with CSS, another front-end web technology that allows us to change the aesthetics of our websites.

Let's reopen `html.js` with our text editor:

```
nano html.js
```

Modify the `requestListener()` function to return the appropriate `Content-Type` header for an HTML response:

first-servers/html.js

```
...
const requestListener = function (req, res) {
    res.setHeader("Content-Type", "text/html");
};
...
```

Now, let's return HTML content to the user. Add the highlighted lines to `html.js` so it looks like this:

first-servers/html.js

```
...
const requestListener = function (req, res) {
    res.setHeader("Content-Type", "text/html");
    res.writeHead(200);
    res.end(`<html><body><h1>This is HTML</h1></body></html>`);
};
...
```

We first add the HTTP status code. We then call `response.end()` with a string argument that contains valid HTML. When we access our server in the browser, we will see an HTML page with one header tag containing `This is HTML`.

Let's save and exit by pressing `CTRL+X`. Now, let's run the server with the `node` command:

```
node html.js
```

We will see `Server is running on http://localhost:8000` when our program has started.

Now go into the browser and visit `http://localhost:8000`. Our page will look like this:

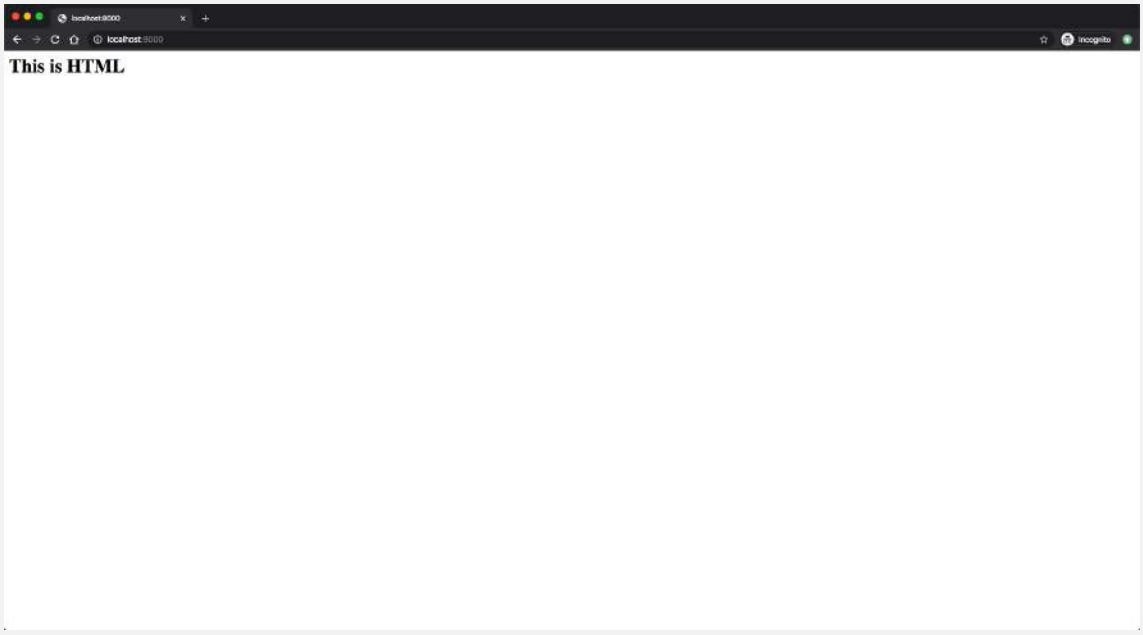


Image of HTML response returned from Node.js server

Let's quit the running server with `CTRL+C` and return to the standard terminal prompt.

It's common for HTML to be written in a file, separate from the server-side code like our Node.js programs. Next, let's see how we can return HTML responses from files.

Step 3 — Serving an HTML Page From a File

We can serve HTML as strings in Node.js to the user, but it's preferable that we load HTML files and serve their content. This way, as the HTML file grows we don't have to maintain long strings in our Node.js code, keeping it more concise and allowing us to work on each aspect of our website independently. This "separation of concerns" is common in many web

development setups, so it's good to know how to load HTML files to support it in Node.js

To serve HTML files, we load the HTML file with the `fs` module and use its data when writing our HTTP response.

First, we'll create an HTML file that the web server will return. Create a new HTML file:

```
touch index.html
```

Now open `index.html` in a text editor:

```
nano index.html
```

Our web page will be minimal. It will have an orange background and will display some greeting text in the center. Add this code to the file:

first-servers/index.html

```
<!DOCTYPE html>

<head>
    <title>My Website</title>
    <style>
        *,
        html {
            margin: 0;
            padding: 0;
            border: 0;
        }

        html {
            width: 100%;
            height: 100%;
        }

        body {
            width: 100%;
            height: 100%;
            position: relative;
            background-color: rgb(236, 152, 42);
        }

        .center {
```

```
            width: 100%;
            height: 50%;
            margin: 0;
            position: absolute;
            top: 50%;
            left: 50%;
            transform: translate(-50%, -50%);
            color: white;
            font-family: "Trebuchet MS", Helvetica, sans-serif;
            text-align: center;
        }

        h1 {
            font-size: 144px;
        }

        p {
            font-size: 64px;
        }
    </style>
</head>

<body>
    <div class="center">
        <h1>Hello Again!</h1>
        <p>This is served from a file</p>
    </div>
```

```
</body>


</html>
```

This single webpage shows two lines of text: `Hello Again!` and `This is served from a file`. The lines appear in the center of the page, one above each other. The first line of text is displayed in a heading, meaning it would be large. The second line of text will appear slightly smaller. All the text will appear white and the webpage has an orange background.

While it's not the scope of this article or series, if you are interested in learning more about HTML, CSS, and other front-end web technologies, you can take a look at Mozilla's Getting Started with the Web guide.

That's all we need for the HTML, so save and exit the file with `CTRL+X`. We can now move on to the server code.

For this exercise, we'll work on `htmlFile.js`. Open it with the text editor:

```
nano htmlFile.js
```

As we have to read a file, let's begin by importing the `fs` module:

first-servers/htmlFile.js

```
const http = require("http");
const fs = require('fs').promises;
...
```

This module contains a `readFile()` function that we'll use to load the HTML file in place. We import the promise variant in keeping with modern JavaScript best practices. We use promises as its syntactically more succinct than callbacks, which we would have to use if we assigned `fs` to just `require('fs')`. To learn more about asynchronous programming best practices, you can read our How To Write Asynchronous Code in Node.js guide.

We want our HTML file to be read when a user requests our system. Let's begin by modifying `requestListener()` to read the file:

first-servers/htmlFile.js

```
...
const requestListener = function (req, res) {
    fs.readFile(__dirname + "/index.html")
};
...
```

We use the `fs.readFile()` method to load the file. Its argument has `__dirname + "/index.html"`. The special variable `__dirname` has the absolute

path of where the Node.js code is being run. We then append `/index.html` so we can load the HTML file we created earlier.

Now let's return the HTML page once it's loaded:

first-servers/htmlFile.js

```
...
const requestListener = function (req, res) {
    fs.readFile(__dirname + "/index.html")
        .then(contents => {
            res.setHeader("Content-Type", "text/html");
            res.writeHead(200);
            res.end(contents);
        })
};
...
```

If the `fs.readFile()` promise successfully resolves, it will return its data. We use the `then()` method to handle this case. The `contents` parameter contains the HTML file's data.

We first set the `Content-Type` header to `text/html` to tell the client that we are returning HTML data. We then write the status code to indicate the request was successful. We finally send the client the HTML page we loaded, with the data in the `contents` variable.

The `fs.readFile()` method can fail at times, so we should handle this case when we get an error. Add this to the `requestListener()` function:

first-servers/htmlFile.js

```
...
const requestListener = function (req, res) {
    fs.readFile(__dirname + "/index.html")
        .then(contents => {
            res.setHeader("Content-Type", "text/html");
            res.writeHead(200);
            res.end(contents);
        })
        .catch(err => {
            res.writeHead(500);
            res.end(err);
            return;
        });
};
...
```

Save the file and exit `nano` with `CTRL+X`.

When a promise encounters an error, it is rejected. We handle that case with the `catch()` method. It accepts the error that `fs.readFile()` returns, sets the status code to `500` signifying that an internal error was encountered, and returns the error to the user.

Run our server with the `node` command:

```
node htmlFile.js
```

In the web browser, visit `http://localhost:8000`. You will see this page:

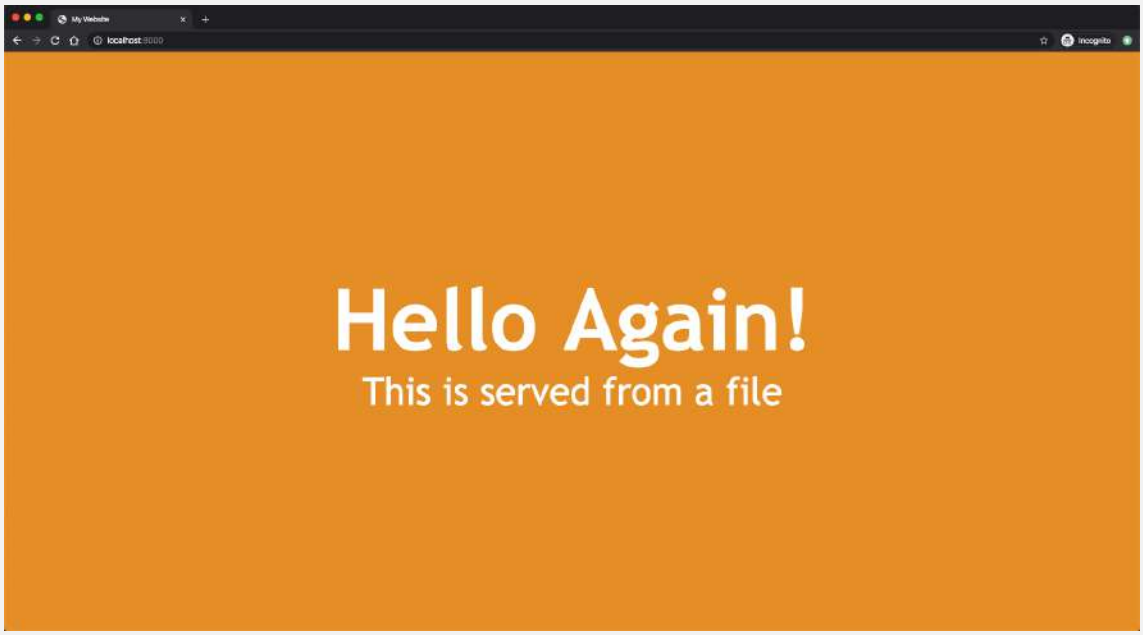


Image of HTML page loaded from a file in Node.js

You have now returned an HTML page from the server to the user. You can quit the running server with `CTRL+C`. You will see the terminal prompt return when you do.

When writing code like this in production, you may not want to load an HTML page every time you get an HTTP request. While this HTML page is roughly 800 bytes in size, more complex websites can be megabytes in size. Large files can take a while to load. If your site is expecting a lot of traffic, it may be best to load HTML files at startup and save their contents. After

they are loaded, you can set up the server and make it listen to requests on an address.

To demonstrate this method, let's see how we can rework our server to be more efficient and scalable.

Serving HTML Efficiently

Instead of loading the HTML for every request, in this step we will load it once at the beginning. The request will return the data we loaded at startup.

In the terminal, re-open the Node.js script with a text editor:

```
nano htmlFile.js
```

Let's begin by adding a new variable before we create the `requestListener()` function:

first-servers/htmlFile.js

```
...
let indexFile;

const requestListener = function (req, res) {
...
```

When we run this program, this variable will hold the HTML file's contents.

Now, let's readjust the `requestListener()` function. Instead of loading the file, it will now return the contents of `indexFile`:

first-servers/htmlFile.js

```
...
const requestListener = function (req, res) {
    res.setHeader("Content-Type", "text/html");
    res.writeHead(200);
    res.end(indexFile);
};
...
```

Next, we shift the file reading logic from the `requestListener()` function to our server startup. Make the following changes as we create the server:

first-servers/htmlFile.js

```
...

const server = http.createServer(requestListener);

fs.readFile(__dirname + "/index.html")
    .then(contents => {
        indexFile = contents;
        server.listen(port, host, () => {
            console.log(`Server is running on http://${host}:${
        });
    })
    .catch(err => {
        console.error(`Could not read index.html file: ${err}`)
        process.exit(1);
    });
```

Save the file and exit `nano` with `CTRL+X`.

The code that reads the file is similar to what we wrote in our first attempt. However, when we successfully read the file we now save the contents to our global `indexFile` variable. We then start the server with the `listen()` method. The key thing is that the file is loaded before the server is run. This way, the `requestListener()` function will be sure to return an HTML page, as `indexFile` is no longer an empty variable.

Our error handler has changed as well. If the file can't be loaded, we capture the error and print it to our console. We then exit the Node.js program with the `exit()` function without starting the server. This way we can see why the file reading failed, address the problem, and then start the server again.

We've now created different web servers that return various types of data to a user. So far, we have not used any request data to determine what should be returned. We'll need to use request data when setting up different routes or paths in a Node.js server, so next let's see how they work together.

Step 4 — Managing Routes Using an HTTP Request Object

Most websites we visit or APIs we use usually have more than one endpoint so we can access various resources. A good example would be a book management system, one that might be used in a library. It would not only need to manage book data, but it would also manage author data for cataloguing and searching convenience.

Even though the data for books and authors are related, they are two different objects. In these cases, software developers usually code each object on different endpoints as a way to indicate to the API user what kind of data they are interacting with.

Let's create a new server for a small library, which will return two different types of data. If the user goes to our server's address at `/books`, they will receive a list of books in JSON. If they go to `/authors`, they will receive a list of author information in JSON.

So far, we have been returning the same response to every request we get. Let's illustrate this quickly.

Re-run our JSON response example:

```
node json.js
```

In another terminal, let's do a cURL request like before:

```
curl http://localhost:8000
```

You will see:

```
Output
{"message": "This is a JSON response"}
```

Now let's try another curl command:

```
curl http://localhost:8000/todos
```

After pressing `Enter`, you will see the same result:

```
Output
{"message": "This is a JSON response"}
```

We have not built any special logic in our `requestListener()` function to handle a request whose URL contains `/todos`, so Node.js returns the same JSON message by default.

As we want to build a miniature library management server, we'll now separate the kind of data that's returned based on the endpoint the user accesses.

First, exit the running server with `CTRL+C`.

Now open `routes.js` in your text editor:

```
nano routes.js
```

Let's begin by storing our JSON data in variables before the `requestListener()` function:

first-servers/routes.js

```
...
const books = JSON.stringify([
    { title: "The Alchemist", author: "Paulo Coelho", year: 198
    { title: "The Prophet", author: "Kahlil Gibran", year: 1923
]);

const authors = JSON.stringify([
    { name: "Paulo Coelho", countryOfBirth: "Brazil", yearOfBir
    { name: "Kahlil Gibran", countryOfBirth: "Lebanon", yearOfB
]);
...
```

The `books` variable is a string that contains JSON for an array of book objects. Each book has a title or name, an author, and the year it was published.

The `authors` variable is a string that contains the JSON for an array of author objects. Each author has a name, a country of birth, and their year of birth.

Now that we have the data our responses will return, let's start modifying the `requestListener()` function to return them to the correct routes.

First, we'll ensure that every response from our server has the correct `Content-Type` header:

first-servers/routes.js

```
...
const requestListener = function (req, res) {
    res.setHeader("Content-Type", "application/json");
}
...
```

Now, we want to return the right JSON depending on the URL path the user visits. Let's create a `switch` statement on the request's URL:

first-servers/routes.js

```
...
const requestListener = function (req, res) {
    res.setHeader("Content-Type", "application/json");
    switch (req.url) {}
}
...
```

To get the URL path from a request object, we need to access its `url` property. We can now add cases to the `switch` statement to return the appropriate JSON.

JavaScript's `switch` statement provides a way to control what code is run depending on the value of an object or JavaScript expression (for example, the result of mathematical operations). If you need a lesson or reminder on how to use them, take a look at our guide on How To Use the Switch Statement in JavaScript.

Let's continue by adding a `case` for when the user wants to get our list of books:

first-servers/routes.js

```
...
const requestListener = function (req, res) {
    res.setHeader("Content-Type", "application/json");
    switch (req.url) {
        case "/books":
            res.writeHead(200);
            res.end(books);
            break
    }
}
...
```

We set our status code to `200` to indicate the request is fine and return the JSON containing the list of our books. Now let's add another `case` for our authors:

first-servers/routes.js

```
...
const requestListener = function (req, res) {
    res.setHeader("Content-Type", "application/json");
    switch (req.url) {
        case "/books":
            res.writeHead(200);
            res.end(books);
            break
        case "/authors":
            res.writeHead(200);
            res.end(authors);
            break
    }
}
...
```

Like before, the status code will be `200` as the request is fine. This time we return the JSON containing the list of our authors.

We want to return an error if the user tries to go to any other path. Let's add the default case to do this:

routes.js

```
...
const requestListener = function (req, res) {
    res.setHeader("Content-Type", "application/json");
    switch (req.url) {
        case "/books":
            res.writeHead(200);
            res.end(books);
            break
        case "/authors":
            res.writeHead(200);
            res.end(authors);
            break
        default:
            res.writeHead(404);
            res.end(JSON.stringify({error:"Resource not found"}
    }
}
...
```

We use the `default` keyword in a `switch` statement to capture all other scenarios not captured by our previous cases. We set the status code to `404` to indicate that the URL they were looking for was not found. We then set a JSON object that contains an error message.

Let’s test our server to see if it behaves as we expect. In another terminal, let’s first run a command to see if we get back our list of books:

```
curl http://localhost:8000/books
```

Press `Enter` to see the following output:

```
Output
[{"title":"The Alchemist","author":"Paulo Coelho","year":198
8},{"title":"The Prophet","author":"Kahlil Gibran","year":192
3}]
```

So far so good. Let’s try the same for `/authors`. Type the following command in the terminal:

```
curl http://localhost:8000/authors
```

You will see the following output when the command is complete:

```
Output
[{"name":"Paulo Coelho","countryOfBirth":"Brazil","yearOfBirt
h":1947},{"name":"Kahlil Gibran","countryOfBirth":"Lebanon","y
earOfBirth":1883}]
```

Last, let’s try an erroneous URL to ensure that `requestListener()` returns the error response:

```
curl http://localhost:8000/notreal
```

Entering that command will display this message:

```
Output
{"error":"Resource not found"}
```

You can exit the running server with `CTRL+C`.

We've now created different avenues for users to get different data. We also added a default response that returns an HTTP error if the user enters a URL that we don't support.

Conclusion

In this tutorial, you've made a series of Node.js HTTP servers. You first returned a basic textual response. You then went on to return various types of data from our server: JSON, CSV, and HTML. From there you were able to combine file loading with HTTP responses to return an HTML page from the server to the user, and to create an API that used information about the user's request to determine what data should be sent in its response.

You're now equipped to create web servers that can handle a variety of requests and responses. With this knowledge, you can make a server that returns many HTML pages to the user at different endpoints. You could also create your own API.

To learn about more HTTP web servers in Node.js, you can read the Node.js documentation on the `http` module. If you'd like to continue learning Node.js, you can return to the How To Code in Node.js series page.

Using Buffers in Node.js

Written by Stack Abuse

The author selected the COVID-19 Relief Fund to receive a donation as part of the Write for DOnations program.

A buffer is a space in memory (typically RAM) that stores binary data. In Node.js, we can access these spaces of memory with the built-in `Buffer` class. Buffers store a sequence of integers, similar to an array in JavaScript. Unlike arrays, you cannot change the size of a buffer once it is created.

You may have used buffers implicitly if you wrote Node.js code already. For example, when you read from a file with `fs.readFile()`, the data returned to the callback or Promise is a buffer object. Additionally, when HTTP requests are made in Node.js, they return data streams that are temporarily stored in an internal buffer when the client cannot process the stream all at once.

Buffers are useful when you're interacting with binary data, usually at lower networking levels. They also equip you with the ability to do fine-grained data manipulation in Node.js.

In this tutorial, you will use the Node.js REPL to run through various examples of buffers, such as creating buffers, reading from buffers, writing to and copying from buffers, and using buffers to convert between binary and encoded data. By the end of the tutorial, you'll have learned how to use the `Buffer` class to work with binary data.

Prerequisites

- You will need Node.js installed on your development machine. This tutorial uses version 10.19.0. To install this on macOS or Ubuntu 18.04, follow the steps in How To Install Node.js and Create a Local Development Environment on macOS or the **Installing Using a PPA** section of How To Install Node.js on Ubuntu 18.04.
- In this tutorial, you will interact with buffers in the Node.js REPL (Read-Evaluate-Print-Loop). If you want a refresher on how to use the Node.js REPL effectively, you can read our guide on How To Use the Node.js REPL.
- For this article we expect the user to be comfortable with basic JavaScript and its data types. You can learn those fundamentals with our How To Code in JavaScript series.

Step 1 — Creating a Buffer

This first step will show you the two primary ways to create a buffer object in Node.js.

To decide what method to use, you need to answer this question: Do you want to create a new buffer or extract a buffer from existing data? If you are going to store data in memory that you have yet to receive, you'll want to create a new buffer. In Node.js we use the `alloc()` function of the `Buffer` class to do this.

Let's open the Node.js REPL to see for ourselves. In your terminal, enter the `node` command:

```
node
```

You will see the prompt begin with `>`.

The `alloc()` function takes the size of the buffer as its first and only required argument. The size is an integer representing how many bytes of memory the buffer object will use. For example, if we wanted to create a buffer that was 1KB (kilobyte) large, equivalent to 1024 bytes, we would enter this in the console:

```
const firstBuf = Buffer.alloc(1024);
```

To create a new buffer, we used the globally available `Buffer` class, which has the `alloc()` method. By providing `1024` as the argument for `alloc()`, we created a buffer that's 1KB large.

By default, when you initialize a buffer with `alloc()`, the buffer is filled with binary zeroes as a placeholder for later data. However, we can change the default value if we'd like to. If we wanted to create a new buffer with `1`s instead of `0`s, we would set the `alloc()` function's second parameter—`fill`.

In your terminal, create a new buffer at the REPL prompt that's filled with `1`s:

```
const filledBuf = Buffer.alloc(1024, 1);
```

We just created a new buffer object that references a space in memory that stores 1KB of `1`s. Although we entered an integer, all data stored in a buffer is binary data.

Binary data can come in many different formats. For example, let's consider a binary sequence representing a byte of data: `01110110`. If this binary sequence represented a string in English using the ASCII encoding

standard, it would be the letter `v`. However, if our computer was processing an image, that binary sequence could contain information about the color of a pixel.

The computer knows to process them differently because the bytes are encoded differently. Byte encoding is the format of the byte. A buffer in Node.js uses the UTF-8 encoding scheme by default if it's initialized with string data. A byte in UTF-8 represents a number, a letter (in English and in other languages), or a symbol. UTF-8 is a superset of ASCII, the American Standard Code for Information Interchange. ASCII can encode bytes with uppercase and lowercase English letters, the numbers 0-9, and a few other symbols like the exclamation mark (**!**) or the ampersand sign (**&**).

If we were writing a program that could only work with ASCII characters, we could change the encoding used by our buffer with the `allo c()` function's third argument—`encoding`.

Let's create a new buffer that's five bytes long and stores only ASCII characters:

```
const asciiBuf = Buffer.alloc(5, 'a', 'ascii');
```

The buffer is initialized with five bytes of the character `a`, using the ASCII representation.

Note: By default, Node.js supports the following character encodings:

- **ASCII**, represented as `ascii`
- **UTF-8**, represented as `utf-8` or `utf8`
- **UTF-16**, represented as `utf-16le` or `utf16le`
- **UCS-2**, represented as `ucs-2` or `ucs2`

- **Base64**, represented as `base64`
- **Hexadecimal**, represented as `hex`
- **ISO/IEC 8859-1**, represented as `latin1` or `binary`

All of these values can be used in Buffer class functions that accept an `encoding` parameter. Therefore, these values are all valid for the `alloc()` method.

So far we've been creating new buffers with the `alloc()` function. But sometimes we may want to create a buffer from data that already exists, like a string or array.

To create a buffer from pre-existing data, we use the `from()` method. We can use that function to create buffers from:

- An array of integers: The integer values can be between `0` and `255`.
- An `ArrayBuffer`: This is a JavaScript object that stores a fixed length of bytes.
- A string.
- Another buffer.
- Other JavaScript objects that have a `Symbol.toPrimitive` property. That property tells JavaScript how to convert the object to a primitive data type: `boolean`, `null`, `undefined`, `number`, `string`, or `symbol`. You can read more about Symbols at Mozilla's JavaScript documentation.

Let's see how we can create a buffer from a string. In the Node.js prompt, enter this:

```
const stringBuf = Buffer.from('My name is Paul');
```

We now have a buffer object created from the string `My name is Paul`. Let's create a new buffer from another buffer we made earlier:

```
const asciiCopy = Buffer.from(asciiBuf);
```

We've now created a new buffer `asciiCopy` that contains the same data as `asciiBuf`.

Now that we have experienced creating buffers, we can dive into examples of reading their data.

Step 2 — Reading from a Buffer

There are many ways to access data in a Buffer. We can access an individual byte in a buffer or we can extract the entire contents.

To access one byte of a buffer, we pass the index or location of the byte we want. Buffers store data sequentially like arrays. They also index their data like arrays, starting at `0`. We can use array notation on the buffer object to get an individual byte.

Let's see how this looks by creating a buffer from a string in the REPL:

```
const hiBuf = Buffer.from('Hi!');
```

Now let's read the first byte of the buffer:

```
hiBuf[0];
```

As you press `ENTER`, the REPL will display:

```
Output
72
```

The integer `72` corresponds the UTF-8 representation for the letter `H`.

Note: The values for bytes can be numbers between `0` and `255`. A byte is a sequence of 8 bits. A bit is binary, and therefore can only have one of two values: `0` or `1`. If we have a sequence of 8 bits and two possible values per bit, then we have a maximum of 2^8 possible values for a byte. That works out to a maximum of 256 values. Since we start counting from zero, that means our highest number is 255.

Let's do the same for the second byte. Enter the following in the REPL:

```
hiBuf[1];
```

The REPL returns `105`, which represents the lowercase `i`.

Finally, let's get the third character:

```
hiBuf[2];
```

You will see `33` displayed in the REPL, which corresponds to `!`.

Let's try to retrieve a byte from an invalid index:

```
hiBuf[3];
```

The REPL will return:

```
Output
undefined
```

This is just like if we tried to access an element in an array with an incorrect index.

Now that we've seen how to read individual bytes of a buffer, let's see our options for retrieving all the data stored in a buffer at once. The buffer object comes with the `toString()` and the `toJSON()` methods, which return the entire contents of a buffer in two different formats.

As its name suggests, the `toString()` method converts the bytes of the buffer into a string and returns it to the user. If we use this method on `hiBuf`, we will get the string `Hi!`. Let's try it!

In the prompt, enter:

```
hiBuf.toString();
```

The REPL will return:

```
Output
'Hi!'
```

That buffer was created from a string. Let's see what happens if we use the `toString()` on a buffer that was not made from string data.

Let's create a new, empty buffer that's `10` bytes large:

```
const tenZeroes = Buffer.alloc(10);
```

Now, let's use the `toString()` method:

```
tenZeroes.toString();
```

We will see the following result:

```
'\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u0000'
```

The string `\u0000` is the Unicode character for `NULL`. It corresponds to the number `0`. When the buffer's data is not encoded as a string, the `toString()` method returns the UTF-8 encoding of the bytes.

The `toString()` has an optional parameter, `encoding`. We can use this parameter to change the encoding of the buffer data that's returned.

For example, if you wanted the hexadecimal encoding for `hiBuf` you would enter the following at the prompt:

```
hiBuf.toString('hex');
```

That statement will evaluate to:

```
Output
'486921'
```

`486921` is the hexadecimal representation for the bytes that represent the string `Hi!`. In Node.js, when users want to convert the encoding of data from one form to another, they usually put the string in a buffer and call `toString()` with their desired encoding.

The `toJSON()` method behaves differently. Regardless of whether the buffer was made from a string or not, it always returns the data as the integer representation of the byte.

Let's re-use the `hiBuf` and `tenZeroes` buffers to practice using `toJSON()`. At the prompt, enter:

```
hiBuf.toJSON();
```

The REPL will return:

```
Output
{ type: 'Buffer', data: [ 72, 105, 33 ] }
```

The JSON object has a `type` property that will always be `Buffer`. That's so programs can distinguish these JSON object from other JSON objects.

The `data` property contains an array of the integer representation of the bytes. You may have noticed that `72`, `105`, and `33` correspond to the values we received when we individually pulled the bytes.

Let's try the `toJSON()` method with `tenZeroes`:

```
tenZeroes.toJSON();
```

In the REPL you will see the following:

```
Output
{ type: 'Buffer', data: [
    0, 0, 0, 0, 0,
    0, 0, 0, 0, 0
  ] }
```

The `type` is the same as noted before. However, the data is now an array with ten zeroes.

Now that we've covered the main ways to read from a buffer, let's look at how we modify a buffer's contents.

Step 3 — Modifying a Buffer

There are many ways we can modify an existing buffer object. Similar to reading, we can modify buffer bytes individually using the array syntax. We can also write new contents to a buffer, replacing the existing data.

Let's begin by looking at how we can change individual bytes of a buffer. Recall our buffer variable `hiBuf`, which contains the string `Hi!`. Let's change each byte so that it contains `Hey` instead.

In the REPL, let's first try setting the second element of `hiBuf` to `e`:

```
hiBuf[1] = 'e';
```

Now, let's see this buffer as a string to confirm it's storing the right data. Follow up by calling the `toString()` method:

```
hiBuf.toString();
```

It will be evaluated as:

```
Output
'H\u0000!'
```

We received that strange output because the buffer can only accept an integer value. We can't assign it to the letter `e`; rather, we have to assign it the number whose binary equivalent represents `e`:

```
hiBuf[1] = 101;
```

Now when we call the `toString()` method:

```
hiBuf.toString();
```

We get this output in the REPL:

```
Output
'He!'
```

To change the last character in the buffer, we need to set the third element to the integer that corresponds to the byte for `y`:

```
hiBuf[2] = 121;
```

Let's confirm by using the `toString()` method once again:

```
hiBuf.toString();
```

Your REPL will display:

```
Output
'Hey'
```

If we try to write a byte that's outside the range of the buffer, it will be ignored and the contents of the buffer won't change. For example, let's try to set the non-existent fourth element of the buffer to `o`:

```
hiBuf[3] = 111;
```

We can confirm that the buffer is unchanged with the `toString()` method:

```
hiBuf.toString();
```

The output is still:

```
Output
'Hey'
```

If we wanted to change the contents of the entire buffer, we can use the `write()` method. The `write()` method accepts a string that will replace the contents of a buffer.

Let's use the `write()` method to change the contents of `hiBuf` back to `Hi!`. In your Node.js shell, type the following command at the prompt:

```
hiBuf.write('Hi!');
```

The `write()` method returned `3` in the REPL. This is because it wrote three bytes of data. Each letter has one byte in size, since this buffer uses UTF-8 encoding, which uses a byte for each character. If the buffer used

UTF-16 encoding, which has a minimum of two bytes per character, then the `write()` function would have returned `6`.

Now verify the contents of the buffer by using `toString()`:

```
hiBuf.toString();
```

The REPL will produce:

```
Output
'Hi!'
```

This is quicker than having to change each element byte-by-byte.

If you try to write more bytes than a buffer's size, the buffer object will only accept what bytes fit. To illustrate, let's create a buffer that stores three bytes:

```
const petBuf = Buffer.alloc(3);
```

Now let's attempt to write `Cats` to it:

```
petBuf.write('Cats');
```

When the `write()` call is evaluated, the REPL returns `3` indicating only three bytes were written to the buffer. Now confirm that the buffer contains the first three bytes:

```
petBuf.toString();
```

The REPL returns:

```
Output
'Cat'
```

The `write()` function adds the bytes in sequential order, so only the first three bytes were placed in the buffer.

By contrast, let's make a `Buffer` that stores four bytes:

```
const petBuf2 = Buffer.alloc(4);
```

Write the same contents to it:

```
petBuf2.write('Cats');
```

Then add some new content that occupies less space than the original content:

```
petBuf2.write('Hi');
```

Since buffers write sequentially, starting from `0`, if we print the buffer's contents:

```
petBuf2.toString();
```

We'd be greeted with:

```
Output
'Hits'
```

The first two characters are overwritten, but the rest of the buffer is untouched.

Sometimes the data we want in our pre-existing buffer is not in a string but resides in another buffer object. In these cases, we can use the `copy()` function to modify what our buffer is storing.

Let's create two new buffers:

```
const wordsBuf = Buffer.from('Banana Nananana');
const catchphraseBuf = Buffer.from('Not sure Turtle!');
```

The `wordsBuf` and `catchphraseBuf` buffers both contain string data. We want to modify `catchphraseBuf` so that it stores `Nananana Turtle!` instead of `Not sure Turtle!`. We'll use `copy()` to get `Nananana` from `wordsBuf` to `catchphraseBuf`.

To copy data from one buffer to the other, we'll use the `copy()` method on the buffer that's the source of the information. Therefore, as `wordsBuf` has the string data we want to copy, we need to copy like this:

```
wordsBuf.copy(catchphraseBuf);
```

The `target` parameter in this case is the `catchphraseBuf` buffer.

When we enter that into the REPL, it returns `15` indicating that 15 bytes were written. The string `Nananana` only uses 8 bytes of data, so we immediately know that our copy did not go as intended. Use the `toString()` method to see the contents of `catchphraseBuf`:

```
catchphraseBuf.toString();
```

The REPL returns:

```
Output
'Banana Nananana!'
```

By default, `copy()` took the entire contents of `wordsBuf` and placed it into `catchphraseBuf`. We need to be more selective for our goal and only copy `Nananana`. Let's re-write the original contents of `catchphraseBuf` before continuing:

```
catchphraseBuf.write('Not sure Turtle!');
```

The `copy()` function has a few more parameters that allow us to customize what data is copied to the other buffer. Here's a list of all the parameters of this function:

- `target` - This is the only required parameter of `copy()`. As we've seen from our previous usage, it is the buffer we want to copy to.
- `targetStart` - This is the index of the bytes in the target buffer where we should begin copying to. By default it's `0`, meaning it copies data starting at the beginning of the buffer.
- `sourceStart` - This is the index of the bytes in the source buffer where we should copy from.
- `sourceEnd` - This is the index of the bytes in the source buffer where we should stop copying. By default, it's the length of the buffer.

So, to copy `Nananana` from `wordsBuf` into `catchphraseBuf`, our `target` should be `catchphraseBuf` like before. The `targetStart` would be `0` as we want `Nananana` to appear at the beginning of `catchphraseBuf`. The `sourceStart` should be `7` as that's the index where `Nananana` begins in `wordsBuf`. The `sourceEnd` would continue to be the length of the buffers.

At the REPL prompt, copy the contents of `wordsBuf` like this:

```
wordsBuf.copy(catchphraseBuf, 0, 7, wordsBuf.length);
```

The REPL confirms that `8` bytes have been written. Note how `wordsBuf.length` is used as the value for the `sourceEnd` parameter. Like arrays, the `length` property gives us the size of the buffer.

Now let's see the contents of `catchphraseBuf`:

```
catchphraseBuf.toString();
```

The REPL returns:

```
Output
'Nananana Turtle!'
```

Success! We were able to modify the data of `catchphraseBuf` by copying the contents of `wordsBuf`.

You can exit the Node.js REPL if you would like to do so. Note that all the variables that were created will no longer be available when you do:

```
.exit
```

Conclusion

In this tutorial, you learned that buffers are fixed-length allocations in memory that store binary data. You first created buffers by defining their size in memory and by initializing them with pre-existing data. You then read data from a buffer by examining their individual bytes and by using the `toString()` and `toJSON()` methods. Finally, you modified the data stored by a buffer by changing its individual bytes and by using the `write()` and `copy()` methods.

Buffers give you great insight into how binary data is manipulated by Node.js. Now that you can interact with buffers, you can observe the different ways character encoding affect how data is stored. For example, you can create buffers from string data that are not UTF-8 or ASCII encoding and observe their difference in size. You can also take a buffer with UTF-8 and use `toString()` to convert it to other encoding schemes.

To learn about buffers in Node.js, you can read the Node.js documentation on the `Buffer` object. If you'd like to continue learning Node.js, you can return to the How To Code in Node.js series, or browse programming projects and setups on our Node topic page.

Using Event Emitters in Node.js

Written by Stack Abuse

The author selected the COVID-19 Relief Fund to receive a donation as part of the Write for DOnations program.

Event emitters are objects in Node.js that trigger an event by sending a message to signal that an action was completed. JavaScript developers can write code that listens to events from an event emitter, allowing them to execute functions every time those events are triggered. In this context, events are composed of an identifying string and any data that needs to be passed to the listeners.

Typically in Node.js, when we want to have an action occur upon completion of another action, we use asynchronous programming techniques like nesting callbacks or chaining promises. However, these techniques tightly couple the triggering action and the resulting action, making it difficult to modify the resulting action in the future. Event emitters provide a different way to structure this relationship: the publish-subscribe pattern. In this software architecture pattern, a publisher (the event emitter) sends a message (an event), and a subscriber receives the event and performs an action. The power of this pattern is that the publisher does not need to know about the subscribers. A publisher publishes a message, and it's up to the subscribers to react to it in their respective ways. If we wanted to change the behavior of our application, we could adjust how the subscribers react to the events without having to change the publisher.

In this article, we will create an event listener for a `TicketManager` JavaScript class that allows a user to buy tickets. We will set up listeners for the `buy` event, which will trigger every time a ticket is bought. This process will also show how to manage erroneous events from the emitter and how to manage event subscribers.

Prerequisites

- Node.js installed on your development machine. This tutorial uses version 10.20.1. To install this on macOS or Ubuntu 18.04, follow the steps in How to Install Node.js and Create a Local Development Environment on macOS or the **Installing Using a PPA** section of How To Install Node.js on Ubuntu 18.04.
- The main example in this article makes use of JavaScript classes as they were introduced in ES2015 (commonly referred to as ES6). If you'd like to learn about classes in JavaScript, read our Understanding Classes in JavaScript tutorial.

Step 1 — Emitting Events

In this step, we'll explore the two most common ways to create an event emitter in Node.js. The first is to use an event emitter object directly, and the second is to create an object that extends the event emitter object.

Deciding which one to use depends on how coupled your events are to the actions of your objects. If the events you want to emit are an effect of an object's actions, you would likely extend from the event emitter object to have access to its functions for convenience. If the events you want to emit are independent of your business objects or are a result of actions from

many business objects, you would instead create an independent event emitter object that's referenced by your objects.

Let's begin by creating a standalone, event-emitting object. We'll begin by creating a folder to store all of our code. In your terminal, make a new folder called `event-emitters`:

```
mkdir event-emitters
```

Then enter that folder:

```
cd event-emitters
```

Open the first event emitter, `firstEventEmitter.js`, in a text editor. We will use `nano` as it's available in the terminal:

```
nano firstEventEmitter.js
```

In Node.js, we emit events via the `EventEmitter` class. This class is part of the `events` module. Let's begin by first loading the `events` module in our file by adding the following line:

event-emitters/firstEventEmitter.js

```
const { EventEmitter } = require("events");
```

With the class imported, we can use it to create a new object instance from it:

event-emitters/firstEventEmitter.js

```
const { EventEmitter } = require("events");

const firstEmitter = new EventEmitter();
```

Let's emit an event by adding the following highlighted line at the end of `firstEventEmitter.js`:

event-emitters/firstEventEmitter.js

```
const { EventEmitter } = require("events");

const firstEmitter = new EventEmitter();


firstEmitter.emit("My first event");
```

The `emit()` function is used to fire events. We need to pass the name of the event to it as a string. We can add any number of arguments after the event name. Events with just a name are fairly limited; the other arguments allow us to send data to our listeners. When we set up our ticket manager, our events will pass data about the purchase when it happens. Keep the name of the event in mind, because event listeners will identify it by this name.

Note: While we don't capture it in this example, the `emit()` function returns `true` if there are listeners for the event. If there are no listeners for an event, it returns `false`.

Let's run this file to see what happens. Save and exit `nano`, then execute the file with the `node` command:

```
node firstEventEmitter.js
```

When the script finishes its execution, you will see no output in the terminal. That's because we do not log any messages in `firstEventEmitter.js` and there's nothing that listens to the event that was sent. The event is emitted, but nothing acts on these events.

Let's work toward seeing a more complete example of publishing, listening to, and acting upon events. We'll do this by creating a ticket manager example application. The ticket manager will expose a function to buy tickets. When a ticket is bought, an event will be sent with details of the purchaser. Later, we'll create another Node.js module to simulate an email being sent to the purchaser's email confirming the purchase.

Let's begin by creating our ticket manager. It will extend the `EventEmitter` class so that we don't have to create a separate event emitter object to emit events.

In the same working directory, create and open a new file called `ticketManager.js`:

```
nano ticketManager.js
```

As with the first event emitter, we need to import the `EventEmitter` class from the `events` module. Put the following code at the beginning of the file:

event-emitters/ticketManager.js

```
const EventEmitter = require("events");
```

Now, make a new `TicketManager` class that will soon define the method for ticket purchases:

event-emitters/ticketManager.js

```
const EventEmitter = require("events");

class TicketManager extends EventEmitter {}
```

In this case, the `TicketManager` class extends the `EventEmitter` class. This means that the `TicketManager` class inherits the methods and properties of the `EventEmitter` class. This is how it gets access to the `emit()` method.

In our ticket manager, we want to provide the initial supply of tickets that can be purchased. We'll do this by accepting the initial supply in our `constructor()`, a special function that's called when a new object of a class is made. Add the following constructor to the `TicketManager` class:

event-emitters/ticketManager.js

```
const EventEmitter = require("events");

class TicketManager extends EventEmitter {
    constructor(supply) {
        super();
        this.supply = supply;
    }
}
```

The constructor has one `supply` argument. This is a number detailing the initial supply of tickets we can sell. Even though we declared that `TicketManager` is a child class of `EventEmitter`, we still need to call `super()` to get access to the methods and properties of `EventEmitter`. The `super()` function calls the constructor of the parent function, which in this case is `EventEmitter`.

Finally, we create a `supply` property for the object with `this.supply` and give it the value passed in by the constructor.

Now, let's add a `buy()` method that will be called when a ticket is purchased. This method will decrease the supply by one and emit an event with the purchase data.

Add the `buy()` method as follows:

event-emitters/ticketManager.js

```
const EventEmitter = require("events");

class TicketManager extends EventEmitter {
    constructor(supply) {
        super();
        this.supply = supply;
    }

    buy(email, price) {
        this.supply--;
        this.emit("buy", email, price, Date.now());
    }
}
```

In the `buy()` function, we take the purchaser's email address and the price they paid for the ticket. We then decrease the supply of tickets by one. We end by emitting a `buy` event. This time, we emit an event with extra data: the email and price that were passed in the function as well as a timestamp of when the purchase was made.

So that our other Node.js modules can use this class, we need to export it. Add this line at the end of the file:

event-emitters/ticketManager.js

```
...


module.exports = TicketManager
```

Save and exit the file.

We've finished our setup for the event emitter `TicketManager`. Now that we've put things in place to push events, we can move on to reading and processing those events. To do that, we will create event listeners in the next step.

Step 2 — Listening for Events

Node.js allows us to add a listener for an event with the `on()` function of an event emitter object. This listens for a particular event name and fires a callback when the event is triggered. Adding a listener typically looks like this:

```
eventEmitter.on(event_name, callback_function) {
    action
}
```

Note:: Node.js aliases the `on()` method with `addListener()`. They perform the same task. In this tutorial, we will continue to use `on()`.

Let's get some first-hand experience with listening to our first event. Create a new file called `firstListener.js`:

```
nano firstListener.js
```

As a demonstration of how the `on()` function works, let's log a simple message upon receiving our first event.

First, let's import `TicketManager` into our new Node.js module. Add the following code into `firstListener.js`:

event-emitters/firstListener.js

```
const TicketManager = require("./ticketManager");


const ticketManager = new TicketManager(10);
```

Recall that `TicketManager` objects need their initial supply of tickets when created. This is why we pass the `10` argument.

Now let's add our first Node.js event listener. It will listen to the `buy` event. Add the following highlighted code:

event-emitters/firstListener.js

```
const TicketManager = require("./ticketManager");

const ticketManager = new TicketManager(10);

ticketManager.on("buy", () => {
    console.log("Someone bought a ticket!");
});
```

To add a new listener, we used the `on()` function that's a part of the `ticketManager` object. The `on()` method is available to all event emitter objects, and since `TicketManager` inherits from the `EventEmitter` class, this method is available on all of the `TicketManager` instance objects.

The second argument to the `on()` method is a callback function, written as an arrow function. The code in this function is run after the event is emitted. In this case, we log `"Someone bought a ticket!"` to the console if a `buy` event is emitted.

Now that we set up a listener, let's use the `buy()` function so that the event will be emitted. At the end of your file add this:

event-emitters/firstListener.js

```
...

ticketManager.buy("test@email.com", 20);
```

This performs the `buy` method with a user email of `test@email.com` and a ticket price of `20`.

Save and exit the file.

Now run this script with `node`:

```
node firstListener.js
```

Your console will display this:

```
Output
Someone bought a ticket!
```

Your first event listener worked. Let's see what happens if we buy multiple tickets. Re-open your `firstListener.js` in your text editor:

```
nano firstListener.js
```

At the end of the file, make another call to the `buy()` function:

event-emitters/firstListener.js

```
...

ticketManager.buy("test@email.com", 20);
ticketManager.buy("test@email.com", 20);
```

Save and exit the file. Let's run the script with Node.js to see what happens:

```
node firstListener.js
```

Your console will display this:

```
Output
Someone bought a ticket!
Someone bought a ticket!
```

Since the `buy()` function was called twice, two `buy` events were emitted. Our listener picked up both.

Sometimes we're only interested in listening to the first time an event was fired, as opposed to all the times it's emitted. Node.js provides an alternative to `on()` for this case with the `once()` function.

Like `on()`, the `once()` function accepts the event name as its first argument, and a callback function that's called when the event is fired as its second argument. Under the hood, when the event is emitted and received

by a listener that uses `once()`, Node.js automatically removes the listener and then executes the code in the callback function.

Let's see `once()` in action by editing `firstListener.js`:

```
nano firstListener.js
```

At the end of the file, add a new event listener using `once()` like the following highlighted lines:

event-emitters/firstListener.js

```
const TicketManager = require("./ticketManager");

const ticketManager = new TicketManager(10);

ticketManager.on("buy", () => {
        console.log("Someone bought a ticket!");
});

ticketManager.buy("test@email.com", 20);
ticketManager.buy("test@email.com", 20);

ticketManager.once("buy", () => {
    console.log("This is only called once");
});
```

Save and exit the file and run this program with `node`:

```
node firstListener.js
```

The output is the same as the last time:

```
Output
Someone bought a ticket!
Someone bought a ticket!
```

While we added a new event listener with `once()`, it was added after the `buy` events were emitted. Because of this, the listener didn't detect these two events. You can't listen for events that already happened in the past. When you add a listener you can only capture events that come after.

Let's add a couple more `buy()` function calls so we can confirm that the `once()` listener only reacts one time. Open `firstListener.js` in your text editor like before:

```
nano firstListener.js
```

Add the following calls at the end of the file:

event-emitters/firstListener.js

```
...

ticketManager.once("buy", () => {
    console.log("This is only called once");
});

ticketManager.buy("test@email.com", 20);
ticketManager.buy("test@email.com", 20);
```

Save and exit, then execute this program:

```
node firstListener.js
```

Your output will be:

```
Output
Someone bought a ticket!
Someone bought a ticket!
Someone bought a ticket!
This is only called once
Someone bought a ticket!
```

The first two lines were from the first two `buy()` calls before the `once()` listener was added. Adding a new event listener does not remove previous

ones, so the first event listener we added is still active and logs messages.

Since the event listener with `on()` was declared before the event listener with `once()`, we see `Someone bought a ticket!` before `This is only called once`. These two lines are both responding to the second-to-last `buy` event.

Finally, when the last call to `buy()` was made, the event emitter only had the first listener that was created with `on()`. As mentioned earlier, when an event listener created with `once()` receives an event, it is automatically removed.

Now that we have added event listeners to detect our emitters, we will see how to capture data with those listeners.

Step 3 — Capturing Event Data

So far, you've set up event listeners to react to emitted events. The emitted events also pass along data. Let's see how we can capture the data that accompanies an event.

We'll begin by creating some new Node.js modules: an email service and a database service. They'll be used to simulate sending an email and saving to a database respectively. We'll then tie them all together with our main Node.js script—`index.js`.

Let's begin by editing our email service module. Open the file in your text editor:

```
nano emailService.js
```

Our email service consists of a class that contains one method—`send()`. This method expects the email that's emitted along with `buy` events. Add

the following code to your file:

event-emitters/emailService.js

```
class EmailService {
    send(email) {
        console.log(`Sending email to ${email}`);
    }
}

module.exports = EmailService
```

This code creates an `EmailService` class that contains a `send()` function. In lieu of sending an actual email, it uses template literals to log a message to the console that would contain the email address of someone buying a ticket. Save and exit before moving on.

Let's set up the database service. Open `databaseService.js` with your text editor:

```
nano databaseService.js
```

The database service saves our purchase data to a database via its `save()` method. Add the following code to `databaseService.js`:

event-emitters/databaseService.js

```
class DatabaseService {

    save(email, price, timestamp) {
        console.log(`Running query: INSERT INTO orders VALUES
 (email, price, created) VALUES (${email}, ${price}, ${timesta
mp})`
        );

    }

}


module.exports = DatabaseService
```

This creates a new `DatabaseService` class that contains a single `save()` method. Similar to the email service's `send()` method, the `save()` function uses the data that accompanies a `buy` event, logging it to the console instead of actually inserting it into a database. This method needs the email of the purchaser, price of the ticket, and the time the ticket was purchased to function. Save and exit the file.

We will use our last file to bring the `TicketManager`, `EmailService`, and `DatabaseService` together. It will set up a listener for the `buy` event and will call the email service's `send()` function and the database service's `save()` function.

Open the `index.js` file in your text editor:

```
nano index.js
```

The first thing to do is import the modules we are using:

event-emitters/index.js

```
const TicketManager = require("./ticketManager");
const EmailService = require("./emailService");
const DatabaseService = require("./databaseService");
```

Next, let's create objects for the classes we imported. We'll set a low ticket supply of three for this demonstration:

event-emitters/index.js

```
const TicketManager = require("./ticketManager");
const EmailService = require("./emailService");
const DatabaseService = require("./databaseService");

const ticketManager = new TicketManager(3);
const emailService = new EmailService();
const databaseService = new DatabaseService();
```

We can now set up our listener with the instantiated objects. Whenever someone buys a ticket, we want to send them an email as well as saving the data to a database. Add the following listener to your code:

event-emitters/index.js

```
const TicketManager = require("./ticketManager");
const EmailService = require("./emailService");
const DatabaseService = require("./databaseService");

const ticketManager = new TicketManager(3);
const emailService = new EmailService();
const databaseService = new DatabaseService();



ticketManager.on("buy", (email, price, timestamp) => {
    emailService.send(email);
    databaseService.save(email, price, timestamp);
});
```

Like before, we add a listener with the `on()` method. The difference this time is that we have three arguments in our callback function. Each argument corresponds to the data that the event emits. As a reminder, this is the emitter code in the `buy()` function:

event-emitters/ticketManager.js

```
this.emit("buy", email, price, Date.now());
```

In our callback, we first capture the `email` from the emitter, then the `price`, and finally the `Date.now()` data, which we capture as `timestamp`.

When our listener detects a `buy` event, it will call the `send()` function from the `emailService` object as well as the `save()` function from `databaseService`. To test that this setup works, let's make a call to the `buy()` function at the end of the file:

event-emitters/index.js

```
...

ticketManager.buy("test@email.com", 10);
```

Save and exit the editor. Now let's run this script with `node` and observe what comes next. In your terminal enter:

```
node index.js
```

You will see the following output:

```
Output
Sending email to test@email.com
Running query: INSERT INTO orders VALUES (email, price, created) VALUES (test@email.com, 10, 1588720081832)
```

The data was successfully captured and returned in our callback function. With this knowledge, you can set up listeners for a variety of emitters with different event names and data. However, there are certain nuances to handling error events with event emitters.

Next, let's look at how to handle error events and what standards we should follow in doing so.

Step 4 — Handling Error Events

If an event emitter cannot perform its action, it should emit an event to signal that the action failed. In Node.js, the standard way for an event emitter to signal failure is by emitting an error event.

An error event must have its name set to `error`. It must also be accompanied by an `Error` object. Let's see a practical example of emitting an error event.

Our ticket manager decreases the supply by one every time the `buy()` function is called. Right now there's nothing stopping it from selling more tickets than it has available. Let's modify the `buy()` function so that if the ticket supply reaches `0` and someone wants to buy a ticket, we emit an error indicating that we're out of stock.

Open `ticketManager.js` in your text editor once more:

```
nano ticketManager.js
```

Now edit the `buy()` function as follows:

event-emitters/ticketManager.js

```
...


buy(email, price) {
    if (this.supply > 0) {
        this.supply--;
        this.emit("buy", email, price, Date.now());
        return;
    }

    this.emit("error", new Error("There are no more tickets le
ft to purchase"));
}
...
```

We've added an `if` statement that allows a ticket purchase if our supply is greater than zero. If we don't have any other tickets, we'll emit an `error` event. The `error` event is emitted with a new `Error` object that contains a description of why we're throwing this error.

Save and exit the file. Let's try to throw this error in our `index.js` file. Right now, we only buy one ticket. We instantiated the `ticketManager` object with three tickets, so we should get an error if we try to buy four tickets.

Edit `index.js` with your text editor:

```
nano index.js
```

Now add the following lines at the end of the file so we can buy four tickets in total:

event-emitters/index.js

```
...

ticketManager.buy("test@email.com", 10);
ticketManager.buy("test@email.com", 10);
ticketManager.buy("test@email.com", 10);
ticketManager.buy("test@email.com", 10);
```

Save and exit the editor.

Let’s execute this file to see what happens:

```
node index.js
```

Your output will be:

Output

```
Sending email to test@email.com
Running query: INSERT INTO orders VALUES (email, price, create
d) VALUES (test@email.com, 10, 1588724932796)
Sending email to test@email.com
Running query: INSERT INTO orders VALUES (email, price, create
d) VALUES (test@email.com, 10, 1588724932812)
Sending email to test@email.com
Running query: INSERT INTO orders VALUES (email, price, create
d) VALUES (test@email.com, 10, 1588724932812)
events.js:196
      throw er; // Unhandled 'error' event
      ^

Error: There are no more tickets left to purchase
    at TicketManager.buy (/home/sammy/event-emitters/ticketMan
ager.js:16:28)
    at Object.<anonymous> (/home/sammy/event-emitters/index.j
s:17:15)
    at Module._compile (internal/modules/cjs/loader.js:1128:3
0)
    at Object.Module._extensions..js (internal/modules/cjs/loa
der.js:1167:10)
    at Module.load (internal/modules/cjs/loader.js:983:32)
    at Function.Module._load (internal/modules/cjs/loader.js:8
91:14)
```

```
    at Function.executeUserEntryPoint [as runMain] (internal/m
odules/run_main.js:71:12)
    at internal/main/run_main_module.js:17:47
Emitted 'error' event on TicketManager instance at:
    at TicketManager.buy (/home/sammy/event-emitters/ticketMan
ager.js:16:14)
    at Object.<anonymous> (/home/sammy/event-emitters/index.j
s:17:15)
    [... lines matching original stack trace ...]
    at internal/main/run_main_module.js:17:47
```

The first three `buy` events were processed correctly, but on the fourth `buy` event our program crashed. Let's examine the beginning of the error message:

```
Output
...
events.js:196
      throw er; // Unhandled 'error' event
      ^

Error: There are no more tickets left to purchase
    at TicketManager.buy (/home/sammy/event-emitters/ticketMan
ager.js:16:28)
...
```

The first two lines highlight that an error was thrown. The comment says `"Unhandled 'error' event"`. If an event emitter emits an error and we did not attach a listener for error events, Node.js throws the error and crashes the program.

It's considered best practice to always listen for `error` events if you're listening to an event emitter. If you do not set up a listener for errors, your entire application will crash if one is emitted. With an error listener, you can gracefully handle it.

To follow best practices, let's set up a listener for errors. Re-open the `index.js` file:

```
nano index.js
```

Add a listener before we start buying tickets. Remember, a listener can only respond to events that are emitted after it was added. Insert an error listener like this:

event-emitters/index.js

```
...

ticketManager.on("error", (error) => {
    console.error(`Gracefully handling our error: ${error}`);
});

ticketManager.buy("test@email.com", 10);
ticketManager.buy("test@email.com", 10);
ticketManager.buy("test@email.com", 10);
ticketManager.buy("test@email.com", 10);
```

When we receive an error event, we will log it to the console with `consol e.error()`.

Save and leave `nano`. Re-run the script to see our error event handled correctly:

```
node index.js
```

This time the following output will be displayed:

```
Output
Sending email to test@email.com
Running query: INSERT INTO orders VALUES (email, price, create
d) VALUES (test@email.com, 10, 1588726293332)
Sending email to test@email.com
Running query: INSERT INTO orders VALUES (email, price, create
d) VALUES (test@email.com, 10, 1588726293348)
Sending email to test@email.com
Running query: INSERT INTO orders VALUES (email, price, create
d) VALUES (test@email.com, 10, 1588726293348)
Gracefully handling our error: Error: There are no more ticket
s left to purchase
```

From the last line, we confirm that our error event is being handled by our second listener, and the Node.js process did not crash.

Now that we've covered the concepts of sending and listening to events, let's look at some additional functionality that can be used to manage event listeners.

Step 5 — Managing Events Listeners

Event emitters come with some mechanisms to monitor and control how many listeners are subscribed to an event. To get an overview of how many listeners are processing an event, we can use the `listenerCount()` method that's included in every object.

The `listenerCount()` method accepts one argument: the event name you want the count for. Let's see it in action in `index.js`.

Open the file with `nano` or your text editor of choice:

```
nano index.js
```

You currently call the `buy()` function four times. Remove those four lines. When you do, add these two new lines so that your entire `index.js` looks like this:

event-emitters/index.js

```
const TicketManager = require("./ticketManager");
const EmailService = require("./emailService");
const DatabaseService = require("./databaseService");

const ticketManager = new TicketManager(3);
const emailService = new EmailService();
const databaseService = new DatabaseService();

ticketManager.on("buy", (email, price, timestamp) => {
    emailService.send(email);
    databaseService.save(email, price, timestamp);
});

ticketManager.on("error", (error) => {
    console.error(`Gracefully handling our error: ${error}`);
});

console.log(`We have ${ticketManager.listenerCount("buy")} lis
tener(s)for the buy event`);
console.log(`We have ${ticketManager.listenerCount("error")} l
istener(s) for the error event`);
```

We've removed the calls to `buy()` from the previous section and instead logged two lines to the console. The first log statement uses the `listenerCount()` function to display the number of listeners for the `buy()` event. The

second log statement shows how many listeners we have for the `error` event.

Save and exit. Now run your script with the `node` command:

```
node index.js
```

You'll get this output:

```
Output
We have 1 listener(s) for the buy event
We have 1 listener(s) for the error event
```

We only used the `on()` function once for the `buy` event and once for the `error` event, so this output matches our expectations.

Next, we'll use the `listenerCount()` as we remove listeners from an event emitter. We may want to remove event listeners when the period of an event no longer applies. For example, if our ticket manager was being used for a specific concert, as the concert comes to an end you would remove the event listeners.

In Node.js we use the `off()` function to remove event listeners from an event emitter. The `off()` method accepts two arguments: the event name and the function that's listening to it.

Note: Similar to the `on()` function, Node.js aliases the `off()` method with `removeListener()`. They both do the same thing, with the same arguments. In this tutorial, we will continue to use `off()`.

For the second argument of the `off()` function, we need a reference to the callback that's listening to an event. Therefore, to remove an event listener, its callback must be saved to some variable or constant. As it stands, we cannot remove the current event listeners for `buy` or `error` with the `off()` function.

To see `off()` in action, let's add a new event listener that we will remove in subsequent calls. First, let's define the callback in a variable so that we can reference it in `off()` later. Open `index.js` with `nano`:

```
nano index.js
```

At the end of `index.js` add this:

event-emitters/index.js

```
...

const onBuy = () => {
    console.log("I will be removed soon");
};
```

Now add another event listener for the `buy` event:

event-emitters/index.js

```
...


ticketManager.on("buy", onBuy);
```

To be sure that we successfully added that event listener, let's print the listener count for `buy` and call the `buy()` function.

event-emitters/index.js

```
...


console.log(`We added a new event listener bringing our total
 count for the buy event to: ${ticketManager.listenerCount("bu
y")}`);

ticketManager.buy("test@email", 20);
```

Save and exit the file, then run the program:

```
node index.js
```

The following message will be displayed in the terminal:

```
Output
We have 1 listener(s) for the buy event
We have 1 listener(s) for the error event
We added a new event listener bringing our total count for the
buy event to: 2
Sending email to test@email
Running query: INSERT INTO orders VALUES (email, price, create
d) VALUES (test@email, 20, 1588814306693)
I will be removed soon
```

From the output, we see our log statement from when we added the new event listener. We then call the `buy()` function, and both listeners react to it. The first listener sent the email and saved data to the database, and then our second listener printed its message `I will be removed soon` to the screen.

Let's now use the `off()` function to remove the second event listener. Re-open the file in `nano`:

```
nano index.js
```

Now add the following `off()` call to the end of the file. You will also add a log statement to confirm the number of listeners, and make another call to `buy()`:

event-emitters/index.js

```
...


ticketManager.off("buy", onBuy);


console.log(`We now have: ${ticketManager.listenerCount("buy")
} listener(s) for the buy event`);

ticketManager.buy("test@email", 20);
```

Note how the `onBuy` variable was used as the second argument of `off()`. Save and exit the file.

Now run this with `node`:

```
node index.js
```

The previous output will remain unchanged, but this time we will find the new log line we added confirming we have one listener once more. When `buy()` is called again, we will only see the output of the callback used by the first listener:

```
Output
We have 1 listener(s) for the buy event
We have 1 listener(s) for the error event
We added a new event listener bringing our total count for the
buy event to: 2
Sending email to test@email
Running query: INSERT INTO orders VALUES (email, price, create
d) VALUES (test@email, 20, 1588816352178)
I will be removed soon
We now have: 1 listener(s) for the buy event
Sending email to test@email
Running query: INSERT INTO orders VALUES (email, price, create
d) VALUES (test@email, 20, 1588816352178)
```

If we wanted to remove all events with `off()`, we could use the `removeAllListeners()` function. This function accepts one argument: the name of the event we want to remove listeners for.

Let's use this function at the end of the file to take off the first event listener we added for the `buy` event. Open the `index.js` file once more:

```
nano index.js
```

You'll first remove all the listeners with `removeAllListeners()`. You'll then log a statement with the listener count using the `listenerCount()` function. To confirm it's no longer listening, you'll buy another ticket. When the event is emitted, nothing will react to it.

Enter the following code at the end of the file:

event-emitters/index.js

```
...


ticketManager.removeAllListeners("buy");
console.log(`We have ${ticketManager.listenerCount("buy")} lis
teners for the buy event`);
ticketManager.buy("test@email", 20);
console.log("The last ticket was bought");
```

Save and exit the file.

Now let's execute our code with the `node` command:

```
node index.js
```

Our final output is:

```
We have 1 listener(s) for the buy event
We have 1 listener(s) for the error event
We added a new event listener bringing our total count for the
buy event to: 2
Sending email to test@email
Running query: INSERT INTO orders VALUES (email, price, create
d) VALUES (test@email, 20, 1588817058088)
I will be removed soon
We now have: 1 listener(s) for the buy event
Sending email to test@email
Running query: INSERT INTO orders VALUES (email, price, create
d) VALUES (test@email, 20, 1588817058088)
We have 0 listeners for the buy event
The last ticket was bought
```

After removing all listeners, we no longer send emails or save to the database for ticket purchases.

Conclusion

In this tutorial, you learned how to use Node.js event emitters to trigger events. You emitted events with the `emit()` function of an `EventEmitter` object, then listened to events with the `on()` and `once()` functions to execute code every time the event is triggered. You also added a listener for an error event and monitored and managed listeners with the `listenerCount()` function.

With callbacks and promises, our ticket manager system would need to be integrated with the email and database service modules to get the same functionality. Since we used event emitters, the event was decoupled from the implementations. Furthermore, any module with access to the ticket manager can observe its event and react to it. If you want Node.js modules, internal or external, to observe what's happening with your object, consider making it an event emitter for scalability.

To learn more about events in Node.js, you can read the Node.js documentation. If you'd like to continue learning Node.js, you can return to the How To Code in Node.js series, or browse programming projects and setups on our Node topic page.

How To Debug Node.js with the Built-In Debugger and Chrome DevTools

Written by Stack Abuse

The author selected the COVID-19 Relief Fund to receive a donation as part of the Write for DOnations program.

In Node.js development, tracing a coding error back to its source can save a lot of time over the course of a project. But as a program grows in complexity, it becomes harder and harder to do this efficiently. To solve this problem, developers use tools like a debugger, a program that allows developers to inspect their program as it runs. By replaying the code line-by-line and observing how it changes the program's state, debuggers can provide insight into how a program is running, making it easier to find bugs.

A common practice programmers use to track bugs in their code is to print statements as the program runs. In Node.js, that involves adding extra `console.log()` or `console.debug()` statements in their modules. While this technique can be used quickly, it is also manual, making it less scalable and more prone to errors. Using this method, it is possible to mistakenly log sensitive information to the console, which could provide malicious agents with private information about customers or your application. On the other hand, debuggers provide a systematic way to observe what's happening in a program, without exposing your program to security threats.

The key features of debuggers are watching objects and adding breakpoints. By watching objects, a debugger can help track the changes of

a variable as the programmer steps through a program. Breakpoints are markers that a programmer can place in their code to stop the code from continuing beyond points that the developer is investigating.

In this article, you will use a debugger to debug some sample Node.js applications. You will first debug code using the built-in Node.js debugger tool, setting up watchers and breakpoints so you can find the root cause of a bug. You will then use Google Chrome DevTools as a Graphical User Interface (GUI) alternative to the command line Node.js debugger.

Prerequisites

- You will need Node.js installed in your development environment. This tutorial uses version 10.19.0. To install this on macOS or Ubuntu 18.04, follow the steps in How To Install Node.js and Create a Local Development Environment on macOS or the **Installing Using a PPA** section of How To Install Node.js on Ubuntu 18.04.
- For this article we expect the user to be comfortable with basic JavaScript, especially creating and using functions. You can learn those fundamentals and more by reading our How To Code in JavaScript series.
- To use the Chrome DevTools debugger, you will need to download and install the Google Chrome web browser or the open-source Chromium web browser.

Step 1 — Using Watchers with the Node.js Debugger

Debuggers are primarily useful for two features: their ability to watch variables and observe how they change when a program is run and their

ability to stop and start code execution at different locations called breakpoints. In this step, we will run through how to watch variables to identify errors in code.

Watching variables as we step through code gives us insight into how the values of variables change as the program runs. Let's practice watching variables to help us find and fix logical errors in our code with an example.

We begin by setting up our coding environment. In your terminal, create a new folder called `debugging`:

```
mkdir debugging
```

Now enter that folder:

```
cd debugging
```

Open a new file called `badLoop.js`. We will use `nano` as it's available in the terminal:

```
nano badLoop.js
```

Our code will iterate over an array and add numbers into a total sum, which in our example will be used to add up the number of daily orders over the course of a week at a store. The program will return the sum of all the numbers in the array. In the editor, enter the following code:

debugging/badLoop.js

```
let orders = [341, 454, 198, 264, 307];

let totalOrders = 0;

for (let i = 0; i <= orders.length; i++) {
  totalOrders += orders[i];
}

console.log(totalOrders);
```

We start by creating the `orders` array, which stores five numbers. We then initialize `totalOrders` to `0`, as it will store the total of the five numbers. In the for loop, we iteratively add each value in `orders` to `totalOrders`. Finally, we print the total amount of orders at the end of the program.

Save and exit from the editor. Now run this program with `node`:

```
node badLoop.js
```

The terminal will show this output:

```
Output
NaN
```

NaN in JavaScript means **Not a Number**. Given that all the input are valid numbers, this is unexpected behavior. To find the error, let's use the Node.js debugger to see what happens to the two variables that are changed in the `for` loop: `totalOrders` and `i`.

When we want to use the built-in Node.js debugger on a program, we include `inspect` before the file name. In your terminal, run the `node` command with this debugger option as follows:

```
node inspect badLoop.js
```

When you start the debugger, you will find output like this:

```
Output
< Debugger listening on ws://127.0.0.1:9229/e1ebba25-04b8-410b
-811e-8a0c0902717a
< For help, see: https://nodejs.org/en/docs/inspector
< Debugger attached.
Break on start in badLoop.js:1
> 1 let orders = [341, 454, 198, 264, 307];
  2
  3 let totalOrders = 0;
```

The first line shows us the URL of our debug server. That's used when we want to debug with external clients, like a web browser as we'll see later on. Note that this server listens on port `:9229` of the `localhost`

(`127.0.0.1`) by default. For security reasons, it is recommended to avoid exposing this port to the public.

After the debugger is attached, the debugger outputs `Break on start in badLoop.js:1`.

Breakpoints are places in our code where we'd like execution to stop. By default, Node.js's debugger stops execution at the beginning of the file.

The debugger then shows us a snippet of code, followed by a special `debug` prompt:

```
Output
...
> 1 let orders = [341, 454, 198, 264, 307];
  2
  3 let totalOrders = 0;
debug>
```

The `>` next to `1` indicates which line we've reached in our execution, and the prompt is where we will type in our commends to the debugger. When this output appears, the debugger is ready to accept commands.

When using a debugger, we step through code by telling the debugger to go to the next line that the program will execute. Node.js allows the following commands to use a debugger:

- `c` or `cont`: Continue execution to the next breakpoint or to the end of the program.
- `n` or `next`: Move to the next line of code.

- `s` or `step`: Step into a function. By default, we only step through code in the block or scope we're debugging. By stepping into a function, we can inspect the code of the function our code calls and observe how it reacts to our data.
- `o`: Step out of a function. After stepping into a function, the debugger goes back to the main file when the function returns. We can use this command to go back to the original function we were debugging before the function has finished execution.
- `pause`: Pause the running code.

We'll be stepping through this code line-by-line. Press `n` to go to the next line:

```
n
```

Our debugger will now be stuck on the third line of code:

```
Output
break in badLoop.js:3
  1 let orders = [341, 454, 198, 264, 307];
  2
> 3 let totalOrders = 0;
  4
  5 for (let i = 0; i <= orders.length; i++) {
```

Empty lines are skipped for convenience. If we press `n` once more in the debug console, our debugger will be situated on the fifth line of code:

```
Output
break in badLoop.js:5
  3 let totalOrders = 0;
  4
> 5 for (let i = 0; i <= orders.length; i++) {
  6   totalOrders += orders[i];
  7 }
```

We are now beginning our loop. If the terminal supports color, the `0` in `let i = 0` will be highlighted. The debugger highlights the part of the code the program is about to execute, and in a `for` loop, the counter initialization is executed first. From here, we can watch to see why `totalOrders` is returning `NaN` instead of a number. In this loop, two variables are changed every iteration—`totalOrders` and `i`. Let's set up watchers for both of those variables.

We'll first add a watcher for the `totalOrders` variable. In the interactive shell, enter this:

```
watch('totalOrders')
```

To watch a variable, we use the built-in `watch()` function with a string argument that contains the variable name. As we press `ENTER` on the `watch()` function, the prompt will move to the next line without providing feedback, but the watch word will be visible when we move the debugger to the next line.

Now let's add a watcher for the variable `i`:

```
watch('i')
```

Now we can see our watchers in action. Press `n` to go to the next step. The debug console will show this:

```
Output
break in badLoop.js:5
Watchers:
  0: totalOrders = 0
  1: i = 0

  3 let totalOrders = 0;
  4
> 5 for (let i = 0; i <= orders.length; i++) {
  6   totalOrders += orders[i];
  7 }
```

The debugger now displays the values of `totalOrders` and `i` before showing the line of code, as shown in the output. These values are updated every time a line of code changes them.

At this point, the debugger is highlighting `length` in `orders.length`. This means the program is about to check the condition before it executes the code within its block. After the code is executed, the final expression `i++` will be executed. You can read more about `for` loops and their execution in our How To Construct For Loops in JavaScript guide.

Enter `n` in the console to enter the `for` loop's body:

```
Output
break in badLoop.js:6
Watchers:
  0: totalOrders = 0
  1: i = 0

  4
  5 for (let i = 0; i <= orders.length; i++) {
> 6   totalOrders += orders[i];
  7 }
  8
```

This step updates the `totalOrders` variable. Therefore, after this step is complete our variable and watcher will be updated.

Press `n` to confirm. You will see this:

```
Output
Watchers:
  0: totalOrders = 341
  1: i = 0

  3 let totalOrders = 0;
  4
> 5 for (let i = 0; i <= orders.length; i++) {
  6   totalOrders += orders[i];
  7 }
```

As highlighted, `totalOrders` now has the value of the first order: `341`.

Our debugger is just about to process the final condition of the loop. Enter `n` so we execute this line and update `i`:

```
Output
break in badLoop.js:5
Watchers:
  0: totalOrders = 341
  1: i = 1

  3 let totalOrders = 0;
  4
> 5 for (let i = 0; i <= orders.length; i++) {
  6   totalOrders += orders[i];
  7 }
```

After initialization, we had to step through the code four times to see the variables updated. Stepping through the code like this can be tedious; this problem will be addressed with breakpoints in Step 2. But for now, by setting up our watchers, we are ready to observe their values and find our problem.

Step through the program by entering `n` twelve more times, observing the output. Your console will display this:

```
Output
break in badLoop.js:5
Watchers:
  0: totalOrders = 1564
  1: i = 5

  3 let totalOrders = 0;
  4
> 5 for (let i = 0; i <= orders.length; i++) {
  6   totalOrders += orders[i];
  7 }
```

Recall that our `orders` array has five items, and `i` is now at position `5`. But since `i` is used as the index of an array, there is no value at `orders[5]`; the last value of the `orders` array is at index `4`. This means that `orders[5]` will have a value of `undefined`.

Type `n` in the console and you'll observe that the code in the loop is executed:

```
Output
break in badLoop.js:6
Watchers:
  0: totalOrders = 1564
  1: i = 5

  4
  5 for (let i = 0; i <= orders.length; i++) {
> 6   totalOrders += orders[i];
  7 }
  8
```

Typing `n` once more shows the value of `totalOrders` after that iteration:

```
Output
break in badLoop.js:5
Watchers:
  0: totalOrders = NaN
  1: i = 5

  3 let totalOrders = 0;
  4
> 5 for (let i = 0; i <= orders.length; i++) {
  6   totalOrders += orders[i];
  7 }
```

Through debugging and watching `totalOrders` and `i`, we can see that our loop is iterating six times instead of five. When `i` is `5`, `orders[5]` is added to `totalOrders`. Since `orders[5]` is `undefined`, adding this to a number will yield `NaN`. The problem with our code therefore lies within our `for` loop's condition. Instead of checking if `i` is less than or equal to the length of the `orders` array, we should only check that it's less than the length.

Let's exit our debugger, make the changes and run the code again. In the debug prompt, type the exit command and press `ENTER`:

```
.exit
```

Now that you've exited the debugger, open `badLoop.js` in your text editor:

```
nano badLoop.js
```

Change the `for` loop's condition:

debugger/badLoop.js

```
...
for (let i = 0; i < orders.length; i++) {
...
```

Save and exit `nano`. Now let's execute our script like this:

```
node badLoop.js
```

When it's complete, the correct result will be printed:

```
Output
1564
```

In this section, we used the debugger's `watch` command to find a bug in our code, fixed it, and watched it work as expected.

Now that we have some experience with the basic use of the debugger to watch variables, let's look at how we can use breakpoints so that we can debug without stepping through all the lines of code from the start of the program.

Step 2 — Using Breakpoints With the Node.js Debugger

It's common for Node.js projects to consist of many interconnected modules. Debugging each module line-by-line would be time consuming, especially as an app scales in complexity. To solve this problem, breakpoints allow us to jump to a line of code where we'd like to pause execution and inspect the program.

When debugging in Node.js, we add a breakpoint by adding the `debugger` keyword directly to our code. We can then go from one breakpoint to the next by pressing `c` in the debugger console instead of `n`. At each breakpoint, we can set up watchers for expressions of interest.

Let's see this with an example. In this step, we'll set up a program that reads a list of sentences and determines the most common word used

throughout all the text. Our sample code will return the first word with the highest number of occurrences.

For this exercise, we will create three files. The first file, `sentences.txt`, will contain the raw data that our program will process. We'll add the beginning text from Encyclopaedia Britannica's article on the Whale Shark as sample data, with the punctuation removed.

Open the file in your text editor:

```
nano sentences.txt
```

Next, enter the following code:

debugger/sentences.txt

```
Whale shark Rhincodon typus gigantic but harmless shark family Rhincodontidae that is the largest living fish
Whale sharks are found in marine environments worldwide but mainly in tropical oceans
They make up the only species of the genus Rhincodon and are classified within the order Orectolobiformes a group containing the carpet sharks
The whale shark is enormous and reportedly capable of reaching a maximum length of about 18 metres 59 feet
Most specimens that have been studied however weighed about 15 tons about 14 metric tons and averaged about 12 metres 39 feet in length
The body coloration is distinctive
Light vertical and horizontal stripes form a checkerboard pattern on a dark background and light spots mark the fins and dark areas of the body
```

Save and exit the file.

Now let's add our code to `textHelper.js`. This module will contain some handy functions we'll use to process the text file, making it easier to determine the most popular word. Open `textHelper.js` in your text editor:

```
nano textHelper.js
```

We'll create three functions to process the data in `sentences.txt`. The first will be to read the file. Type the following into `textHelper.js`:

debugger/textHelper.js

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

const readFile = () => {
  let data = fs.readFileSync('sentences.txt');
  let sentences = data.toString();
  return sentences;
};
```

First, we import the fs Node.js library so we can read files. We then create the `readFile()` function that uses `readFileSync()` to load the data from `sentences.txt` as a Buffer object and the `toString()` method to return it as a string.

The next function we'll add processes a string of text and flattens it to an array with its words. Add the following code into the editor:

textHelper.js

```
...

const getWords = (text) => {
  let allSentences = text.split('\n');
  let flatSentence = allSentences.join(' ');
  let words = flatSentence.split(' ');
  words = words.map((word) => word.trim().toLowerCase());
  return words;
};
```

In this code, we are using the methods `split()`, `join()`, and `map()` to manipulate the string into an array of individual words. The function also lowercases each word to make counting easier.

The last function needed returns the counts of different words in a string array. Add the last function like this:

debugger/textHelper.js

```
...

const countWords = (words) => {
  let map = {};
  words.forEach((word) => {
    if (word in map) {
      map[word] = 1;
    } else {
      map[word] += 1;
    }
  });

  return map;
};
```

Here we create a JavaScript object called `map` that has the words as its keys and their counts as the values. We loop through the array, adding one to a count of each word when it's the current element of the loop. Let's complete this module by exporting these functions, making them available to other modules:

debugger/textHelper.js

```
...

module.exports = { readFile, getWords, countWords };
```

Save and exit.

Our third and final file we'll use for this exercise will use the `textHelper.js` module to find the most popular word in our text. Open `index.js` with your text editor:

```
nano index.js
```

We begin our code by importing the `textHelpers.js` module:

debugger/index.js

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

Continue by creating a new array containing stop words:

debugger/index.js

```
...

const stopwords = ['i', 'me', 'my', 'myself', 'we', 'our', 'ou
rs', 'ourselves', 'you', 'your', 'yours', 'yourself', 'yoursel
ves', 'he', 'him', 'his', 'himself', 'she', 'her', 'hers', 'he
rself', 'it', 'its', 'itself', 'they', 'them', 'their', 'their
s', 'themselves', 'what', 'which', 'who', 'whom', 'this', 'tha
t', 'these', 'those', 'am', 'is', 'are', 'was', 'were', 'be',
'been', 'being', 'have', 'has', 'had', 'having', 'do', 'does',
'did', 'doing', 'a', 'an', 'the', 'and', 'but', 'if', 'or', 'b
ecause', 'as', 'until', 'while', 'of', 'at', 'by', 'for', 'wit
h', 'about', 'against', 'between', 'into', 'through', 'during'
, 'before', 'after', 'above', 'below', 'to', 'from', 'up', 'do
wn', 'in', 'out', 'on', 'off', 'over', 'under', 'again', 'furt
her', 'then', 'once', 'here', 'there', 'when', 'where', 'why',
'how', 'all', 'any', 'both', 'each', 'few', 'more', 'most', 'o
ther', 'some', 'such', 'no', 'nor', 'not', 'only', 'own', 'sam
e', 'so', 'than', 'too', 'very', 's', 't', 'can', 'will', 'jus
t', 'don', 'should', 'now', ''];
```

Stop words are commonly used words in a language that we filter out before processing a text. We can use this to find more meaningful data than the result that the most popular word in English text is `the` or `a`.

Continue by using the `textHelper.js` module functions to get a JavaScript object with words and their counts:

debugger/index.js

```
...

let sentences = textHelper.readFile();
let words = textHelper.getWords(sentences);
let wordCounts = textHelper.countWords(words);
```

We can then complete this module by determining the words with the highest frequency. To do this, we'll loop through each key of the object with the word counts and compare its count to the previously stored maximum. If the word's count is higher, it becomes the new maximum.

Add the following lines of code to compute the most popular word:

debugger/index.js

```
...

let max = -Infinity;
let mostPopular = '';

Object.entries(wordCounts).forEach(([word, count]) => {
  if (stopwords.indexOf(word) === -1) {
    if (count > max) {
      max = count;
      mostPopular = word;
    }
  }
});

console.log(`The most popular word in the text is "${mostPopul
ar}" with ${max} occurrences`);
```

In this code, we are using `Object.entries()` to transform the key-value pairs in the `wordCounts` object into individual arrays, all of which are nested within a larger array. We then use the `forEach()` method and some conditional statements to test the count of each word and store the highest number.

Save and exit the file.

Let’s now run this file to see it in action. In your terminal enter this command:

```
node index.js
```

You will see the following output:

```
Output
The most popular word in the text is "whale" with 1 occurrence
s
```

From reading the text, we can see that the answer is incorrect. A quick search in `sentences.txt` would highlight that the word `whale` appears more than once.

We have quite a few functions that can cause this error: We may not be reading the entire file, or we may not be processing the text into the array and JavaScript object correctly. Our algorithm for finding the maximum word could also be incorrect. The best way to figure out what’s wrong is to use the debugger.

Even without a large codebase, we don’t want to spend time stepping through each line of code to observe when things change. Instead, we can use breakpoints to go to those key moments before the function returns and observe the output.

Let’s add breakpoints in each function in the `textHelper.js` module. To do so, we need to add the keyword `debugger` into our code.

Open the `textHelper.js` file in the text editor. We’ll be using `nano` once again:

```
nano textHelper.js
```

First, we'll add the breakpoint to the `readFile()` function like this:

debugger/textHelper.js

```
...

const readFile = () => {
  let data = fs.readFileSync('sentences.txt');
  let sentences = data.toString();
  debugger;
  return sentences;
};

...
```

Next, we'll add another breakpoint to the `getWords()` function:

debugger/textHelper.js

```
...

const getWords = (text) => {
  let allSentences = text.split('\n');
  let flatSentence = allSentences.join(' ');
  let words = flatSentence.split(' ');
  words = words.map((word) => word.trim().toLowerCase());
  debugger;
  return words;
};

...
```

Finally, we'll add a breakpoint to the `countWords()` function:

debugger/textHelper.js

```
...

const countWords = (words) => {
  let map = {};
  words.forEach((word) => {
    if (word in map) {
      map[word] = 1;
    } else {
      map[word] += 1;
    }
  });

  debugger;
  return map;
};

...
```

Save and exit `textHelper.js`.

Let's begin the debugging process. Although the breakpoints are in `textHelpers.js`, we are debugging the main point of entry of our application: `index.js`. Start a debugging session by entering the following command in your shell:

```
node inspect index.js
```

After entering the command, we'll be greeted with this output:

Output

```
< Debugger listening on ws://127.0.0.1:9229/b2d3ce0e-3a64-4836
-bdbf-84b6083d6d30
< For help, see: https://nodejs.org/en/docs/inspector
< Debugger attached.
Break on start in index.js:1
> 1 const textHelper = require('./textHelper');
  2
  3 const stopwords = ['i', 'me', 'my', 'myself', 'we', 'our',
'ours', 'ourselves', 'you', 'your', 'yours', 'yourself', 'your
selves', 'he', 'him', 'his', 'himself', 'she', 'her', 'hers',
 'herself', 'it', 'its', 'itself', 'they', 'them', 'their', 't
heirs', 'themselves', 'what', 'which', 'who', 'whom', 'this',
 'that', 'these', 'those', 'am', 'is', 'are', 'was', 'were',
 'be', 'been', 'being', 'have', 'has', 'had', 'having', 'do',
 'does', 'did', 'doing', 'a', 'an', 'the', 'and', 'but', 'if',
'or', 'because', 'as', 'until', 'while', 'of', 'at', 'by', 'fo
r', 'with', 'about', 'against', 'between', 'into', 'through',
 'during', 'before', 'after', 'above', 'below', 'to', 'from',
 'up', 'down', 'in', 'out', 'on', 'off', 'over', 'under', 'aga
in', 'further', 'then', 'once', 'here', 'there', 'when', 'wher
e', 'why', 'how', 'all', 'any', 'both', 'each', 'few', 'more',
'most', 'other', 'some', 'such', 'no', 'nor', 'not', 'only',
 'own', 'same', 'so', 'than', 'too', 'very', 's', 't', 'can',
 'will', 'just', 'don', 'should', 'now', ''];
```

This time, enter `c` into the interactive debugger. As a reminder, `c` is short for continue. This jumps the debugger to the next breakpoint in the code. After pressing `c` and typing `ENTER`, you will see this in your console:

```
Output
break in textHelper.js:6
  4   let data = fs.readFileSync('sentences.txt');
  5   let sentences = data.toString();
> 6   debugger;
  7   return sentences;
  8 };
```

We've now saved some debugging time by going directly to our breakpoint.

In this function, we want to be sure that all the text in the file is being returned. Add a watcher for the `sentences` variable so we can see what's being returned:

```
watch('sentences')
```

Press `n` to move to the next line of code so we can observe what's in `sentences`. You will see the following output:

Output

```
break in textHelper.js:7
Watchers:
  0: sentences =
    'Whale shark Rhincodon typus gigantic but harmless shark f
amily Rhincodontidae that is the largest living fish\n' +
      'Whale sharks are found in marine environments worldwide
but mainly in tropical oceans\n' +
      'They make up the only species of the genus Rhincodon an
d are classified within the order Orectolobiformes a group con
taining the carpet sharks\n' +
      'The whale shark is enormous and reportedly capable of r
eaching a maximum length of about 18 metres 59 feet\n' +
      'Most specimens that have been studied however weighed a
bout 15 tons about 14 metric tons and averaged about 12 metres
39 feet in length\n' +
      'The body coloration is distinctive\n' +
      'Light vertical and horizontal stripes form a checkerboa
rd pattern on a dark background and light spots mark the fins
 and dark areas of the body\n'


  5   let sentences = data.toString();
  6   debugger;
> 7   return sentences;
  8 };
  9
```

It seems that we aren’t having any problems reading the file; the problem must lie elsewhere in our code. Let’s move to the next breakpoint by pressing `c` once again. When you do, you’ll see this output:

Output

```
break in textHelper.js:15
Watchers:
  0: sentences =
    ReferenceError: sentences is not defined
        at eval (eval at getWords (your_file_path/debugger/tex
tHelper.js:15:3), <anonymous>:1:1)
        at Object.getWords (your_file_path/debugger/textHelpe
r.js:15:3)
        at Object.<anonymous> (your_file_path/debugger/index.j
s:7:24)
        at Module._compile (internal/modules/cjs/loader.js:112
5:14)
        at Object.Module._extensions..js (internal/modules/cj
s/loader.js:1167:10)
        at Module.load (internal/modules/cjs/loader.js:983:32)
        at Function.Module._load (internal/modules/cjs/loader.
js:891:14)
        at Function.executeUserEntryPoint [as runMain] (intern
al/modules/run_main.js:71:12)
        at internal/main/run_main_module.js:17:47

 13   let words = flatSentence.split(' ');
 14   words = words.map((word) => word.trim().toLowerCase());
>15   debugger;
```

```
16   return words;
17 };
```

We get this error message because we set up a watcher for the `sentences` variable, but that variable does not exist in our current function scope. A watcher lasts for the entire debugging session, so as long as we keep watching `sentences` where it's not defined, we'll continue to see this error.

We can stop watching variables with the `unwatch()` command. Let's unwatch `sentences` so we no longer have to see this error message every time the debugger prints its output. In the interactive prompt, enter this command:

```
unwatch('sentences')
```

The debugger does not output anything when you unwatch a variable.

Back in the `getWords()` function, we want to be sure that we are returning a list of words that are taken from the text we loaded earlier. Let's watch the value of the `words` variable:

```
watch('words')
```

Then enter `n` to go to the next line of the debugger, so we can see what's being stored in `words`. The debugger will show the following:

Output

```
break in textHelper.js:16
Watchers:
  0: words =
    [ 'whale',
      'shark',
      'rhincodon',
      'typus',
      'gigantic',
      'but',
      'harmless',
      ...
      'metres',
      '39',
      'feet',
      'in',
      'length',
      '',
      'the',
      'body',
      'coloration',
      ... ]

 14   words = words.map((word) => word.trim().toLowerCase());
 15   debugger;
>16   return words;
```

```
 17 };
 18
```

The debugger does not print out the entire array as it's quite long and would make the output harder to read. However, the output meets our expectations of what should be stored: the text from `sentences` split into lowercase strings. It seems that `getWords()` is functioning correctly.

Let's move on to observe the `countWords()` function. First, unwatch the `words` array so we don't cause any debugger errors when we are at the next breakpoint. In the command prompt, enter this:

```
unwatch('words')
```

Next, enter `c` in the prompt. At our last breakpoint, we will see this in the shell:

```
Output
break in textHelper.js:29
 27   });
 28
>29   debugger;
 30   return map;
 31 };
```

In this function, we want to be sure that the `map` variable correctly contains the count of each word from our sentences. First, let's tell the

debugger to watch the `map` variable:

```
watch('map')
```

Press `n` to move to the next line. The debugger will then display this:

```
Output
break in textHelper.js:30
Watchers:
  0: map =
    { 12: NaN,
      14: NaN,
      15: NaN,
      18: NaN,
      39: NaN,
      59: NaN,
      whale: 1,
      shark: 1,
      rhincodon: 1,
      typus: NaN,
      gigantic: NaN,
      ... }


 28
 29   debugger;
>30   return map;
 31 };
 32
```

That does not look correct. It seems as though the method for counting words is producing erroneous results. We don’t know why those values are being entered, so our next step is to debug what’s happening in the loop

used on the `words` array. To do this, we need to make some changes to where we place our breakpoint.

First, exit the debug console:

```
.exit
```

Open `textHelper.js` in your text editor so we can edit the breakpoints:

```
nano textHelper.js
```

First, knowing that `readFile()` and `getWords()` are working, we will remove their breakpoints. We then want to remove the breakpoint in `countWords()` from the end of the function, and add two new breakpoints to the beginning and end of the `forEach()` block.

Edit `textHelper.js` so it looks like this:

debugger/textHelper.js

```
...

const readFile = () => {
  let data = fs.readFileSync('sentences.txt');
  let sentences = data.toString();
  return sentences;
};

const getWords = (text) => {
  let allSentences = text.split('\n');
  let flatSentence = allSentences.join(' ');
  let words = flatSentence.split(' ');
  words = words.map((word) => word.trim().toLowerCase());
  return words;
};

const countWords = (words) => {
  let map = {};
  words.forEach((word) => {
    debugger;
    if (word in map) {
      map[word] = 1;
    } else {
      map[word] += 1;
    }
```

```
    debugger;
  });

  return map;
};

...
```

Save and exit `nano` with `CTRL+X`.

Let's start the debugger again with this command:

```
node inspect index.js
```

To get insight into what's happening, we want to debug a few things in the loop. First, let's set up a watcher for `word`, the argument used in the `forEach()` loop containing the string that the loop is currently looking at. In the debug prompt, enter this:

```
watch('word')
```

So far, we have only watched variables. But watches are not limited to variables. We can watch any valid JavaScript expression that's used in our code.

In practical terms, we can add a watcher for the condition `word in map`, which determines how we count numbers. In the debug prompt, create this watcher:

```
watch('word in map')
```

Let's also add a watcher for the value that's being modified in the `map` variable:

```
watch('map[word]')
```

Watchers can even be expressions that aren't used in our code but could be evaluated with the code we have. Let's see how this works by adding a watcher for the length of the `word` variable:

```
watch('word.length')
```

Now that we've set up all our watchers, let's enter `c` into the debugger prompt so we can see how the first element in the loop of `countWords()` is evaluated. The debugger will print this output:

```
Output
break in textHelper.js:20
Watchers:
  0: word = 'whale'
  1: word in map = false
  2: map[word] = undefined
  3: word.length = 5

 18   let map = {};
 19   words.forEach((word) => {
>20     debugger;
 21     if (word in map) {
 22       map[word] = 1;
```

The first word in the loop is `whale`. At this point, the `map` object has no key with `whale` as its empty. Following from that, when looking up `whale` in `map`, we get `undefined`. Lastly, the length of `whale` is `5`. That does not help us debug the problem, but it does validate that we can watch any expression that could be evaluated with the code while debugging.

Press `c` once more to see what's changed by the end of the loop. The debugger will show this:

```
Output
break in textHelper.js:26
Watchers:
  0: word = 'whale'
  1: word in map = true
  2: map[word] = NaN
  3: word.length = 5

 24       map[word] += 1;
 25     }
>26     debugger;
 27   });
 28
```

At the end of the loop, `word in map` is now true as the `map` variable contains a `whale` key. The value of `map` for the `whale` key is `NaN`, which highlights our problem. The `if` statement in `countWords()` is meant to set a word's count to one if it's new, and add one if it existed already.

The culprit is the `if` statement's condition. We should set `map[word]` to `1` if the `word` is not found in `map`. Right now, we are adding one if `word` is found. At the beginning of the loop, `map["whale"]` is `undefined`. In JavaScript, `undefined + 1` evaluates to `NaN`—not a number.

The fix for this would be to change the condition of the `if` statement from `(word in map)` to `(!(word in map))`, using the `!` operator to test if `word` is not in `map`. Let's make that change in the `countWords()` function to see what happens.

First, exit the debugger:

```
.exit
```

Now open the `textHelper.js` file with your text editor:

```
nano textHelper.js
```

Modify the `countWords()` function as follows:

debugger/textHelper.js

```
...

const countWords = (words) => {
  let map = {};
  words.forEach((word) => {
    if (!(word in map)) {
      map[word] = 1;
    } else {
      map[word] += 1;
    }
  });

  return map;
};

...
```

Save and close the editor.

Now let's run this file without a debugger. In the terminal, enter this:

```
node index.js
```

The script will output the following sentence:

```
Output
The most popular word in the text is "whale" with 3 occurrence
s
```

This output seems a lot more likely than what we received before. With the debugger, we figured out which function caused the problem and which functions did not.

We've debugged two different Node.js programs with the built-in CLI debugger. We are now able to set up breakpoints with the `debugger` keyword and create various watchers to observe changes in internal state. But sometimes, code can be more effectively debugged from a GUI application.

In the next section, we'll use the debugger in Google Chrome's DevTools. We'll start the debugger in Node.js, navigate to a dedicated debugging page in Google Chrome, and set up breakpoints and watchers using the GUI.

Step 3 — Debugging Node.js with Chrome DevTools

Chrome DevTools is a popular choice for debugging Node.js in a web browser. As Node.js uses the same V8 JavaScript engine that Chrome uses, the debugging experience is more integrated than with other debuggers.

For this exercise, we'll create a new Node.js application that runs an HTTP server and returns a JSON response. We'll then use the debugger to set up breakpoints and gain deeper insight into what response is being generated for the request.

Let’s create a new file called `server.js` that will store our server code. Open the file in the text editor:

```
nano server.js
```

This application will return a JSON with a `Hello World` greeting. It will have an array of messages in different languages. When a request is received, it will randomly pick a greeting and return it in a JSON body.

This application will run on our `localhost` server on port `:8000`. If you’d like to learn more about creating HTTP servers with Node.js, read our guide on How To Create a Web Server in Node.js with the HTTP Module.

Type the following code into the text editor:

debugger/server.js

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


const host = 'localhost';

const port = 8000;


const greetings = ["Hello world", "Hola mundo", "Bonjour le mo
nde", "Hallo Welt", "Salve mundi"];


const getGreeting = function () {
  let greeting = greetings[Math.floor(Math.random() * greeting
s.length)];

  return greeting

}
```

We begin by importing the `http` module, which is needed to create an HTTP server. We then set up the `host` and `port` variables that we will use later to run the server. The `greetings` array contains all the possible greetings our server can return. The `getGreeting()` function randomly selects a greeting and returns it.

Let's add the request listener that processes HTTP requests and add code to run our server. Continue editing the Node.js module by typing the following:

debugger/server.js

```
...

const requestListener = function (req, res) {
  let message = getGreeting();
  res.setHeader("Content-Type", "application/json");
  res.writeHead(200);
  res.end(`{"message": "${message}"}`);
};

const server = http.createServer(requestListener);
server.listen(port, host, () => {
  console.log(`Server is running on http://${host}:${port}`);
});
```

Our server is now ready for use, so let's set up the Chrome debugger. We can start the Chrome debugger with the following command:

```
node --inspect server.js
```

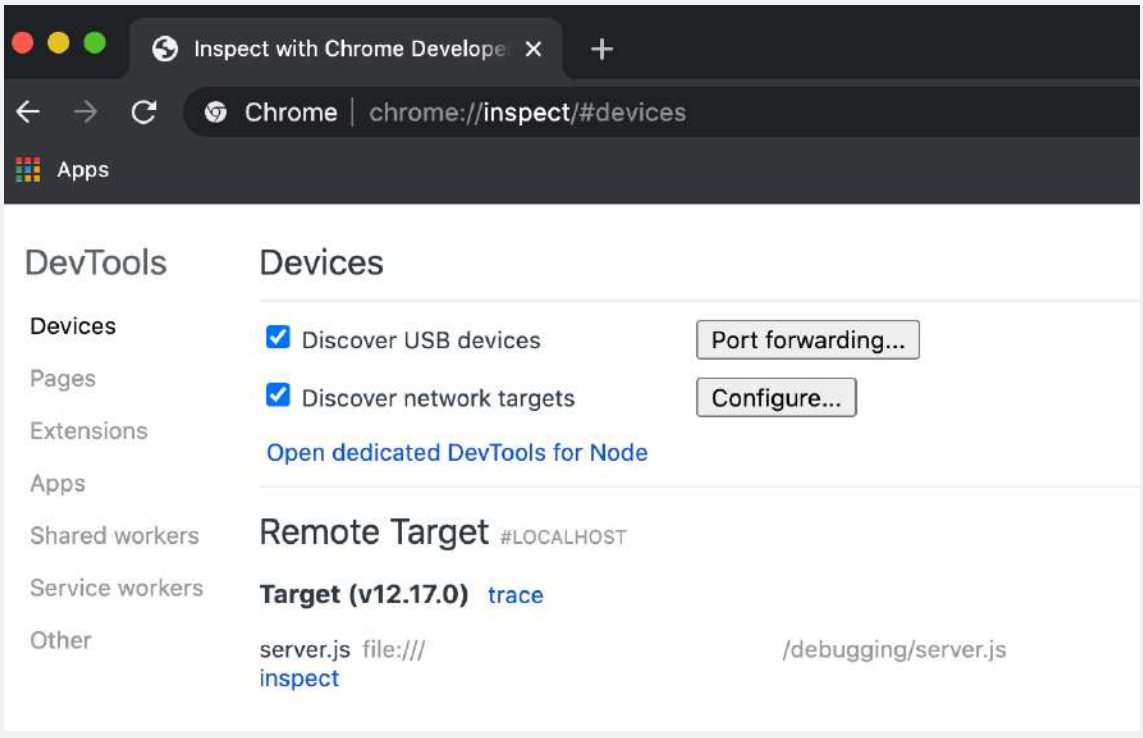
Note: Keep in mind the difference between the CLI debugger and the Chrome debugger commands. When using the CLI you use `inspect`. When using Chrome you use `--inspect`.

After starting the debugger, you'll find the following output:

```
Output
Debugger listening on ws://127.0.0.1:9229/996cfbaf-78ca-4ebd-9
fd5-893888efe8b3
For help, see: https://nodejs.org/en/docs/inspector
Server is running on http://localhost:8000
```

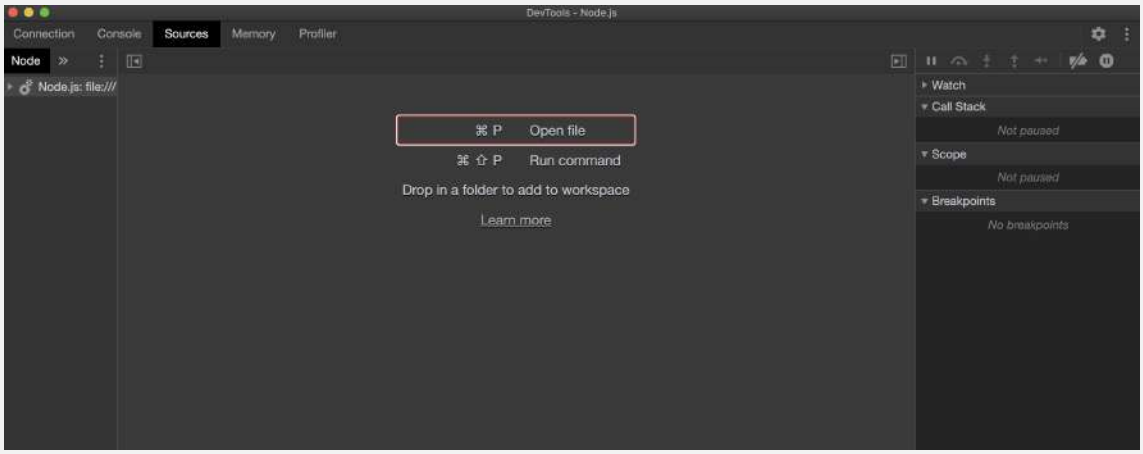
Now open Google Chrome or Chromium and enter `chrome://inspect` in the address bar. Microsoft Edge also uses the V8 JavaScript engine, and can thus use the same debugger. If you are using Microsoft Edge, navigate to `edge://inspect`.

After navigating to the URL, you will see the following page:



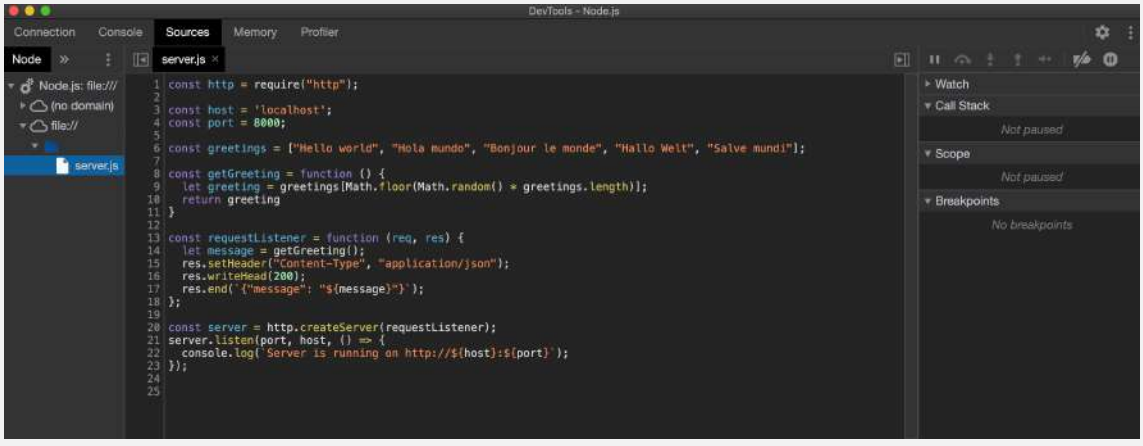
Screenshot of Google Chome's "inspect" page

Under the **Devices** header, click the **Open dedicated DevTools for Node** button. A new window will pop up:



Screenshot of debug window

We're now able to debug our Node.js code with Chrome. Navigate to the **Sources** tab if not already there. On the left-hand side, expand the file tree and select `server.js`:



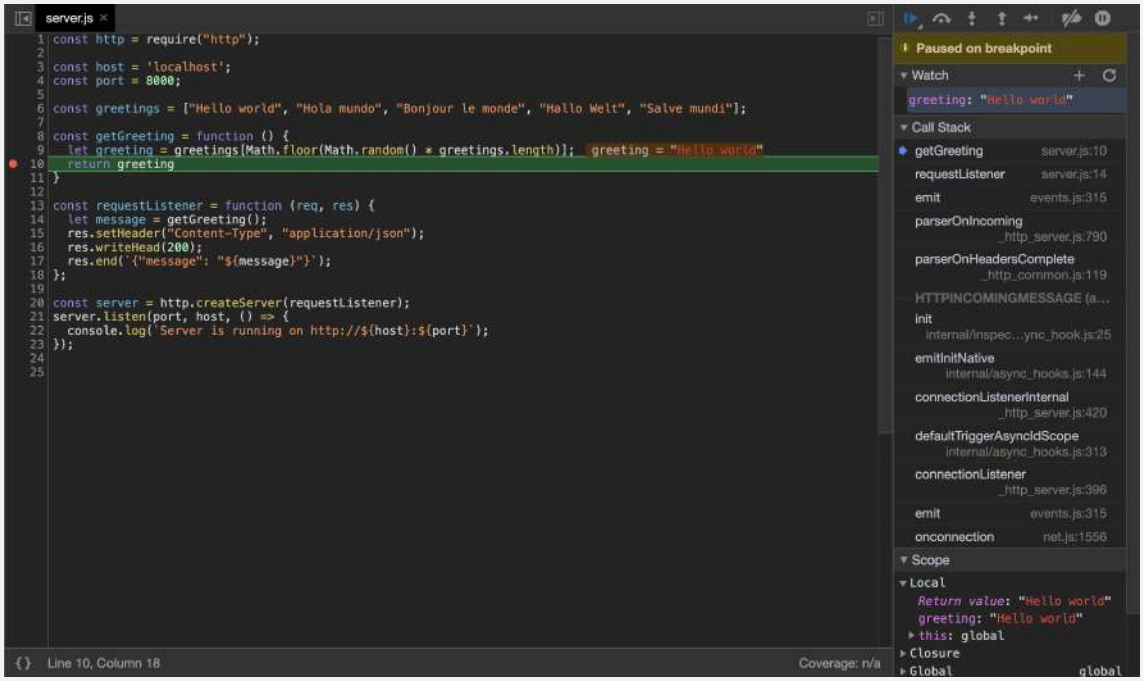
Screenshot of debugger window's Sources tab

Let’s add a breakpoint to our code. We want to stop when the server has selected a greeting and is about to return it. Click on the line number **10** in the debug console. A red dot will appear next to the number and the right-hand panel will indicate a new breakpoint was added:

Screenshot of adding a breakpoint in the Chrome debugger

Now let’s add a watch expression. On the right panel, click the arrow next to the **Watch** header to open the watch words list, then click +. Enter `greeting` and press `ENTER` so that we can observe its value when processing a request.

Next, let’s debug our code. Open a new browser window and navigate to `http://localhost:8000`—the address the Node.js server is running on. When pressing `ENTER`, we will not immediately get a response. Instead, the debug window will pop up once again. If it does not immediately come into focus, navigate to the debug window to see this:



Screenshot of the program's execution paused in Chrome

The debugger pauses the server's response where we set our breakpoint. The variables that we watch are updated in the right panel and also in the line of code that created it.

Let's complete the response's execution by pressing the continue button at the right panel, right above **Paused on breakpoint**. When the response is complete, you will see a successful JSON response in the browser window used to speak with the Node.js server:

```
{"message": "Hello world"}
```

In this way, Chrome DevTools does not require changes to the code to add breakpoints. If you prefer to use graphical applications over the

command line to debug, the Chrome DevTools are more suitable for you.

Conclusion

In this article, we debugged sample Node.js applications by setting up watchers to observe the state of our application, and then by adding breakpoints to allow us to pause execution at various points in our program's execution. We accomplished this using both the built-in CLI debugger and Google Chrome's DevTools.

Many Node.js developers log to the console to debug their code. While this is useful, it's not as flexible as being able to pause execution and watch various state changes. Because of this, using debugging tools is often more efficient, and will save time over the course of developing a project.

To learn more about these debugging tools, you can read the Node.js documentation or the Chrome DevTools documentation. If you'd like to continue learning Node.js, you can return to the How To Code in Node.js series, or browse programming projects and setups on our Node topic page.

How To Launch Child Processes in Node.js

Written by Stack Abuse

The author selected the COVID-19 Relief Fund to receive a donation as part of the Write for DOnations program.

When a user executes a single Node.js program, it runs as a single operating system (OS) process that represents the instance of the program running. Within that process, Node.js executes programs on a single thread. As mentioned earlier in this series with the How To Write Asynchronous Code in Node.js tutorial, because only one thread can run on one process, operations that take a long time to execute in JavaScript can block the Node.js thread and delay the execution of other code. A key strategy to work around this problem is to launch a child process, or a process created by another process, when faced with long-running tasks. When a new process is launched, the operating system can employ multiprocessing techniques to ensure that the main Node.js process and the additional child process run concurrently, or at the same time.

Node.js includes the `child_process` module, which has functions to create new processes. Aside from dealing with long-running tasks, this module can also interface with the OS and run shell commands. System administrators can use Node.js to run shell commands to structure and maintain their operations as a Node.js module instead of shell scripts.

In this tutorial, you will create child processes while executing a series of sample Node.js applications. You'll create processes with the `child_proces`

`s` module by retrieving the results of a child process via a buffer or string with the `exec()` function, and then from a data stream with the `spawn()` function. You'll finish by using `fork()` to create a child process of another Node.js program that you can communicate with as it's running. To illustrate these concepts, you will write a program to list the contents of a directory, a program to find files, and a web server with multiple endpoints.

Prerequisites

- You must have Node.js installed to run through these examples. This tutorial uses version 10.22.0. To install this on macOS or Ubuntu 18.04, follow the steps in How To Install Node.js and Create a Local Development Environment on macOS or the **Installing Using a PPA** section of How To Install Node.js on Ubuntu 18.04.
- This article uses an example that creates a web server to explain how the `fork()` function works. To get familiar with creating web servers, you can read our guide on How To Create a Web Server in Node.js with the HTTP Module.

Step 1 — Creating a Child Process with `exec()`

Developers commonly create child processes to execute commands on their operating system when they need to manipulate the output of their Node.js programs with a shell, such as using shell piping or redirection. The `exec()` function in Node.js creates a new shell process and executes a command in that shell. The output of the command is kept in a buffer in memory, which you can accept via a callback function passed into `exec()`.

Let’s begin creating our first child processes in Node.js. First, we need to set up our coding environment to store the scripts we’ll create throughout this tutorial. In the terminal, create a folder called `child-processes`:

```
mkdir child-processes
```

Enter that folder in the terminal with the `cd` command:

```
cd child-processes
```

Create a new file called `listFiles.js` and open the file in a text editor. In this tutorial we will use nano, a terminal text editor:

```
nano listFiles.js
```

We’ll be writing a Node.js module that uses the `exec()` function to run the `ls` command. The `ls` command list the files and folders in a directory. This program takes the output from the `ls` command and displays it to the user.

In the text editor, add the following code:

~/child-processes/listFiles.js

```
const { exec } = require('child_process');

exec('ls -lh', (error, stdout, stderr) => {
  if (error) {
    console.error(`error: ${error.message}`);
    return;
  }

  if (stderr) {
    console.error(`stderr: ${stderr}`);
    return;
  }

  console.log(`stdout:\n${stdout}`);
});
```

We first import the `exec()` command from the `child_process` module using JavaScript destructuring. Once imported, we use the `exec()` function. The first argument is the command we would like to run. In this case, it's `ls -lh`, which lists all the files and folders in the current directory in long format, with a total file size in human-readable units at the top of the output.

The second argument is a callback function with three parameters: `error`, `stdout`, and `stderr`. If the command failed to run, `error` will

capture the reason why it failed. This can happen if the shell cannot find the command you're trying to execute. If the command is executed successfully, any data it writes to the standard output stream is captured in `stdout`, and any data it writes to the standard error stream is captured in `stderr`.

Note: It's important to keep the difference between `error` and `stderr` in mind. If the command itself fails to run, `error` will capture the error. If the command runs but returns output to the error stream, `stderr` will capture it. The most resilient Node.js programs will handle all possible outputs for a child process.

In our callback function, we first check if we received an error. If we did, we display the error's `message` (a property of the `Error` object) with `console.error()` and end the function with `return`. We then check if the command printed an error message and `return` if so. If the command successfully executes, we log its output to the console with `console.log()`.

Let's run this file to see it in action. First, save and exit `nano` by pressing `CTRL+X`.

Back in your terminal, run your application with the `node` command:

```
node listFiles.js
```

Your terminal will display the following output:

```
Output
stdout:
total 4.0K
-rw-rw-r-- 1 sammy sammy 280 Jul 27 16:35 listFiles.js
```

This lists the contents of the `child-processes` directory in long format, along with the size of the contents at the top. Your results will have your own user and group in place of `sammy`. This shows that the `listFiles.js` program successfully ran the shell command `ls -lh`.

Now let's look at another way to execute concurrent processes. Node.js's `child_process` module can also run executable files with the `execFile()` function. The key difference between the `execFile()` and `exec()` functions is that the first argument of `execFile()` is now a path to an executable file instead of a command. The output of the executable file is stored in a buffer like `exec()`, which we access via a callback function with `error`, `stdout`, and `stderr` parameters.

Note: Scripts in Windows such as `.bat` and `.cmd` files cannot be run with `execFile()` because the function does not create a shell when running the file. On Unix, Linux, and macOS, executable scripts do not always need a shell to run. However, a Windows machines needs a shell to execute scripts. To execute script files on Windows, use `exec()`, since it creates a new shell. Alternatively, you can use `spawn()`, which you'll use later in this Step.

However, note that you can execute `.exe` files in Windows successfully using `execFile()`. This limitation only applies to script files that require a

shell to execute.

Let's begin by adding an executable script for `execFile()` to run. We'll write a bash script that downloads the Node.js logo from the Node.js website and Base64 encodes it to convert its data to a string of ASCII characters.

Create a new shell script file called `processNodejsImage.sh`:

```
nano processNodejsImage.sh
```

Now write a script to download the image and base64 convert it:

~/child-processes/processNodejsImage.sh

```
#!/bin/bash
curl -s https://nodejs.org/static/images/logos/nodejs-new-pantone-black.svg > nodejs-logo.svg

base64 nodejs-logo.svg
```

The first statement is a shebang statement. It's used in Unix, Linux, and macOS when we want to specify a shell to execute our script. The second statement is a `curl` command. The cURL utility, whose command is `curl`, is a command-line tool that can transfer data to and from a server. We use cURL to download the Node.js logo from the website, and then we use redirection to save the downloaded data to a new file `nodejs-logo.svg`. The last statement uses the `base64` utility to encode the `nodejs-logo.svg` file we downloaded with cURL. The script then outputs the encoded string to the console.

Save and exit before continuing.

In order for our Node program to run the bash script, we have to make it executable. To do this, run the following:

```
chmod u+x processNodejsImage.sh
```

This will give your current user the permission to execute the file.

With our script in place, we can write a new Node.js module to execute it. This script will use `execFile()` to run the script in a child process, catching any errors and displaying all output to console.

In your terminal, make a new JavaScript file called `getNodejsImage.js`:

```
nano getNodejsImage.js
```

Type the following code in the text editor:

~/child-processes/getNodejsImage.js

```
const { execFile } = require('child_process');


execFile(__dirname + '/processNodejsImage.sh', (error, stdout,
stderr) => {
  if (error) {
    console.error(`error: ${error.message}`);
    return;
  }


  if (stderr) {
    console.error(`stderr: ${stderr}`);
    return;
  }


  console.log(`stdout:\n${stdout}`);
});
```

We use JavaScript destructuring to import the `execFile()` function from the `child_process` module. We then use that function, passing the file path as the first name. `__dirname` contains the directory path of the module in which it is written. Node.js provides the `__dirname` variable to a module when the module runs. By using `__dirname`, our script will always find the `processNodejsImage.sh` file across different operating systems, no matter

where we run `getNodejsImage.js`. Note that for our current project setup, `getNodejsImage.js` and `processNodejsImage.sh` must be in the same folder.

The second argument is a callback with the `error`, `stdout`, and `stderr` parameters. Like with our previous example that used `exec()`, we check for each possible output of the script file and log them to the console.

In your text editor, save this file and exit from the editor.

In your terminal, use `node` to execute the module:

```
node getNodejsImage.js
```

Running this script will produce output like this:

```
Output
stdout:
PHN2ZyB4bWxucz0iaHR0cDovL3d3dy53My5vcmcvMjAwMC9zdmciIHhtbG5zOn
hsaW5rPSJodHRwOi8vd3d3LnczLm9yZy8xOTk5L3hsaW5rIiB2aWV3Qm94PSIw
IDAgNDQyLjQgMjcwLjkiPjxkZWZzPjxsaW5lYXJHcmFkaWVudCBpZD0iYiIgeD
E9IjE4MC43IiB5MT0iODAuNyIge
...
```

Note that we truncated the output in this article because of its large size.

Before base64 encoding the image, `processNodejsImage.sh` first downloads it. You can also verify that you downloaded the image by inspecting the current directory.

Execute `listFiles.js` to find the updated list of files in our directory:

```
node listFiles.js
```

The script will display content similar to the following on the terminal:

```
Output
stdout:
total 20K
-rw-rw-r-- 1 sammy sammy  316 Jul 27 17:56 getNodejsImage.js
-rw-rw-r-- 1 sammy sammy  280 Jul 27 16:35 listFiles.js
-rw-rw-r-- 1 sammy sammy 5.4K Jul 27 18:01 nodejs-logo.svg
-rwxrw-r-- 1 sammy sammy  129 Jul 27 17:56 processNodejsImage.
sh
```

We've now successfully executed `processNodejsImage.sh` as a child process in Node.js using the `execFile()` function.

The `exec()` and `execFile()` functions can run commands on the operating system's shell in a Node.js child process. Node.js also provides another method with similar functionality, `spawn()`. The difference is that instead of getting the output of the shell commands all at once, we get them in chunks via a stream. In the next section we'll use the `spawn()` command to create a child process.

Step 2 — Creating a Child Process with `spawn()`

The `spawn()` function runs a command in a process. This function returns data via the stream API. Therefore, to get the output of the child process, we need to listen for stream events.

Streams in Node.js are instances of event emitters. If you would like to learn more about listening for events and the foundations of interacting with streams, you can read our guide on Using Event Emitters in Node.js.

It's often a good idea to choose `spawn()` over `exec()` or `execFile()` when the command you want to run can output a large amount of data. With a buffer, as used by `exec()` and `execFile()`, all the processed data is stored in the computer's memory. For large amounts of data, this can degrade system performance. With a stream, the data is processed and transferred in small chunks. Therefore, you can process a large amount of data without using too much memory at any one time.

Let's see how we can use `spawn()` to make a child process. We will write a new Node.js module that creates a child process to run the `find` command. We will use the `find` command to list all the files in the current directory.

Create a new file called `findFiles.js`:

```
nano findFiles.js
```

In your text editor, begin by calling the `spawn()` command:

~/child-processes/findFiles.js

```
const { spawn } = require('child_process');

const child = spawn('find', ['.']);
```

We first imported the `spawn()` function from the `child_process` module. We then called the `spawn()` function to create a child process that executes the `find` command. We hold the reference to the process in the `child` variable, which we will use to listen to its streamed events.

The first argument in `spawn()` is the command to run, in this case `find`. The second argument is an array that contains the arguments for the executed command. In this case, we are telling Node.js to execute the `find` command with the argument `.`, thereby making the command find all the files in the current directory. The equivalent command in the terminal is `find .`.

With the `exec()` and `execFile()` functions, we wrote the arguments along with the command in one string. However, with `spawn()`, all arguments to commands must be entered in the array. That's because `spawn()`, unlike `exec()` and `execFile()`, does not create a new shell before running a process. To have commands with their arguments in one string, you need Node.js to create a new shell as well.

Let's continue our module by adding listeners for the command's output. Add the following highlighted lines:

~/child-processes/findFiles.js

```
const { spawn } = require('child_process');

const child = spawn('find', ['.']);

child.stdout.on('data', data => {
  console.log(`stdout:\n${data}`);
});

child.stderr.on('data', data => {
  console.error(`stderr: ${data}`);
});
```

Commands can return data in either the `stdout` stream or the `stderr` stream, so you added listeners for both. You can add listeners by calling the `on()` method of each streams' objects. The `data` event from the streams gives us the command's output to that stream. Whenever we get data on either stream, we log it to the console.

We then listen to two other events: the `error` event if the command fails to execute or is interrupted, and the `close` event for when the command has finished execution, thus closing the stream.

In the text editor, complete the Node.js module by writing the following highlighted lines:

~/child-processes/findFiles.js

```
const { spawn } = require('child_process');

const child = spawn('find', ['.']);

child.stdout.on('data', (data) => {
  console.log(`stdout:\n${data}`);
});

child.stderr.on('data', (data) => {
  console.error(`stderr: ${data}`);
});

child.on('error', (error) => {
  console.error(`error: ${error.message}`);
});

child.on('close', (code) => {
  console.log(`child process exited with code ${code}`);
});
```

For the `error` and `close` events, you set up a listener directly on the `child` variable. When listening for `error` events, if one occurs Node.js provides an `Error` object. In this case, you log the error's `message` property.

When listening to the `close` event, Node.js provides the exit code of the command. An exit code denotes if the command ran successfully or not. When a command runs without errors, it returns the lowest possible value for an exit code: `0`. When executed with an error, it returns a non-zero code.

The module is complete. Save and exit `nano` with `CTRL+X`.

Now, run the code with the `node` command:

```
node findFiles.js
```

Once complete, you will find the following output:

```
Output
stdout:
.
./findFiles.js
./listFiles.js
./nodejs-logo.svg
./processNodejsImage.sh
./getNodejsImage.js


child process exited with code 0
```

We find a list of all files in our current directory and the exit code of the command, which is `0` as it ran successfully. While our current directory has a small number of files, if we ran this code in our home directory, our program would list every single file in every accessible folder for our user.

Because it has such a potentially large output, using the `spawn()` function is most ideal as its streams do not require as much memory as a large buffer.

So far we've used functions to create child processes to execute external commands in our operating system. Node.js also provides a way to create a child process that executes other Node.js programs. Let's use the `fork()` function to create a child process for a Node.js module in the next section.

Step 3 — Creating a Child Process with `fork()`

Node.js provides the `fork()` function, a variation of `spawn()`, to create a child process that's also a Node.js process. The main benefit of using `fork()` to create a Node.js process over `spawn()` or `exec()` is that `fork()` enables communication between the parent and the child process.

With `fork()`, in addition to retrieving data from the child process, a parent process can send messages to the running child process. Likewise, the child process can send messages to the parent process.

Let's see an example where using `fork()` to create a new Node.js child process can improve the performance of our application. Node.js programs run on a single process. Therefore, CPU intensive tasks like iterating over large loops or parsing large JSON files stop other JavaScript code from running. For certain applications, this is not a viable option. If a web server is blocked, then it cannot process any new incoming requests until the blocking code has completed its execution.

Let's see this in practice by creating a web server with two endpoints. One endpoint will do a slow computation that blocks the Node.js process. The other endpoint will return a JSON object saying `hello`.

First, create a new file called `httpServer.js`, which will have the code for our HTTP server:

```
nano httpServer.js
```

We'll begin by setting up the HTTP server. This involves importing the `http` module, creating a request listener function, creating a server object, and listening for requests on the server object. If you would like to dive deeper into creating HTTP servers in Node.js or would like a refresher, you can read our guide on How To Create a Web Server in Node.js with the HTTP Module.

Enter the following code in your text editor to set up an HTTP server:

~/child-processes/httpServer.js

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

const host = 'localhost';
const port = 8000;

const requestListener = function (req, res) {};

const server = http.createServer(requestListener);
server.listen(port, host, () => {
  console.log(`Server is running on http://${host}:${port}`);
});
```

This code sets up an HTTP server that will run at `http://localhost:8000`. It uses template literals to dynamically generate that URL.

Next, we will write an intentionally slow function that counts in a loop 5 billion times. Before the `requestListener()` function, add the following code:

~/child-processes/httpServer.js

```
...
const port = 8000;

const slowFunction = () => {
  let counter = 0;
  while (counter < 5000000000) {
    counter++;
  }

  return counter;
}

const requestListener = function (req, res) {};
...
```

This uses the arrow function syntax to create a `while` loop that counts to `5000000000`.

To complete this module, we need to add code to the `requestListener()` function. Our function will call the `slowFunction()` on subpath, and return a small JSON message for the other. Add the following code to the module:

~/child-processes/httpServer.js

```
...
const requestListener = function (req, res) {
  if (req.url === '/total') {
    let slowResult = slowFunction();
    let message = `{"totalCount":${slowResult}}`;

    console.log('Returning /total results');
    res.setHeader('Content-Type', 'application/json');
    res.writeHead(200);
    res.end(message);
  } else if (req.url === '/hello') {
    console.log('Returning /hello results');
    res.setHeader('Content-Type', 'application/json');
    res.writeHead(200);
    res.end(`{"message":"hello"}`);
  }
};
...
```

If the user reaches the server at the `/total` subpath, then we run `slowFunction()`. If we are hit at the `/hello` subpath, we return this JSON message: `{"message":"hello"}`.

Save and exit the file by pressing `CTRL+X`.

To test, run this server module with `node`:

```
node httpServer.js
```

When our server starts, the console will display the following:

```
Output
Server is running on http://localhost:8000
```

Now, to test the performance of our module, open two additional terminals. In the first terminal, use the `curl` command to make a request to the `/total` endpoint, which we expect to be slow:

```
curl http://localhost:8000/total
```

In the other terminal, use `curl` to make a request to the `/hello` endpoint like this:

```
curl http://localhost:8000/hello
```

The first request will return the following JSON:

```
Output
{"totalCount":5000000000}
```

Whereas the second request will return this JSON:

```
Output
{"message":"hello"}
```

The request to `/hello` completed only after the request to `/total`. The `slowFunction()` blocked all other code from executing while it was still in its loop. You can verify this by looking at the Node.js server output that was logged in your original terminal:

```
Output
Returning /total results
Returning /hello results
```

To process the blocking code while still accepting incoming requests, we can move the blocking code to a child process with `fork()`. We will move the blocking code into its own module. The Node.js server will then create a child process when someone accesses the `/total` endpoint and listen for results from this child process.

Refactor the server by first creating a new module called `getCount.js` that will contain `slowFunction()`:

```
nano getCount.js
```

Now enter the code for `slowFunction()` once again:

~/child-processes/getCount.js

```
const slowFunction = () => {
  let counter = 0;
  while (counter < 5000000000) {
    counter++;
  }

  return counter;
}
```

Since this module will be a child process created with `fork()`, we can also add code to communicate with the parent process when `slowFunction()` has completed processing. Add the following block of code that sends a message to the parent process with the JSON to return to the user:

~/child-processes/getCount.js

```
const slowFunction = () => {
  let counter = 0;
  while (counter < 5000000000) {
    counter++;
  }

  return counter;
}

process.on('message', (message) => {
  if (message == 'START') {
    console.log('Child process received START message');
    let slowResult = slowFunction();
    let message = `{"totalCount":${slowResult}}`;
    process.send(message);
  }
});
```

Let's break down this block of code. The messages between a parent and child process created by `fork()` are accessible via the Node.js global process object. We add a listener to the `process` variable to look for `message` events. Once we receive a `message` event, we check if it's the `START` event. Our server code will send the `START` event when someone accesses the `/total` endpoint. Upon receiving that event, we run `slowFunction()` and

create a JSON string with the result of the function. We use `process.send()` to send a message to the parent process.

Save and exit `getCount.js` by entering `CTRL+X` in nano.

Now, let's modify the `httpServer.js` file so that instead of calling `slowFunction()`, it creates a child process that executes `getCount.js`.

Re-open `httpServer.js` with `nano`:

```
nano httpServer.js
```

First, import the `fork()` function from the `child_process` module:

~/child-processes/httpServer.js

```
const http = require('http');
const { fork } = require('child_process');
...
```

Next, we are going to remove the `slowFunction()` from this module and modify the `requestListener()` function to create a child process. Change the code in your file so it looks like this:

~/child-processes/httpServer.js

```
...
const port = 8000;

const requestListener = function (req, res) {
  if (req.url === '/total') {
    const child = fork(__dirname + '/getCount');

    child.on('message', (message) => {
      console.log('Returning /total results');
      res.setHeader('Content-Type', 'application/json');
      res.writeHead(200);
      res.end(message);
    });

    child.send('START');
  } else if (req.url === '/hello') {
    console.log('Returning /hello results');
    res.setHeader('Content-Type', 'application/json');
    res.writeHead(200);
    res.end(`{"message":"hello"}`);
  }
};
...
```

When someone goes to the `/total` endpoint, we now create a new child process with `fork()`. The argument of `fork()` is the path to the Node.js module. In this case, it is the `getCount.js` file in our current directory, which we receive from `__dirname`. The reference to this child process is stored in a variable `child`.

We then add a listener to the `child` object. This listener captures any messages that the child process gives us. In this case, `getCount.js` will return a JSON string with the total number counted by the `while` loop. When we receive that message, we send the JSON to the user.

We use the `send()` function of the `child` variable to give it a message. This program sends the message `START`, which begins the execution of `slowFunction()` in the child process.

Save and exit `nano` by entering `CTRL+X`.

To test the improvement using `fork()` made on HTTP server, begin by executing the `httpServer.js` file with `node`:

```
node httpServer.js
```

Like before, it will output the following message when it launches:

```
Output
Server is running on http://localhost:8000
```

To test the server, we will need an additional two terminals as we did the first time. You can re-use them if they are still open.

In the first terminal, use the `curl` command to make a request to the `/total` endpoint, which takes a while to compute:

```
curl http://localhost:8000/total
```

In the other terminal, use `curl` to make a request to the `/hello` endpoint, which responds in a short time:

```
curl http://localhost:8000/hello
```

The first request will return the following JSON:

```
Output
{"totalCount":5000000000}
```

Whereas the second request will return this JSON:

```
Output
{"message":"hello"}
```

Unlike the first time we tried this, the second request to `/hello` runs immediately. You can confirm by reviewing the logs, which will look like this:

```
Output
Child process received START message
Returning /hello results
Returning /total results
```

These logs show that the request for the `/hello` endpoint ran after the child process was created but before the child process had finished its task.

Since we moved the blocking code in a child process using `fork()`, the server was still able to respond to other requests and execute other JavaScript code. Because of the `fork()` function's message passing ability, we can control when a child process begins an activity and we can return data from a child process to a parent process.

Conclusion

In this article, you used various functions to create a child process in Node.js. You first created child processes with `exec()` to run shell commands from Node.js code. You then ran an executable file with the `execFile()` function. You looked at the `spawn()` function, which can also run commands but returns data via a stream and does not start a shell like `exec()` and `execFile()`. Finally, you used the `fork()` function to allow for two-way communication between the parent and child process.

To learn more about the `child_process` module, you can read the Node.js documentation. If you'd like to continue learning Node.js, you can return to the How To Code in Node.js series, or browse programming projects and setups on our Node topic page.

How To Work with Files using the fs Module in Node.js

Written by Stack Abuse

The author selected the COVID-19 Relief Fund to receive a donation as part of the Write for DOnations program.

Working with files is as common for development purposes as it is for non-development purposes. In daily computer use, a user would likely read and write data to files in various directories in order to accomplish tasks like saving a downloaded file or accessing data to be used in another application. In the same way, a back-end program or command line interface (CLI) tool might need to write downloaded data to a file in order to save it, or a data-intensive application may need to export to JSON, CSV, or Excel formats. These programs would need to communicate with the filesystem of the operating system on which they are running.

With Node.js, you can programmatically manipulate files with the built-in `fs` module. The name is short for "file system," and the module contains all the functions you need to read, write, and delete files on the local machine. This unique aspect of Node.js makes JavaScript a useful language for back-end and CLI tool programming.

In this article, you will use the `fs` module to read a file created via the command line, create and write to a new file, delete the file that you created, and move the first file into a different folder. The `fs` module supports interacting with files synchronously, asynchronously, or via

streams; this tutorial will focus on how to use the asynchronous, Promise-based API, the most commonly used method for Node.js developers.

Prerequisites

- You must have Node.js installed on your computer to access the `fs` module and follow the tutorial. This tutorial uses Node.js version 10.22.0. To install Node.js on macOS or Ubuntu 18.04, follow the steps in How To Install Node.js and Create a Local Development Environment on macOS or the **Installing Using a PPA** section of How To Install Node.js on Ubuntu 18.04.
- This article uses JavaScript Promises to work with files, particularly with the `async/await` syntax. If you're not familiar with Promises, `async/await` syntax, or asynchronous programming, check out our guide on How To Write Asynchronous Code in Node.js.

Step 1 — Reading Files with `readFile()`

In this step, you'll write a program to read files in Node.js. To do this, you'll need to import the `fs` module, a standard Node.js module for working with files, and then use the module's `readFile()` function. Your program will read the file, store its contents in a variable, then log its contents to the console.

The first step will be to set up the coding environment for this activity and the ones in the later sections.

Create a folder to store your code. In your terminal, make a folder called `node-files`:

```
mkdir node-files
```

Change your working directory to the newly created folder with the `cd` command:

```
cd node-files
```

In this folder, you'll create two files. The first file will be a new file with content that your program will read later. The second file will be the Node.js module that reads the file.

Create the file `greetings.txt` with the following command:

```
echo "hello, hola, bonjour, hallo" > greetings.txt
```

The `echo` command prints its string argument to the terminal. You use `>` to redirect `echo`'s output to a new file, `greetings.txt`.

Now, create and open `readFile.js` in your text editor of choice. This tutorial uses `nano`, a terminal text editor. You can open this file with `nano` like this:

```
nano readFile.js
```

The code for this file can be broken up into three sections. First, you need to import the Node.js module that allows your program to work with files. In your text editor, type this code:

node-files/readFile.js

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

As mentioned earlier, you use the `fs` module to interact with the filesystem. Notice, though, that you are importing the `.promises` part of the module.

When the `fs` module was first created, the primary way to write asynchronous code in Node.js was through callbacks. As promises grew in popularity, the Node.js team worked to support them in the `fs` module out of the box. In Node.js version 10, they created a `promises` object in the `fs` module that uses promises, while the main `fs` module continues to expose functions that use callbacks. In this program, you are importing the promise version of the module.

Once the module is imported, you can create an asynchronous function to read the file. Asynchronous functions begin with the `async` keyword. With an asynchronous function, you can resolve promises using the `await` keyword, instead of chaining the promise with the `.then()` method.

Create a new function `readFile()` that accepts one argument, a string called `filePath`. Your `readFile()` function will use the `fs` module to load the file into a variable using `async/await` syntax.

Enter the following highlighted code:

node-files/readFile.js

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


async function readFile(filePath) {
  try {
    const data = await fs.readFile(filePath);
    console.log(data.toString());
  } catch (error) {
    console.error(`Got an error trying to read the file: ${error.message}`);
  }
}
```

You define the function with the `async` keyword so you can later use the accompanying `await` keyword. To capture errors in your asynchronous file reading operation, you enclose the call to `fs.readFile()` with a `try...catch` block. Within the `try` section, you load a file to a `data` variable with the `fs.readFile()` function. The only required argument for that function is the file path, which is given as a string.

The `fs.readFile()` returns a `buffer` object by default. A `buffer` object can store any kind of file type. When you log the contents of the file, you convert those bytes into text by using the `toString()` method of the buffer object.

If an error is caught, typically if the file is not found or the program does not have permission to read the file, you log the error you received in the

console.

Finally, call the function on the `greetings.txt` file with the following highlighted line:

node-files/readFile.js

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


async function readFile(filePath) {
  try {
    const data = await fs.readFile(filePath);
    console.log(data.toString());
  } catch (error) {
    console.error(`Got an error trying to read the file: ${err
or.message}`);
  }
}


readFile('greetings.txt');
```

Be sure to save your contents. With `nano`, you can save and exit by pressing `CTRL+X`.

Your program will now read the `greetings.txt` file you created earlier and log its contents to the terminal. Confirm this by executing your module with `node`:

```
node readFile.js
```

You will receive the following output:

```
Output
hello, hola, bonjour, hallo
```

You've now read a file with the `fs` module's `readFile()` function using the `async/await` syntax.

Note: In some earlier versions of Node.js, you will receive the following warning when using the `fs` module:

```
(node:13085) ExperimentalWarning: The fs.promises API is exper
imental
```

The `promises` object of the `fs` module was introduced in Node.js version 10, so some earlier versions still call the module experimental. This warning was removed when the API became stable in version 12.6.

Now that you've read a file with the `fs` module, you will next create a file and write text to it.

Step 2 — Writing Files with `writeFile()`

In this step, you will write files with the `writeFile()` function of the `fs` module. You will create a CSV file in Node.js that keeps track of a grocery bill. The first time you write the file, you will create the file and add the headers. The second time, you will append data to the file.

Open a new file in your text editor:

```
nano writeFile.js
```

Begin your code by importing the `fs` module:

node-files/writeFile.js

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

You will continue to use `async/await` syntax as you create two functions. The first function will be to make the CSV file. The second function will be to add data to the CSV file.

In your text editor, enter the following highlighted code:

node-files/writeFile.js

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


async function openFile() {
  try {
    const csvHeaders = 'name,quantity,price'
    await fs.writeFile('groceries.csv', csvHeaders);
  } catch (error) {
    console.error(`Got an error trying to write to a file: ${error.message}`);
  }
}
```

This asynchronous function first creates a `csvHeaders` variable that contains the column headings of your CSV file. You then use the `writeFile()` function of the `fs` module to create a file and write data to it. The first argument is the file path. As you provided just the file name, Node.js will create the file in the same directory that you're executing the code in. The second argument is the data you are writing, in this case the `csvHeaders` variable.

Next, create a new function to add items to your grocery list. Add the following highlighted function in your text editor:

node-files/writeFile.js

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


async function openFile() {
  try {
    const csvHeaders = 'name,quantity,price'
    await fs.writeFile('groceries.csv', csvHeaders);
  } catch (error) {
    console.error(`Got an error trying to write to a file: ${e
rror.message}`);
  }
}


async function addGroceryItem(name, quantity, price) {
  try {
    const csvLine = `\n${name},${quantity},${price}`
    await fs.writeFile('groceries.csv', csvLine, { flag: 'a'
 });
  } catch (error) {
    console.error(`Got an error trying to write to a file: ${e
rror.message}`);
  }
}
```

The asynchronous `addGroceryItem()` function accepts three arguments: the name of the grocery item, the amount you are buying, and the price per

unit. These arguments are used with template literal syntax to form the `csvLine` variable, which is the data you are writing to the file.

You then use the `writeFile()` method as you did in the `openFile()` function. However, this time you have a third argument: a JavaScript object. This object has a `flag` key with the value `a`. Flags tell Node.js how to interact with the file on the system. By using the flag `a`, you are telling Node.js to append to the file, not overwrite it. If you don't specify a flag, it defaults to `w`, which creates a new file if none exists or overwrites a file if it already exists. You can learn more about filesystem flags in the Node.js documentation.

To complete your script, use these functions. Add the following highlighted lines at the end of the file:

node-files/writeFile.js

```
...
async function addGroceryItem(name, quantity, price) {
  try {
    const csvLine = `\n${name},${quantity},${price}`
    await fs.writeFile('groceries.csv', csvLine, { flag: 'a'
 });
  } catch (error) {
    console.error(`Got an error trying to write to a file: ${e
rror.message}`);
  }
}

(async function () {
  await openFile();
  await addGroceryItem('eggs', 12, 1.50);
  await addGroceryItem('nutella', 1, 4);
})();
```

To call the functions, you first create a wrapper function with `async function`. Since the `await` keyword can not be used from the global scope as of the writing of this tutorial, you must wrap the asynchronous functions in an `async function`. Notice that this function is anonymous, meaning it has no name to identify it.

Your `openFile()` and `addGroceryItem()` functions are asynchronous functions. Without enclosing these calls in another function, you cannot

guarantee the order of the content. The wrapper you created is defined with the `async` keyword. Within that function you order the function calls using the `await` keyword.

Finally, the `async function` definition is enclosed in parentheses. These tell JavaScript that the code inside them is a function expression. The parentheses at the end of the function and before the semicolon are used to invoke the function immediately. This is called an Immediately-Invoked Function Expression (IIFE). By using an IIFE with an anonymous function, you can test that your code produces a CSV file with three lines: the column headers, a line for `eggs`, and the last line for `nutella`.

Save and exit `nano` with `CTRL+X`.

Now, run your code with the `node` command:

```
node writeFile.js
```

There will be no output. However, a new file will exist in your current directory.

Use the `cat` command to display the contents of `groceries.csv`:

```
cat groceries.csv
```

You will receive the following output:

node-files/groceries.csv

```
name,quantity,price
eggs,12,1.5
nutella,1,4
```

Your call to `openFile()` created a new file and added the column headings for your CSV. The subsequent calls to `addGroceryItem()` then added your two lines of data.

With the `writeFile()` function, you can create and edit files. Next, you will delete files, a common operation when you have temporary files or need to make space on a hard drive.

Step 3 — Deleting Files with `unlink()`

In this step, you will delete files with the `unlink()` function in the `fs` module. You will write a Node.js script to delete the `groceries.csv` file that you created in the last section.

In your terminal, create a new file for this Node.js module:

```
nano deleteFile.js
```

Now you will write code that creates an asynchronous `deleteFile()` function. That function will accept a file path as an argument, passing it to the `unlink()` function to remove it from your filesystem.

In your text editor, write the following code:

node-files/deleteFile.js

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


async function deleteFile(filePath) {
  try {
    await fs.unlink(filePath);
    console.log(`Deleted ${filePath}`);
  } catch (error) {
    console.error(`Got an error trying to delete the file: ${error.message}`);
  }
}


deleteFile('groceries.csv');
```

The `unlink()` function accepts one argument: the file path of the file you want to be deleted.

Warning: When you delete the file with the `unlink()` function, it is not sent to your recycle bin or trash can but permanently removed from your filesystem. This action is not reversible, so please be certain that you want to remove the file before executing your code.

Exit `nano`, ensuring that you save the contents of the file by entering `CTRL+X`.

Now, execute the program. Run the following command in your terminal:

```
node deleteFile.js
```

You will receive the following output:

```
Output
Deleted groceries.csv
```

To confirm that the file no longer exists, use the `ls` command in your current directory:

```
ls
```

This command will display these files:

```
Output
deleteFile.js   greetings.txt   readFile.js     writeFile.js
```

You've now confirmed that your file was deleted with the `unlink()` function.

So far you've learned how to read, write, edit, and delete files. The following section uses a function to move files to different folders. After learning that function, you will be able to do the most critical file management tasks in Node.js.

Step 4 — Moving Files with `rename()`

Folders are used to organize files, so being able to programmatically move files from one folder to another makes file management easier. You can

move files in Node.js with the `rename()` function. In this step, you'll move a copy of the `greetings.txt` file into a new folder.

Before you can code your Node.js module, you need to set a few things up. Begin by creating a folder that you'll be moving your file into. In your terminal, create a `test-data` folder in your current directory:

```
mkdir test-data
```

Now, copy the `greetings.txt` file that was used in the first step using the `cp` command:

```
cp greetings.txt greetings-2.txt
```

Finish the setup by opening a JavaScript file to contain your code:

```
nano moveFile.js
```

In your Node.js module, you'll create a function called `moveFile()` that calls the `rename()` function. When using the `rename()` function, you need to provide the file path of the original file and the path of the destination location. For this example, you'll use a `moveFile()` function to move the `greetings-2.txt` file into the `test-data` folder. You'll also change its name to `salutations.txt`.

Enter the following code in your open text editor:

node-files/moveFile.js

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


async function moveFile(source, destination) {
  try {
    await fs.rename(source, destination);
    console.log(`Moved file from ${source} to ${destination}`)
;
  } catch (error) {
    console.error(`Got an error trying to move the file: ${err
or.message}`);
  }
}


moveFile('greetings-2.txt', 'test-data/salutations.txt');
```

As mentioned earlier, the `rename()` function takes two arguments: the source and destination file paths. This function can move files to other folders, rename a file in its current directory, or move and rename at the same time. In your code, you are moving and renaming your file.

Save and exit `nano` by pressing `CTRL+X`.

Next, execute this program with `node`. Enter this command to run the program:

```
node moveFile.js
```

You will receive this output:

```
Output
Moved file from greetings-2.txt to test-data/salutations.txt
```

To confirm that the file no longer exists in your current directory, you can use the `ls` command:

```
ls
```

This command will display these files and folder:

```
Output
deleteFile.js   greetings.txt   moveFile.js     readFile.js
test-data       writeFile.js
```

You can now use `ls` to list the files in the `test-data` subfolder:

```
ls test-data
```

Your moved file will appear in the output:

```
Output
salutations.txt
```

You have now used the `rename()` function to move a file from your current directory into a subfolder. You also renamed the file with the same function call.

Conclusion

In this article, you learned various functions to manage files with Node.js. You first loaded the contents of a file with `readFile()`. You then created new files and appended data to an existing file with the `writeFile()` function. You permanently removed a file with the `unlink()` function, and then move and renamed a file with `rename()`.

Working with files programmatically is an important function of Node.js. Programs might need to output files for a user to use, or may need to store data for an application that is not always running. With the `fs` module's functions, developers have control of how files are used in our Node.js programs.

To learn more about the `fs` module, you can read the Node.js documentation. If you'd like to continue learning Node.js, you can return to the How To Code in Node.js series, or browse programming projects and setups on our Node topic page.

How To Create an HTTP Client with Core HTTP in Node.js

Written by Stack Abuse

The author selected the COVID-19 Relief Fund to receive a donation as part of the Write for DOnations program.

It's common for a modern web application to communicate with other servers to accomplish a task. For example, a web app that allows you to purchase a book online may involve communication between a customer orders server, a book inventory server, and a payment server. In this design, the different services communicate via web APIs—standard formats that allow you to programmatically send and receive data. In a Node.js app, you can communicate with web APIs by making HTTP requests.

Node.js comes bundled with an `http` and an `https` module. These modules have functions to create an HTTP server so that a Node.js program can respond to HTTP requests. They can also make HTTP requests to other servers. This key functionality equips Node.js programmers to create modern, API-driven web applications with Node.js. As it's a core module, you do not need to install any libraries to use it.

In this tutorial, you will use the `https` module to make HTTP requests to JSON Placeholder, a fake REST API for testing purposes. You will begin by making a `GET` request, the standard HTTP request to receive data. You will then look at ways to customize your request, such as by adding headers. Finally, you will make `POST`, `PUT`, and `DELETE` requests so that you can modify data in an external server.

Prerequisites

- This tutorial requires that you have Node.js installed. Once installed, you will be able to access the `https` module that's used throughout the tutorial. This tutorial uses Node.js version 10.19.0. To install Node.js on macOS or Ubuntu 18.04, follow the steps in How To Install Node.js and Create a Local Development Environment on macOS or the **Installing Using a PPA** section of How To Install Node.js on Ubuntu 18.04.
- The methods used to send HTTP requests have a Stream-based API. In Node.js, streams are instances of event emitters. The way in which you respond to data coming from a stream is the same as the way in which you respond to data from events. If you are curious, you can get more in-depth knowledge of event emitters by reading our Using Event Emitters in Node.js guide.

Step 1 — Making a `GET` Request

When you interact with an API, you typically make `GET` requests to retrieve data from web servers. In this step, you'll look at two functions to make `GET` requests in Node.js. Your code will retrieve a JSON array of user profiles from a publicly accessible API.

The `https` module has two functions to make `GET` requests—the `get()` function, which can only make `GET` requests, and the `request()` function, which makes other types of requests. You will begin by making a request with the `get()` function.

Making Requests with `get()`

HTTP requests using the `get()` function have this format:

```
https.get(URL_String, Callback_Function) {
    Action
}
```

The first argument is a string with the endpoint you're making the request to. The second argument is a callback function, which you use to handle the response.

First, set up your coding environment. In your terminal, create a folder to store all your Node.js modules for this guide:

```
mkdir requests
```

Enter that folder:

```
cd requests
```

Create and open a new file in a text editor. This tutorial will use `nano` as it's available in the terminal:

```
nano getRequestWithGet.js
```

To make HTTP requests in Node.js, import the `https` module by adding the follow line:

requests/getRequestWithGet.js

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

Note:: Node.js has an `http` and an `https` module. They have the same functions and behave in a similar manner, but `https` makes the requests through the Transport Layer Security (TLS/SSL). As the web servers you are using are available via HTTPS, you will use the `https` module. If you are making requests to and from URLs that only have HTTP, then you would use the `http` module.

Now use the `http` object to make a `GET` request to the API to retrieve a list of users. You will use JSON Placeholder, a publicly available API for testing. This API does not keep a record of any changes you make in your requests. It simulates a real server, and returns mocked responses as long as you send a valid request.

Write the following highlighted code in your text editor:

requests/getRequestWithGet.js

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


let request = https.get('https://jsonplaceholder.typicode.
com/users?_limit=2', (res) => { });
```

As mentioned in the function signature, the `get()` function takes two parameters. The first is the API URL you are making the request to in string format and the second is a callback to handle the HTTP response. To read the data from your response, you have to add some code in the callback.

HTTP responses come with a status code. A status code is a number that indicates how successful the response was. Status codes between 200 and 299 are positive responses, while codes between 400 and 599 are errors. You can learn more about status codes in our How To Troubleshoot Common HTTP Error Codes guide.

For this request, a successful response would have a 200 status code. The first thing you'll do in your callback will be to verify that the status code is what you expect. Add the following code to the callback function:

requests/getRequestWithGet.js

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


let request = https.get('https://jsonplaceholder.typicode.com/
users?_limit=2', (res) => {

  if (res.statusCode !== 200) {

    console.error(`Did not get an OK from the server. Code: ${
res.statusCode}`);

    res.resume();

    return;

  }

});
```

The response object that's available in the callback has a `statusCode` property that stores the status code. If the status code is not 200, you log an error to the console and exit.

Note the line that has `res.resume()`. You included that line to improve performance. When making HTTP requests, Node.js will consume all the data that's sent with the request. The `res.resume()` method tells Node.js to ignore the stream's data. In turn, Node.js would typically discard the data more quickly than if it left it for garbage collection—a periodic process that frees an application's memory.

Now that you've captured error responses, add code to read the data. Node.js responses stream their data in chunks. The strategy for retrieving data will be to listen for when data comes from the response, collate all the chunks, and then parse the JSON so your application can use it.

Modify the request callback to include this code:

requests/getRequestWithGet.js

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


let request = https.get('https://jsonplaceholder.typicode.com/
users?_limit=2', (res) => {

  if (res.statusCode !== 200) {

    console.error(`Did not get an OK from the server. Code: ${
res.statusCode}`);

    res.resume();

    return;

  }


  let data = '';


  res.on('data', (chunk) => {

    data += chunk;

  });


  res.on('close', () => {

    console.log('Retrieved all data');

    console.log(JSON.parse(data));

  });

});
```

You begin by creating a new variable `data` that's an empty string. You can store data as an array of numbers representing byte data or a string. This

tutorial uses the latter as it's easier to convert a JSON string to a JavaScript object.

After creating the `data` variable, you create an event listener. Node.js streams the data of an HTTP response in chunks. Therefore, when the response object emits a `data` event, you will take the data it received and add it to your `data` variable.

When all the data from the server is received, Node.js emits a `close` event. At this point, you parse the JSON string stored in `data` and log the result to the console.

Your Node.js module can now communicate with the JSON API and log the list of users, which will be a JSON array of three users. However, there's one small improvement you can make first.

This script will throw an error if you are unable to make a request. You may not be able to make a request if you lose your internet connection, for example. Add the following code to capture errors when you're unable to send an HTTP request:

requests/getRequestWithGet.js

```
...
  res.on('data', (chunk) => {
    data += chunk;
  });

  res.on('close', () => {
    console.log('Retrieved all data');
    console.log(JSON.parse(data));
  });

});

request.on('error', (err) => {
  console.error(`Encountered an error trying to make a reques
t: ${err.message}`);
});
```

When a request is made but cannot be sent, the request object emits an `error` event. If an `error` event is emitted but not listened to, the Node.js program crashes. Therefore, to capture errors you add an event listener with the `on()` function and listen for `error` events. When you get an error, you log its message.

That's all the code for this file. Save and exit `nano` by pressing `CTRL+X`.

Now execute this program with `node`:

```
node getRequestWithGet.js
```

Your console will display this response:

Output

```
Retrieved all data
[
  {
    id: 1,
    name: 'Leanne Graham',
    username: 'Bret',
    email: 'Sincere@april.biz',
    address: {
      street: 'Kulas Light',
      suite: 'Apt. 556',
      city: 'Gwenborough',
      zipcode: '92998-3874',
      geo: [Object]
    },
    phone: '1-770-736-8031 x56442',
    website: 'hildegard.org',
    company: {
      name: 'Romaguera-Crona',
      catchPhrase: 'Multi-layered client-server neural-net',
      bs: 'harness real-time e-markets'
    }
  },
  {
    id: 2,
    name: 'Ervin Howell',
```

```
    username: 'Antonette',
    email: 'Shanna@melissa.tv',
    address: {
      street: 'Victor Plains',
      suite: 'Suite 879',
      city: 'Wisokyburgh',
      zipcode: '90566-7771',
      geo: [Object]
    },
    phone: '010-692-6593 x09125',
    website: 'anastasia.net',
    company: {
      name: 'Deckow-Crist',
      catchPhrase: 'Proactive didactic contingency',
      bs: 'synergize scalable supply-chains'
    }
  }
]
```

This means you've successfully made a `GET` request with the core Node.js library.

The `get()` method you used is a convenient method Node.js provides because `GET` requests are a very common type of request. Node.js provides a `request()` method to make a request of any type. Next, this tutorial will examine how to make a `GET` request with `request()`.

Making Requests with `request()`

The `request()` method supports multiple function signatures. You'll use this one for the subsequent example:

```
https.request(URL_String, Options_Object, Callback_Function) {
    Action
}
```

The first argument is a string with the API endpoint. The second argument is a JavaScript object containing all the options for the request. The last argument is a callback function to handle the response.

Create a new file for a new module called `getRequestWithRequest.js`:

```
nano getRequestWithRequest.js
```

The code you will write is similar to the `getRequestWithGet.js` module you wrote earlier. First, import the `https` module:

requests/getRequestWithRequest.js

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

Next, create a new JavaScript object that contains a `method` key:

requests/getRequestWithRequest.js

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


const options = {
  method: 'GET'
};
```

The `method` key in this object will tell the `request()` function what HTTP method the request is using.

Next, make the request in your code. The following codeblock highlights code that was different from the request made with the `get()` method. In your editor, enter all of the following lines:

requests/getRequestWithRequest.js

```
...

let request = https.request('https://jsonplaceholder.typicode.
com/users?_limit=2', options, (res) => {
  if (res.statusCode !== 200) {
    console.error(`Did not get an OK from the server. Code: ${
res.statusCode}`);
    res.resume();
    return;
  }

  let data = '';

  res.on('data', (chunk) => {
    data += chunk;
  });

  res.on('close', () => {
    console.log('Retrieved all data');
    console.log(JSON.parse(data));
  });
});

request.end();

request.on('error', (err) => {
```

```
  console.error(`Encountered an error trying to make a reques
t: ${err.message}`);

});
```

To make a request using `request()`, you provide the URL in the first argument, an object with the HTTP options in the second argument, and a callback to handle the response in the third argument.

The `options` variable you created earlier is the second argument, telling Node.js that this is a `GET` request. The callback is unchanged from when you first wrote it.

You also call the `end()` method of the `request` variable. This is an important method that must be called when using the `request()` function. It completes the request, allowing it to be sent. If you don't call it, the program will never complete, as Node.js will think you still have data to add to the request.

Save and exit `nano` with `CTRL+X`, or the equivalent with your text editor.

Run this program in your terminal:

```
node getRequestWithRequest.js
```

You will receive this output, which is the same as the first module:

Output

```
Retrieved all data
[
  {
    id: 1,
    name: 'Leanne Graham',
    username: 'Bret',
    email: 'Sincere@april.biz',
    address: {
      street: 'Kulas Light',
      suite: 'Apt. 556',
      city: 'Gwenborough',
      zipcode: '92998-3874',
      geo: [Object]
    },
    phone: '1-770-736-8031 x56442',
    website: 'hildegard.org',
    company: {
      name: 'Romaguera-Crona',
      catchPhrase: 'Multi-layered client-server neural-net',
      bs: 'harness real-time e-markets'
    }
  },
  {
    id: 2,
    name: 'Ervin Howell',
```

```
    username: 'Antonette',
    email: 'Shanna@melissa.tv',
    address: {
      street: 'Victor Plains',
      suite: 'Suite 879',
      city: 'Wisokyburgh',
      zipcode: '90566-7771',
      geo: [Object]
    },
    phone: '010-692-6593 x09125',
    website: 'anastasia.net',
    company: {
      name: 'Deckow-Crist',
      catchPhrase: 'Proactive didactic contingency',
      bs: 'synergize scalable supply-chains'
    }
  }
]
```

You have now used the `request()` method to make a `GET` request. It's important to know this function as it allows you to customize your request in ways the `get()` method cannot, like making requests with other HTTP methods.

Next, you will configure and customize your requests with the `request()` function.

Step 2 — Configuring HTTP `request()` Options

The `request()` function allows you to send HTTP requests without specifying the URL in the first argument. In this case, the URL would be contained with the `options` object, and the `request()` would have this function signature:

```
https.request(Options_Object, Callback_Function) {
    Action
}
```

In this step, you will use this functionality to configure your `request()` with the `options` object.

Node.js allows you to enter the URL in the `options` object you pass to the request. To try this out, reopen the `getRequestWithRequest.js` file:

```
nano getRequestWithRequest.js
```

Remove the URL from the `request()` call so that the only arguments are the `options` variable and the callback function:

requests/getRequestWithRequest.js

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

const options = {
  method: 'GET',
};

let request = https.request(options, (res) => {
...
```

Now add the following properties to the `options` object:

requests/getRequestWithRequest.js

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

const options = {
  host: 'jsonplaceholder.typicode.com',
  path: '/users?_limit=2',
  method: 'GET'
};

let request = https.request(options, (res) => {
...
```

Instead of one string URL, you have two properties—`host` and `path`. The `host` is the domain name or IP address of the server you're accessing. The path is everything that comes after the domain name, including query parameters (values after the question mark).

The options object can hold other useful data that goes into a request. For example, you can provide request headers in the options. Headers typically send metadata about the request.

When developers create APIs, they may choose to support different data formats. One API endpoint may be able to return data in JSON, CSV, or XML. In those APIs, the server may look at the `Accept` header to determine the correct response type.

The `Accept` header specifies the type of data the user can handle. While the API being used in these examples only return JSON, you can add the `Accept` header to your request to explicitly state that you want JSON.

Add the following lines of code to append the `Accept` header:

requests/getRequestWithRequest.js

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

const options = {
  host: 'jsonplaceholder.typicode.com',
  path: '/users?_limit=2',
  method: 'GET',
  headers: {
    'Accept': 'application/json'
  }
};
```

By adding headers, you've covered the four most popular options that are sent in Node.js HTTP requests: `host`, `path`, `method`, and `headers`. Node.js supports many more options; you can read more at the official Node.js docs for more information.

Enter `CTRL+X` to save your file and exit `nano`.

Next, run your code once more to make the request by only using options:

```
node getRequestWithRequest.js
```

The results will be the same as your previous runs:

Output

```
Retrieved all data
[
  {
    id: 1,
    name: 'Leanne Graham',
    username: 'Bret',
    email: 'Sincere@april.biz',
    address: {
      street: 'Kulas Light',
      suite: 'Apt. 556',
      city: 'Gwenborough',
      zipcode: '92998-3874',
      geo: [Object]
    },
    phone: '1-770-736-8031 x56442',
    website: 'hildegard.org',
    company: {
      name: 'Romaguera-Crona',
      catchPhrase: 'Multi-layered client-server neural-net',
      bs: 'harness real-time e-markets'
    }
  },
  {
    id: 2,
    name: 'Ervin Howell',
```

```
    username: 'Antonette',
    email: 'Shanna@melissa.tv',
    address: {
      street: 'Victor Plains',
      suite: 'Suite 879',
      city: 'Wisokyburgh',
      zipcode: '90566-7771',
      geo: [Object]
    },
    phone: '010-692-6593 x09125',
    website: 'anastasia.net',
    company: {
      name: 'Deckow-Crist',
      catchPhrase: 'Proactive didactic contingency',
      bs: 'synergize scalable supply-chains'
    }
  }
]
```

As APIs can vary from provider to provider, being comfortable with the `options` object is key to adapting to their differing requirements, with the data types and headers being some of the most common variations.

So far, you have only done `GET` requests to retrieve data. Next, you will make a `POST` request with Node.js so you can upload data to a server.

Step 3 — Making a `POST` Request

When you upload data to a server or want the server to create data for you, you typically send a `POST` request. In this section, you'll create a `POST` request in Node.js. You will make a request to create a new user in the `users` API.

Despite being a different method from `GET`, you will be able to reuse code from the previous requests when writing your `POST` request. However, you will have to make the following adjustments:

- Change the method in the `options` object to `POST`
- Add a header to state you are uploading JSON
- Check the status code to confirm a user was created
- Upload the new user's data

To make these changes, first create a new file called `postRequest.js`. Open this file in `nano` or an alternative text editor:

```
nano postRequest.js
```

Begin by importing the `https` module and creating an `options` object:

requests/postRequest.js

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

const options = {
  host: 'jsonplaceholder.typicode.com',
  path: '/users',
  method: 'POST',
  headers: {
    'Accept': 'application/json',
    'Content-Type': 'application/json; charset=UTF-8'
  }
};
```

You change the `path` to match what's required for `POST` requests. You also updated the `method` to `POST`. Lastly, you added a new header in your options `Content-Type`. This header tells the server what type of data you are uploading. In this case, you'll be uploading JSON data with UTF-8 encoding.

Next, make the request with the `request()` function. This is similar to how you made `GET` requests, but now you look for a different status code than 200. Add the following lines to the end of your code:

requests/postRequest.js

```
...
const request = https.request(options, (res) => {
  if (res.statusCode !== 201) {
    console.error(`Did not get a Created from the server. Cod
e: ${res.statusCode}`);
    res.resume();
    return;
  }

  let data = '';

  res.on('data', (chunk) => {
    data += chunk;
  });

  res.on('close', () => {
    console.log('Added new user');
    console.log(JSON.parse(data));
  });
});
```

The highlighted line of code checks if the status code is 201. The 201 status code is used to indicate that the server created a resource.

This `POST` request is meant to create a new user. For this API, you need to upload the user details. Create some user data and send that with your `POST` request:

requests/postRequest.js

```
...

const requestData = {
  name: 'New User',
  username: 'digitalocean',
  email: 'user@digitalocean.com',
  address: {
    street: 'North Pole',
    city: 'Murmansk',
    zipcode: '12345-6789',
  },
  phone: '555-1212',
  website: 'digitalocean.com',
  company: {
    name: 'DigitalOcean',
    catchPhrase: 'Welcome to the developer cloud',
    bs: 'cloud scale security'
  }
};

request.write(JSON.stringify(requestData));
```

You first created the `requestData` variable, which is a JavaScript object containing user data. Your request does not include an `id` field, as servers

typically generate these while saving the new data.

You next use the `request.write()` function, which accepts a string or buffer object to send along with the request. As your `requestData` variable is an object, you used the `JSON.stringify` function to convert it to a string.

To complete this module, end the request and check for errors:

requests/postRequest.js

```
...


request.end();


request.on('error', (err) => {
  console.error(`Encountered an error trying to make a reques
t: ${err.message}`);
});
```

It's important that you write data before you use the `end()` function. The `end()` function tells Node.js that there's no more data to be added to the request and sends it.

Save and exit `nano` by pressing `CTRL+X`.

Run this program to confirm that a new user was created:

```
node postRequest.js
```

The following output will be displayed:

```
Output
Added new user
{
  name: 'New User',
  username: 'digitalocean',
  email: 'user@digitalocean.com',
  address: { street: 'North Pole', city: 'Murmansk', zipcode:
 '12345-6789' },
  phone: '555-1212',
  website: 'digitalocean.com',
  company: {
    name: 'DigitalOcean',
    catchPhrase: 'Welcome to the developer cloud',
    bs: 'cloud scale security'
  },
  id: 11
}
```

The output confirms that the request was successful. The API returned the user data that was uploaded, along with the ID that was assigned to it.

Now that you have learned how to make `POST` requests, you can upload data to servers in Node.js. Next you will try out `PUT` requests, a method used to update data in a server.

Step 4 — Making a `PUT` Request

Developers make a `PUT` request to upload data to a server. While this may be similar to `POST` requests, `PUT` requests have a different function. `PUT` requests are idempotent—you can run a `PUT` request multiple times and it will have the same result.

In practice, the code you write is similar to that of a `POST` request. You set up your options, make your request, write the data you want to upload, and verify the response.

To try this out, you're going to create a `PUT` request that updates the first user's username.

As the code is similar to the `POST` request, you'll use that module as a base for this one. Copy the `postRequest.js` into a new file, `putRequest.js`:

```
cp postRequest.js putRequest.js
```

Now open `putRequest.js` in a text editor:

```
nano putRequest.js
```

Make these highlighted changes so that you send a `PUT` request to `https://jsonplaceholder.typicode.com/users/1`:

requests/putRequest.js

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


const options = {
  host: 'jsonplaceholder.typicode.com',
  path: '/users/1',
  method: 'PUT',
  headers: {
    'Accept': 'application/json',
    'Content-Type': 'application/json; charset=UTF-8'
  }
};


const request = https.request(options, (res) => {
  if (res.statusCode !== 200) {
    console.error(`Did not get an OK from the server. Code: ${
res.statusCode}`);
    res.resume();
    return;
  }


  let data = '';


  res.on('data', (chunk) => {
    data += chunk;
  });
```

```
  res.on('close', () => {
    console.log('Updated data');
    console.log(JSON.parse(data));
  });
});

const requestData = {
  username: 'digitalocean'
};

request.write(JSON.stringify(requestData));

request.end();

request.on('error', (err) => {
  console.error(`Encountered an error trying to make a reques
t: ${err.message}`);
});
```

You first change the `path` and `method` properties of the `options` object. `path` in this case identifies the user that you are going to update. When you make the request, you check if the response code was 200, meaning that the request was OK. The data you are uploading now only contains the property you are updating.

Save and exit `nano` with `CTRL+X`.

Now execute this Node.js program in your terminal:

```
node putRequest.js
```

You will receive this output:

```
Output
Updated data
{ username: 'digitalocean', id: 1 }
```

You sent a `PUT` request to update a pre-existing user.

So far you have learned how to retrieve, add, and update data. To give us a full command of managing data via APIs, you'll next make a `DELETE` request to remove data from a server.

Step 5 — Making a `DELETE` Request

The `DELETE` request is used to remove data from a server. It can have a request body, but most APIs tend not to require them. This method is used to delete an entire object from the server. In this section, you are going to delete a user using the API.

The code you will write is similar to that of a `GET` request, so use that module as a base for this one. Copy the `getRequestWithRequest.js` file into a new `deleteRequest.js` file:

```
cp getRequestWithRequest.js deleteRequest.js
```

Open `deleteRequest.js` with `nano`:

```
nano deleteRequest.js
```

Now modify the code at the highlighted parts, so you can delete the first user in the API:

requests/putRequest.js

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


const options = {
  host: 'jsonplaceholder.typicode.com',
  path: '/users/1',
  method: 'DELETE',
  headers: {
    'Accept': 'application/json',
  }
};


const request = https.request(options, (res) => {
  if (res.statusCode !== 200) {
    console.error(`Did not get an OK from the server. Code: ${
res.statusCode}`);
    res.resume();
    return;
  }


  let data = '';


  res.on('data', (chunk) => {
    data += chunk;
  });


  res.on('close', () => {
```

```
    console.log('Deleted user');
    console.log(JSON.parse(data));
  });
});


request.end();


request.on('error', (err) => {
  console.error(`Encountered an error trying to make a reques
t: ${err.message}`);
});
```

For this module, you begin by changing the `path` property of the options object to the resource you want to delete—the first user. You then change the method to `DELETE`.

Save and exit this file by pressing `CTRL+X`.

Run this module to confirm it works. Enter the following command in your terminal:

```
node deleteRequest.js
```

The program will output this:

```
Output
Deleted user
{}
```

While the API does not return a response body, you still got a 200 response so the request was OK.

You've now learned how to make `DELETE` requests with Node.js core modules.

Conclusion

In this tutorial, you made `GET`, `POST`, `PUT`, and `DELETE` requests in Node.js. No libraries were installed; these requests were made using the standard `https` module. While `GET` requests can be made with a `get()` function, all other HTTP methods are done via the `request()` method.

The code you wrote was written for a publicly available, test API. However, the way you write requests will work for all types of APIs. If you would like to learn more about APIs, check out our API topic page. For more on developing in Node.js, return to the How To Code in Node.js series.