Nodejs Cheat Sheet

by raffi001 via cheatography.com/41042/cs/12495/

Query String

```
const querystring =
require('querystring');
querystring.parse('w=%D6%D0%CE%C4&
foo=bar', null, null,
  { decodeURIComponent:
gbkDecodeURIComponent });
querystring.stringify({ foo: 'bar',
baz: ['qux', 'quux'], corge: '' });
```

Server Example

```
const http = require('http');
const hostname = '127.0.0.1';
const port = 3000;
const server =
http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type',
'text/plain');
  res.end('Hello World\n');
});
server.listen(port, hostname, () =>
{
  console.log(Server running at
http://${hostname}:${port}/);
});
```

Cluster

```
const cluster = require('cluster');
const http = require('http');
const numCPUs =
require('os').cpus().length;
if (cluster.isMaster) {
  console.log(Master
${process.pid} is running);
  // Fork workers.
  for (let i = 0; i < numCPUs; i++)
{
    cluster.fork();
```

Cluster (cont)

```
  }
  cluster.on('exit', (worker, code,
signal) => {
    console.log(worker
${worker.process.pid} died);
  });
} else {
  // Workers can share any TCP
connection
  // In this case it is an HTTP
server
  http.createServer((req, res) => {
    res.writeHead(200);
    res.end('hello world\n');
  }).listen(8000);
  console.log(Worker
${process.pid} started);
}
```

A single instance of Node.js runs in a single thread. To take advantage of multi-core systems the user will sometimes want to launch a cluster of Node.js processes to handle the load.

The cluster module allows you to easily create child processes that all share server ports.

DNS

```
const dns = require('dns');
dns.lookup('nodejs.org', (err,
addresses, family) => {
  console.log('addresses:',
addresses);
});
const dns = require('dns');
dns.resolve4('archive.org', (err,
addresses) => {
  if (err) throw err;
```

DNS (cont)

```
  console.log(addresses:
${JSON.stringify(addresses)});
  addresses.forEach((a) => {
    dns.reverse(a, (err, hostnames)
=> {
      if (err) {
        throw err;
      }
      console.log(reverse for
${a}:
${JSON.stringify(hostnames)});
    });
  });
});
```

Globals

__dirname	__filename
clearImmediate(immediateObject)	clearInterval(intervalObject)
clearTimeout(timeoutObject)	console
exports	global
module	process
require()	require.cache
require.resolve()	setImmediate(callback[, ...args])
setInterval(callback, delay[, ...args])	setTimeout(callback, delay[, ...args])

http

```
const http = require('http');
const keepAliveAgent = new
http.Agent({ keepAlive: true });
options.agent = keepAliveAgent;
http.request(options,
onResponseCallback);
http.get({
  hostname: 'localhost',
  port: 80,
  path: '/',
  agent: false // create a new
agent just for this one request
}, (res) => {
  // Do stuff with response
});
```

Readline

```
const readline =
require('readline');
const rl =
readline.createInterface({
  input: process.stdin,
  output: process.stdout
});
rl.question('What do you think of
Node.js? ', (answer) => {
  // TODO: Log the answer in a
database
  console.log(Thank you for your
valuable feedback: ${answer});
  rl.close();
});
```

Assert

```
const assert = require('assert');
const obj1 = {
  a: {
    b: 1
  }
};
const obj2 = {
  a: {
    b: 2
  }
};
const obj3 = {
  a: {
    b: 1
  }
};
const obj4 = Object.create(obj1);
assert.deepEqual(obj1, obj1);
// OK, object is equal to itself
assert.deepEqual(obj1, obj2);
// AssertionError: { a: { b: 1 } }
deepEqual { a: { b: 2 } }
// values of b are different
assert.deepEqual(obj1, obj3);
// OK, objects are equal
assert.deepEqual(obj1, obj4);
// AssertionError: { a: { b: 1 } }
deepEqual {}
// Prototypes are ignored
```

Console

```
const out = getStreamSomehow();
const err = getStreamSomehow();
const myConsole = new
console.Console(out, err);
myConsole.log('hello world');
// Prints: hello world, to out
myConsole.log('hello %s', 'world');
// Prints: hello world, to out
myConsole.error(new Error('Whoops,
something bad happened'));
// Prints: [Error: Whoops,
something bad happened], to err
const name = 'Will Robinson';
myConsole.warn(Danger ${name}!
Danger!);
// Prints: Danger Will Robinson!
Danger!, to err
```

eRROR

```
try {
  const m = 1;
  const n = m + z;
} catch (err) {
  // Handle the error here.
}
const fs = require('fs');
  fs.readFile('a file that does not
exist', (err, data) => {
    if (err) {
      console.error('There was an
error reading the file!', err);
      return;
    }
```

eRROR (cont)

```
    // Otherwise handle the data
  });
const net = require('net');
const connection =
net.connect('localhost');
// Adding an 'error' event handler
to a stream:
connection.on('error', (err) => {
  // If the connection is reset by
the server, or if it can't
  // connect at all, or on any sort
of error encountered by
  // the connection, the error will
be sent here.
  console.error(err);
});
connection.pipe(process.stdout);
```

https

```
// curl -k https://localhost:8000/
const https = require('https');
const fs = require('fs');
const options = {
  key:
fs.readFileSync('test/fixtures/keys
/agent2-key.pem'),
  cert:
fs.readFileSync('test/fixtures/keys
/agent2-cert.pem')
};
https.createServer(options, (req,
res) => {
  res.writeHead(200);
  res.end('hello world\n');
}).listen(8000);
```

Stream

```
const http = require('http');
const server =
http.createServer((req, res) => {
  // req is an
http.IncomingMessage, which is a
Readable Stream
  // res is an
http.ServerResponse, which is a
Writable Stream
  let body = '';
  // Get the data as utf8 strings.
  // If an encoding is not set,
Buffer objects will be received.
  req.setEncoding('utf8');
  // Readable streams emit 'data'
events once a listener is added
  req.on('data', (chunk) => {
    body += chunk;
  });
  // the end event indicates that
the entire body has been received
  req.on('end', () => {
    try {
      const data =
JSON.parse(body);
      // write back something
interesting to the user:
      res.write(typeof data);
      res.end();
    } catch (er) {
      // uh oh! bad json!
      res.statusCode = 400;
      return res.end( error:
${er.message});
    }
```

Stream (cont)

```
  });
});
server.listen(1337);
```

Buffer

```
// Creates a zero-filled Buffer of
length 10.
const buf1 = Buffer.alloc(10);
// Creates a Buffer of length 10,
filled with 0x1.
const buf2 = Buffer.alloc(10, 1);
// Creates an uninitialized buffer
of length 10.
// This is faster than calling
Buffer.alloc() but the returned
// Buffer instance might contain
old data that needs to be
// overwritten using either fill()
or write().
const buf3 =
Buffer.allocUnsafe(10);
// Creates a Buffer containing
[0x1, 0x2, 0x3].
const buf4 = Buffer.from([1, 2,
3]);
// Creates a Buffer containing
UTF-8 bytes [0x74, 0xc3, 0xa9,
0x73, 0x74].
const buf5 = Buffer.from('tést');
// Creates a Buffer containing
Latin-1 bytes [0x74, 0xe9, 0x73,
0x74].
const buf6 = Buffer.from('tést',
'latin1');
```

Events

```
const myEmitter = new MyEmitter();
myEmitter.on('event', function(a,
b) {
  console.log(a, b, this);
  // Prints:
```

Events (cont)

```
  // a b MyEmitter {
  // domain: null,
  // _events: { event: [Function]
},
  // _eventsCount: 1,
  // _maxListeners: undefined }
});
myEmitter.emit('event', 'a', 'b');
```

File System

```
fs.open('myfile', 'wx', (err, fd)
=> {
  if (err) {
    if (err.code === 'EEXIST') {
      console.error('myfile already
exists');
      return;
    }
    throw err;
  }
  writeMyData(fd);
});
fs.watch('./tmp', {encoding:
'buffer'}, (eventType, filename) =>
{
  if (filename)
    console.log(filename);
    // Prints: <Buffer ...>
});
```

Child Process

```
const spawn =
require('child_process').spawn;
const ls = spawn('ls', ['-lh',
'/usr']);
ls.stdout.on('data', (data) => {
  console.log(stdout: ${data});
});
ls.stderr.on('data', (data) => {
  console.log(stderr: ${data});
});
ls.on('close', (code) => {
  console.log(child process exited
with code ${code});
});
```

The child_process.spawn() method spawns the child process asynchronously, without blocking the Node.js event loop. The child_process.spawnSync() function provides equivalent functionality in a synchronous manner that blocks the event loop until the spawned process either exits or is terminated.