Mastering Node

TJ Holowaychuk, et al.

Table of Contents

Table of Contents

Mastering Node

Node is an exciting new platform developed by *Ryan Dahl*, allowing JavaScript developers to create extremely high performance servers by leveraging Google 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.

Why Node?

Node is an evented I/O framework for server-side JavaScript. What does that mean, though? In many programming languages, I/O operations are blocking. This means that when you open a file, no other code executes until that file is fully opened.

Using a busy office environment as an example, blocking I/O operations are like a very busy and very focussed desk worker name Frank. Work may pile up on the desk, but Frank finishes every task that he starts (however, when he finds the smallest problem, he gives up). This is traditionally how procedural programming languages perform tasks.

Node.js is more like an office manager. Sure, there are management tasks, but all of those I/O operations are handled fairly well by the operating system or node addons. So, the manager sees a task, assigns it to Frank. The manager may check back on Frank from time to time to see how he's doing, but he doesn't need to waste his time on the task because he knows that when it's completed, Frank will give it back.

Don't worry, though. Node.js isn't a jerk, so look forward to a healthy relationship.

Why JavaScript?

JavaScript is a well-known dynamic language.

In fact, many people who don't consider themselves programmers know JavaScript. Node.js uses Google's V8 JavaScript engine. V8 is an implementation of the JavaScript standards specified in ECMA-262, Version 3. It is a high-performance, garbage-collected execution environment which can be embedded in any stand-alone application.

Everything in JavaScript is an `Object`.

Arrays are `Objects`. Object literals are `Objects`. Even functions are `Objects`.

JavaScript supports some fairly advanced functional programming concepts. ?

Currying, closures, functions-as-arguments, and anonymous functions are concepts typically found in functional programming languages. There are libraries available, such as underscore.js, which provider more functional programming functionality without changing JavaScript itself.

It's awesome. I had to say it.

For a deep understanding of what JavaScript really has to offer, check out JavaScript Patterns.

Hello World!

Becuase it has to be done

```
// hello.js
sys = require('sys');
```

```
sys.puts("Hello, World!");
```

To run the 'Hello, World!' example, execute `node hello.js` on the command line.

Conventions in this book

Code Blocks

There are a few conventions used in the code example blocks (such as the Hello, World! example above).

First, if the example is a script (or a part of a script) that needs to be run from the command line, it will begin with a `//` styled comment of the filename. These files are located under the `./src` directory of the *Mastering Node* project. Running these examples is fairly simple: open a terminal (CTRL+SHIFT+T might do it) and type `node` followed by a space and the relative path of the script. In other words, if you're in */home/jim/masteringnode* and you want to run a script under */home/jim/masteringnode/scripts/* which is called *hello.js*, you would run `node scripts/hello.js` or `node ./scripts/hello.js`. For an example of how this code snippet is displayed, see the Hello, World! example above.

Second, if the example is a command to be entered directly into what will be referred to as a 'node terminal', you should first open a terminal then type `node` and ENTER (assuming the node executable is in your PATH variable, and is called node). At the prompt, you should enter the commands following the > character. Text following the > character are commands you should enter, while text on a line that does not begin with > is the output. This is done to closely match what you should see in your terminal. These examples are not included in files under the *src* directory. For instance:

```
> console.log("Hello, World!")
Hello, World.
```

Third, if an example is to be run from the terminal itself, it is prefixed by a `$`. This represents a bash environment, and may look differently depending on your configuration. For instance, my bash terminal displays:

```
jim@cr-48:~/projects/masteringnode$
```

These code examples will look like this:

```
$ ruby -e "puts 'Hello, World'"
```

Finally, code without a prefix is a sample of a possible solution or expected output from *stdout*. This is not necessarily code you should type into your terminal or run as a script, unless of course you want to experiment. For example, suppose we're discussing CommonJS and we'd like to show how something *would* be done in CommonJS, we might show an example such as:

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

This wouldn't contain a filename unless there is a working example (in which case, it's a concrete example and no longer hypothetical).

Inline Code

Occasionally, there will be code or commands that are being explained and displayed inline to separate them visually from the rest of the sentence. For instance, if we were talking about how to include modules into your code, we may say to add a `var fs = require('fs');` line to the top of your file. Code displayed like this is generally accompanied by an explanation of the code itself or instructions for where to add the code.

Functions may be displayed as `require()` or `require`, depending on the context for readability. That is, when we're talking about any JavaScript objects, we'll most likely use `require` since a function is also an object.

If we're talking about multiple functions, we may say "use the functions: `doSomething1` and `doSomething2`" instead of "use the functions: `doSomething1()` and `doSomething2()`" because the parentheses can be implied and the absence of parentheses is easier on the eyes.

Files and Directories

Files and directories will be displayed in italics when inline, or as a relative path to the *Mastering Node* source directory in code comments when in a code example block. For instance, if we reference */home/jim/masteringnode/src/modules/fake.js* in a code example, it will look like:

```
// modules/fake.js
var num = 1;
```

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, 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. Homebrew can be installed in a number of ways. Possibly the easiest way is to perform a quick install to `/usr/local/`:

```
$ ruby -e "$(curl -fsSL https://gist.github.com/raw/323731/install_homebrew.rb)"
```

Next, to install node via the `brew` executable, simply run:

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

For more information on packages and commands available to homebrew, checkout the README at github.

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/joyent/node.git && cd node
$ git checkout v0.4.0
```

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-v0.4.0.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 && sudo make install
```

Installing from Distribution Sources

Installing node.js from a distribution's repositories is not highly recommended. This is because the version included with your distribution may be very outdated. As an example, the original version of this document was written to target node.js 0.1.99. In Ubuntu 10.10, the version of node.js included in the official repository is 0.1.97-1build1. Because node.js is a relatively young project with a large community, changes to the API occur quickly and often.

If you still wish to install nodejs from an official repository, it can be done in the usual way:

```
$ sudo apt-get install nodejs
$ yum install nodejs
```

Refer to your distribution's documention or wiki for the proper commannd for installation from the official repository.

npm: Node Package Manager

npm is a node package manager with a relatively large category of available modules. To install directly in one line, run:

```
$ curl http://npmjs.org/install.sh | sh
```

If this one-liner fails, run:

```
$ git clone http://github.com/isaacs/npm.git
$ cd npm
$ sudo make install
```

For more information on npm, check out the repo.

Installing other packages

Using npm, we'll install other packages needed for the examples in this book.

```
$ sudo npm install connect express nodeunit geddy zombie node-inspector
```

Let npm do it's thing. It will install these modules any any dependencies. You'll see output similar to the following:

```
npm info activate jsdom@0.1.23
npm info postactivate jsdom@0.1.23
npm info build Success: express@1.0.7
npm info build Success: nodeunit@0.5.1
npm info build Success: mime@1.2.1
npm info build Success: geddy@0.1.3
npm info build Success: websocket-client@1.0.0
npm info build Success: connect@1.0.1
```

Setting up an editor

Part of setting up a development environment is using the right tools. While some may prefer a full IDE, others will go with a very simple text editor. I enjoy using *vim*, and this is my *~/.vimrc* file's contents:

```
syntax on " enabled syntax highlighting
:set number " line numbers
:set ai " autoindent
:set tabstop=4 " sets tabs to 4 characters
:set shiftwidth=4
:set expandtab
:set softtabstop=4 " makes the spaces feel like real tabs

" CSS (tabs = 2, lines = 79)
autocmd FileType css set omnifunc=csscomplete#CompleteCSS
autocmd FileType css set sw=2
autocmd FileType css set ts=2
autocmd FileType css set sts=2

" JavaScript (tabs = 4, lines = 79)
autocmd FileType javascript set omnifunc=javascriptcomplete#CompleteJS
autocmd FileType javascript set sw=4
autocmd FileType javascript set ts=4
autocmd FileType javascript set sts=4
" autocmd FileType javascript set tw=79

" Jade
autocmd FileType jade set omnifunc=jadecomplete#CompleteJade
```

```
autocmd FileType jade set sw=2
autocmd FileType jade set ts=2
autocmd FileType jade set sts=2

" ejs
autocmd FileType ejs set sw=2
autocmd FileType ejs set ts=2
autocmd FileType ejs set sts=2

" Highlight current line only in insert mode
autocmd InsertLeave * set nocursorline
autocmd InsertEnter * set cursorline

" Makefiles require TAB instead of whitespace
autocmd FileType make setlocal noexpandtab

" Highlight cursor
highlight CursorLine ctermbg=8 cterm=NONE
```

If your install of *vim* doesn't include the **complete* functions, you can download them from vim.org/scripts. Most likely, you'll need the jade.vim script.

Other excellent IDEs include:

1. cloud9 IDE
2. geany
3. Aptana Studio

Debugging

One of the most important aspects of programming in any language is the ability to debug code.

node-inspector

node-inspector is a Web Inspector node.js debugger. This means you'll need a WebKit browser like Chrome or Safari to run node-inspector.

Installation

If you've performed the npm installation from the previous section, you should be ready to go. If not, you can install node-inspector from source or through npm.

```
$ npm install node-inspector
```

If you'd like to install from source, check out the Getting Started from scratch wiki entry from the node-inspector repository.

Usage

Open one terminal to host a node-inspector server. This terminal will remain open and provide a debug listener.

```
$ node-inspector&
> visit http://0.0.0.0:8080/debug?port=5858 to start debugging
```

You will see output directing you to open a browser and point it to the node-inspector server at the default port (8080).

Next, open another terminal and navigate to `./src/events` and run node with the `--debug-brk` switch. This will connect to the debugger on port 5858.

```
$ node --debug-brk basic.js
> debugger listening on port 5858
```

Now, open a browser to http://127.0.0.1:8080/debug?port=5858 and you will see a breakpoint on what would be the first line of your file. I phrased it like this intentionally because the debugger shows you how node.js sees your file:

```
(function (exports, require, module, filename, dirname) {
/* Your code here */
});
```

If you've used the Web Inspector view in Google Chrome, you should be familiar with using node-inspector's version of Web Inspector. There is a console drawer if you click on the bottom-left icon which allows you to evaluate JavaScript and filter log messages by type (Errors, Warnings, Logs). There is also a 'Console' tab if you need a full view of the console.

Expand the `Watch Expressions` section on the right of the Web Inspector's Scripts tab. Click the 'Add' button and you can add variables which will be updated as you step through code. In version 0.1.6 of node-inspector, when I click the 'Add' button, it automatically adds two single-quotes. If this happens to you, delete the quotes and enter your text.

You'll see as you step through the code that you step directly into node.js functions. This is very useful if you are exploring node.js or if you want to see how something is implemented, like when you call something like `console.log("%d days without incident", 5);`.

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

Let's 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 *./modules/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:

```
// modules/utils.js
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

To get started with requiring modules, first create a second file named *./modules/app.js* with the code shown below. The first line `require('./utils')` fetches the contents of *./modules/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()`.

```
// modules/app.js
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 *./modules/math/add.js* and *./modules/math/sub.js* files. So where does *index.js* come into play? we can populate *./modules/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')`.

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

The contents of *./modules/math/add.js* show us a new technique; here we use `module.exports` instead of `exports`. As previously mentioned, `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. We simply define the module export object to a new object, which happens to be a function.

```
// modules/math/add.js
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

To examine the functionality of `require`, `module`, and `exports`, open a node console by running the command `node`. You should then enter a node prompt as seen below:

```
$ node
>
```

To view these objects, enter the following commands into node:

```
> require
> module
> module.exports
> module.filename
```

If you'd like to examine other objects, the node console supports the well-known `<TAB><TAB>` auto-completion.

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:

```
// modules/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

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. By using `require.registerExtension()` we can have node compile CoffeeScript to JavaScript in an automated fashion.

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

```
// modules/compiler/example.ejs
::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 let's create the module that will actually be doing the ejs to JavaScript compilation. In this example it is located at *./modules/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.

```
// modules/compiler/extended.js
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 *./modules/compiler/extended.js* so we are requiring it in. Prior to node.js 0.3.0, we would pass 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);
```

The new way to register an extension is to add a key to the require.extensions object with a function which specifies how to process the file. For compatibility, we can use `require.extensions` and fallback to `require.registerExtension`.

```
// modules/compiler.js
if(require.extensions) {
  require.extensions['.ejs'] = function(module,filename){
    var content = require('fs').readFileSync(filename, 'utf8');
    var newContent = require('./compiler/extended').compile(content);
    module._compile(newContent, filename);
  };
} else {
  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.

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

This should display the following output when run with `node ./src/modules/compiler.js` in a terminal:

```
$ node compiler.js
{ min: [Function: min], max: [Function: max] }
2
10
```

Patterns

...

Best Practices

...

Addons

The node documentation for addons is self-admittedly pretty weak as of v0.4.0.

This chapter doesn't aim to be a replacement for the official documentation. Instead, we'd hope this can expand on some of the basics a little more than a simple "Hello, World!" and drive you as a developer more on the path toward mastering node through its source code.

In fact, for now, we're only going to cover some shortcuts for creating properties on an object, functions, and interacting with function prototypes. This doesn't reach the evented level of node's awesomeness, but you should be able to look at examples in node's source and the documention for *libev* and *libeio* to find answers.

Pre-requisites

- Some C/C++ knowledge
- V8 JavaScript
- Internal Node libraries
- libev
- libeio

hello.node

Our first example is the very same one from node's docs. We're going to include it for those who haven't read through the docs and have instead trusted in the knowledge of this ebook's authors (thanks, by the way).

A node addon begins with a source file containing a single entry point:

```
// addons/hello/hello.cc
#include <v8.h>

using namespace v8;

extern "C" void
init (Handle<Object> target)
{
  HandleScope scope;
  target->Set(String::New("hello"), String::New("world"));
}
```

This is a C/C++ file which begins by including the *v8.h* header. It then uses the *v8* namespace to make the code a little cleaner. Finally, the entry point accepts a single parameter, `Handle<Object> target`. If you don't use the *v8* namespace as we've done here, your parameter will read `v8::Handle<v8::Object> target`. As you can see, `Object` is a *v8* class. This is actually the same object we would otherwise refer to using `exports` in a regular node source file.

Within the `init` method, we do two things. First, we create a scope. Again, this is the same as is done with the `exports` object in a JavaScript file. Then we set a property on the `target` called `hello` which returns the string "world".

If you were to perform the same actions as this source file (ignoring the scope part) in a node REPL console, it would look like:

```
$ node
> var target = {};
> target.hello = "world";
'world'
```

Building hello.node

Building a node addon requires WAF. *node-waf* should be installed if you've installed node.js from source or distro.

```
// addons/hello/wscript
srcdir = '.'
blddir = 'build'
VERSION = '0.0.1'

def set_options(opt):
  opt.tool_options('compiler_cxx')

def configure(conf):
  conf.check_tool('compiler_cxx')
  conf.check_tool('node_addon')

def build(bld):
  obj = bld.new_task_gen('cxx', 'shlib', 'node_addon')
  obj.target = 'hello'
  obj.source = 'hello.cc'
```

This file is relatively simple, there are two tasks: *configure* and *build*. To run this script:

```
$ cd src/addons/hello/wscript && node-waf configure build
```

Notice how you have to run node-waf from the directory of the `wscript` build script. Afterward, your built addon will be located at *./build/default/hello.node*. You can load and play with the addon when you're finished.

```
$ cd build/default && node
> var hello = require('./hello.node');
> hello
{ hello: 'world' }
>
```

Basic Functions

The world of functions begins with an object that will resemble the following object:

```
{ version: '0.1', echo: [Function] }
```

As you can imagine, node is already coding functions in the same way as you would for an addon. Luckily, *node.h* provides a helper definition for setting a function to a callback, or method defined within your source file. If you open *node.h* and look at line 33, you'll see:

```
// [node repo]/src/node.h
#define NODE\_SET\_METHOD(obj, name, callback)                            \
  obj->Set(v8::String::NewSymbol(name),                                  \
           v8::FunctionTemplate::New(callback)->GetFunction())
```

Let's reuse this bit of code. One thing to note here is that since node is already defining how to set a method on a target object, changes to this functionality in v8 will most likely be reflected in this macro. Because the addon compilation process links node anyway, including the header here shouldn't be an issue. Plus, reusing the macro ensures that we're creating functions in the same way as the rest of the framework.

The following example will compile our desired object mentioned earlier.

```
// addons/functions/v0.1/func.cc
#include <v8.h>
```

```
#include <node.h>

using namespace v8;

static Handle<Value> Echo(const Arguments& args) {
  HandleScope scope;

  if (args.Length() < 1) {
    return ThrowException(Exception::TypeError(String::New("Bad argument")));
  }
  return scope.Close(args[0]);
}

extern "C" void
init (Handle<Object> target)
{
    HandleScope scope;

    target->Set(String::New("version"), String::New("0.1"));

    NODE_SET_METHOD(target, "echo", Echo);
}
```

This example differs in a couple of ways from the *Hello, World!* example. First, it includes the *node.h* header containing the `NODE_SET_METHOD` macro. Second, this contains a callback which is providing the functionality of our function.

Just as you would expect in a JavaScript function, the entrance and exit points of the function define a `scope` in which the *context* of the function executes (i.e. 'this'). Our function will throw an error if it doesn't receive the proper number of arguments, then returns the first argument regardless of how many others are passed.

The *WAF* script used to build this file is nearly identical to the one used for the *Hello, World!* example.

Now, build the source and toy with it. Here's a dump of my console, run from *./src/addons/functions/v0.1*.

```
$ node-waf configure build && cd build/default && node
Checking for program g++ or c++          : /usr/bin/g++
Checking for program cpp                 : /usr/bin/cpp
Checking for program ar                  : /usr/bin/ar
Checking for program ranlib              : /usr/bin/ranlib
Checking for g++                         : ok
Checking for node path                   : ok /home/jim/.node_libraries
Checking for node prefix                 : ok /usr/local
'configure' finished successfully (0.164s)
Waf: Entering directory `/media/16GB/projects/masteringnode/src/addons/functions/v0.1/build'
[1/2] cxx: func.cc -> build/default/func_1.o
Waf: Leaving directory `/media/16GB/projects/masteringnode/src/addons/functions/v0.1/build'
'build' finished successfully (0.587s)
> var func = require('./func');
> func
{ version: '0.1', echo: [Function] }
> func.echo
[Function]
> func.echo()
TypeError: Bad argument
    at [object Context]:1:6
    at Interface.<anonymous> (repl.js:144:22)
    at Interface.emit (events.js:42:17)
    at Interface._onLine (readline.js:132:10)
    at Interface._line (readline.js:387:8)
    at Interface._ttyWrite (readline.js:564:14)
    at ReadStream.<anonymous> (readline.js:52:12)
    at ReadStream.emit (events.js:59:20)
```

```
    at ReadStream._emitKey (tty_posix.js:280:10)
    at ReadStream.onData (tty_posix.js:43:12)
> func.echo(1)
1
> func.echo("asdf", 20)
'asdf'
> func.echo(function() { return 1; }, 222)
[Function]
> func.echo(function() { return 1; }, 222)()
1
```

Notice the construct of the last line is `func.echo()()`, which executes the function that is passed as the first argument.

FunctionTemplate

In v8, a FunctionTemplate is used to create the equivalent to:

```
var template = function() { }
```

The function at this point is an object and not an *instance* of the function.

As an example, we will use the linux package *uuid* to generate a uuid. We will define the header for this addon as:

```
// addons/uuid/v0.1/uuid.h
#ifndef __node_uuid_h__
#define __node_uuid_h__

#include <string>
#include <v8.h>
#include <node.h>
#include "uuid/uuid.h"

using namespace v8;
using namespace node;

class Uuid : public node::ObjectWrap {
    public:
        Uuid() { }
        static Persistent<FunctionTemplate> constructor;
        static void Init(Handle<Object> target);
        static Handle<Value> New(const Arguments &args);
        static Handle<Value> Generate(const Arguments &args);
        static Handle<Value> GenerateRandom(const Arguments &args);
        static Handle<Value> GenerateTime(const Arguments &args);
    private:
        ~Uuid();
        static std::string GetString(uuid_t id);
};

#endif // __node_uuid_h__
```

This addon will showcase three methods, `Generate`, `GenerateRandom`, and `GenerateTime`. It will also include a trivial private `GetString` method to demonstrate how to *Unwrap* a `node::ObjectWrap` object and interact with C++ code that is not specific to node or v8.

A lot of the public function definitions should look similar to the *Echo* example. One notable difference is that instead of using a macro and hiding the `FunctionTemplate` method, we are defining `static Persistent<FunctionTemplate> constructor;`. The `Persistent<T>` type is used "when you need to keep a reference to an object for more than one function call, or when handle lifetimes do not

correspond to C++ scopes." source. Since we'd expect our object's constructor to last longer than a single function, we declare it separately and as a persistent handle. Another point to notice is that all of the method we're pulling from *uuid.h* have the signature `static Handle<Value> Method(const Arguments &args)` even though we will plan to call it as `uuid.generate()`. This is because we will be accessing the *scope* of the call via `args.This()`.

Although more methods are implemented in *uuid.cc*, we will look at three:

- `Uuid::Init(Handle<Object> target)`
- `Handle<Value> Uuid::New(const Arguments &args)`
- `Handle<Value> Uuid::Generate(const Arguments &args)`

Just as before, node expects to find a signature of `extern "C" void init(Handle<Object> target)` in order to initialize the addon. Inside this method, we may set parameters such as the version number from the previous example. We may also pass-through initialization to any modules within our node addon. In this example, our addon will be *uuid.node* and contain a single module, *Uuid*. There is no reason we can't later add *Uuid2* which, instead of returning a normalized string value might return a `Buffer` object. To initialize the Uuid module, we pass the `target` object along to `Uuid::Init` and add the module definition to `target`:

```
// addons/uuid/v0.1/uuid.cc
void Uuid::Init(Handle<Object> target) {
    HandleScope scope;

    // Setup the constructor
    constructor = Persistent<FunctionTemplate>::New(FunctionTemplate::New(Uuid::New));
    constructor->InstanceTemplate()->SetInternalFieldCount(1); // for constructors
    constructor->SetClassName(String::NewSymbol("Uuid"));

    // Setup the prototype
    NODE_SET_PROTOTYPE_METHOD(constructor, "generate", Generate);
    NODE_SET_PROTOTYPE_METHOD(constructor, "generateRandom", GenerateRandom);
    NODE_SET_PROTOTYPE_METHOD(constructor, "generateTime", GenerateTime);

    target->Set(String::NewSymbol("Uuid"), constructor->GetFunction());
}
```

In this scope, we are instantiating the `constructor` using `Uuid::New` as a new `FunctionTemplate`. We then call `InstanceTemplate()` and on that object we call `SetInternalFieldCount(1)`. This tells v8 that this object holds a reference to one object.

Next, we setup the prototype using another macro provided by node. These calls say, for instance, "Add a method called *generate* to the constructor function which executes the native method *Generate*".

Lastly, we have to create a "Uuid" module on the object returned by the call to `require()`. Here, `Uuid` will point to a function (`constructor`) which returns a function that internally executes `Uuid::New`. In other words, we have created something akin to:

```
var Uuid = function() { };
Uuid.constructor = function() {
    return function() {
        // Uuid::New executes here.
    }
}
```

Although the above is not exactly what we have done, it may provide a better view for some to understand `FunctionTemplate` references and why we assign one to the constructor object in such a way.

The `Uuid::New` method is defined as:

```
// addons/uuid/v0.1/uuid.cc
Handle<Value> Uuid::New(const Arguments &args) {
    HandleScope scope;
    // no args are checked
    Uuid *uuid = new Uuid();
    uuid->Wrap(args.This());
    return args.This();
}
```

As you would expect, calling the constructor function multiple times will create newly-scoped `Uuid` objects on the heap. In this method, we wrap the parameter (scoped object) by setting a reference to `Uuid` in the args as a contextual scope (i.e. `this`) and then returns `this`.

Within the `Generate` method, we will want to unwrap the contextual `Uuid` object and call the private method `GetString`.

```
// addons/uuid/v0.1/uuid.cc
Handle<Value> Uuid::Generate(const Arguments &args) {
    HandleScope scope;
    uuid_t id;
    uuid_generate(id);

    Uuid *uuid = ObjectWrap::Unwrap<Uuid>(args.This());
    return scope.Close(String::New(uuid->GetString(id).c_str()));
}
```

As with any JavaScript function call, we have to ensure functional scope. Scoped methods should create a HandleScope object at the start and call `scope.Close()` at the end. `HandleScope` will get rid of temporary handles when the scope is closed.

Within each of the *Generate** methods, we will create a `uuid_t` type and call the corresponding method defined in */usr/includes/uuid/uuid.h* (location may vary per system). To demonstrate accessing the pointer to our original `Uuid` object, we unwrap the contextual scope of this function using `ObjectWrap::Unwrap<Uuid>(args.This())`. From this pointer, we can access any private methods such as `GetString`. Be careful with your returned values, though. `String::New` in the v8 library does not take `std::string` in any of its signatures. Simply enough, `std::string` provides a `c_str()` method to return a `const char*` which `String::New` does accept.

uuid.node demo

Navigate to *addons/uuid/v0.1/* and execute:

```
$ node-waf configure build
```

If there are build errors and the *func.cc* example from before built successfully, check that you have the *uuid-dev* package installed and rerun the above command. Then, navigate to *build/default* and try out the Uuid addon:

```
$ node
> var Uuid = require('./uuid.node').Uuid;
> var uuid = new Uuid();
> uuid.generate();
'83475e0c-212b-402c-bdc7-b81ebb7b34f8'
> uuid.generateRandom();
'4d597bda-8f5f-4c3c-b2fa-1cd6cd4a6903'
> uuid.generateTime();
'25a0dd30-5076-11e1-96be-0022fb93b24c'
> var util = require('util');
```

```
> util.inspect(uuid);
'{}'
> util.inspect(Uuid);
'[Function: Uuid]'
>
```

The above output may surprise you. Firstly, where is the `version` option?! It's at the required module level: `require('./uuid.node').version;`. Secondly, if we can access `uuid.generate()` and others, why don't they display when inspecting the object? That's because we defined those methods on the prototype:

```
> uuid.__proto__
{ generate: [Function: generate],
  generateRandom: [Function: generateRandom],
  generateTime: [Function: generateTime] }
>
```

Thirdly, you may have noticed that I didn't say anything about `constructor->SetClassName(String::NewSymbol("Uuid"));` in `Uuid::Init`. You may have also wondered where `SetClassName` actually sets a class name, considering JavaScript is a prototypal language. That string value is what is displayed when you call inspect and get the value `'[Function: Uuid]'`. Just as you would expect, `Uuid` is the constructor and it is named `Uuid`.

Now, if you've played around with this a bit, you may have noticed that `uuid.__proto__` gives us our three functions but `uuid.prototype` is empty. Why is that? That's because `uuid.__proto__` really is `uuid.constructor.prototype`, which is also really `Uuid.prototype`. This is the essence of prototypal inheritance. If this concept is foreign or difficult to grasp, be sure to check out the excellent explanation on JavaScript Garden.

Logically, the next step would be to understand how to declare a prototype of our own.

Function Prototypes

TODO

Function Constructor

TODO

Making it Evented

TODO

Globals

As we have learned, node's module system discourages the use of globals. However, node provides a few important globals for us. The first, and probably the most important, is the `process` global which exposes process manipulation (e.g. signalling, exiting, process id (pid), and others). Other globals, such as the `console` object, provide JavaScript functionality common in most browsers.

console

The `console` object contains several methods which are used to output information to *stdout* or *stderr*.

```
> console
{ log: [Function],
  info: [Function],
  warn: [Function],
  error: [Function],
  dir: [Function],
  time: [Function],
  timeEnd: [Function],
  trace: [Function],
  assert: [Function] }
```

Let's 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' });
{ foo: 'bar' }
```

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. Here, you can also use `console.trace()` and `console.error()` to print a stack trace and a helpful message when the exception is caught.

```
try {
   console.assert( (1 != 2), '1 != 2: Should be true' );
   console.assert( (1 == 2), '1 == 2: Should be false');
} catch(e) {
```

```
    console.error('Caught error ' + e);
    console.trace();
}
```

process

The `process` object is plastered with goodies. In a node console, type `process` and look at what it has to offer. First we will take a look at some properties that provide information about the node process itself.

process.version(s)

The `process.version` property contains the node version string, for example "v0.4.0". The `process.versions` object displays the versions of node, v8, ares, ev, and openssl:

```
> process.versions
{ node: '0.4.0',
  v8: '3.1.2',
  ares: '1.7.4',
  ev: '4.3',
  openssl: '0.9.8o' }
```

process.installPrefix

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

process.execPath

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

process.platform

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

process.pid

The process id.

process.cwd()

Returns the current working directory, for example:

```
$ cd /var && node
> process.cwd()
'/var'
```

process.chdir()

Changes the current working directory to the path passed.

```
> process.chdir('lib')
> process.cwd()
'/var/lib'
```

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, where 501 is TJ's *uid*.

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/X11/bin'
, 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/exceptions.js
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/kill.js
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);
```

process.binding()

Node performs lazy initialization and caching for a number of built-in modules through the `process.binding` function. If you'd like to poke around in node's source code and view this binding cache implementation, navigate to your node checkout directory and open *./src/node.cc*. The code for the binding cache starts on like 1749 (of node.js v0.4.0).

Currently, the cacheable process bindings include: built-in modules, `constants`, `io_watcher`, `timer` and `natives`. To access the constants, instead of using `var constants = require('constants');`, you could lazily bind the constants as it is done in the `fs` module:

```
// [node.js source]/lib/fs.js
var util = require('util');

var binding = process.binding('fs');
var constants = process.binding('constants');
var fs = exports;
var Stream = require('stream').Stream;
// ...
```

Binding in this way allows us to perform the initialization of those modules bound to `process` only once. We can then extend those bindings because they are an exported module like any other `require`.

Another reason for caching the bindings in this way is because these bindings differ across platforms. If you were to open *node*constants.cc*, you would see conditional includes for* MINGW32 *and* `ifdefs` *for each constant. You wouldn't want to hardcode these constants in a module's exports. If you opened opened the* constants_ module definition, you would see one line:

```
// [node.js source]/lib/constants.js
module.exports = process.binding('constants');
```

After a single require of this module, or call to `process.binding('constants')`, the constants are initialized and cached. This lazy initialization improves node's startup.

errno

To access `errno` constants, you'll need to either add the include `var constants = require('constants');` or use the `var constants = process.binding('constants');` method described earlier.

The `constants` object is host of the error numbers, these reference what you would find in C-land, for example `constants.EPERM` represents a permission based error, while `constants.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 as in the http request error object:

```
if (err.errno === constants.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.

```
// events/basic.js
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

Perhaps a 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 its 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*.

```
// events/subclass.js
var EventEmitter = require('events').EventEmitter;

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

- Note: JavaScript doesn't have 'classes'. The file above is called 'subclass' because this is a term commonly used for inheritance in an object-oriented language.*

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);
```

You may look at the above code and think, "Why don't I just add a *bark* function?" That's a good question. The power of events is in subscribing to events that occur when a function on the object is executed. For instance, if you have a writeable `Stream` object with the function `write()`, you may subscribe to a data event. The following event example is directly from the node api documentation:

```
// events/streams.js
var util = require("util");
var events = require("events");

function MyStream() {
    events.EventEmitter.call(this);
}

util.inherits(MyStream, events.EventEmitter);

MyStream.prototype.write = function(data) {
    this.emit("data", data);
}

var stream = new MyStream();

console.log(stream instanceof events.EventEmitter); // true
console.log(MyStream.super_ === events.EventEmitter); // true

stream.on("data", function(data) {
        console.log('Received data: "' + data + '"');
});
stream.write("It works!"); // Received data: "It works!"
```

This example is fairly complicated compared to most of the examples so far. It does a number of things that haven't yet been covered, but are included here for completeness. First of all, we're creating an object of `MyStream` called `stream`, which inherits from `EventEmitter`. This was mentioned earlier, but the api's example provides a nice use of the util module for inheritance. Then, we set the `write` function to emit the *data* event to any subscribers (which are called *EventListeners*). In this function, we could also provide some default functionality (more than emitting an event), such as writing to an inner sink, logging, or any number of operations.

After setting up the object, this example writes to `console.log()` to demonstrate that both the stream and the `super_` object are instances of `EventEmitter`. Afterward, we subscribe to the `"data"` event and log the data that is emitted from the `MyStream.prototype.write` function. This means that anything passed to the `write()` function is emitted to all subscribers as the single object passed to the callback function. In other words, anytime we call `write()`, we're going to log with a little message to show that the parameter is being written by the emitted event.

So, unlike the dog barking example, this shows that a single function can have multiple operations occur via events.

Note: when adding an event, it is pushed onto the end of an array of existing events

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 `emitter.on()` originally. To complement this method is `removeAllListeners(type)`, which removes all listeners associated to the given *type*.

```
// events/removing.js
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 pass a string as the first argument to `Buffer`'s constructor. 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.

```
> var h = [0x48, 0x65, 0x6c, 0x6c, 0x6f];
> h
[ 72, 101, 108, 108, 111 ]

> h.toString();
'72,101,108,108,111'

> var hello = new Buffer(h);
<Buffer 48 65 6c 6c 6f>

> hello.toString();
'Hello'
```

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 create a buffer large enough to hold the string "Hello World!" After writing 'Hello', we see the bytes 5 through 12 are unused bytes. We then write ' World' starting at byte 6 and examine the output. Whoops! We skipped a byte. We can overwrite this part of the buffer with ' World!' starting at byte 5. We then call `toString()` on the buffer and see that the buffer is now filled with the desired string.

```
> var hello = new Buffer(12);
> hello.write('Hello');
5

> hello.toString();
'Hello\u0000ï¿½Ìµ\u0001\u0000\u0000'
> hello.write(' World', 6);
6
```

```
> hello.toString();
'Hello\u0000 World'

> hello.write(' World!', 5);
7

> hello.toString();
'Hello World!'

> hello.length
12
```

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

> ellipsis
<Buffer e2 80 a6>
```

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

```
> Buffer.byteLength('â ¦');
3
```

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 buf = new Buffer('For some other string!');
> var chunk = buf.slice(4, 8);
> console.log(chunk.toString());
'some'
```

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

```
> var buffer = new Buffer('The quick brown fox');
> buffer.toString('ascii', 4, 9);
'quick'
```

A Buffer object has a number of helper functions: `.utf8Write()`, `.utf8Slice()`, `.asciiWrite()`, `.asciiSlice()`, `.binaryWrite()`, `.binarySlice()`. These methods provide similar functionality while enforcing proper encoding.

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()` on a buffer to return a string representation of the binary data. 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 important 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".

```
// streams/echo.js
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.

```
// streams/pump.js
var http = require('http'),
    sys = require('sys');

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

File System

To work with the filesystem, node provides the 'fs' module. The commands follow the POSIX operations, with most methods supporting an asynchronous and synchronous method call. We will look at how to use both and then establish which is the better option.

Working with the filesystem

Lets start with a basic example of working with the filesystem, this example creates a directory, it then creates a file in it. Once the file has been created the contents of the file are written to console:

```
// fs/basics.js
var fs = require('fs');

fs.mkdir('./helloDir',0777, function (err) {
  if (err) throw err;

  fs.writeFile('./helloDir/message.txt', 'Hello Node', function (err) {
    if (err) throw err;
    console.log('file created with contents:');

    fs.readFile('./helloDir/message.txt','UTF-8' ,function (err, data) {
      if (err) throw err;
      console.log(data);
    });
  });
});
```

As evident in the example above, each callback is placed in the previous callback - this is what is referred to as chainable callbacks. When using asynchronous methods this pattern should be used, as there is no guarantee that the operations will be completed in the order that they are created. This could lead to unpredictable behavior.

The example can be rewritten to use a synchronous approach:

```
fs.mkdirSync('./helloDirSync',0777);
fs.writeFileSync('./helloDirSync/message.txt', 'Hello Node');
var data = fs.readFileSync('./helloDirSync/message.txt','UTF-8');
console.log('file created with contents:');
console.log(data);
```

It is better to use the asynchronous approach on servers with a high load, as the synchronous methods will cause the whole process to halt and wait for the operation to complete. This will block any incoming connections and other events.

File information

The fs.Stats object contains information about a particular file or directory. This can be used to determine what type of object we are working with. In this example we are getting all the file objects in a directory and displaying whether they are a file or a directory object.

```
// fs/stat.js
var fs = require('fs');

fs.readdir('/etc/', function (err, files) {
  if (err) throw err;

  files.forEach( function (file) {
```

```
    fs.stat('/etc/' + file, function (err, stats) {
      if (err) throw err;

      if (stats.isFile()) {
        console.log("%s is file", file);
      }
      else if (stats.isDirectory ()) {
      console.log("%s is a directory", file);
      }
    console.log('stats:  %s',JSON.stringify(stats));
    });
  });
});
```

Watching files

The fs.watchfile monitors a file and will fire the event whenever the file is changed.

```
// fs/watch.js
var fs = require('fs');

fs.watchFile('./testFile.txt', function (curr, prev) {
  console.log('the current mtime is: ' + curr.mtime);
  console.log('the previous mtime was: ' + prev.mtime);
});

fs.writeFile('./testFile.txt', "changed", function (err) {
  if (err) throw err;

  console.log("file write complete");
});
```

A file can also be unwatched using the fs.unwatchFile method call. This is used once monitoring of a file is no longer required.

Nodejs Docs for further reading

The node api docs are very detailed and list all the possible filesystem commands available when working with Nodejs.

TCP

...

TCP Servers

Many times, a server will do two things: write data to the console and send data to a client connection. Below is a simple TCP server which echos "You were connected!" to a client connection and outputs "A client has connected!" to *stdout*.

```
// tcp/tcp_server.js
var sys = require('sys');
var tcp = require('net');
var server = tcp.createServer({ allowHalfOpen:false }, function(socket) {
    sys.puts("A client has connected!");
    socket.write("You were connected!\n", "utf8");
    socket.end();
}).listen(8000, '127.0.0.1');
```

To test this server, enter `node src/tcp/tcp_server.js` in a terminal. Then, open another terminal to telnet into this server.

```
$ telnet 127.0.0.1 8000
```

In the server's terminal, you should see the message output from `sys.puts`, while the client's server should display, "You were connected!"

TCP Clients

...

HTTP

Node's HTTP module offers a chunkable, buffered interface to the HTTP protocol. Headers are JSON objects with lower-case keys and original values.

```
{ 'content-length': '123', 'content-type': 'text/plain' }
```

The HTTP module is low-level, meaning it handles the headers and the message (and that's about it). This creates a solid framework ontop of which modules can be built for web frameworks (express, geddy, file servers, browsers (zombie.js, and even SaaS (cloud9 IDE.

Let's take a look at the `request` and `response` objects of an HTTP server created by node. To do this, we're going to create a server, inspect both of these objects in the request, and write a dump of the inspection out to a file.

```
// http/view_request.js
var http = require('http'),
fs = require('fs'),
util = require('util'),
tty = require('tty');

var server = http.createServer(function(req, res) {
    console.log("Received a request!");
    fs.writeFile('./request.txt', util.inspect(req) );
    res.writeHead(200, { 'Content-Type' : 'text/html' });
    res.write("<html><head></head><body><h1>Hello, World!</h1><p>We're serving up html!</p></body></html>");
    res.end();
    fs.writeFile('./response.txt', util.inspect(res) );
});

// ... cleanup on exit (see file)

console.log("Server is running on http://localhost:9111");
console.log("Hit CTRL+C to shutdown the http server");
```

Run this sample with `node src/http/view_request.js` from a terminal. Then, navigate to the server location displayed in the terminal. This will generate two files: `./src/http/request.txt` and `./src/http/response.txt`.

!IMPORTANT! You must always remember to provide a way to close a server. Leaving an open server running developmental code can be a security risk.

HTTP Request

Let's take a look at what the request object looks like from the server example above. You should see output similar to:

```
// http/request.txt
// ...
httpVersion: '1.1',
  complete: false,
  headers:
   { host: 'localhost:9111',
     connection: 'keep-alive',
     accept: '*/*',
     'user-agent': 'Mozilla/5.0 (X11; Linux i686) AppleWebKit/534.24 (KHTML, like Gecko) Chrome/11.0.696.14 Safari/534.24',
     'accept-encoding': 'gzip,deflate,sdch',
     'accept-language': 'en-US,ru;q=0.8',
     'accept-charset': 'ISO-8859-1,utf-8;q=0.7,*;q=0.3' },
```

```
  trailers: {},
  readable: true,
  url: '/favicon.ico',
  method: 'GET',
  statusCode: null,
// ...
```

HTTP Response

Let's take a look at what the request object looks like from the server example above. You should see output similar to:

```
{ output: [],
  outputEncodings: [],
  writable: true,
  _last: false,
  chunkedEncoding: true,
  shouldKeepAlive: true,
  useChunkedEncodingByDefault: true,
  _hasBody: true,
  _trailer: '',
  finished: true,
  socket: null,
  connection: null,
  _events: { finish: [Function] },
  _header: 'HTTP/1.1 200 OK\r\nContent-Type: text/html\r\nConnection: keep-alive\r\nTransfer-Encoding: chunked\r\n\r\n',
  _headerSent: true }
```

HTTP Servers

http.Server exposes a number of events (request, connection, close, response, etc.) and three functions for creation, listening for requests, and closing the server. We saw examples of a few of these at the beginning of this section. Up to this point, most of the code we've looked at has been mainly procedural. To create a server, we will have to implement quite a few things we've learned up to this point:

- Write a module for an http server
- Expose events which occur on the server
- Serve/compile files

Like the first example in this chapter, we'll be using the *http*, *fs* and *tty* modules. To make this interesting, though, we'll also do a compilation from markdown to html so that our server is doing a little more than serving static files. To simplify the example, we won't also be serving static files or even sending a full range of HTTP status codes; everything is `200 OK` in this example.

Creating the module

Our module will be located at `./src/http/server/`. The directory is structured in the following way:

```
$ ls -l
total 20
drwxr-xr-x 3 jim jim 4096 2011-03-23 16:31 lib
-rw-r--r-- 1 jim jim   56 2011-03-23 17:13 package.json
drwxr-xr-x 2 jim jim 4096 2011-03-24 07:25 pages
-rw-r--r-- 1 jim jim 3034 2011-03-24 07:57 server.js
drwxr-xr-x 2 jim jim 4096 2011-03-23 20:31 templates
```

The file `server.js` is our module. So we don't need to call `var server = require('./server/server.js')` at the top of our node application, we're also going to include

`package.json` which specifies where the main routine for our module is located.

```
// http/server/package.json
{ "name" : "example server",
  "main" : "./server.js" }
```

Recall from the *Modules* chapter that it is also possible to include an `index.js` file.

Next, let's identify the functionality we'll require in this module. First, we know that we'll have to read a file. We'll call that function `getFile()`. Then, since we're simplifying things for example's sake, we're assuming all files are in markdown format and converting all files to html. We'll call that function `getHtml()`. I know, I'm really creative.

Finally, we'll have a function called `run()` which creates the server and listens on the specified port. To make our module useful, we have to expose functionality which, in our case, is only the run method. Here is how we will interface with our module:

```
// http/server_example.js
var server = require('./server');
server.run({ port: 9222 });
```

We also have the option of performing a function before starting the server:

```
server.run({ port: 9222,
    beforeStart: function(){
        console.log("before start");
    }
});
```

Now that we have a plan in place for the interaction with the module, let's take a look at the three functions, `getFile()`, `getHtml()`, and `run()`.

getFile(request)

The `getFile()` function assumes that all requests occur at the base of the server's uri. It also ignores requests for `/favicon.ico` and for the base uri `/`. It then builds a full path to the file and attempts to read the file in a single blocking call. This blocking call helps ensure that we're passing data correctly to the Markdown module.

```
// http/server/server.js
var getFile = function(request) {
    if(!request) {
        console.log("getFile called with empty request");
        noop();
    }

   var callback = arguments[arguments.length - 1];
   if (typeof(callback) !== 'function') callback = noop;

   var filename = url.parse(request).pathname.replace('html', 'md');
   if(filename === "/favicon.ico") return;
   if(filename === "/" || filename === "") { filename = "/index.md"; }

   var requestedFile = __dirname + '/pages' + filename;

   console.log('Requested: ' + requestedFile);
   var data = fs.readFileSync(requestedFile, "utf8");
   callback(data);
};
```

This function uses the same pattern of identifying a callback as the node libraries. The `noop()` function is defined as `var noop = function() { };` which saves us from creating numerous empty callbacks and checking for functions where those callbacks are required. For those unfamiliar with the term *noop*, it means the function performs **no op**eration.

getHtml(request)

The second function in our module performs the conversion between markdown and html. It does this in the callack to the `getFile()` function, transforms the markdown to html, and passes the result to its own callback.

```
// http/server/server.js
var getHtml = function(request) {
    var callback = arguments[arguments.length - 1];
    if (typeof(callback) !== 'function') callback = noop;

    // get the data and call markdown
    try {
        getFile(request, function(data) {
            if(!data) {
                callback("Nothing to see here!");
            }
            var html = template.replace("{{content}}", markdown.toHTML(markdown.parse(data), {xhtml:true}));
            console.log(html);
            callback(html);
        });
    } catch(err) {
        console.log(err);
        callback("Nothing to see here!");
    }
};
```

If there are any errors, it returns a string: "Nothing to see here!" A message like this usually accompanies a `404 - Not Found` HTTP status, but we're keeping this pretty simple.

run()

The run function requires the following server to be created. The `requestListener` function calls `getHtml` and writes the html to the response. Here, we're using a buffer to get the proper byte length and attempt to output a properly-encoded html string.

```
// http/server/server.js
var server = http.createServer(function(req, res) {
    getHtml(req.url, function(html) {
        var buf = new Buffer(html, "utf8");
        res.writeHead(200, { 'Content-Type' : 'text/html', 'Content-Length' : buf.length });
        res.write(buf.toString());
        res.end();
    });
});
```

Next, the run function sets the options we're allowing (port number and a function to call before starting the server). We're also catching all errors and logging the output to the console.

```
// http/server/server.js
var run = function(opts) {
    try {
        if(opts && typeof opts['beforeStart'] === 'function'){
            opts['beforeStart']();
        }
        var port = (opts && opts.port) || 9111;
```

```
        // ... Removed CTRL+C interrupt code from previous example

        server.listen(port, function() {
            console.log("Server is running on http://localhost:" + port);
            console.log("Hit CTRL+C to shutdown the http server");
        });
    } catch(err) {
        console.log(err);
    }
};
```

Notice how we've moved the console logging to the `server.listen` callback. This makes more sense than the procedural example from before- if the port isn't available and `server.listen` throws an error, you don't want to tell the developer that the server has started! This is how things should be programmed, and it's part of what makes node.js so awesome.

Exposing `run()`

The last thing to do to make our module run is to expose the run function. To revisit from the *Modules* chapter, you can do this a few ways:

```
// 1. Multiple assignment
var server = exports.server = http.createServer(requestListener);

// 2. Inline assignment
exports.server = http.createServer(requestListener);

// 3. Post-facto assignment
var server = http.createServer(requestListener);
exports.server = server;
```

There's no 'correct' method, and each has it's benefits. Please refer to the *Best Practices* section for more information.

Run it!

```
$ node src/http/server_example.js
Server is running on http://localhost:9222
Hit CTRL+C to shutdown the http server
```

After receiving the output to confirm the server is running, visit http://localhost:9222 to check it out. Then, hit `CTRL+C` to be sure the server's `close` function is working as expected.

Where are the events?

You may have noticed that we met only half of the requirements with the above implementation of our server. To expose events, our `run` method would have to inherit from `EventEmitter`. That doesn't *really* make sense. For the sake of simplicity and brevity, we previously only had three methods. We had no class-like objects, and we didn't touch an object's prototype. Also, the three methods we did have didn't make good use of callbacks. So, there is another take on this example at *./src/http/server/server2.js*.

This is set up so that our `run` function returns our `Example` object. This object has properties for our configurables (such as the pages directory, template name, etc.). It also has the functions from the previous example, which have been slightly refactored.

First, you'll notice the inheritence of the `Example` object from `EventEmitter`:

```
// http/server/server2.js
```

```
util.inherits(Example, EventEmitter);
```

This allows us to call `this.emit('eventName')` at different times in our code, which executes all of the listeners bound to these events in the order they were declared.

Now, at different points of the code, you'll see a `this.emit()` or `self.emit()` call. For instance, in the `getServer` function, we emit the `request` and `requestComplete` events.

```
// http/server/server2.js
Example.prototype.getServer = function() {
        var self = this, ct = 'text/html; charset=utf-8';
        return http.createServer(function(req, res) {
                self.emit('request', req.url);
                self.getHtml(req.url, function(html) {
                        var buf = new Buffer(html, "utf8");
                        res.writeHead(200,
                                {
                                        'Content-Type' : ct,
                                        'Content-Length' : buf.length
                                });
                        res.write(buf.toString());
                        res.end();
                        self.emit('requestComplete');
                });
        });
};
```

Because we know that all responses will be *html*, the request comes in and immediate emits the `request` event. Our `getHtml` function is written to do one of two things. First, it gets a valid string of html, emits it as *data* to the `data` event, then passes it to `getHtml`'s callback as the *html* parameter, which ultimately writes that string to the response.

The alternative is when a request is ignored either explicitly with the `option.ignorePaths` array, or implicitly by `fs.readFileSync` throwing an error. Either way, we emit the `data` event with the content as "ignored". The `getHtml` function above then writes an empty response.

Note: `http.ServerRequest`, the type of the `request` parameter in the *requestListener* passed to `http.createServer`, emits its own *data* and *end* events. Instead of emitting our own events, we could expose the existing events instead of emitting completely new and renamed events.

HTTP Clients

Node also provides client functionality in the *http* module. To begin with the http client functionality, we're going to open one terminal and run the server from the previous example: `node src/http/server_example2.js`. The first client code we're going to run is slightly modified from node's v0.4.0 documentation.

http.get(options, callback)

This example uses the `http.get` function. This function is a wrapper around the `http.request` function which assumes that a get doesn't write a body to the server request, ultimately only expecting a response. If you're familiar with jQuery, node's `http.get` is a wrapper around `http.request` in much the same way as jQuery's `$.get` is a wrapper around the `$.ajax` function; the big difference is that in node, we're not working with an *XmlHttpRequest* object-- we're working directly with *sockets* and *streams*.

```
// http/http_get.js
var http = require('http'),
        util = require('util');
```

```
var opt = {
        host: 'localhost',
        port: 9222,
        path: '/index.html'
};

http.get(opt, function(response) {
        console.log("http.get [Status]: %s", response.statusCode);
        console.log("http.get [method]: %s", response.client._httpMessage.method);
        console.log("http.get [path]: %s", response.client._httpMessage.path);
        console.log("http.get [method]:\n%s", response.client._httpMessage._header);
        // console.log("http.get [client]:\n%s", util.inspect(response.client));
}).on('error', function(err) {
        console.log("http.get [Error]: " + err.message);
});
```

Note: see the Globals chapter for information on `err.errno` constants

With the server from the previous example running, we can run the *./src/http/http*get.js_ example and examine the output.

```
$ node src/http/http_get.js
http.get [Status]: 200
http.get [method]: GET
http.get [path]: /index.html
http.get [method]:
GET /index.html HTTP/1.1
Host: localhost
Connection: close
```

If you'd like to view the full client object, uncomment the last line from the callback in the file *./src/http/http*get.js_.

Url Module

...

Micro-Frameworks

Geddy

...

ZombieJS

...

cloud9 IDE

...

HTTPS

HTTPS Server

HTTPS Client

Connect

Connect is a ...

Express

Express is a robust and feature-rich web application framework for node.js written by TJ Holowaychuk, Ciaran Jessup, Aaron Heckmann, and Guillermo Rauch.

Install

Simple First Project

Testing

Testing is a very large topic. There are different types of testing, each with different methodologies. Then, there are also strong feelings of developers toward which type of testing is the best or most effective.

- Assertion Testing
  - Tests a true or false condition against known objects or values
- Behavioral Testing
  - Tests how one object acts when interacting with another
- Functional/Acceptance Testing
  - "Black box" testing of a system, usually code-agnostic
- Regression Testing
  - Tests for consistency after changes have been made to code
- Others...

There are many other types of testing, but these are some large topics in the testing arena. Development practices include Test-Driven Development (TDD), Behavior-Driven Development, and others. Some programmers don't believe in testing, others believe tests should be written before the code, while others still believe tests should only be written for the most important code. As you can see, this is a large and complex topic.

Whether you write tests before you code, test only the most important code, or casually write tests when there is time, it is important to choose a testing tool that fits your needs.

assert Module

To begin, require the assert module:

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

This module exposes a few functions common to assertion testing:

```
// equality
assert.equal(actual, expected, [message])
assert.notEqual(actual, expected, [message])
assert.deepEqual(actual, expected, [message])
assert.notDeepEqual(actual, expected, [message])
assert.strictEqual(actual, expected, [message])
assert.notStrictEqual(actual, expected, [message])

// exception
assert.throws(block, [error], [message])
assert.doesNotThrow(block, [error], [message])
assert.ifError(value)

// condition
assert.ok(value, [message])
assert.fail(actual, expected, message, operator)
```

These are pretty self-explanatory if you've written assertion tests before. Let's look at a pretty simple test, anyway. We'll code in an object for the test that will do what we want for each test.

Synchronous Testing with Assert

```
// testing/equality_no_errors.js
var assert = require('assert'),
    tester_a = {
```

```
      val : 'a'
    },
    tester_b = {
      val : 'b'
    };

assert.equal(tester_a.val, 'a');
assert.equal(tester_b.val, 'b');
```

In this example, we're using the assert module to check that a value on each of these objects is the expected value. This works well because the values are equal. But, the assert module throws an `AssertionError` whenever an assertion fails. So, it doesn't really make sense to have a single file with end-to-end assertions as in the `./src/testing/equality\_no\_errors.js` file. This is where testing frameworks like *nodeunit* or *expresso* come in very handy. They offer common functionality of test frameworks (like metrics, etc.).

To see how the `AssertionError` being thrown from `assert.equal` can be problematic, change the expected value in the first test to an incorrect value. Then, change the expected value in the very last test to an incorrect value and run `node src/testing/equality_no_errors.js` again. You'll see that, because of the procedural style of the code, the last test never runs!

In order to run multiple assertions and provide feedback, the simplest test (without a testing framework) would use a `try/catch` and provide output to *stdout*.

```
// testing/equality_with_errors.js
var assert = require('assert'),
    tester_a = {
      val : 'aa'
    }, total = 0, good = 0;

// assert.equal(actual, expected, [message])
try {
    console.log("assert.equal(tester_a.val, 'a')");
    assert.equal(tester_a.val, 'a');
    passed();
} catch (err) { writeException(err); }

console.log("%d of %d tests passed", good, total);

function writeException(err) {
    console.log("Test failed!");
    util.inspect(err);
    if(err["name"] === "AssertionError") {
        console.log("Message: " + (err["message"] || "None"));
        console.log("Expected: " + err["expected"]);
        console.log("Actual: " + err["actual"]);
        console.log("Operation: " + err["operator"]);
    }
    console.log("");
    total = total + 1;
}

function passed() {
    good = good + 1;
    total = total + 1;
    console.log("Test passed!\n");
}
```

The above code is part of the code from within *./src/testing/equality_with_errors.js*. It shows how to run synchronous tests with a minimal amount of redundant code, *without* writing a lightweight testing module for your tests. This may be what you want, but most likely it isn't.

The problems

You *could* write a simple helper module to run these tests for you. But, how do you verify the order of your tests? This can become complex very quickly.

You can partially resolve this with the `try/catch` block example above, but this requires a lot of redundant code. Compare *./src/testing/equality_with_errors.js* and *./src/testing/equality_no_errors.js* to see how the testing quickly expands! Run `node testing/equality_with_errors.js` to see the output. That's more like it! Isn't that nice?

Yes and no. There are a few problems with this code:

1. It isn't evented
2. It hard-codes test objects
3. It is very redundant
4. It isn't scalable

Well, then, *what are other options?*

Nodeunit

Nodeunit is a framework similar to nunit in that it allows for multiple test cases (running in parallel), and supports mocks and stubs. It is easy to use, and even allows you to run tests in the browser.

Nodeunit testing starts with exporting a test or two from a module.

```
// testing/nodeunit_basics.js
module.exports = {
    'Test 1' : function(test) {
        test.expect(1);
        test.ok(true, "This shouldn't fail");
        test.done();
    },
    'Test 2' : function(test) {
        test.expect(2);
        test.ok(1 === 1, "This shouldn't fail");
        test.ok(false, "This should fail");
        test.done();
    }
};
```

Here, we have a module exporting two tests, `Test 1` and `Test 2`. You may have noticed that the `test` object in the functions have an `ok()` method just like the `assert` module mentioned above. Good eye. In fact, `test` supports all of the assert functions and adds two others: `expect(number)` and `done()`. The `expect` function tells nodeunit how many tests are being run within the context of the current test case. When all tests are finished, call `test.done()` to let nodeunit know the test case has completed (and a callback may have failed).

The output of nodeunit is visually helpful.

```
$ nodeunit src/testing/nodeunit_basics.js

nodeunit_basics.js
â   Test 1
â   Test 2

Assertion Message: This should fail
AssertionError: false == true
```

```
    at Object.ok (/usr/local/lib/node/.npm/nodeunit/0.5.1/package/lib/types.js:81:39)
    at /home/jim/projects/masteringnode/src/testing/nodeunit_basics.js:10:14
    at Object.runTest (/usr/local/lib/node/.npm/nodeunit/0.5.1/package/lib/core.js:54:9)
    at /usr/local/lib/node/.npm/nodeunit/0.5.1/package/lib/core.js:90:21
    at /usr/local/lib/node/.npm/nodeunit/0.5.1/package/deps/async.js:508:13
    at /usr/local/lib/node/.npm/nodeunit/0.5.1/package/deps/async.js:118:25
    at /usr/local/lib/node/.npm/nodeunit/0.5.1/package/deps/async.js:129:25
    at /usr/local/lib/node/.npm/nodeunit/0.5.1/package/deps/async.js:510:17
    at Array.<anonymous> (/usr/local/lib/node/.npm/nodeunit/0.5.1/package/lib/types.js:144:17)
    at EventEmitter._tickCallback (node.js:108:26)


FAILURES: 1/3 assertions failed (8ms)
```

Nodeunit will list all test cases run within the test, followed by any `AssertionError` output and the number of passing or failing assertions. This is the default (minimal) output.

Nodeunit reporters

Nodeunit ships with a number of reporters and it is possible to add custom reporters.

```
$ nodeunit --list-reporters
Build-in reporters:
  * default: Default tests reporter
  * minimal: Pretty minimal output
  * junit: jUnit XML test reports
  * html: Report tests result as HTML
  * skip_passed: Skip passed tests output
  * browser: Browser-based test reporter
```

To output to junit, run the command as:

```
$ cd src/testing && nodeunit --reporter junit nodeunit_basics.js --output junit.out
```

This creates an xml file (whitespace has been condensed):

```
// testing/junit.out/nodeunit_basics.js.xml
<?xml version="1.0" encoding="UTF-8" ?>
<testsuite name="nodeunit_basics.js"
         errors="0"
         failures="0"
         tests="2">
  <testcase name="Test 1">
  </testcase>
  <testcase name="Test 2">
  </testcase>
</testsuite>
```

This is great, but in a real-world development environment, you may be asked to write reports with a specific format, wording, or links (in html) to files with failing test cases. Luckily, nodeunit allows us to customize this output.

Nodeunit Custom reporters

To write a custom reporter for nodeunit, first decide how you'd like nodeunit to report information about your tests. As a simple example, let's take a look at a different take on the minimal reporter.

Here are the modifications I'd like to see:

- Show the start time of the test
- Show the filename as bold/green

- Add *[PASS]* or *[FAIL]* after the *â* or *â*
- Prefix a test case with a '|'

These may seem like contrived requirements. But, what if my manager wants me to parse textual output for *[FAIL]* and, for whatever reason, it has to say *[FAIL]*?

The modifications to the *minimal* reporter are too spread out to include inline here, so be sure to check out the file at *./src/testing/reporterse/example.js*.

Here is the slightly modified minimal output, meeting all of the requirements.

```
$ cd src/testing && nodeunit --reporter reporters/example.js nodeunit_basics.js
Tests started: Tue Mar 22 2011 21:24:42 GMT-0700 (PDT)
nodeunit_basics.js:

â   [PASS] | Test 1
â   [FAIL] | Test 2


AssertionError: false == true
    at Object.ok (/usr/local/lib/node/.npm/nodeunit/0.5.1/package/lib/types.js:81:39)
    at /home/jim/projects/masteringnode/src/testing/nodeunit_basics.js:10:14
    at Object.runTest (/usr/local/lib/node/.npm/nodeunit/0.5.1/package/lib/core.js:54:9)
    at /usr/local/lib/node/.npm/nodeunit/0.5.1/package/lib/core.js:90:21
    at /usr/local/lib/node/.npm/nodeunit/0.5.1/package/deps/async.js:508:13
    at /usr/local/lib/node/.npm/nodeunit/0.5.1/package/deps/async.js:118:25
    at /usr/local/lib/node/.npm/nodeunit/0.5.1/package/deps/async.js:129:25
    at /usr/local/lib/node/.npm/nodeunit/0.5.1/package/deps/async.js:510:17
    at Array.<anonymous> (/usr/local/lib/node/.npm/nodeunit/0.5.1/package/lib/types.js:144:17)
    at EventEmitter._tickCallback (node.js:108:26)


FAILURES: 1/3 assertions failed (9ms)
```

These were small modifications, but following through the reporters included in nodeunit, it will be easy to output test reports to your desired format.

Nodeunit Mocks and Stubs

...

Expresso

...

Vows

...

Fixtures

...

Deployment

Deploying an application depends on the host environment. Most shared hosts don't currently support node.js.

Heroku is a cloud-based hosting platform with support for Node.js. It also supports Ruby, Clojure, Java, Python, and Scala.

Other hosting platforms include:
Joyent Node
Windows Azure
* Amazon EC2

The level of effort for hosting in these platforms varies. Heroku is relatively simple.

Deploying to Heroku

For complete instructions, refer to Heroku's Documentation.

Assuming you have already created your app and an account on Heroku, the next step is to install the heroku-toolbelt. In Debian/Ubuntu, this can be accomplished with:

```
$ sudo -s
$ echo "deb http://toolbelt.herokuapp.com/ubuntu ./" > /etc/apt/sources.list.d/heroku.list
$ wget -q -O - http://toolbelt.herokuapp.com/apt/release.key | apt-key add -
$ apt-get update
$ apt-get install heroku-toolbelt
```

Once installed, run the command `heroku login` and follow the prompts.

Next, in the root of your app directory, you will need to create a *Procfile* file containing:

```
web: node app.js
```

In this file, `node app.js` is the command you would normally use to run your app locally. If your app requires parameters, either enter those on this line or modify your script to read defaults from `env`.

The Heroku documentation for deploying a node.js suggests you test the *Procfile* using *foreman*. This isn't a necessity. However, if you want to install *foreman*, instructions for Ubuntu can be found here.

Next, you'll need to create a cedar stack and push your code to heroku:

```
$ heroku create --stack cedar
$ git push heroku master
```

You'll also need to configure some settings on heroku using the toolbelt. You'll need to set web scaling to 1 and `NODE_ENV=production`:

```
$ heroku ps:scale web=1
$ heroku config:add NODE_ENV=production
```

With the heroku toolbelt, you can also run commands within your cedar stack. Try the following to become familiar with the toolbelt:

```
$ heroku ps
$ heroku logs
$ heroku run node
```

Heroku also offers free database addons (postgresql, redistogo, etc.). There are also services such as Iris Couch which offer free services that are not tied to the heroku service.

At any time, you can login to heroku.com and modify settings for your newly-created cedar stack. You can also opt-in to logging, notifications, and a plethora of other free and paid services.