Mastering Node

Node is an exciting new platform developed by *Ryan Dahl*, allowing JavaScript developers to create extremely high performance servers by leveraging Google's V8 JavaScript engine, and asynchronous I/O. In *Mastering Node* we will discover how to write high concurrency web servers, utilizing the CommonJS module system, node's core libraries, third party modules, high level web development and more.

Installing Node

In this chapter we will be looking at the installation and compilation of node. Although there are several ways we may install node, we will be looking at homebrew, nDistro, and the most flexible method of course, compiling from source.

Homebrew

Homebrew is a package management system for *OSX* written in Ruby, is extremely well adopted, and easy to use. To install node via the `brew` executable simply run:

```
$ brew install node.js
```

nDistro

nDistro is a distribution toolkit for node, which allows creation and installation of node distros within seconds. An *nDistro* is simply a dotfile named *.ndistro* which defines module and node binary version dependencies. In the example below we specify the node binary version *0.1.102*, as well as several 3rd party modules.

```
node 0.1.102
module senchalabs connect
module visionmedia express 1.0.0beta2
module visionmedia connect-form
module visionmedia connect-redis
module visionmedia jade
module visionmedia ejs
```

Any machine that can run a shell script can install distributions, and keeps dependencies defined to a single directory structure, making it easy to maintain an deploy. nDistro uses pre-compiled node binaries making them extremely fast to install, and module tarballs which are fetched from GitHub via *wget* or *curl* (auto detected).

To get started we first need to install nDistro itself, below we *cd* to our bin directory of choice, *curl* the shell script, and pipe the response to *sh* which will install nDistro to the current directory:

```
$ cd /usr/local/bin && curl http://github.com/visionmedia/ndistro/raw/master/install | sh
```

Next we can place the contents of our example in *./.ndistro*, and execute *ndistro* with no arguments, prompting the program to load the config, and start installing:

```
$ ndistro
```

Installation of the example took less than 17 seconds on my machine, and outputs the following *stdout* indicating success, not bad for an entire stack!

```
... installing node-0.1.102-i386
... installing connect
... installing express 1.0.0beta2
... installing bin/express
... installing connect-form
... installing connect-redis
... installing jade
... installing bin/jade
... installing ejs
... installation complete
```

Building From Source

To build and install node from source, we first need to obtain the code. The first method of doing so is via `git`, if you have git installed you can execute:

```
$ git clone http://github.com/ry/node.git && cd node
```

For those without *git*, or who prefer not to use it, we can also download the source via *curl*, *wget*, or similar:

```
$ curl -# http://nodejs.org/dist/node-v0.1.99.tar.gz > node.tar.gz
$ tar -zxf node.tar.gz
```

Now that we have the source on our machine, we can run `./configure` which discovers which libraries are available for node to utilize such as *OpenSSL* for transport security support, C and C++ compilers, etc. `make` which builds node, and finally `make install` which will install node.

```
$ ./configure && make && make install
```

CommonJS Module System

CommonJS is a community driven effort to standardize packaging of JavaScript libraries, known as *modules*. Modules written which comply to this standard provide portability between other compliant frameworks such as narwhal, and in some cases even browsers.

Although this is ideal, in practice modules are often not portable due to relying on apis that are currently only provided by, or are tailored to node specifically. As the framework matures, and additional standards emerge our modules will become more portable.

Creating Modules

Lets create a utility module named *utils*, which will contain a `merge()` function to copy the properties of one object to another. Typically in a browser, or environment without CommonJS module support, this may look similar to below, where `utils` is a global variable.

```
var utils = {};
  utils.merge = function(obj, other) {};
```

Although namespacing can lower the chance of collisions, it can still become an issue, and when further namespacing is applied it can look flat-out silly. CommonJS modules aid in removing this issue by "wrapping" the contents of a JavaScript file with a closure similar to what is shown below, however more pseudo globals are available to the module in addition to `exports`, `require`, and `module`. The `exports` object is then returned when a user invokes `require('utils')`.

```
var module = { exports: {}};
  (function(module, exports){
      function merge(){};
      exports.merge = merge;
  })(module, module.exports);
```

First create the file *./utils.js*, and define the `merge()` function as seen below. The implied anonymous wrapper function shown above allows us to seemingly define globals, however these are not accessible until exported.

```
 function merge(obj, other) {
      var keys = Object.keys(other);
      for (var i = 0, len = keys.length; i < len; ++i) {
          var key = keys[i];
          obj[key] = other[key];
      }
      return obj;
  };

  exports.merge = merge;
```

The typical pattern for public properties is to simply define them on the `exports` object like so:

```
exports.merge = function(obj, other) {
    var keys = Object.keys(other);
    for (var i = 0, len = keys.length; i < len; ++i) {
        var key = keys[i];
        obj[key] = other[key];
    }
    return obj;
};
```

Next we will look at utilizing out new module in other libraries.

Requiring Modules

There are four main ways to require a module in node, first is the *synchronous* method, which simply returns the module's exports, second is the *asynchronous* method which accepts a callback, third is the *asynchronous http* method which can load remote modules, and lastly is requiring of shared libraries or "node addons" which we will cover later.

To get started create a second file named *./app.js* with the code shown below. The first line `require('./utils')` fetches the contents of *./utils.js* and returns the `exports` of which we later utilize our `merge()` method and display the results of our merged object using `console.dir()`.

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

var a = { one: 1 };
var b = { two: 2 };
utils.merge(a, b);
console.dir(a);
```

Core modules such as the *sys* which are bundled with node can be required without a path, such as `require('sys')`, however 3rd-party modules will iterate the `require.paths` array in search of a module matching the given path. By default `require.paths` includes *~/.node_libraries*, so if *~/.node_libraries/*utils.js_ exists we may simply `require('utils')`, instead of our relative example `require('./utils')` shown above.

Node also supports the concept of *index* JavaScript files. To illustrate this example lets create a *math* module that will provide the `math.add()`, and `math.sub()` methods. For organizational purposes we will keep each method in their respective *./math/add.js* and *./math/sub.js* files. So where does *index.js* come into play? we can populate *./math/index.js* with the code shown below, which is used when `require('./math')` is invoked, which is conceptually identical to invoking `require('./math/index')`.

```
module.exports = {
    add: require('./add'),
    sub: require('./sub')
};
```

The contents of *./math/add.js* show us a new technique, here we use `module.exports` instead of `exports`. Previously mentioned was the fact that `exports` is not the only object exposed to the module file when evaluated, we also have access to `__dirname`, `__filename`, and `module` which represents the current module. Here we simply define the module export object to a new object, which happens to be a function.

```
module.exports = function add(a, b){
    return a + b;
};
```

This technique is usually only helpful when your module has one aspect that it wishes to expose, be it a single function, constructor, string, etc. Below is an example of how we could provide the `Animal` constructor:

```
exports.Animal = function Animal(){};
```

which can then be utilized as shown:

```
var Animal = require('./animal').Animal;
```

if we change our module slightly, we can remove `.Animal`:

```
module.exports = function Animal(){};
```

which can now be used without the property:

```
var Animal = require('./animal');
```

Require Paths

We talked about `require.paths`, the `Array` utilized by node's module system in order to discover modules. By default node checks the following directories for modules:

- `<node binary>`/../../lib/node
- **$HOME**/.node_libraries
- **$NODE_PATH**

The **NODE_PATH** environment variable is much like **PATH**, as it allows several paths delimited by the colon (`:`) character.

Runtime Manipulation

Since `require.paths` is just an array, we can manipulate it at runtime in order to expose libraries. In our previous example we defined the libraries *./math/{add,sub}.js*, in which we would typically `require('./math')` or `require('./math/add')` etc. Another approach is to prepend or "unshift" a directory onto `require.paths` as shown below, after which we can simply `require('add')` since node will iterate the paths in order to try and locate the module.

```
require.paths.unshift(__dirname + '/math');

var add = require('add'),
    sub = require('sub');

console.log(add(1,2));
console.log(sub(1,2));
```

Pseudo Globals

As mentioned above, modules have several pseudo globals available to them, these are as follows:

- `require` the require function itself
- `module` the current `Module` instance
- `exports` the current module's exported properties
- `__filename` absolute path to the current module's file
- `__dirname` absolute path to the current module's directory

require()

Although not obvious at first glance, the `require()` function is actually re-defined for the current module, and calls an internal function `loadModule` with a reference to the current `Module` to resolve relative paths and to populate `module.parent`.

module

When we `require()` a module, typically we only deal with the module's `exports`, however the `module` variable references the current module's `Module` instance. This is why the following is valid, as we may re-assign the module's `exports` to any object, even something trivial like a string:

```
// css.js
module.exports = 'body { background: blue; }';
```

To obtain this string we would simply `require('./css')`. The `module` object also contains these useful properties:

- `id` the module's id, consisting of a path. Ex: `./app`
- `parent` the parent `Module` (which required this one) or `undefined`
- `filename` absolute path to the module
- `moduleCache` an object containing references to all cached modules

Asynchronous Require

Node provides us with an asynchronous version of `require()`, aptly named `require.async()`. Below is the sample example previously shown for our *utils* module, however non blocking. `require.async()` accepts a callback of which the first parameter `err` is `null` or an instanceof `Error`, and then the module exports. Passing the error (if there is one) as the first argument is an extremely common idiom in node for async routines.

```
require.async('./utils', function(err, utils){
    console.dir(utils.merge({ foo: 'bar' }, { bar: 'baz' }));
});
```

Requiring Over HTTP

Asynchronous requires in node also have the added bonus of allowing module loading via **HTTP** and **HTTPS**. To require a module via http all we have to do is pass a valid url as shown in the *sass* to *css* compilation example below:

```
var sassUrl = 'http://github.com/visionmedia/sass.js/raw/master/lib/sass.js',
    sassStr = ''
        + 'body\n'
        + '  a\n'
        + '    :color #eee';

require.async(sassUrl, function(err, sass){
    var str = sass.render(sassStr);
    console.log(str);
});
```

Outputs:

```
body a {
  color: #eee;}
```

Registering Module Compilers

Another cool feature that node provides us, is the ability to register compilers for a specific file extension. A good example of this is the CoffeeScript language, which is a ruby/python inspired language compiling to

vanilla JavaScript, and through the use of `require.registerExtension()` can do so in an automated fashion.

To illustrate it's usage, lets create a small (and useless) Extended JavaScript language, or "ejs" for our example which will live at *./compiler/example.ejs*, it's syntax will look like this:

```
::min(a, b) a < b ? a : b
::max(a, b) a > b ? a : b
```

which will be compiled to:

```
exports.min = function min(a, b) { return a < b ? a : b }
exports.max = function max(a, b) { return a > b ? a : b }
```

First lets create the module that will actually be doing the ejs to JavaScript compilation. In this example it is located at *./compiler/extended.js*, and exports a single method named `compile()`. This method accepts a string, which is the raw contents of what node is requiring, transformed to vanilla JavaScript via regular expressions.

```
exports.compile = function(str){
    return str
        .replace(/(\w+)\(/g, '$1 = function $1(')
        .replace(/\)(.+?)\n/g, '){ return $1 }\n')
        .replace(/::/g, 'exports.');
};
```

Next we have to "register" the extension to assign out compiler. As previously mentioned our compiler lives at *./compiler/extended.js* so we are requiring it in, and passing the `compile()` method to `require.registerExtension()` which simply expects a function accepting a string, and returning a string of JavaScript.

```
require.registerExtension('.ejs', require('./compiler/extended').compile);
```

Now when we require our example, the ".ejs" extension is detected, and will pass the contents through our compiler, and everything works as expected.

```
var example = require('./compiler/example');
console.dir(example)
console.log(example.min(2, 3));
console.log(example.max(10, 8));

// => { min: [Function], max: [Function] }
// => 2
// => 10
```

Globals

As we have learnt node's module system discourages the use of globals, however node provides a few important globals for use to utilize. The first and most important is the `process` global which exposes process manipulation such as signalling, exiting, the process id (pid), and more. Other globals help drive to be similar to other familiar JavaScript environments such as the browser, by providing a `console` object.

console

The `console` object contains several methods which are used to output information to *stdout* or *stderr*. Lets take a look at what each method does.

console.log()

The most frequently used console method is `console.log()` simply writing to *stdout* with a line feed (`\n`). Currently aliased as `console.info()`.

```
console.log('wahoo');
// => wahoo

console.log({ foo: 'bar' });
// => [object Object]
```

console.error()

Identical to `console.log()`, however writes to *stderr*. Aliased as `console.warn()` as well.

```
console.error('database connection failed');
```

console.dir()

Utilizes the *sys* module's `inspect()` method to pretty-print the object to *stdout*.

```
console.dir({ foo: 'bar' });
// => { foo: 'bar' }
```

console.assert()

Asserts that the given expression is truthy, or throws an exception.

```
console.assert(connected, 'Database connection failed');
```

process

The `process` object is plastered with goodies, first we will take a look at some properties that provide information about the node process itself.

process.version

The version property contains the node version string, for example "v0.1.103".

process.installPrefix

Exposes the installation prefix, in my case "*/usr/local*", as node's binary was installed to "*/usr/local/bin/node*".

process.execPath

Path to the executable itself "*/usr/local/bin/node*".

process.platform

Exposes a string indicating the platform you are running on, for example "darwin".

process.pid

The process id.

process.cwd()

Returns the current working directory, for example:

```
cd ~ && node
node> process.cwd()
"/Users/tj"
```

process.chdir()

Changes the current working directory to the path passed.

```
process.chdir('/foo');
```

process.getuid()

Returns the numerical user id of the running process.

process.setuid()

Sets the effective user id for the running process. This method accepts both a numerical id, as well as a string. For example both `process.setuid(501)`, and `process.setuid('tj')` are valid.

process.getgid()

Returns the numerical group id of the running process.

process.setgid()

Similar to `process.setuid()` however operates on the group, also accepting a numerical value or string representation. For example `process.setgid(20)` or `process.setgid('www')`.

process.env

An object containing the user's environment variables, for example:

```
{ PATH: '/Users/tj/.gem/ruby/1.8/bin:/Users/tj/.nvm/current/bin:/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:/usr/X1
```

```
, PWD: '/Users/tj/ebooks/masteringnode'
, EDITOR: 'mate'
, LANG: 'en_CA.UTF-8'
, SHLVL: '1'
, HOME: '/Users/tj'
, LOGNAME: 'tj'
, DISPLAY: '/tmp/launch-YCkT03/org.x:0'
, _: '/usr/local/bin/node'
, OLDPWD: '/Users/tj'
}
```

process.argv

When executing a file with the `node` executable `process.argv` provides access to the argument vector, the first value being the node executable, second being the filename, and remaining values being the arguments passed.

For example our source file *./src/process/misc.js* can be executed by running:

```
$ node src/process/misc.js foo bar baz
```

in which we call `console.dir(process.argv)`, outputting the following:

```
[ 'node'
, '/Users/tj/EBooks/masteringnode/src/process/misc.js'
, 'foo'
, 'bar'
, 'baz'
]
```

process.exit()

The `process.exit()` method is synonymous with the C function `exit()`, in which a exit code > 0 is passed indicating failure, or 0 to indicate success. When invoked the *exit* event is emitted, allowing a short time for arbitrary processing to occur before `process.reallyExit()` is called with the given status code.

process.on()

The process itself is an `EventEmitter`, allowing you to do things like listen for uncaught exceptions, via the *uncaughtException* event:

```
process.on('uncaughtException', function(err){
    console.log('got an error: %s', err.message);
    process.exit(1);
});

setTimeout(function(){
    throw new Error('fail');
}, 100);
```

process.kill()

`process.kill()` method sends the signal passed to the given *pid*, defaulting to **SIGINT**. In our example below we send the **SIGTERM** signal to the same node process to illustrate signal trapping, after which we output "terminating" and exit. Note that our second timeout of 1000 milliseconds is never reached.

```
process.on('SIGTERM', function(){
```

```
    console.log('terminating');
    process.exit(1);
});

setTimeout(function(){
    console.log('sending SIGTERM to process %d', process.pid);
    process.kill(process.pid, 'SIGTERM');
}, 500);

setTimeout(function(){
    console.log('never called');
}, 1000);
```

errno

The `process` object is host of the error numbers, these reference what you would find in C-land, for example `process.EPERM` represents a permission based error, while `process.ENOENT` represents a missing file or directory. Typically these are used within bindings to bridge the gap between c++ and JavaScript, however useful for handling exceptions as well:

```
if (err.errno === process.ENOENT) {
    // Display a 404 "Not Found" page
} else {
    // Display a 500 "Internal Server Error" page
}
```

Events

The concept of an "event" is crucial to node, and used greatly throughout core and 3rd-party modules. Node's core module *events* supplies us with a single constructor, *EventEmitter*.

Emitting Events

Typically an object inherits from *EventEmitter*, however our small example below illustrates the api. First we create an `emitter`, after which we can define any number of callbacks using the `emitter.on()` method which accepts the *name* of the event, and arbitrary objects passed as data. When `emitter.emit()` is called we are only required to pass the event *name*, followed by any number of arguments, in this case the `first` and `last` name strings.

```
var EventEmitter = require('events').EventEmitter;

var emitter = new EventEmitter;

emitter.on('name', function(first, last){
    console.log(first + ', ' + last);
});

emitter.emit('name', 'tj', 'holowaychuk');
emitter.emit('name', 'simon', 'holowaychuk');
```

Inheriting From EventEmitter

A perhaps more practical use of `EventEmitter`, and commonly used throughout node is to inherit from it. This means we can leave `EventEmitter`'s prototype untouched, while utilizing it's api for our own means of world domination!

To do so we begin by defining the `Dog` constructor, which of course will bark from time to time, also known as an *event*. Our `Dog` constructor accepts a `name`, followed by `EventEmitter.call(this)`, which invokes the `EventEmitter` function in context to the given argument. Doing this is essentially the same as a "super" or "parent" call in languages that support classes. This is a crucial step, as it allows `EventEmitter` to set up the `_events` property which it utilizes internally to manage callbacks.

```
var EventEmitter = require('events').EventEmitter;

function Dog(name) {
    this.name = name;
    EventEmitter.call(this);
}
```

Here we inherit from `EventEmitter`, so that we may use the methods provided such as `EventEmitter#on()` and `EventEmitter#emit()`. If the `__proto__` property is throwing you off, no worries! we will be touching on this later.

```
Dog.prototype.__proto__ = EventEmitter.prototype;
```

Now that we have our `Dog` set up, we can create .... simon! When simon barks we can let *stdout* know by calling `console.log()` within the callback. The callback it-self is called in context to the object, aka `this`.

```
var simon = new Dog('simon');
```

```
simon.on('bark', function(){
    console.log(this.name + ' barked');
});
```

Bark twice a second:

```
setInterval(function(){
    simon.emit('bark');
}, 500);
```

Removing Event Listeners

As we have seen event listeners are simply functions which are called when we `emit()` an event. Although not seen often we can remove these listeners by calling the `removeListener(type, callback)` method. In the example below we emit the *message* "foo bar" every `300` milliseconds, which has the callback of `console.log()`. After 1000 milliseconds we call `removeListener()` with the same arguments that we passed to `on()` originally. To compliment this method is `removeAllListeners(type)` which removes all listeners associated to the given *type*.

```
var EventEmitter = require('events').EventEmitter;

var emitter = new EventEmitter;

emitter.on('message', console.log);

setInterval(function(){
    emitter.emit('message', 'foo bar');
}, 300);

setTimeout(function(){
    emitter.removeListener('message', console.log);
}, 1000);
```

Buffers

To handle binary data, node provides us with the global `Buffer` object. Buffer instances represent memory allocated independently to that of V8's heap. There are several ways to construct a `Buffer` instance, and many ways you can manipulate it's data.

The simplest way to construct a `Buffer` from a string is to simply pass a string as the first argument. As you can see by the log output, we now have a buffer object containing 5 bytes of data represented in hexadecimal.

```
var hello = new Buffer('Hello');

console.log(hello);
// => <Buffer 48 65 6c 6c 6f>

console.log(hello.toString());
// => "Hello"
```

By default the encoding is "utf8", however this can be specified by passing as string as the second argument. The ellipsis below for example will be printed to stdout as the '&' character when in "ascii" encoding.

```
var buf = new Buffer('â ¦');
console.log(buf.toString());
// => â ¦

var buf = new Buffer('â ¦', 'ascii');
console.log(buf.toString());
// => &
```

An alternative method is to pass an array of integers representing the octet stream, however in this case functionality equivalent.

```
var hello = new Buffer([0x48, 0x65, 0x6c, 0x6c, 0x6f]);
```

Buffers can also be created with an integer representing the number of bytes allocated, after which we may call the `write()` method, providing an optional offset and encoding. As shown below we provide the offset of 2 bytes to our second call to `write()`, buffering "Hel", and then we continue on to write another two bytes with an offset of 3, completing "Hello".

```
var buf = new Buffer(5);
buf.write('He');
buf.write('l', 2);
buf.write('lo', 3);
console.log(buf.toString());
// => "Hello"
```

The `.length` property of a buffer instance contains the byte length of the stream, opposed to JavaScript strings which will simply return the number of characters. For example the ellipsis character 'â ¦' consists of three bytes, however the buffer will respond with the byte length, and not the character length.

```
var ellipsis = new Buffer('â ¦', 'utf8');

console.log('â ¦ string length: %d', 'â ¦'.length);
// => â ¦ string length: 1

console.log('â ¦ byte length: %d', ellipsis.length);
// => â ¦ byte length: 3

console.log(ellipsis);
```

```
// => <Buffer e2 80 a6>
```

When dealing with JavaScript strings, we may pass it to the `Buffer.byteLength()` method to determine it's byte length.

The api is written in such a way that it is String-like, so for example we can work with "slices" of a `Buffer` by passing offsets to the `slice()` method:

```
var chunk = buf.slice(4, 9);
console.log(chunk.toString());
// => "some"
```

Alternatively when expecting a string we can pass offsets to `Buffer#toString()`:

```
var buf = new Buffer('just some data');
console.log(buf.toString('ascii', 4, 9));
// => "some"
```

Streams

Streams are an important concept in node. The stream api is a unified way to handle stream-like data, for example data can be streamed to a file, streamed to a socket to respond to an HTTP request, or a stream can be read-only such as reading from *stdin*. However since we will be touching on stream specifics in later chapters, for now we will concentrate on the api.

Readable Streams

Readable streams such as an HTTP request inherit from `EventEmitter` in order to expose incoming data through events. The first of these events is the *data* event, which is an arbitrary chunk of data passed to the event handler as a `Buffer` instance.

```
req.on('data', function(buf){
    // Do something with the Buffer
});
```

As we know, we can call `toString()` a buffer to return a string representation of the binary data, however in the case of streams if desired we may call `setEncoding()` on the stream, after which the *data* event will emit strings.

```
req.setEncoding('utf8');
req.on('data', function(str){
    // Do something with the String
});
```

Another import event is the *end* event, which represents the ending of *data* events. For example below we define an HTTP echo server, simply "pumping" the request body data through to the response. So if we **POST** "hello world", our response will be "hello world".

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

http.createServer(function(req, res){
    res.writeHead(200);
    req.on('data', function(data){
        res.write(data);
    });
    req.on('end', function(){
        res.end();
    });
}).listen(3000);
```

The *sys* module actually has a function designed specifically for this "pumping" action, aptly named `sys.pump()`, which accepts a read stream as the first argument, and write stream as the second.

```
var http = require('http'),
    sys = require('sys');

http.createServer(function(req, res){
    res.writeHead(200);
    sys.pump(req, res);
}).listen(3000);
```

File System

...

TCP

...

TCP Servers

...

TCP Clients

...

HTTP

...

HTTP Servers

...

HTTP Clients

...

Connect

Connect is a ...

Express

Express is a ...

Testing

...

Expresso

...

Vows

...

Deployment

...