# NodeJS



# Introduction

NodeJS is a server-side Javascript platform based on the Chrome's V8 Javascript engine. It was initially released in 2009 by Ryan Dahl.

In NodeJS's case, server-side means that it can run Javascript code locally instead of needing a browser. You get access to I/O resources typical with most other languages that run locally (e.g. networking, file, etc..), but you don't get access to browser specific features (e.g. DOM search and manipulation).

> **NOTE**: There are packages you can use to simulate a browser within NodeJS.

# Setup

The following sections discuss common setup and usage instructions for NodeJS.

## Installing Latest Version

To install nodejs, you can use the instructions at https://nodejs.org/en/download/package-manager.

For Linux Mint, this was…

```
~ $ curl -sL https://deb.nodesource.com/setup_8.x | sudo -E bash -
~ $ sudo apt-get install -y nodejs
```

## Installing Different Versions

You also have the option of using node version manager (nvm), which will let you install multiple different versions of node locally and switch between them as needed -- very useful for testing between different versions of node.

> **NOTE**: You don't need sudo for any of this. NodeJS packages installed via nvm will be installed locally.

For Linux Mint, this was…

```
~ $ wget -qO-
https://raw.githubusercontent.com/creationix/nvm/v0.33.6/install.sh | bash
=> Downloading nvm as script to '/home/user/.nvm'

=> Appending nvm source string to /home/user/.bashrc
=> Appending bash_completion source string to /home/user/.bashrc
=> Close and reopen your terminal to start using nvm or run the following
to use it now:

export NVM_DIR="$HOME/.nvm"
[ -s "$NVM_DIR/nvm.sh" ] && \. "$NVM_DIR/nvm.sh"  # This loads nvm
```

```
[ -s "$NVM_DIR/bash_completion" ] && \. "$NVM_DIR/bash_completion"  # This
loads nvm bash_completion
```

> **NOTE**: After setting up nvm, open a new terminal and type nvm and you should see the nvm help page.

To install a specific version of NodeJS, you can use nvm install vx.x.x…

```
~ $ nvm install v8.9.1
Downloading and installing node v8.9.1...
Downloading https://nodejs.org/dist/v8.9.1/node-v8.9.1-linux-x64.tar.xz...
######################################################################## 
100.0%
Computing checksum with sha256sum
Checksums matched!
Now using node v8.9.1 (npm v5.5.1)
Creating default alias: default -> v8.9.1
```

To list installed versions of NodeJS, you can use nvm ls…

```
~ $ nvm list
->       v6.6.0
         v8.9.1
default -> v8.9.1
node -> stable (-> v8.9.1) (default)
stable -> 8.9 (-> v8.9.1) (default)
iojs -> N/A (default)
lts/* -> lts/carbon (-> N/A)
lts/argon -> v4.8.7 (-> N/A)
lts/boron -> v6.12.2 (-> N/A)
lts/carbon -> v8.9.3 (-> N/A)
```

To use a specific version of NodeJS, you can use nvm use vx.x.x (remember that you need to do this for every new shell)...

```
~ $ nvm use v8.9.1
Now using node v8.9.1 (npm v5.5.1)
~ $ node -v
v8.9.1
```

To set the default version of NodeJS for any new shells, you can use nvm alias default vx.x.x (remember that this change only takes effect in new shells, not the current shell)…

```
~ $ nvm alias default v6.6.0
default -> v6.6.0
```

## Running Files

You can run a Javascript file by using node filenamehere…

```
~/test $ node index.js
5
7
```

## Running REPL

You can also start a REPL loop (just like with python) just by running the node command without a filename…

```
~ $ node
> console.log('hi!');
hi!
undefined
> console.log('bye!');
bye!
undefined
>
```

> **NOTE**: The REPL comes with autocomplete. Use the Tab key.
>
> **NOTE**: In addition to allowing you to type in Javascript code, the REPL tool has some built-in commands that you can use to do things like exit the REPL or get help. These commands all start with a dot (.) and work with autocomplete. For example: .help or .exit.

## Using IDEs

There are lots of different IDEs you can use, but the most popular one at the moment seems to be VSCode. It has a large list of extensions available and provides several useful features out of the box (different languages, autocomplete, built-in terminal, etc..).

https://code.visualstudio.com/docs/setup/linux for setup instructions on Linux Mint…

```
~ $ curl https://packages.microsoft.com/keys/microsoft.asc | gpg --dearmor
> microsoft.gpg
~ $ sudo mv microsoft.gpg /etc/apt/trusted.gpg.d/microsoft.gpg
~ $ sudo sh -c 'echo "deb [arch=amd64]
https://packages.microsoft.com/repos/vscode stable main" >
```

```
/etc/apt/sources.list.d/vscode.list'
~ $ sudo apt-get update
~ $ sudo apt-get install code
```

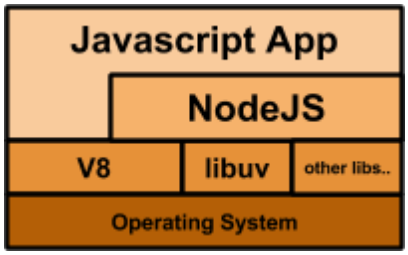
Regardless of which IDE you use, you should set up editorconfig: http://editorconfig.org. It supported by nearly every IDE.

# Architecture

NodeJS's architecture is centered around two components…

- V8 → Google Chrome's Javascript engine
- libuv → Event loop library that perform all async I/O (and uses a thread pool to deal with I/O that can't be done async)

Javascript functionality in NodeJS is delegated to V8, while I/O functionality is delegated to libuv. NodeJS acts as the bridge between these two components, in addition to providing several higher-level user modules and nice-ities.



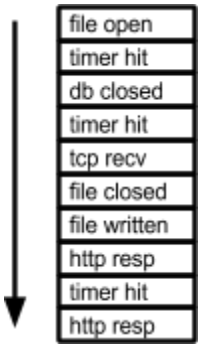

> **NOTE**: It looks like there's effort being put in to decouple NodeJS from V8, such that it can use other Javascript engines to run: https://github.com/nodejs/node-chakracore. But, this is not an officially supported project.
>
> **NOTE**: libuv was originally written for NodeJS, but has since become its own project and is used by other platforms/languages.

# Programming Model

> **NOTE**: This is super important. It's the foundation of the entire platform.

NodeJS follows a typical event loop model -- similar to a game loop or Win32API's GUI message loop. Events come in and are queued for processing by the loop. The event loop continually takes the next item off the queue and processes it by giving it to your code.

The events coming in are almost always IO-based events: mouse click, keyboard button pressed, file open, tcp connection closed, db connection open, etc…




In Javascript, the standard way to handle these events in your code are callbacks. Unlike other languages (e.g. Java), you can't / shouldn't be blocking the thread waiting for an event. The reason is that a NodeJS (and other Javascript engines) typically execute your code in a single thread and don't come with any multithreading APIs.

> **NOTE**: The workaround to not having threading support is to spawn multiple NodeJS processes, similar to what you do with Python (because Python's multithreading support suffers from global interpreter lock issues). There's even a module to help with this called cluster.

## Callbacks

In Java, we may have code like this…

```
Socket skt = new Socket("localhost", 1234);
BufferedReader in;

in = new BufferedReader(new InputStreamReader(skt.getInputStream()));
System.out.println(in.readLine());
in.close();
skt.close()
```

We can't block like this in Javascript/NodeJS, so that same logic has to be re-written using callbacks.

For each function that has the potential to block, you pass in a callback. When that function is finished, it'll invoke that callback and pass any relevant information/results to it...

```
Socket.open("localhost", 1234, function(err, conn) {
  conn.readLine(function(err, line) {
```

```
    console.out(line);
    conn.close();
  });
});
```

The conventions for callbacks in NodeJS are…

- functions that take in a callback should take it in as the last parameter.
  e.g. getRecordsFromDatabase(connection, searchterms, callbackResult) ...
- callback functions should take in the error as the first parameter while all other parameters should be the result
  e.g. function(error, callbackResult) ...

If your callbacks are small and only used once, it's typical to have them inlined as anonymous functions / closures. For example...

```
Socket.open("localhost", 1234, function(err, conn) {
  lineEmitter = conn.readLine();
  lineEmitter.on("newLine", function(line) {
    console.out(line);
    conn.close();
  });
});
```

instead of...

```
var incomingLineCallback = function(line) {
  console.out(line);
  conn.close();
};
var connEstablishedCallback = function(err, conn) {
  lineEmitter = conn.readLine();
  lineEmitter.on("newLine", incomingLineCallback);
};
Socket.open("localhost", 1234, connEstablishedCallback);
```

## Emitters

Emitters are pretty much the same as callback functions, but a bit more flexible.  You can think of an event emitter as pubsub (publisher-subscriber) style callbacks. Instead of passing the callback directly into a function, that function will return an event emitter object that "publishes" events. You can then "subscribe" to listen for events as they happen.

Let's take the same example from the callback section. In Java, we may have code like this…

```
Socket skt = new Socket("localhost", 1234);
BufferedReader in;

in = new BufferedReader(new InputStreamReader(skt.getInputStream()));
System.out.println(in.readLine());
in.close();
skt.close()
```

We can't block like this in Javascript/NodeJS, so that same logic has to be re-written using emitters...

```
Socket.open("localhost", 1234, function(err, conn) {
  lineEmitter = conn.readLine();
  lineEmitter.on("newLine", function(line) {
    console.out(line);
    conn.close();
  });
});
```

> **NOTE**: In addition to the on() function, the emitter object also has a emit() function that can be used to publish events. This may be useful for mocks.

You may be thinking that there's a race condition here: there's a chance the event could hit between when the event emitter is given back and when the callback is registered on the event emitter.

This is not correct… Remember that NodeJS is single threaded and runs a event loop. So long as we register with the event emitter before we release control (return from the function in this case), we won't' miss any events.

## Streams

NodeJS comes with a streams similar to Java's InputStream and OutputStream. The difference is that NodeJS's streams extend event emitters -- you listen for events such as data, error, close, etc… on a stream instead of invoking functions and potentially blocking.

> **NOTE**: In addition to Readable and Writable streams, NodeJS also has Duplex and Transform streams. Duplex is a stream that's both readable and writable. Transform streams are also readable and writable but are used to transform data as its read/written (e.g. compression/encryption/etc..).

NodeJS streams even provide higher-level functions such aspipe(), which automatically register for events on the relevant event emitters such that data can flow between streams…

```
var request = require('request');
var process = require('process');

var is = request('http://www.pluralsight.com'); // readable stream
var os = process.stdout;                        // writeable stream
is.pipe(os);                                    // pipe data is -> os
```

> **NOTE**: A lot of modules return streams, and you can chain pipe() calls together. For example you can pipe the output of a website to a gzip pipe and feed that to a filestream pipe. This is very similar to linux commandline piping (e.g. cat somefile.txt | grep searchterm).

## Gotcha: Computationally Heavy Code

Code that's computationally heavy will stop the event loop from processing events while it's running. This means that if …

- you a UI, your program will feel like it's frozen.
- the event queue gets too backed up while your code works, it may end up crashing.

Here's an example…

```
setTimeout(() => { console.log(new Date()) }, 1000);
setTimeout(() => { console.log(new Date()) }, 2000);
setTimeout(() => { console.log(new Date()) }, 3000);
for (i = 0; i < 1999999999; i++) {}
```

The code above should output the current date once every second for 3 seconds, but that's not what happens…

```
~/test $ node index.js
2017-12-09T17:52:07.298Z
2017-12-09T17:52:07.299Z
2017-12-09T17:52:07.303Z
```

Notice how the times in the output above are much closer than 1 second apart. The reason is the for loop at the end -- it causes the event loop processing to block for 10+ seconds while it iterates.

While that for loop is iterating, the 3 timeouts hit and their callbacks get put onto the event loop's queue. But, the event loop won't get a chance to process those events (by invoking their callbacks) until that for loop ends.

## Gotcha: Sync vs Async Callbacks

Your functions should either be sync or async, it doesn't make sense for callbacks to be invoked async in some cases (via the event loop) and invoked sync in other cases (via your code).

Here's a trivial example…

```
function timer(cb, timeout) {
   if (timeout % 1000 != 0) {
       cb("timeout must be whole seconds");
       return;
   }

   setTimeout(cb, timeout);
}

timer((error) => {
   if (error) {
       console.log('ERROR!');
   } else {
       console.log(new Date());
   }
}, 1500);
```

Notice what's happening in timer. If the timeout value…

- isn't divisible by 1000, it'll invoke the callback sync (directly invoked by you).
- is divisible by 1000, it'll invoke the callback async (invoked by the event loop).

This is bad design. Code that uses timer() may break because that it expects the cb parameter to always be invoked asynchronously (that's what happens in the happy path -- setTimeout will put it onto the event loop). For example, that code may call timer() and then immediately follow up by doing some other critical piece of work because it doesn't expect the callback to be invoked.

There are a few ways around this. If your code is intended to be async, you should always use one of the following for invoking a callback...

- setTimeout(callback, 0);
- setImmediate(callback);
- process.nextTick(callback);

These all pretty much do the same thing, but they're slightly different...

- setImmediate is like setTimeout with a 0ms wait, but it looks like setImmediate will always take precedence over setTimeout in the event queue.

- process.nextTick seems to skip the event loop queue entirely -- it forces your callback to be invoked as soon as you release control.

In most cases, if your code is intended to be async then process.nextTick is what you want when invoking a callback. It'll let the rest of the user code execute. As soon as it's done, it'll invoke the callback.

## Gotcha: Callback Hell

The problem with callbacks is that in most non-trivial cases, you need to chain multiple small one-time use callbacks together. If you use anonymous functions for these callbacks, your code will quickly become unmaintainable...

```
doAsync1(function(err, result) {
  doAsync2(function(err, result) {
    doAsync3(function(err, result) {
      doAsync4(function(err, result) {
      });
    });
  });
});
```

The name for this is the callback hell. For a long time, the only workaround for this was to be diligent when writing your code such that you didn't nest too heavily. However, with newer versions of the ECMA standard, the best way to workaround callback hell is to use Javascript generators.

Javascript generators are essentially lightweight coroutines. For more information on how to use generators, see https://strongloop.com/strongblog/node-js-callback-hell-promises-generators.

> **NOTE**: If you're using Javascript directly then generators are available right now in almost all browsers as well as NodeJS. Babel probably shims in this feature if it isn't available. Typescript also seems to have generator support.

The following example was taken directly from the website...

```
var co = require('co')
var thunkify = require('thunkify')
var fs = require('fs')
var path = require('path')
var readdir = thunkify(fs.readdir)
var stat = thunkify(fs.stat)
```

```
var myfunction = co(function* (dir) {
  var files = yield readdir(dir)
  var stats = yield files.map(function (file) {
    return stat(path.join(dir,file))
  })
  var largest = stats
    .filter(function (stat) { return stat.isFile() })
    .reduce(function (prev, next) {
      if (prev.size > next.size) return prev
      return next
    })
  return files[stats.indexOf(largest)]
});
```

# Modules

A module is a file that exports some state or functionality. The file can be…

- another Javascript file (.js extension)...
  Javascript files will be the types of modules you'll be working with most frequently.

- a JSON file (.json extension)...
  JSON files are parsed by node and loaded as a Javascript object.

- a compiled addon module (.node extension)...
  Compiled addon modules are native binaries that export some functionality.

  **NOTE**: Remember that there's a difference between a module and a package. See the Node Package Manager document for more information on packages.

## Loading Modules

To get access to a module, you can pull it into your code via the require() function...

```
var myMod1 = require('./module1');
var CarMod = require('Car');
```

The string passed into the require function is either…

- a relative path pointing (with or without a file extension)
  (e.g. ./FancyModule1 -- YOU CANNOT OMIT THE ./)
- a string identifying a built-in NodeJS module
  (e.g. os, fs, process, http, crypto, etc…)

If you're pointing to a file but you don't specify the file extension, NodeJS will attempt to first load a .js extension, failing that it'll try a .json extension, and failing that it will try a .node extension. The typical pattern here is to skip putting in the extension if your file is a Javascript file, but explicitly put in the extension if your file is a JSON file or a node compiled module.

If you're pointing to a folder, what gets loaded depends on if there's a package.json file in the folder. If…

- there is a package.json file and it has a main field, the file that the main field points to is the file that gets loaded.
- there isn't a package.json file, index.js within that folder will be the file that gets loaded.

  **NOTE**: I'm not 100% sure how folders works work with with JSON files and node compiled modules. I imagine that they work similarly in that the .json/.node file gets loaded from the folder.

## Using Modules

Once pulled in, you can access whatever that module has decided to export…

```
var x = myMod1.count + 5;  // access exported variables
var y = myMod1.func() * 10;// access exported functions
var z = new CarMod();      // instantiated as if module was a class
```

Notice the naming convention used for the variables which hold onto the require functoin's result. A module that...

- only exports variables/functions is assigned to a camelcase variable name starting with a lowercase letter (e.g. myMod1)
- is designed to be instantiated is assigned to a camelcase variable name starting with an uppercase letter (e.g. CarMod)

  **NOTE**: The next section has an important part about module caching -- it applies to this section as well.

## Creating Modules

To make variables/functions available from your Javascript file, simply assign them as properties of the module.exports object. For example…

```
module.exports.myInt = 5;
module.exports.myFunc = function(x,y) { return x + y };
```

Then, you can use those properties in any other Javascript files that require() your module…

```
var myModule = require('./myModule');

console.log(myModule.myInt);
console.log(myModule.myFunc(3,4));
```

> **NOTE**: Remember that we need to specify a relative path in require unless we're trying to pull in a built-in NodeJS module. If the module is in the same path as the file that require()'s it, you need to prepend ./ to it.

The above script outputs…

```
~/test $ node index.js
5
7
```

## Gotcha: Cached Modules

When a module gets pulled in, it's likely a cached instance. That means every require() for that module will pull likely in the same instance. In the example above, if a Javascript file pulls in ./myModule and changes myInt, that change will be visible to every other Javascript file that pulled in ./myModule.

The way to work around this is to export a function that creates an object every time it's invoked

```
module.exports = function() {
   ret = {}
   ret.myInt = 5;
   ret.myFunc = function(x,y) { return x + y };
   return ret;
}
```

```
var MyModule = require('./myModule');

var instance = MyModule();
console.log(instance.myInt);
console.log(instance.myFunc(3,4));
```

> **NOTE**: Notice how the variable name for the require is using camelcase with an uppercase. This is the convention to use when what's being exported is designed to be be instantiated. See previous section for more information.

Another workaround may be to manually invalidate the module cache: https://stackoverflow.com/a/11477602.

## Gotcha: module.exports vs exports

You can add exports by either assigning to module.exports or exports (they point to the same object), but in almost all cases you should be using modules.exports. In certain cases, what you're exporting will fail to export if you don't. For example…

```
exports = function() {
   ret = {}
   ret.myInt = 5;
   ret.myFunc = function(x,y) { return x + y };
   return ret;
}
```

If you try to require() this module and use it, you'll get a failure…

```
var MyModule = require('./myModule');

var instance = MyModule();
console.log(instance.myInt);
console.log(instance.myFunc(3,4));
```

```
~/test $ node index.js
/home/user/test/index.js:3
var instance = MyModule();
               ^

TypeError: MyModule is not a function
    at Object.<anonymous> (/home/user/test/index.js:3:16)
    at Module._compile (module.js:556:32)
    at Object.Module._extensions..js (module.js:565:10)
    at Module.load (module.js:473:32)
    at tryModuleLoad (module.js:432:12)
    at Function.Module._load (module.js:424:3)
    at Module.runMain (module.js:590:10)
    at run (bootstrap_node.js:394:7)
    at startup (bootstrap_node.js:149:9)
    at bootstrap_node.js:509:3
```

The reason for this is because when you load a module, that module is actually wrapped in and called from a wrapper function…

```
function (exports, require, module, __filename, __dirname) { ... }
```

The first parameter of that function is exports, which references the same thing as module.exports. The problem is that when we set the exports directly like we did in our example, we're not actually changing the reference in module.exports. We're changing the reference in the exports parameter of the wrapper function invocation, but not what the module object's exports property references.

If we had set module.export in our example instead, it would have worked as we expected.

# Common Modules

The following is a list of common NodeJS modules and their closest Java equivalents…

- os (built-in)
  → Runtime
- process (built-in)
  → System
  → System.in
  → System.out
  → System.err
  → Process
  → ProcessBuilder
- fs (built-in)
  → FileInputStream
  → FileOutputStream
  → File
  → Files
  → Path
  → Paths
- buffer (built-in)
  → ByteBuffer
  → byte[]
  → DataInput
  → DataOutput
- http (built-in) / request (third-party)
  → HttpUrlConnection
  → Apache HttpClient
- url (built-in) / querystring (built-in)
  → URL
  → URI
- net (built-in) / dns (built-in) / dgram (built-in)
  → Netty
  → Socket

→ SocketChannel
→ ServerSocket
→ ServerSocketChannel
→ DatagramSocket
→ DatagramSocketChannel

- http (built-in) / socket.io (third-party) / express (third-party) / etc...
  → Jetty
  → Tomcat
  → etc..
- assert (built-in) / mocha (third-party) / chai (third-party) / jasmine (third-party) / etc..
  → JUnit
  → TestNG
  → etc..

## Global

Variables and functions defined inside of a Javascript file are scoped to that Javascript file. That means that if you did something like this…

```
var counter = 40;
```

That variable (counter) won't be accessible outside of the Javascript file/module it was declared in. To make things visible globally (to all Javascript files/modules), you need to assign it as a property on the global object.

```
global.counter = 40;
```

It's highly recommended that you don't do this, but if you did you would be able to access counter anywhere in your program after setting it on the global object -- you don't need to use the global object to access it, you can access it directly: global.counter vs counter.

The global object comes with useful properties that you can make use of in your code. Many of these properties are self-explanatory. For example, isNaN is probably a function that determines if your variable is a NaN floating point number.

The entire list of global object properties is too large to replicate, but if you want you can start a REPL loop and type global. Followed by hitting the Tab key to get a list of properties (or just hit the Tab key on a blank line -- same thing). You'll see that many of the things commonly used are actually coming from the global object.

For example, when you use require() to import a module you're actually using global.require(). Another example is the global object itself -- the global object has a global property which references itself, so you can do something like global.global.global.global… forever.

## Module

The module objects contains information on the current module such as the file it comes from and any modules it has loaded. So for example, if I loaded a module and then queried the module object, I would see the module I loaded listed as a child…

```
> require('./index.js')
5
7
{}
> module
Module {
  id: '<repl>',
  exports: {},
  parent: undefined,
  filename: null,
  loaded: false,
  children:
   [ Module {
       id: '/home/user/test/index.js',
       exports: {},
       parent: [Circular],
       filename: '/home/user/test/index.js',
       loaded: true,
       children: [Array],
       paths: [Array] } ],
  paths:
   [ '/home/user/test/repl/node_modules',
     '/home/user/test/node_modules',
     '/home/user/node_modules',
     '/home/node_modules',
     '/node_modules',
     '/home/user/.node_modules',
     '/home/user/.node_libraries',
     '/home/user/.nvm/versions/node/v8.9.1/lib/node' ] }
```

## Process

The process object let you interface with the current process as well as other processes on the system. For example, you can use process to get the arguments passed into your app, get access to stdin/stdout/stderr, access environment variables, or kill another process...

```
process {
  title: 'node',
  version: 'v8.9.1',
...
  stdout: [Getter],
  stderr: [Getter],
  stdin: [Getter],
  openStdin: [Function],
  exit: [Function],
  kill: [Function],
...
}
```

## Os

The os object lets you access particulars of the operating system. For example, you can use os to get the amount of total RAM, the amount of free RAM, the number of cores, etc…

```
> os
{ arch: { [Function: arch] [Symbol(Symbol.toPrimitive)]: [Function] },
  cpus: [Function: cpus],
  EOL: '\n',
  endianness: { [Function: endianness] [Symbol(Symbol.toPrimitive)]:
[Function] },
  freemem: { [Function: getFreeMem] [Symbol(Symbol.toPrimitive)]:
[Function] },
  homedir: { [Function: getHomeDirectory] [Symbol(Symbol.toPrimitive)]:
[Function] },
  hostname: { [Function: getHostname] [Symbol(Symbol.toPrimitive)]:
[Function] },
  loadavg: [Function: loadavg],
  networkInterfaces: [Function: networkInterfaces],
  platform: { [Function: platform] [Symbol(Symbol.toPrimitive)]: [Function]
},
  release: { [Function: getOSRelease] [Symbol(Symbol.toPrimitive)]:
[Function] },
  tmpdir: { [Function: tmpdir] [Symbol(Symbol.toPrimitive)]: [Function] },
```

```
...
}
```

## Buffer

Buffer objects are essentially encapsulations around raw binary data that seem to borrow heavily from java.nio.ByteBuffer. Buffer objects in NodeJS are allocated outside of the V8 heap, just like ByteBuffer.allocateDirect().

The common pattern with a lot of I/O operations (e.g. file reads or socket reads) is that if you don't specify a character encoding (e.g. UTF-8), you'll get back a Buffer object instead of a string.

```
> buffer
{ Buffer:
   { [Function: Buffer]
     poolSize: 8192,
     from: [Function],
     alloc: [Function],
     allocUnsafe: [Function],
     allocUnsafeSlow: [Function],
     isBuffer: [Function: isBuffer],
     compare: [Function: compare],
     isEncoding: [Function],
     concat: [Function],
     byteLength: [Function: byteLength],
     [Symbol(node.isEncoding)]: [Function] },
  SlowBuffer: [Function: SlowBuffer],
  INSPECT_MAX_BYTES: 50,
  kMaxLength: 2147483647,
  constants: { MAX_LENGTH: 2147483647, MAX_STRING_LENGTH: 268435440 },
  kStringMaxLength: 268435440,
  transcode: [Function: transcode] }
```

# Experimental Features

NodeJS uses V8 for its Javascript engine. V8 comes with experimental JS features, but they aren't enabled by default. If you want to enable these features you need to use the relevant --harmony flags when running NodeJS…

```
~ $ node --v8-options | grep "harmony"
```

```
  --es_staging (enable test-worthy harmony features (for internal use
only))
  --harmony (enable all completed harmony features)
  --harmony_shipping (enable all shipped harmony features)
  --harmony_array_prototype_values (enable "harmony Array.prototype.values"
(in progress))
  --harmony_function_sent (enable "harmony function.sent" (in progress))
  --harmony_sharedarraybuffer (enable "harmony sharedarraybuffer" (in
progress))
  --harmony_do_expressions (enable "harmony do-expressions" (in progress))
  --harmony_class_fields (enable "harmony public fields in class literals"
(in progress))
  --harmony_async_iteration (enable "harmony async iteration" (in
progress))
  --harmony_promise_finally (enable "harmony Promise.prototype.finally" (in
progress))
  --harmony_number_format_to_parts (enable "Intl.NumberFormat.prototype."
"formatToParts" (in progress))
  --harmony_function_tostring (enable "harmony
Function.prototype.toString")
  --harmony_regexp_dotall (enable "harmony regexp dotall flag")
  --harmony_regexp_lookbehind (enable "harmony regexp lookbehind")
  --harmony_regexp_named_captures (enable "harmony regexp named captures")
  --harmony_regexp_property (enable "harmony unicode regexp property
classes")
  --harmony_strict_legacy_accessor_builtins (enable "treat __defineGetter__
and related functions as strict")
  --harmony_template_escapes (enable "harmony invalid escapes in tagged
template literals")
  --harmony_restrict_constructor_return (enable "harmony disallow non
undefined primitive return value from class " "constructor")
  --harmony_dynamic_import (enable "harmony dynamic import")
  --harmony_restrictive_generators (enable "harmony restrictions on
generator declarations")
  --harmony_object_rest_spread (enable "harmony object rest spread
properties"
```