

# Node.js

---

## The basics

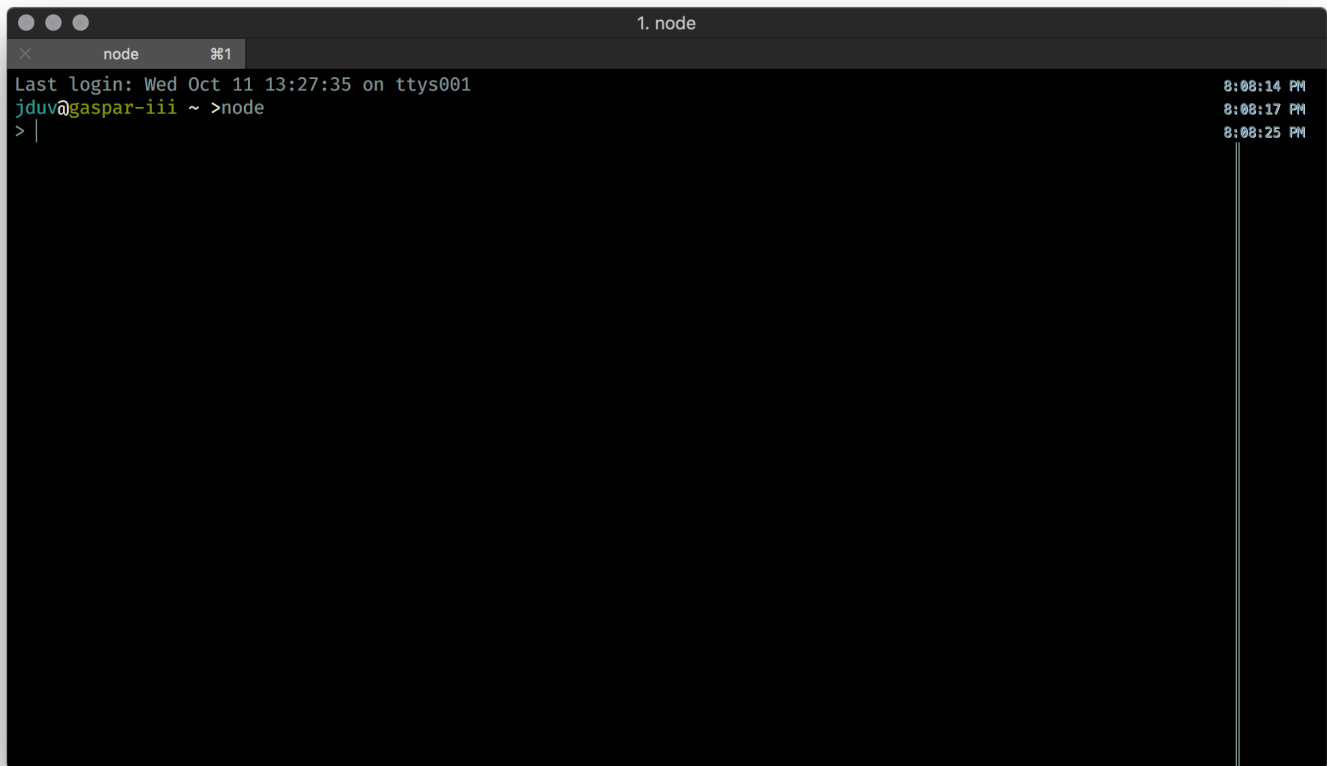
---

Node is a run time environment similar to python. Instead of running scripts that are `.py` we run scripts that end in `.js`. Which means, of course, that any code we write that runs on the node framework is written in JavaScript.

## How it works

---

First, let's fire up the node interpreter and play around with it. This is important because it will help you understand, just like we did in python, how code runs: top to bottom left to right.

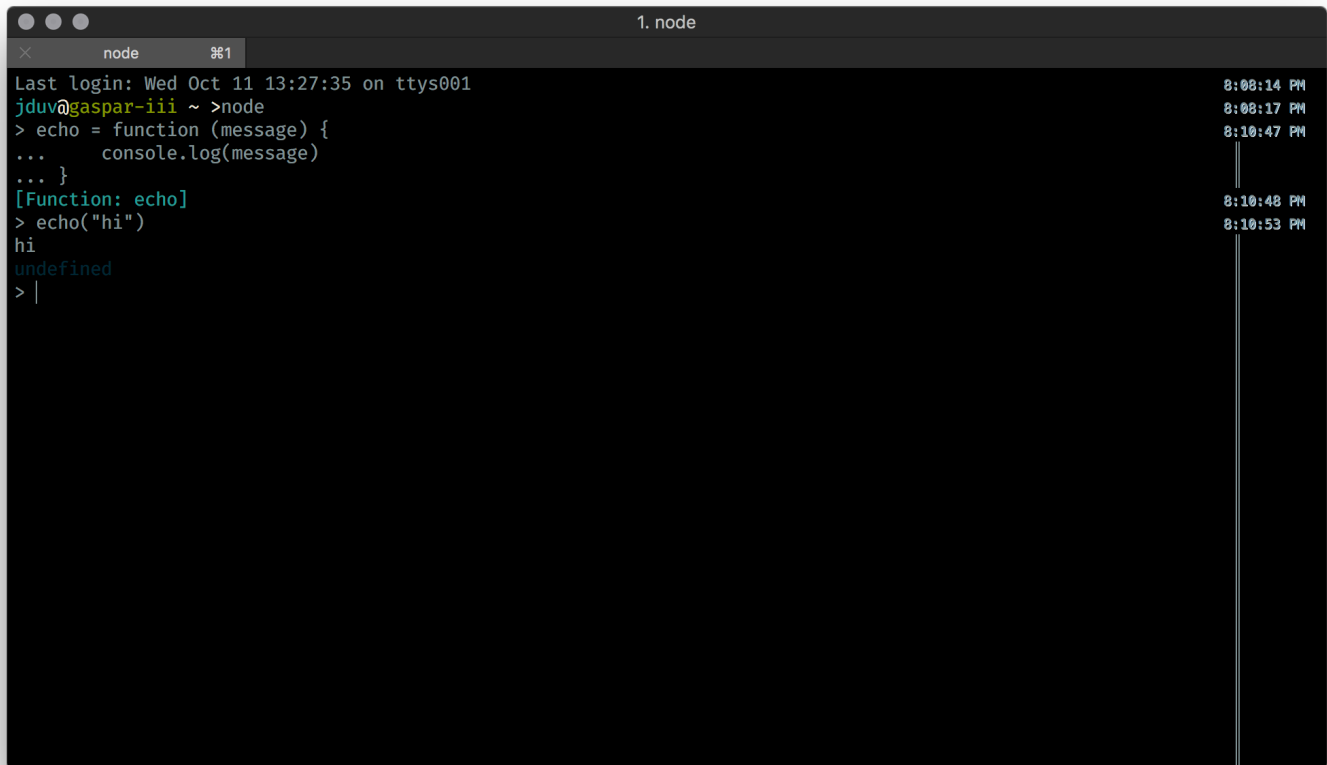
A terminal window titled "1. node" with a tab labeled "node". The terminal shows the login prompt "Last login: Wed Oct 11 13:27:35 on ttys001" and the user "jduv@gaspar-iii" in the shell. The command ">node" has been entered, and the prompt has changed to "> |". On the right side of the terminal, there are three timestamps: "8:08:14 PM", "8:08:17 PM", and "8:08:25 PM".

```
node 1. node
Last login: Wed Oct 11 13:27:35 on ttys001
jduv@gaspar-iii ~ >node
> |
8:08:14 PM
8:08:17 PM
8:08:25 PM
```

Let's write a simple program that writes a message to the console. The program looks like this:

```
echo = function (message) {  
  console.log(message)  
}
```

Paste or write this function line for line in your node.js interpreter. You'll get something like this:



```
1. node  
Last login: Wed Oct 11 13:27:35 on ttys001  
jduv@gaspar-iii ~ > node  
> echo = function (message) {  
...   console.log(message)  
... }  
[Function: echo]  
> echo("hi")  
hi  
undefined  
> |
```

This function will remain in the interpreters memory as long as you have the current shell open. So, if you wanted to keep this function safe we might as well save it in the current directory as a `.js` file. Like so:

```
1. bash
bash %1
jduv@gaspar-iii wk10_node_sql >touch echo.js 8:14:06 PM
jduv@gaspar-iii wk10_node_sql >tree 8:14:09 PM
.
├── echo.js
├── img
│   ├── echo-func.png
│   └── start-node.png
└── intro-to-node.md

1 directory, 4 files 8:13:54 PM
jduv@gaspar-iii wk10_node_sql > 8:14:09 PM
8:13:54 PM
```

Paste the function along with a line executing the function into `echo.js` like so:

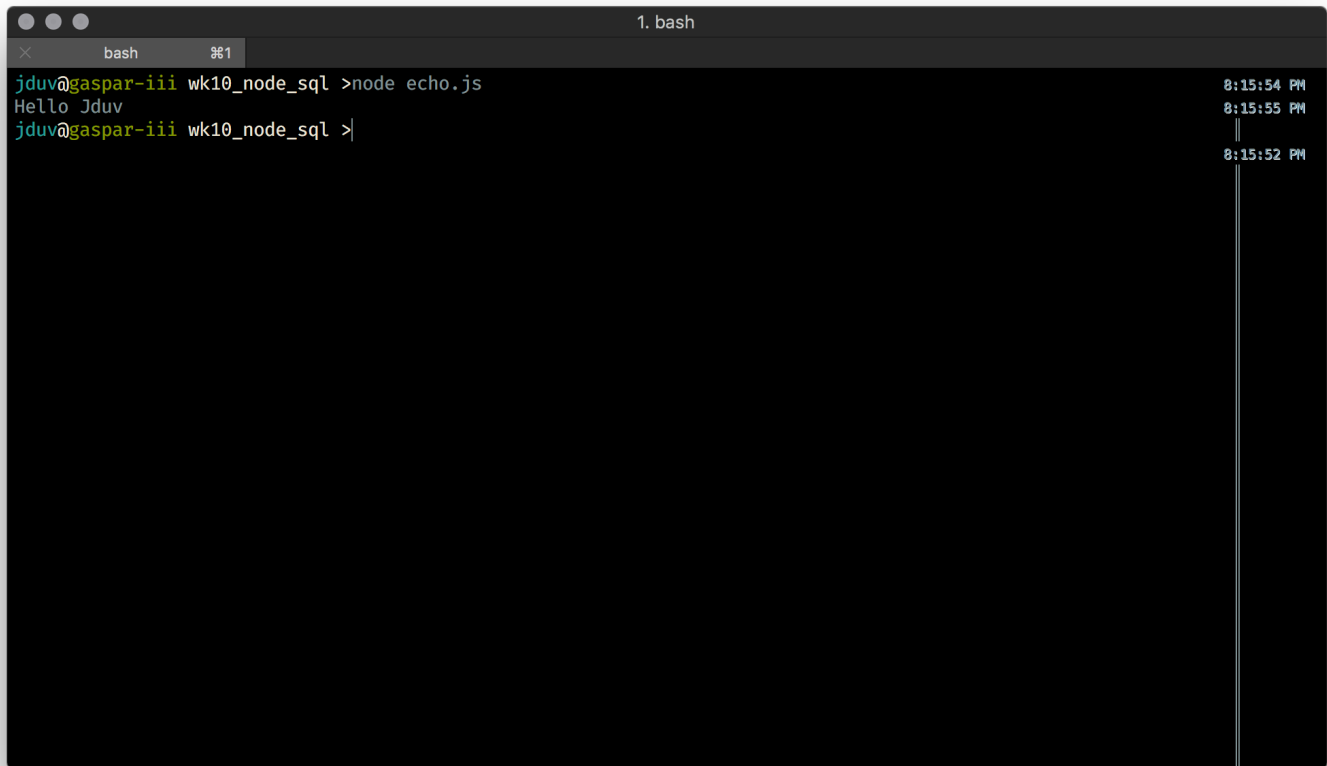
```
echo = function (message) {
  console.log(message)
}

echo("Hello Jduv")
```

Now, run the file like so:

```
> node echo.js
```

You should see something like the terminal below:

A terminal window titled "1. bash" with a tab labeled "bash" and a cursor icon. The prompt is "jduv@gaspar-iii wk10\_node\_sql >". The user enters "node echo.js" and the output is "Hello Jduv". The prompt returns to "jduv@gaspar-iii wk10\_node\_sql >". On the right side, there are three timestamps: "8:15:54 PM", "8:15:55 PM", and "8:15:52 PM" (likely a typo for 53 PM).

```
jduv@gaspar-iii wk10_node_sql >node echo.js
Hello Jduv
jduv@gaspar-iii wk10_node_sql >
```

**Exercise 1::** Using the node interpreter, write a function that doubles a number. Write another function that reverses a string.

**Exercise 2::** Save your work to files called `double.js` and `reverse.js` . Run them using the node interpreter from the command line.

## As a framework

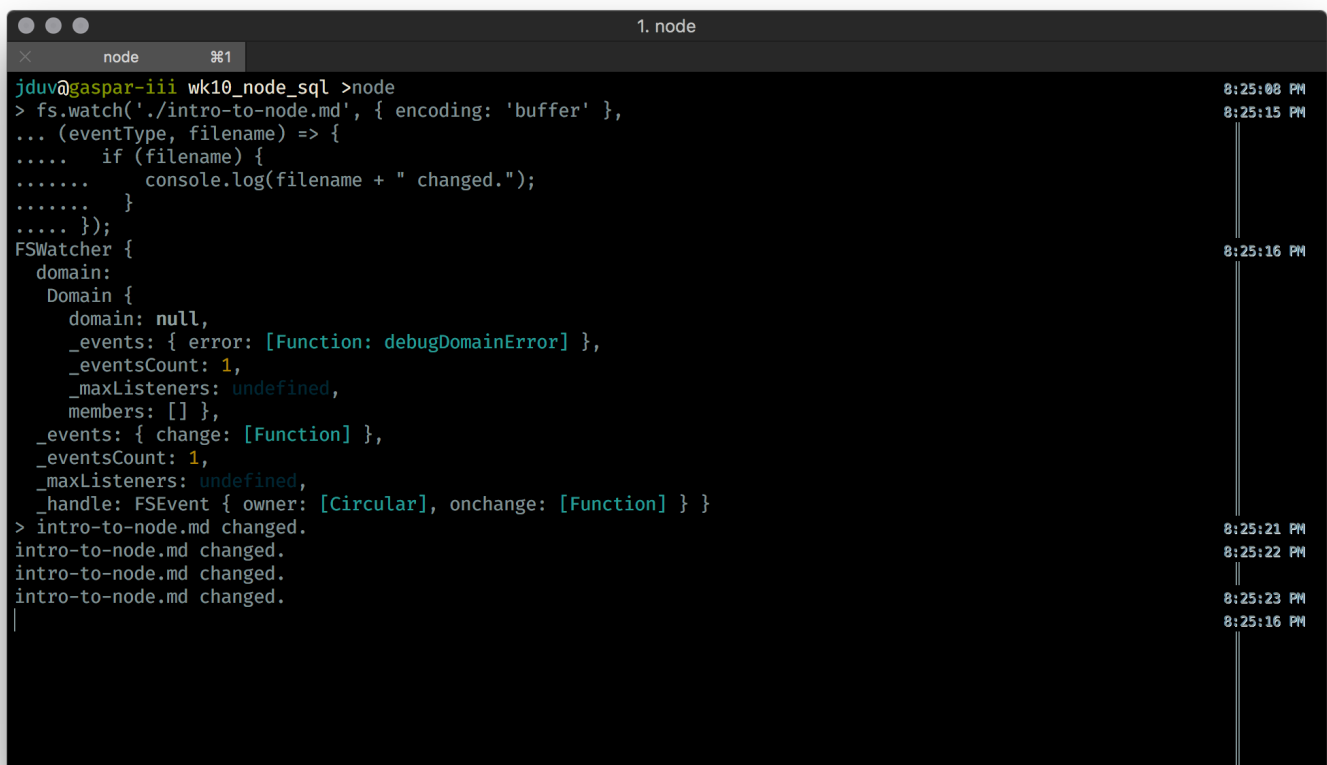
Just like python, node ships with lots of built in libraries that do interesting things. Take, for example, the `fs` package. What does it do? Well, a lot. At least based on the documentation linked above. But, let's explore an interesting piece of it called a `FSWatcher` object.

### Code along solution

```
// Example when handled through fs.watch listener
fs.watch('./intro-to-node.md', { encoding: 'buffer' },
(eventType, filename) => {
  if (filename) {
    console.log(filename + " changed.");
  }
});
```

```
}  
});
```

The `fs.watch()` function will, using system calls and voodoo magic, watch the target file for any modifications and execute the given callback function on a change. Note that the implementation of this module is operating system specific because the semantics of file systems are native to their respective operating systems. Therefore, you may get different results on Windows, Linux, and Mac OSX systems.

A screenshot of a terminal window titled "1. node" showing a Node.js process. The user runs a command to start a Node.js script. The script uses `fs.watch()` to monitor a file named `intro-to-node.md`. The terminal output shows the file being watched, the watcher object being created, and several "changed" events being triggered. On the right side of the terminal, a vertical timeline of timestamps is visible, indicating the sequence of events.

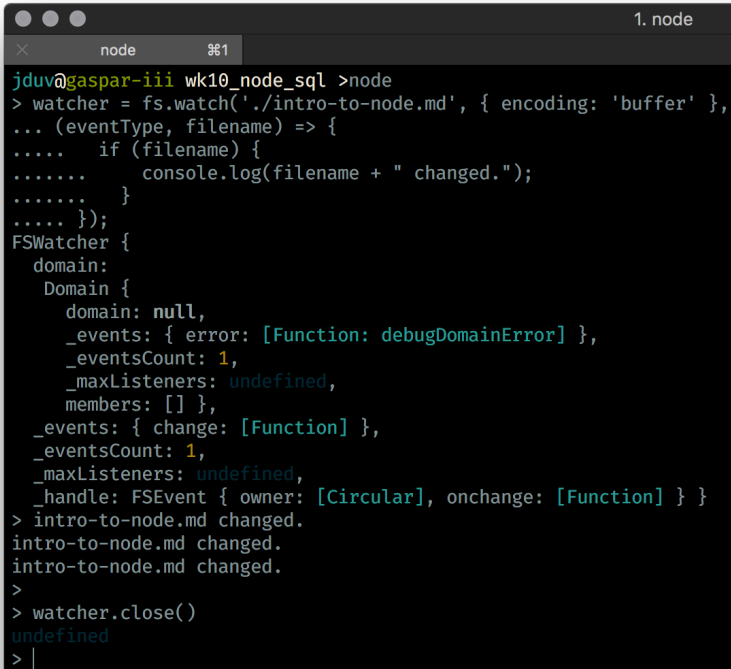
```
node wk10_node_sql >node  
> fs.watch('./intro-to-node.md', { encoding: 'buffer' },  
... (eventType, filename) => {  
..... if (filename) {  
.....   console.log(filename + " changed.");  
..... }  
..... });  
FSWatcher {  
  domain:  
    Domain {  
      domain: null,  
      _events: { error: [Function: debugDomainError] },  
      _eventsCount: 1,  
      _maxListeners: undefined,  
      members: [] },  
  _events: { change: [Function] },  
  _eventsCount: 1,  
  _maxListeners: undefined,  
  _handle: FSEvent { owner: [Circular], onchange: [Function] } }  
> intro-to-node.md changed.  
intro-to-node.md changed.  
intro-to-node.md changed.  
intro-to-node.md changed.  
|
```

Note a few interesting things about this code and its execution: the code runs immediately, then returns and gives you back control. This is because the `fs.watch()` function runs *asynchronously* to your node terminal. That is, you can do other stuff while this code runs in the background. In order to stop a watcher, according to the documentation you need to call `watcher.close()`. So, let's modify our code a bit so we can do so:

```
// Example when handled through fs.watch listener  
watcher = fs.watch('./intro-to-node.md', { encoding: 'buffer' },  
(eventType, filename) => {
```

```
if (filename) {  
  console.log(filename + " changed.");  
}  
});
```

Now that we have stored our watcher in a variable, we can access it! So, let's close the watcher like so:

A terminal window titled '1. node' showing a Node.js REPL session. The user enters a command to create a file watcher for 'intro-to-node.md'. The watcher is stored in a variable named 'watcher'. The user then triggers a file change, and the watcher logs the message 'intro-to-node.md changed.'. Finally, the user calls 'watcher.close()' to stop the watcher, and the terminal returns 'undefined'.

```
node wk10_node_sql >node  
> watcher = fs.watch('./intro-to-node.md', { encoding: 'buffer' },  
... (eventType, filename) => {  
.....   if (filename) {  
.....     console.log(filename + " changed.");  
.....   }  
..... });  
FSWatcher {  
  domain:  
    Domain {  
      domain: null,  
      _events: { error: [Function: debugDomainError] },  
      _eventsCount: 1,  
      _maxListeners: undefined,  
      members: [] },  
  _events: { change: [Function] },  
  _eventsCount: 1,  
  _maxListeners: undefined,  
  _handle: FSEvent { owner: [Circular], onchange: [Function] } }  
> intro-to-node.md changed.  
intro-to-node.md changed.  
intro-to-node.md changed.  
>  
> watcher.close()  
undefined  
> |
```