Node.js by Example

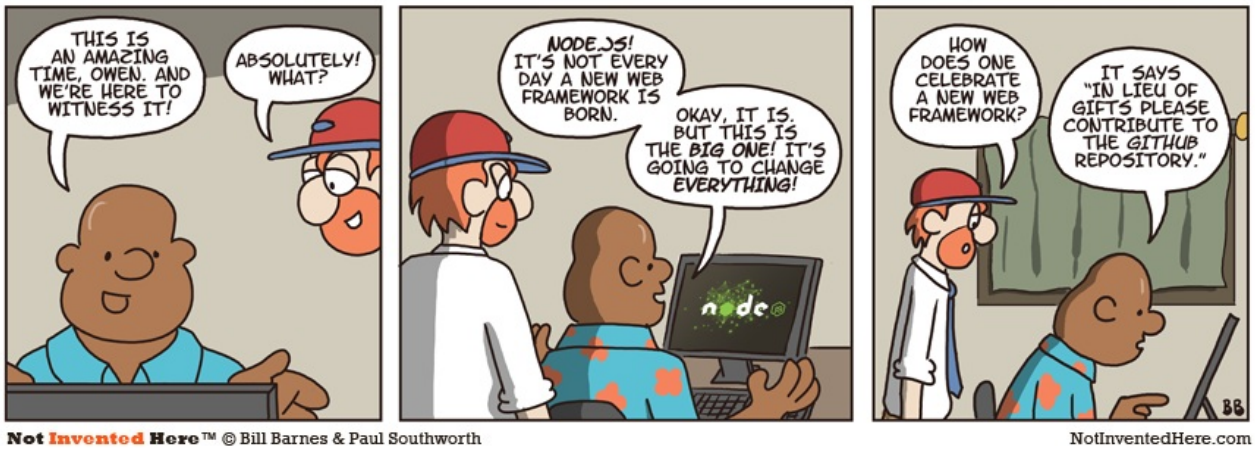
Nelson

Published with GitBook

Table of Contents

Learn Node.js by Example

This *Book* is available *free* online at: http://nelsonic.gitbooks.io/node-js-by-example/ in MOBI (Kindle), PDF (everything) and ePUB (iBooks).



Requirements

- [x] A computer with internet access
- [ ] Time: 30h (e.g. 2 months 30 mins per day or **1 week intensive**)

What is Node.js ?

Node.js lets you *easily* build networked software (websites, applications "apps" using JavaScript).

Node.js is not "*point-and-click*" like WordPress, SquareSpace or Salesforce.com; you will need to *write* some "code". But as I will demonstrate, that's a *lot* easier than it sounds (if you're *new* to writing code), gives you more power/flexibility and puts you in full control.

Frequently Asked Questions (FAQ)

Q: I am *New* to Programming Should I *Start* with Node.js?

If you have *never* written any code of any sort before, I recommend you starting with: http://gitbookio.gitbooks.io/javascript/ You need to have **JavaScript** basics to make the most of Node.js

Q: Node.js is not "Version 1.0" yet can we used it in Production?

Yes! Some of the biggest organisations/companies in the world are using Node.js in Production systems.

A Short List of Companies using Node.js

Alibaba, Ajax.org, Box.com, British Sky Broadcasting (Sky/Now TV), CNN, Cloudup, Conde Nast, DirectTV, Dow Jones, eBay, Etsy, FeedHenry, GitHub, Google, Groupon, HBO, Help.com, HP, iTV, Joyent (duh!), Klout, LinkedIn, McDonalds, Medium, Mozilla, NetFlix, OpenTable, PayPal, Pearson, ~~Q~~, Revolt, Square, Tesco, ThomasCook, Trello, Uber, Voxer, Walmart, Wikimedia (in progress of moving to SOA with node!) Yahoo, Yammer, Yandex, Zendesk

Want more? See: http://nodejs.org/industry/ and
https://github.com/joyent/node/wiki/Projects,-Applications,-and-Companies-Using-Node

Try it!

Download & Install

> http://nodejs.org/download/

Node.js (Core) API

The node.js ("core") has many useful modules.

Bookmark: http://nodejs.org/api (you will come back to it)

Stability (Can we use it?)

> *Which node.js **core** package(s) can/should I use?*

Every core module has a "***Stability Index***" rating on the node.js API.

General rule: If you are being *paid* to write code that runs in node.js,
pick core modules/methods with stability **Stable**, **API Frozen** and **Locked**.

```
Stability: 0 - Deprecated
This feature is known to be problematic, and changes are
planned.  Do not rely on it.  Use of the feature may cause warnings.  Backwards
compatibility should not be expected.
```

```
Stability: 1 - Experimental
This feature was introduced recently, and may change
or be removed in future versions.  Please try it out and provide feedback.
If it addresses a use-case that is important to you, tell the node core team.
```

```
Stability: 2 - Unstable
The API is in the process of settling, but has not yet had
sufficient real-world testing to be considered stable. Backwards-compatibility
will be maintained if reasonable.
```

```
Stability: 3 - Stable
The API has proven satisfactory, but cleanup in the underlying
code may cause minor changes.  Backwards-compatibility is guaranteed.
```

```
Stability: 4 - API Frozen
This API has been tested extensively in production and is
unlikely to ever have to change.
```

```
Stability: 5 - Locked
Unless serious bugs are found, this code will not ever
change.  Please do not suggest changes in this area; they will be refused.
```

Avoid Playground Work / Production

Examples

- **cluster** is ***Experimental*** - don't use
- **domain** is ***Unstable*** - don't use
- **path** is ***Stable*** - use
- **events** is ***Frozen*** - use
- **assert** is ***Locked*** - use

Core Modules to Learn

- path
- os

Community Modules to Learn:

- jscs - code style checker
- q - promises library
- nd - view documentation for a module

About

This book is for everyone who wants to learn node.js but doesn't know where to start.

Node.js (Core) API

The node.js ("core") has many useful modules.

Bookmark: http://nodejs.org/api (you will come back to it)

We are going to cover the modules labeled **Stability**: **Stable**, **API Frozen** and **Locked**. Only use the ***Experimental*** or ***Unstable*** core modules when you want to try the *latest & greatest* and at your own risk; avoid using them on a project for a client (unless you get agreement *in writing* that they want to sail close to the wind for a specific user feature).

Assert lets you *test* your code

Prerequisites

- Node.js installed
- No prior knowledge

Skill Level

- Basic/Beginner

What?

Assert - *verb* - state a fact or belief confidently and forcefully. (dictionary definition)

When writing code we write *tests* to *automatically* check our code is working as we *expect* it to.

Assert is the most *rudimentary* way of writing tests. It provides no feedback when running your tests *unless* one fails.

The assert module has 11 methods but you will only (regularly) use a few of them: **assert.equal**, **assert.deepEqual** and **assert.throws**. Each are explained *with examples* below.

assert.fail(actual, expected, message, operator)

The first method (alphabetically), but the *least useful* for practical purposes (unless you expect a method/test to always fail).

Throws an exception that displays the values for actual and expected separated by the provided operator.

Example:

```
assert.fail(21, 42, 'Test Failed', '###')
```

Output:

```
  throw new assert.AssertionError({
        ^
AssertionError: Faild
    at Object.<anonymous> (assert.js:2:8)
    at Module._compile (module.js:456:26)
    at Object.Module._extensions..js (module.js:474:10)
    at Module.load (module.js:356:32)
    at Function.Module._load (module.js:312:12)
    at Function.Module.runMain (module.js:497:10)
    at startup (node.js:119:16)
    at node.js:906:3
```

Usefulness: virtually *none*. I haven't found a *practical* use for this.
http://nodejs.org/api/assert.html#assert_assert_fail_actual_expected_message_operator

assert(value, message), assert.ok(value, [message])

Tests if value is "truthy", it is equivalent to:

```
assert.equal(true, value, message);
```

The simplest assertion.

Example:

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

function add (a, b) {
  return a + b;
}

var expected = add(1,2);
assert( expected === 3, 'one plus two is three');
```

This will not have any output. If you want to see output, you need to make the test *fail*:

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

function add (a, b) {
  return a + b;
}

var expected = add(1,2);
assert( expected === 4, 'one plus two is three');
```

Output:

```
assert.js:92
  throw new assert.AssertionError({
        ^
AssertionError: one plus two is NOT four
    at Object.<anonymous> (/Users/n/code/node-js-by-example/core/assert/assert.js:8:1)
    at Module._compile (module.js:456:26)
```

`assert.ok(value, [message])` is essentially the same as `assert(value, message);`

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

function add (a, b) {
  return a + b;
}

var expected = add(1,2);
assert.ok( expected === 3, 'one plus two is three');
```

Again, no output because the test passes. To see some feedback, make the test fail.

Usefulness: ***universal***. **assert** can be used to test *any* code.

assert.equal(actual, expected, [message])

Tests shallow, coercive equality with the (double) equal comparison operator (==).

Why would you use assert.equal() instead of assert() ?

If you want to make your test clearer use assert.equal otherwise there is no benefit to the additional verbosity.

Example:

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

function add (a, b) {
  return a + b;
}

var expected = add(1,2);

// these three assertions are equivalent:
assert(expected == 3, 'one plus two is three');
assert.ok(expected == 3, 'one plus two is three');
assert.equal(expected, 3, 'one plus two is three');
```

assert.notEqual(actual, expected, [message])

Tests shallow, coercive non-equality with the not equal comparison operator (!=).

Example:

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

function add (a, b) {
  return a + b;
}

var expected = add(1,2);

// these three assertions are equivalent:
assert(expected != 4, 'one plus two is three');
assert.ok(expected != 4, 'one plus two is three');
assert.notEqual(expected, 4, 'one plus two is three (NOT Four!)');
```

Why would you use assert.notEqual(1, 2) instead of assert(1 != 2) ?

Again, verbosity/clarity in your tests.

assert.deepEqual(actual, expected, [message])

Tests for deep equality.

assert.deepEqual is the second of the *useful* methods. We use it for comparing two objects (or arrays for equality).

Example:

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

var list1 = [1, 2, 3, 4, 5];
var list2 = [1, 2, 3, 4, 5];

assert.deepEqual(list1, list2, 'deepEqual checks the elements in the arrays are identical');

var person1 = { "name":"john", "age":"21" };
var person2 = { "name":"john", "age":"21" };

// deepEqual checks the elements in the objects are identical
assert.deepEqual(person1, person2, 'these two objects are the same');
```

assert.notDeepEqual(actual, expected, [message])

Tests for any deep inequality. Useful when confirming two objects or arrays are not equal.

Example:

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

var list1 = [1, 2, ,3, 4, 5];
var list2 = [1, 2, 3, 4, 5];

assert.deepEqual(list1, list2, 'deepEqual checks the elements in the arrays are identical');

var person1 = { "name":"john", "age":"21" };
var person2 = { "name":"jane", "age":"19" };

// deepEqual checks the elements in the objects are identical
assert.notDeepEqual(person1, person2, 'these two objects are NOT the same');
```

assert.strictEqual(actual, expected, [message])

Tests strict equality, as determined by the strict equality operator (===)

Similar to **assert.equal** but "strict" (type coersion).

Example:

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

assert.strictEqual(1, '1', 'not the same'); // this will fail
```

assert.notStrictEqual(actual, expected, [message])

Tests strict non-equality, as determined by the strict not equal operator (`!==`)

The opposite of the strictEqual.

Example:

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

assert.notStrictEqual(1, true, 'not the same (strictly)');
```

assert.throws(block, [error], [message])

Expects block to throw an error. error can be constructor, RegExp or validation function.

The **assert.throws** lets you check for *specific* errors in your functions.

Validate instanceof using constructor:

Example:

```
assert.throws(
  function() {
    throw new Error("Wrong value");
  },
  Error
);
```

assert.doesNotThrow(block, [message])

Expects `block` not to throw an error, see `assert.throws` for details.

Example:

```
assert.doesNotThrow(
  function() {
    console.log("Nothing to see here");
  },
  Error
);
```

Not particularly useful method because its *too vague*. Its good to know your method did not throw an error under normal circumstances.

assert.ifError(value)

Tests if value is not a false value, throws if it is a true value. Useful when testing the first argument, error in callbacks.

Example:

```
// define a simple function with callback(err, value)
function sayHello(name, callback) {
  var error = false;
  var str   = "Hello "+name;
  callback(error, str);
}

// use the function
sayHello('World', function(err, value){
  assert.ifError(err);
  assert.equal(value, "Hello World");
})
```

Try it!

As with all code, you can't expect to learn without *trying*. Open the **assert.js** file in your editor and try a few examples. Remember you won't see any output unless your test *fails*. Run it with the command:

```
node assert.js
```

Useful Links

- Assert API: http://nodejs.org/api/assert.html
- Chris Johansen's intro to node testing: http://cjohansen.no/en/node_js/unit_testing_node_js_apps

In Practice (The "Real World")

In the "real world" people rarely use the node **assert** module by its' self. Instead a **test runner** is used.

Examples of Node.js test runners include:

- Mocha: https://github.com/visionmedia/mocha
- Tape: https://github.com/substack/tape
- Lab: https://github.com/hapijs/lab

Each of these has their merits and will be discussed in a later chapter on testing.

Buffers

Buffers are a series of bytes.

What does that mean?

If you are completely new to computer programming/science, here's a *quick intro* to Bits, Bytes and Octets.

Buffers

Pure JavaScript is *Unicode friendly* but not nice to binary data. When dealing with TCP streams or the file system, it's necessary to handle octet streams. Node has several ways of manipulating, creating, and consuming octet streams.

Buffers are used *internally* by Node.js but available to everyone.

Here's a simple example of creating a new **Buffer** containing the word "hello". You will notice that the buffer stores the the ***hexadecimal*** values for the characters.

new Buffer(size)

Allocates a new buffer of size octets.

Feeding **new Buffer()** a numeric parameter simply allocates that amount of memory to the buffer for later use.

```
var buf = new Buffer(10); // <Buffer 10 d6 33 00 01 00 00 00 00 c8>
var str = buf.toString(); // 3  (incomprehensible - non-printable chars)
```

Practical use? None. (If I find one I will let you know.)

new Buffer(array)

Allocates a new buffer using an array of octets.

Similar to the **new Buffer(number)** but allows you to pass in an array; node creates a placeholder for each element in the array in your new Buffer.

```
var buf = new Buffer(['hello','world']); // <Buffer 00 00>
var str = buf.toString(); // empty
```

Practical use? None.

new Buffer(str, [encoding])

- **str** *String* - string to encode.
- **encoding** *String* - encoding to use, Optional.

Allocates a new buffer containing the given str. encoding defaults to 'utf8'.

```
var buf = new Buffer('hello'); // <Buffer 68 65 6c 6c 6f>
var str = buf.toString();      // hello
```

Class Method: Buffer.isEncoding(encoding)

- **encoding** *String* The encoding string to test

Returns true if the encoding is a valid encoding argument, or false otherwise.

```
console.log("Buffer.isEncoding('utf8') >> " + Buffer.isEncoding('utf8') ); // true
```

Again, can't see a *practical* application for this.

buf.toJSON()

Returns a JSON-representation of the Buffer instance, which is identical to the output for JSON Arrays. JSON.stringify implicitly calls this function when stringifying a Buffer instance.

```
var buf = new Buffer('test');
var json = JSON.stringify(buf);

console.log(json); // '[116,101,115,116]'

var copy = new Buffer(JSON.parse(json));

console.log(copy); // <Buffer 74 65 73 74>
```

- http://nodejs.org/api/buffer.html#buffer_buffer
- Ascii to hex to binary conversion table: http://www.ascii-code.com

Bits

A **bit** is the basic unit of information in computing and digital communications usually expressed as 0 (Zero) or 1 (One). The word bit is short for **b**inary dig**it**. *Binary* means there are two options e.g: true/false, 0/1 or on/off

Counting in Binary (bits)

Starting at Zero, the first 16 numbers in Binary are:

```
+---+---+----+----+-----+-----+-----+-----+------+------+------+------+------+------+------+------+
| 0 | 1 |  2 |  3 |   4 |   5 |   6 |   7 |    8 |    9 |   10 |   11 |   12 |   13 |   14 |   15 |
+---+---+----+----+-----+-----+-----+-----+------+------+------+------+------+------+------+------+
| 0 | 1 | 10 | 11 | 100 | 101 | 110 | 111 | 1000 | 1001 | 1010 | 1011 | 1100 | 1101 | 1110 | 1111 |
+---+---+----+----+-----+-----+-----+-----+------+------+------+------+------+------+------+------+
```

Did you spot the pattern(s)? If not, I suggest you read up on counting binary: http://en.wikipedia.org/wiki/Binary_number

> watch: https://www.youtube.com/watch?v=VBDoT8o4q00 (how computers count)

Bytes

A **byte** is a unit of digital information in computing and telecommunications that most commonly consists of **eight bits**.

Hello! in Binary (8-Bit ASCII)

```
01001000 01100101 01101100 01101100 01101111 00100001
    H        e        l        l        o        !
```

Try it: http://text2binary.herokuapp.com/

Hexadecimal

Hexadecimal

In mathematics and computing, hexadecimal (also base 16, or hex) is a positional numeral system with a radix, or base, of 16. It uses sixteen distinct symbols, most often the symbols 0–9 to represent values zero to nine, and A, B, C, D, E, F (or alternatively a–f) to represent values ten to fifteen.