Node.js Development JavaScript

By Stack Abuse
Published on September 18, 2019 42.8k

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

Introduction

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.__lookupGetter__
Math.__lookupSetter__      Math.__proto__             Math.constructor
Math.hasOwnProperty        Math.isPrototypeOf         Math.propertyIsEnumerable
Math.toLocaleString        Math.toString              Math.valueOf
```

```
Math.E                  Math.LN10               Math.LN2
Math.LOG10E             Math.LOG2E              Math.PI
Math.SQRT1_2            Math.SQRT2              Math.abs
Math.acos               Math.acosh              Math.asin
Math.asinh              Math.atan               Math.atan2
Math.atanh              Math.cbrt               Math.ceil
Math.clz32              Math.cos                Math.cosh
Math.exp                Math.expm1              Math.floor
Math.fround             Math.hypot              Math.imul
Math.log                Math.log10              Math.log1p
Math.log2               Math.max                Math.min
Math.pow                Math.random             Math.round
Math.sign               Math.sin                Math.sinh
Math.sqrt               Math.tan                Math.tanh
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

Next in series: How To Use Node.js Modules with npm and package.json →