

HANDS - ON

NODE.JS

THE NODE.JS INTRODUCTION AND API REFERENCE
BY PEDRO TEIXEIRA

Table of Content

Table of Content	2
Credits	8
About this book	8
Acknowledgements	8
Introduction	9
Why the sudden, exponential popularity?	9
What does this book cover?	9
What does this book not cover?	10
Prerequisites	10
Exercises	10
Source code	10
Where will this book lead you?	10
Chapter Overview	12
Why?	12
Starting up	12
Understanding Basics	12
API Quick Tour	12
Utilities	12
Buffers	12
Event Emitter	12
Timers	12
Low-level File System	13
HTTP	13
Streams	13
TCP Server	13
UNIX Sockets	13
Datagrams (UDP)	13
Child Processes	13
Streaming HTTP Chunked Responses	13
TLS / SSL	13
HTTPS	13
Making Modules	14
Debugging	14
Automated Unit Testing	14
Callback Flow	14
Why?	15
Why the event loop?	15
Solution 1: Create more call stacks	15
Solution 2: Use event callbacks	16
Why Javascript?	17
How I Learned to Stop Fearing and Love Javascript	18
Function Declaration Styles	19
Functions are first-class objects	20
JSLint	22
Javascript versions	22
Handling callbacks	22
References	23

Starting up	25
Install Node	25
NPM - Node Package Manager	26
NPM commands	26
npm ls [filter]	27
npm install package[@filters]	27
npm rm package_name[@version] [package_name[@version] ...]	28
npm view [@] [.]...	28
Understanding	29
Understanding the Node event loop	29
An event-queue processing loop	29
Callbacks that will generate events	30
Don't block!	30
Understanding Modules	31
How Node resolves a module path	32
Core modules	32
Modules with complete or relative path	32
As a file	32
As a directory	32
As an installed module	32
API quick tour	34
Processes	34
process	34
child_process	34
File system	34
fs	34
path	34
Networking	35
net	35
dgram	35
http	35
tls (ssl)	35
https	35
dns	35
Utilities	36
console	36
util	37
Buffers	39
Slice a buffer	39
Copy a buffer	40
Buffer Exercises	40
Exercise 1	40
Exercise 2	40
Exercise 3	40
Event Emitter	42
.addListener	42
.once	42
.removeAllListeners	43
Creating an Event Emitter	43
Event Emitter Exercises	44
Exercise 1	44
Exercise 2	44
Timers	45
setTimeout	45
clearTimeout	45

setInterval	46
clearInterval	46
process.nextTick	46
Escaping the event loop	47
A note on tail recursion	48
Low-level file-system	49
fs.stat and fs.fstat	49
Open a file	50
Read from a file	51
Write into a file	51
Close Your files	52
Advanced Tip: careful when appending concurrently	53
File-system Exercises	55
Exercise 1 - get the size of a file	55
Exercise 2 - read a chunk from a file	55
Exercise 3 - read two chunks from a file	55
Exercise 4 - Overwrite a file	55
Exercise 5 - append to a file	55
Exercise 6 - change the content of a file	56
HTTP	57
HTTP Server	57
The http.ServerRequest object	58
req.url	58
req.method	58
req.headers	58
The http.ServerResponse object	59
Write a header	59
Change or set a header	59
Remove a header	60
Write a piece of the response body	60
HTTP Client	60
http.get()	60
http.request()	61
HTTP Exercises	62
Exercise 1	62
Exercise 2	62
Exercise 3	63
Exercise 4	63
Streams, pump and pipe	64
ReadStream	64
Wait for data	64
Know when it ends	64
Pause it	65
Resume it	65
WriteStream	65
Write	65
Wait for it to drain	66
Some stream examples	66
Filesystem streams	66
Network streams	67
The Slow Client Problem	67
What can we do?	67
Pump	68
... and his cousin "pipe"	68
Making your own	70
ReadStream	70
WriteStream	70
TCP	72

Write a string or a buffer	72
end	73
...and all the other methods	73
Idle sockets	73
Keep-alive	74
Delay or no delay	74
server.close()	74
Listening	74
TCP client	75
Error handling	76
TCP Exercises	76
Exercise 1	76
Exercise 2	76
UNIX sockets	78
Server	78
Client	78
Passing file descriptors around	78
Read or write into that file	79
Listen to the server socket	80
Datagrams (UDP)	82
Datagram server	82
Datagram client	83
Datagram Multicast	84
Receiving multicast messages	85
Sending multicast messages	85
What can be the datagram maximum size?	86
UDP Exercises	86
Exercise 1	86
Child processes	87
Executing commands	87
Spawning processes	88
Killing processes	88
Child Processes Exercises	89
Exercise 1	89
Streaming HTTP chunked responses	90
A streaming example	90
Streaming Exercises	91
Exercise 1	91
TLS / SSL	92
Public / private keys	92
Private key	92
Public key	92
TLS Client	93
TLS Server	93
Verification	94
TLS Exercises	94
Exercise 1	94
Exercise 2	94
Exercise 3	94
Exercise 4	95
Exercise 5	95

HTTPS	96
HTTPS Server	96
HTTPS Client	96
Making modules	98
CommonJS modules	98
One file module	98
An aggregating module	99
A pseudo-class	100
A pseudo-class that inherits	100
node_modules and npm bundle	101
Bundling	102
Debugging	103
console.log	103
Node built-in debugger	103
Node Inspector	104
Live edit	107
Automated Unit Testing	108
A test runner	108
Assertion testing module	109
should.js	109
Assert truthfulness:	109
or untruthfulness:	110
=== true	110
=== false	110
emptiness	110
equality	110
equal (strict equality)	110
assert numeric range (inclusive) with within	110
test numeric value is above given value:	110
test numeric value is below given value:	110
matching regular expressions	110
test length	110
substring inclusion	111
assert typeof	111
property existence	111
array containment	111
own object keys	111
responds to, asserting that a given property is a function:	111
Putting it all together	111
Callback flow	113
The boomerang effect	114
Step	116
Parallel execution	118
Appendix - Exercise Results	120
Chapter: Buffers	120
Exercise 1	120
One solution:	120
Exercise 2	121
One solution:	121
Exercise 3	122
One Solution:	122
Chapter: Event Emitter	123
Exercise 1	123
One solution:	123
Exercise 2	124
One solution:	124
Chapter: Low-level File System	125
Exercise 1 - get the size of a file	125
One solution	125

Exercise 2 - read a chunk from a file	126
One solution	126
Exercise 3 - read two chunks from a file	127
One solution	127
Exercise 4 - Overwrite a file	128
One solution:	128
Exercise 5 - append to a file	129
One Solution:	129
Exercise 6 - change the content of a file	130
One solution:	130
Chapter: HTTP	131
Exercise 1	131
One solution:	131
Exercise 2	132
One solution:	132
Exercise 3	133
One solution:	133
Exercise 4	134
One solution:	134
Chapter: Child processes	135
Exercise 1	135
One solution:	135
Chapter: Streaming HTTP Chunked responses	137
Exercise 1	137
One solution:	137
Chapter: UDP	138
Exercise 1	138
One solution:	138
Chapter: TCP	139
Exercise 1	139
One solution:	139
Exercise 2	140
One solution:	140
Chapter: SSL / TLS	141
Exercise 1	141
One solution:	141
Exercise 2	144
One solution:	144
Exercise 3	145
One solution:	145
Exercise 4	146
One solution:	146
Exercise 5	147
One solution:	147

Credits

This second revision of this book could not have been done without the the help of Timothy Kevin Oxley, who co-reviewed end edited most of this book.

About this book

This book was built using Node.js, Express, the Jade templating engine, wkhtmltopdf, bash scripts, git and github.

Version: 1.1

Acknowledgements

I'm grateful to my dear wife Susana for putting up with my mood swings about this book, to my Father for teaching me how to program Basic on a ZX Spectrum when I was 10 years old, to my Parents and my Grandmother Maria do Carmo for offering me my first IBM PC clone a little later and to my friend Pedro Mendes who gave me the idea for making this book.

Introduction

At the European JSConf 2009, a young programmer by the name of Ryan Dahl, introduced a project he had been working on. This project was a platform that combining Google's V8 Javascript engine and an event loop. The project then took a different direction from other server-side Javascript platforms: all I/O primitives were event-driven, and there was no way around it. Leveraging the power and simplicity of Javascript, it turned the difficult task of writing asynchronous applications into an easy one. Since receiving a standing ovation at the end of his talk, Dahl's project has been met with unprecedented growth, popularity and adoption.

The project was named Node.js, now known to developers simply as 'Node'. Node provides purely evented, non-blocking infrastructure for building highly concurrent software.



Node allows you to easily construct fast and scalable network services.

Why the sudden, exponential popularity?

Server-side Javascript has been around for some time, what makes this platform so appealing?

In previous server-side Javascript implementations, javascript was the *raison d'être*, and the approach focussed on translating common practices from other platforms like Ruby, PERL and Python, into Javascript. Node takes a leap from this and says: "Let's use the successful event-driven programming model of the web and use it to make an easy way to build scalable servers. And let's make it the only way people can do anything on this platform."

It can be argued that Javascript itself contributed to much of Node's success, but that would not explain why other the server-side projects proceeding Node have not yet come close in popularity. The ubiquity of Javascript surely has played a role, but, as Ryan Dahl points out, unlike other SSJS attempts, a unifying client/server language was not the primary goal for Node.

In the perspective of this author, there are three factors contributing to Node's success:

1. Node is Easy - Node makes event-driven I/O programming, the best way to do I/O programming, much easier to understand and achieve than ever before.
2. Node is Lean - Node does not try to solve all problems. It lays the foundation and supports the basic internet protocols using clean, functional APIs.
3. Node does not Compromise - Node does not try to be compatible with pre-existing software, it takes a fresh look at what many believe is the right direction.

What does this book cover?

We will analyze what makes Node a different proposal to other server-side solutions, why you should use it, and how to get started. We will start with an overview but quickly dive into some code module-by-module. By the end of this book you should be able to build and test your own Node modules, service producers/consumers and feel comfortable using Node's conventions and API.

What does this book not cover?

This book does not attempt to cover the complete Node API. Instead, we will cover what the author thinks is required to build most applications he would build on Node.

This book does not cover any Node frameworks; Node is a great tool for building frameworks and many are available, such as cluster management, inter-process communication, web frameworks, network traffic collection tools, game engines and many others. Before you dive into any of those you should be familiar with Node's infrastructure and what it provides to these building blocks.

Prerequisites

This book does not assume you have any prior knowledge of Node, but the code examples are written in Javascript, so familiarity with the Javascript language will help.

Exercises

This book has exercises in some chapters. At the end of this book you can find the exercise solutions, but I advise you to try do them yourself. Consult this book or use the comprehensive API documentation on the official <http://nodejs.org> website.

Source code

You can find some of the source code and exercises used in this book on GitHub:

https://github.com/pgte/handson_nodejs_source_code

or you can download it directly:

https://github.com/pgte/handson_nodejs_source_code/zipball/master

Where will this book lead you?

By the end of it, you should understand the Node API and be able to pursue the exploration of other things built on top of it, being adaptors, frameworks and modules.

Let's get started!

Chapter Overview

While this book does not need to be read from cover to cover, it will help since some common concepts presented earlier in the book will probably not be repeated.

Why?

Why does Node use event-driven programming and Javascript together? Why is Javascript a great language?

Starting up

How to install Node and get Node modules using NPM.

Understanding Basics

How Node loads modules, how the Event Loop functions and what to look out for in order not to block it.

API Quick Tour

Quick overview of Node's core modules.

Utilities

Useful Node utilities.

Buffers

Learn to create, modify and access buffer data, an essential part of the Node fundamentals.

Event Emitter

How the Event Emitter pattern is used throughout Node and how to use it for flexibility in your code.

Timers

Node's timer API, reminiscent of browsers.

Low-level File System

How to use Node to open, read and write files.

HTTP

Node's rich HTTP server and client implementation.

Streams

The richness of this great abstraction in Node.

TCP Server

Quickly setup a bare TCP server.

UNIX Sockets

How to use UNIX sockets and use them to pass file descriptors around.

Datagrams (UDP)

The power of datagrams in Node

Child Processes

Launching, watching, piping and killing other processes.

Streaming HTTP Chunked Responses

HTTP in Node is streamable from the get go.

TLS / SSL

How to provide and consume secure streams.

HTTPS

How to build a secure HTTPS server or client.

Making Modules

How to make your app more modular.

Debugging

How to debug your Node app.

Automated Unit Testing

How to unit test your modules.

Callback Flow

How to manage intricate callback flow in a sane way.

Why?

Why the event loop?

The Event Loop is a software pattern that facilitates non-blocking I/O (network, file or inter-process communication). Traditional blocking programming does I/O in the same fashion as regular function calls; processing may not continue until the operation is complete. Here is some pseudo-code that demonstrates blocking I/O:

```
1. var post = db.query('SELECT * FROM posts where id = 1');
2. // processing from this line onward cannot execute until the
   line above completes
3. doSomethingWithPost(post);
4. doSomethingElse();
```

What is happening here? While the database query is being executed, the whole process/thread idles, waiting for the response. This is called blocking. The response to this query may take many thousands of CPU cycles, rendering the entire process unusable during this time. The process could have been servicing other client requests instead of just waiting.

This does not allow you to parallelize I/O such as performing another database query or communicating with a remote web service, without involving concurrent programming trickery. The call stack becomes frozen, waiting for the database server to reply.

This leaves you with two possible solutions to keep the process busy while it's waiting: create more call stacks or use event callbacks.

Solution 1: Create more call stacks

In order for your process to handle more concurrent I/O, you have to have more concurrent call stacks. For this, you can use threads or some kind of cooperative multi-threading scheme like co-routines, fibers, continuations, etc.

The multi-threaded concurrency model can be very difficult to configure, understand and debug, mainly because of the complexity of synchronization when accessing shared information; you never know when the thread you are running is going to be preempted, which can lead to strange and inconsistent bugs creeping up when the thread's interleaving is not as expected.

On the other hand, cooperative multi-threading is a "trick" where you have more than one stack, and each "thread" of execution explicitly de-schedules itself to give time to another "thread". This can relax the synchronization requirements but can become complex and error-prone, since the thread scheduling is left at the hands of the programmers.

Solution 2: Use event callbacks

An event callback is a function that gets invoked when something significant happens (e.g. the result of a database query is available.)

To use event callbacks on the previous example, you could change it like so:

```
1. callback = function(post) {
2.   doSomethingWithPost(post); // this will only execute when
   the db.query function returns.
3. };
4. db.query('SELECT * FROM posts where id = 1', callback);
5. doSomethingElse(); // this will execute independent of the
   returned status of the db.query call.
```

Here you are defining a function to be invoked when the db operation is complete, then passing this function as an callback argument to the db query operation. The db operation becomes responsible for executing the callback when it completes.

You can use an inline anonymous function to express this in a more compact fashion:

```
1. db.query('SELECT * FROM posts where id = 1',
2.   function(post) {
3.     doSomethingWithPost(post); // this will only execute when
   the db.query function returns.
4.   }
5. );
6. doSomethingElse(); // this will execute independent of the
   returned status of the db.query call.
```

While `db.query()` is executing, the process is free to continue running `doSomethingElse()`, and even service new client requests.

For quite some time, the C systems-programming "hacker" community has known that event-driven programming is the best way to scale a server to handle many concurrent connections. It has been known to be more efficient regarding memory: less context to store, and time: less context-switching.

This knowledge has been infiltrating other platforms and communities: some of the most well-known event loop implementations are Ruby's Event Machine, Perl's AnyEvent and Python's Twisted, and there plenty of others.



Tip: For more info about event-driven server implementations, see http://en.wikipedia.org/wiki/Reactor_pattern.

Implementing an application using one of these frameworks requires framework-specific knowledge and framework-specific libraries. For example: when using Event Machine, you should avoid using synchronous libraries (i.e. most libraries). To gain the benefit of not blocking, you are limited to using only asynchronous libraries, built specifically for Event Machine. Your server will not be able to scale

optimally if the event loop is constantly blocking which prevents timely processing of I/O events.

If you choose to go down the non-blocking rabbit hole, you have to go all the way down, never compromising, and hope you come out on the other side in one piece.

Node has been devised as a non-blocking I/O server platform from day one, so generally you should expect everything built on top of it is non-blocking. Since Javascript itself is very minimal and does not impose any way of doing I/O (it does not have a standard I/O library), Node has a clean slate to build upon.

Why Javascript?

Ryan Dahl began this project building a C platform, but maintaining the context between callbacks was too complicated and could to poorly structured code, so he then turned to Lua. Lua already has several blocking I/O libraries and this mix of blocking and non-blocking could confuse average developers and prevent many of them from building scalable applications, so Lua wasn't ideal either. Dahl then thought to Javascript. Javascript has closures and first-class functions, making it indeed a powerful match with evented I/O programming.

Closures are functions that inherit the variables from their enclosing environment. When a function callback executes it will magically remember the context in which it was declared, along with all the variables available in that context and any "parent" contexts. This powerful feature is at the heart of Node's success among programming communities.

In the web browser, if you want to listen for an event, a button click for instance, you may do something like:

```
1. var clickCount = 0;
2. document.getElementById('mybutton').onclick = function() {
3.
4.     clickCount ++;
5.
6.     alert('Clicked ' + clickCount + ' times.');
```


or, using jQuery:

```
1. var clickCount = 0;
2. $('button#mybutton').click(function() {
3.     clickedCount ++;
4.     alert('Clicked ' + clickCount + ' times.');
```

In both examples we assign or pass a function as an argument, which may be executed later. The click handling function has access every variable in scope at the point where the function is declared, i.e. practically speaking, the click handler has access to the `clickCount` variable, declared in the parent closure.


Here we are using a global variable, "clickCount", where we store the number of times the user has clicked a button. We can also avoid having a global variable accessible to the rest of the system, by wrapping it inside another closure, making the **clicked** variable only accessible within the closure we created:

```
1. (function() {  
2.   var clickCount = 0;  
3.   $('button#mybutton').click(function() {  
4.     clickCount ++;  
5.     alert('Clicked ' + clickCount + ' times.');6.   });  
7. })();
```

 In line 7 we are invoking a function immediately after defining it. If this is strange to you, don't worry! We will cover this pattern later.

How I Learned to Stop Fearing and Love Javascript

Javascript has good and bad parts. It was created in 1995 by Netscape's Brendan Eich, in a rush to ship the latest version of the Netscape web browser. Due to this rush some good, even wonderful, parts got into Javascript, but also some bad parts.

 This book will not cover the distinction between Javascript good and bad parts. (For all we know, we will only provide examples using the good parts.) For more on this topic you should read Douglas Crockford book named "Javascript, The Good Parts", edited by O'Reilly.

In spite of its drawbacks, Javascript quickly - and somewhat unpredictably - became the de-facto language for web browsers. Back then, Javascript was used to primarily to inspect and manipulate HTML documents, allowing the creation the first dynamic, client-side web applications.

In late 1998, the World Wide Web Consortium (W3C), standardized the Document Object Model (DOM), an API devised to inspect and manipulate HTML documents on the client side. In response to Javascript's quirks and the initial hatred towards the DOM API, Javascript quickly gained a bad reputation, also due to some incompatibilities between browser vendors (and sometimes even between products from the same vendor!).

Despite mild to full-blown hate in some developer communities, Javascript became widely adopted. For better or for worse, today Javascript is the most widely deployed programming language on Earth.

If you learn the good features of the language - such as prototypical inheritance, function closures, etc. - and learn to avoid or circumvent the bad parts, Javascript can be a very pleasant language to work in.

Function Declaration Styles

A function can be declared in many ways in Javascript. The simplest is declaring it anonymously:

```
1. function() {  
2.   console.log('hello');  
3. }
```

Here we declare a function, but it's not of much use, because do not invoke it. What's more, we have no way to invoke it as it has no name.

We can invoke an anonymous function in-place:

```
1. (function() {  
2.   console.log('hello');  
3. })();
```

Here we are executing the function immediately after declaring it. Notice we wrap the entire function declaration in parenthesis.

We can also name functions like this:

```
1. function myFunction () {  
2.   console.log('hello');  
3. }
```

Here we are declaring a named function with the name: "myFunction". myFunction will be available inside the scope it's declared

```
myFunction();
```

and also within inner scopes:

```
1. function myFunction () {  
2.   console.log('hello');  
3. }  
4.  
5. (function() {  
6.   myFunction();  
7. })();
```

A result of Javascript treating functions as first-class objects means we can assign a function to a variable:

```
1. var myFunc = function() {  
2.   console.log('hello');  
3. }  
4.
```

This function is now available as the value of the **myFunc** variable.

We can assign that function to another variable:

```
var myFunc2 = myFunc;
```

And invoke them just like any other function:

```
1. myFunc();
2. myFunc2();
```

We can mix both techniques, having a named function stored in a variable:

```
1. var myFunc2 = function myFunc() {
2.   console.log('hello');
3. }
4. myFunc2();
```

Note though, we cannot access myFunc from outside the scope of myFunc itself!

We can then use a variable or a function name to pass variables into functions like this:

```
1. var myFunc = function() {
2.   console.log('hello');
3. }
4.
5. console.log(myFunc);
```

or simply declare it inline if we don't need it for anything else:

```
1. console.log(function() {
2.   console.log('hello');
3. }());
```

Functions are first-class objects

In fact, there are no second-class objects in Javascript. Javascript is the ultimate object-oriented language, where everything is indeed, an object. As that, a function is an object where you can set properties, pass it around inside arguments and return them. Always a pleasure.

Example:

```
1. var schedule = function(timeout, callbackfunction) {
2.     return {
3.         start: function() {
4.             setTimeout(callbackfunction, timeout)
5.         }
6.     };
7. };
8.
9. (function() {
10.     var timeout = 1000; // 1 second
11.     var count = 0;
12.     schedule(timeout, function doStuff() {
13.         console.log(++ count);
14.         schedule(timeout, doStuff);
15.     }).start(timeout);
16. })();
17.
18. // "timeout" and "count" variables
19. // do not exist on this scope.
```

In this little example we create a function and store it in a function called "schedule" (starting on line 1). This function just returns an object that has one property called "start" (line 3). This value of the "start" property is a function that, when called, sets a timeout (line 4) to call a function that is passed in as the "timeout" argument. This timeout will schedule callback function to be called within the number of seconds defined in the `timeout` variable passes.

On line 9 we declare a function that will immediately be executed on line 16. This is a normal way to create new scopes in Javascript. Inside this scope we create 2 variables: "timeout" (line 10) and "count" (line 11). Note that these variables will not be accessible to the outer scope.

Then, on line 12, we invoke the `schedule` function, passing in the timeout value as first argument and a function called `doStuff` as second argument. When the timeout occurs, this function will increment the variable `count` and log it, and also call the schedule all over again.

So in this small example we have: functions passed as argument, functions to create scope, functions to serve as asynchronous callbacks and returning functions. We also here present the notions of encapsulation (by hiding local variables from the outside scope) and recursion (the function is calling itself at the end).

In Javascript you can even set and access attributes in a function, something like this:

```
1. var myFunction = function() {
2.     // do something crazy
3. };
4. myFunction.someProperty = 'abc';
5. console.log(myFunction.someProperty);
6. // #=> "abc"
```

Javascript is indeed a powerful language, and if you don't already do, you should learn it and embrace

it's good parts.

JSLint

It's not to be covered here, but Javascript indeed has some bad parts, and they should be avoided at all costs.

One tool that's proven invaluable to the author is JSLint, by Douglas Crockford. JSLint analyzes your Javascript file and outputs a series of errors and warnings, including some known misuses of Javascript, like using globally-scoped variables (like when you forget the "var" keyword), and freezing values inside iteration that have callbacks that use them, and many others that are useful.

JSLint can be installed using

```
$ npm install jshint
```



If you don't have NPM installed see section about [NPM](#).

and can be run from the command line like this:

```
$ jshint myfile.js
```

To the author of this book, JSLint has proven itself to be an invaluable tool to guarantee he doesn't fall into some common Javascript traps.



Javascript versions

Javascript is a standard with it's own name - ECMAScript - and it has gone through various iterations. Currently Node natively supports everything the V8 Javascript engine supports ECMA 3rd edition and parts of the new ECMA 5th edition.

These parts of ECMA 5 are nicely documented on the following github wiki page:

<https://github.com/joyent/node/wiki/ECMA-5-Mozilla-Features-Implemented-in-V8>

Handling callbacks

In Node you can implement your own functions that perform asynchronous I/O.

To do so, you can accept a callback function. You will invoke this function when the I/O is done.

```
1. var myAsyncFunction = function(someArgument1, someArgument2,
   callback) {
2.   // simulate some I/O was done
3.   setTimeout(function() {
4.     // 1 second later, we are done with the I/O, call the
       callback
5.     callback();
6.   }, 1000)
7. }
```

On line 3 we are invoking a `setTimeout` to simulate the delay and asynchronism of an I/O call. This `setTimeout` function will call the first argument - a function we declare inline - after 1000 milliseconds (the second argument) have gone by.

This inline function will then call the callback that was passed as the third argument to `myAsyncFunction`, notifying the caller the operation has ended.

Using this convention and others (like the Event Emitter pattern which we will cover later) you can embrace Javascript as the ultimate language for event-driven applications.



Tip: To follow the Node convention, this function should receive the error (or null if there was no error) as first argument, and then some "real" arguments if you wish to do so.

```
1. fs.open('/path/to/file', function(err, fd) {
2.   if (err) { /* handle error */; return; }
3.   console.log('opened file and got file descriptor ' +
       fd);
4. })
```

Here we are using a Node API function `fs.open` that receives 2 arguments: the path to the file and a function, which will be invoked with an error or null on the first argument and a file descriptor on the second.

References

Event Machine: <http://rubyeventmachine.com/>

Twisted: <http://twistedmatrix.com/trac/>

AnyEvent: <http://software.schmorp.de/pkg/AnyEvent.html>

Javascript, the Good Parts - Douglas Crockford - O'Reilly -
<http://www.amazon.com/exec/obidos/ASIN/0596517742/wrrldwideweb>

JSLint <http://www.jshint.com/>

Starting up

Install Node

The typical way of installing Node on your development machine is by following the steps on the nodejs.org website. Node should install out of the box on Linux, Macintosh, and Solaris.



With some effort you should be able to get it running on other Unix platforms and Windows (either via Cygwin or MinGW).



Node has several dependencies, but fortunately most of them are distributed along with it. If you are building from source you should only need 2 things:

- python - version 2.4 or higher. The build tools distributed with Node run on python.
- libssl-dev - If you plan to use SSL/TLS encryption in your networking, you'll need this. Libssl is the library used in the openssl tool. On Linux and Unix systems it can usually be installed with your favorite package manager. The lib comes pre-installed on OS X.

Pick the latest stable version and download it:

```
$ wget http://nodejs.org/dist/node-v0.4.7.tar.gz
```

Expand it:

```
$ tar xvfz node-v0.4.7.tar.gz
```

Build it:

1. `$ cd node-v0.4.7`
2. `$./configure`
3. `$ make`
4. `$ make install`



Tip: if you are having permission problems on this last step you should run the install step as a super user like by:

```
$ sudo make install
```

After you are done, you should be able to run the node executable on the command line:

1. `$ node -v`
2. `v0.4.7`

The **node** executable can be executed in two main fashions: CLI (command-line interface) or file.

To launch the CLI, just type

```
$ node
```

and you will get a Javascript command line prompt, which you can use to evaluate Javascript. It's great for kicking the tires and trying out some stuff quickly.

You can also launch Node on a file, which will make Node parse and evaluate the Javascript on that file, and when it ends doing that, it enters the event loop. Once inside the event loop, node will exit if it has nothing to do, or will wait and listen for events.

You can launch Node on a file like this:

```
$ node myfile.js
```

or, if you wish, you can also make your file directly executable by changing the permissions like this:

```
$ chmod o+x myfile.js
```

and insert the following as the first line of the file:

```
#!/usr/bin/env node
```

You can then execute the file directly:

```
$ ./myfile.js
```

NPM - Node Package Manager

NPM has become the standard for managing Node packages throughout time, and tight collaboration between Isaac Schlueter - the original author of NPM - and Ryan Dahl - the author and maintainer on Node - has further tightened this relationship to the point where, starting at version 0.4.0, Node supports the **package.json** file format to indicate dependencies and package starting file.

To install it you type:

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



If that fails, try this on a temporary directory:

1.

```
$ git clone http://github.com/isaacs/npm.git
```
2.

```
$ cd npm
```
3.

```
$ sudo make install
```

NPM commands

NPM can be used on the command line. The basic commands are:

npm ls [filter]

Use this to see the list of all packages and their versions (npm ls with no filter), or filter by a tag (npm filter tag). Examples:

List all installed packages:

```
$ npm ls installed
```

List all stable packages:

```
$ npm ls stable
```

You can also combine filters:

```
$ npm ls installed stable
```

You can also use npm ls to search by name:

```
$ npm ls fug
```

(this will return all packages that have "fug" inside its name or tags)

You can also query it by version, prefixed with the "@" character:

```
$ npm ls @1.0
```

npm install package[@filters]

With this command you can install a package and all the packages it depends on.

To install the latest version of a package do:

```
$ npm install package_name
```

Example:

```
$ npm install express
```

To install a specific version of a package do:

```
$ npm install package_name@version
```

Example:

```
$ npm install express@2.0.0beta
```

To install the latest within a version range you can specify, for instance:

```
$ npm install express@>=0.1.0 <0.2.0"
```

You can also combine many filters to select a specific version, combining version range and / or tags like this:

```
$ npm install sax@">=0.1.0 <0.2.0" bench supervisor
```

npm rm package_name[@version] [package_name[@version] ...]

Use this command to uninstall packages. If versions are omitted, then all the found versions are removed.

Example:

```
$ npm rm sax
```

npm view [@] [[.]...]

To view all of a package info. Defaults to the latest version if version is omitted.

View the latest info on the "connect" package:

```
$ npm view connect
```

View information about a specific version:

```
$ npm view connect@1.0.3
```



Further on we will look more into NPM and how it can help us bundle and "freeze" application dependencies.

Understanding

Understanding the Node event loop

Node makes evented I/O programming simple and accessible, putting speed and scalability on the fingertips of the common programmer.

But the event loop comes with a price. Even though you are not aware of it (and Node makes a good job at this), you should understand how it works. Every good programmer should know the intricacies of the platforms he / she is building for, its do's and don'ts, and in Node it should be no different.

An event-queue processing loop

You should think of the event loop as a loop that processes an event queue. Interesting events happen, and when they do, they go in a queue, waiting for their turn. Then, there is an event loop popping out these events, one by one, and invoking the associated callback functions, one at a time. The event loop pops one event out of the queue and invokes the associated callback. When the callback returns the event loop pops the next event and invokes the associated callback function. When the event queue is empty, the event loop waits for new events if there are some pending calls or servers listening, or just quits if there are none.

So, let's jump into our first Node example. Write a file named `hello.js` with the following content:

Source code in
chapters/understanding/1_hello.js

```
1. setTimeout(function() {  
2.   console.log('World!');  
3. }, 2000);  
4. console.log('Hello');
```

Run it using the node command line tool:

```
$ node hello.js
```

You should see the word "Hello" written out, and then, 2 seconds later, "World!". Shouldn't "World!" have been written first since it appears first on the code? No, and to answer that properly we must analyze what happens when executing this small program.

On line 1 we declare an anonymous function that prints out "World!". This function, which is not yet executed, is passed in as the first argument to a `setTimeout` call, which schedules this function to run in 2000 milliseconds. Then, on line 4, we output "Hello" into the console.

Two seconds later the anonymous function we passed in as an argument to the `setTimeout` call is invoked, printing "World!".

So, the first argument to the `setTimeout` call is a function we call a "callback". It's a function which will be called later, when the event we set out to listen to (in this case, a time-out of 2 seconds) occurs.



We can also pass callback functions to be called on events like when a new TCP connection is established, some file data is read or some other type of I/O event.

After our callback is invoked, printing "World", Node understands that there is nothing more to do and exits.

Callbacks that will generate events

Let's complicate this a bit further. Let's keep Node busy and keep on scheduling callbacks like this:

Source code in
`chapters/understanding/2_repeat.js`

```
1. (function schedule() {
2.   setTimeout(function() {
3.     console.log('Hello World!');
4.     schedule();
5.   }, 1000);
6. })();
```

Here we are wrapping the whole thing inside a function named "schedule", and we are invoking it immediately after declaring it on line 6. This function will schedule a callback to execute in 1 second. This callback, when invoked, will print "Hello World!" and then run **schedule** again.


On every callback we are registering a new one to be invoked one second later, never letting Node finish. This little script will just keep printing "Hello World".


Don't block!

Node primary concern and the main use case for an event loop is to create highly scalable servers. Since an event loop runs in a single thread, it only processes the next event when the callback finishes. If you could see the call stack of a busy Node application you would see it going up and down really fast, invoking callbacks and picking up the next event in line. But for this to work well you have to clear the event loop as fast as you can.

There are two main categories of things that can block the event loop: synchronous I/O and big loops.

Node API is not all asynchronous. Some parts of it are synchronous like, for instance, some file operations. Don't worry, they are very well marked: they always terminate in "Sync" - like `fs.readFileSync` -, and they should not be used, or used only when initializing. On a working server you should never use a blocking I/O function inside a callback, since you're blocking the event loop and preventing other callbacks - probably belonging to other client connections - from being served.

 One function that is synchronous and does not end in "Sync" is the "require" function, which should only be used when initializing an app or a module.

 Tip: Don't put a **require** statement inside a callback, since it is synchronous and thus will slow down your event loop.

The second category of blocking scenarios is when you are performing loops that take a lot of time, like iterating over thousands of objects or doing complex time-taking operations in memory. There are several techniques that can be used to work around that, which we'll cover later.

Here is a case where we present some simple code that blocks the event loop:

```
1. var open = false;
2.
3. setTimeout(function() {
4.   open = true;
5. }, 1000)
6.
7. while(!open) {
8.   // wait
9. }
10.
11. console.log('opened!');
```

Here we are setting a timeout, on line 3, that invokes a function that will set the **open** variable to true. This function is set to be triggered in one second.

On line 7 we are waiting for the variable to become true.

We could be lead to believe that, in one second the timeout will happen and set **open** to **true**, and that the **while** loop will stop and that we will get "opened!" (line 11) printed.

But this never happens. Node will never execute the timeout callback because the event loop is stuck on this while loop started on line 7, never giving it a chance to process the timeout event!

Understanding Modules

Client-side Javascript has a bad reputation also because of the common namespace shared by all scripts, which can lead to conflicts and security leaks.

Node implements the CommonJS modules standard, where each module is separated from the other modules, having a separate namespace to play with, and exporting only the desired properties.

To include an existing module you can use the **require** function like this:

```
var module = require('module_name');
```

This will fetch a module that was installed by npm. If you want to author modules (as you should

when doing an application), you can also use the relative notation like this:

```
var module = require("./path/to/module_name");
```

This will fetch the module relatively to the current file we are executing. We will cover creating modules on a later section.



In this format you can use an absolute path (starting with "/") or a relative one (starting with ".");

Modules are loaded only once per process, that is, when you have several **require** calls to the same module, Node caches the require call if it resolves to the same file.

Which leads us to the next chapter.

How Node resolves a module path

So, how does node resolve a call to "require(module_path)"? Here is the recipe:

Core modules

There are a list of core modules, which Node includes in the distribution binary. If you require one of those modules, Node just returns that module and the require() ends.

Modules with complete or relative path

If the module path begins with "./" or "/", Node tries to load the module as a file. If it does not succeed, it tries to load the module as a directory.

As a file

When loading as a file, if the file exists, Node just loads it as Javascript text.

If not, it tries doing the same by appending ".js" to the given path.

If not, it tries appending ".node" and load it as a binary add-on.

As a directory

If appending "/package.json" is a file, try loading the package definition and look for a "main" field. Try to load it as a file.

If unsuccessful, try to load it by appending "/index" to it.

As an installed module

If the module path does not begin with "." or "/" or if loading it with complete or relative paths does not work, Node tries to load the module as a module that was previously installed. For that it adds "/node_modules" to the current directory and tries to load the module from there. If it does not

succeed it tries adding `"/node_modules"` to the parent directory and load the module from there. If it does not succeed it moves again to the parent directory and so on, until either the module is found or the root of the tree is found.

This means that you can put your bundle your Node modules into your app directory, and Node will find those.

Later we will see how using this feature together with NPM we can bundle and "freeze" your application dependencies.



Also you can, for instance, have a `node_modules` directory on the home folder of each user, and so on. Node tries to load modules from these directories, starting first with the one that is closest up the path.

API quick tour

Node provides a platform API that covers mainly 4 aspects:

- processes
- filesystem
- networking
- utilities



This book is not meant to be a comprehensive coverage of the whole Node API. For that you should consult the Node online documentation on <http://nodejs.org>.

Processes

Node allows you to analyze your process (environment variables, etc.) and manage external processes. The involved modules are:

process

Inquire the current process to know the PID, environment variables, platform, memory usage, etc.

child_process

Spawn and kill new processes, execute commands and pipe their outputs.

File system

Node also provides a low-level API to manipulate files, which is inspired by the POSIX standard, and is comprised by the following modules:

fs

File manipulation: create, remove, load, write and read files. Create read and write streams (covered later).

path

Normalize and join file paths. Check if a file exists or is a directory.

Networking

net

Create a TCP server or client.

dgram

Receive and send UDP packets.

http

Create an HTTP server or Client.

tls (ssl)

The tls module uses OpenSSL to provide Transport Layer Security and/or Secure Socket Layer: encrypted stream communication.

https

Implementing http over TLS/SSL.

dns

Asynchronous DNS resolution.

Utilities

console

Node provides a global "console" object to which you can output strings using:

```
console.log("Hello");
```

This simply outputs the string into the process `stdout` after formatting it. You can pass in, instead of a string, an object like this:

```
1. var a = {1: true, 2: false};
2. console.log(a); // => { '1': true, '2': false }
```

In this case `console.log` outputs the object using `util.inspect` (covered later);

You can also use string interpolation like this:

```
1. var a = {1: true, 2: false};
2. console.log('This is a number: %d, and this is a string: %s,
    and this is an object outputted as JSON: %j', 42, 'Hello', a);
3.
```

Which outputs:

```
This is a number: 42, and this is a string: Hello, and this is an
object outputted as JSON: {"1":true,"2":false}
```

`console` also allows you to write into the `stderr` using:

```
console.warn("Warning!");
```

and to print a stack trace:

```
console.trace();
```

```
1. Trace:
2.   at [object Context]:1:9
3.   at Interface.<anonymous> (repl.js:171:22)
4.   at Interface.emit (events.js:64:17)
5.   at Interface._onLine (readline.js:153:10)
6.   at Interface._line (readline.js:408:8)
7.   at Interface._ttyWrite (readline.js:585:14)
8.   at ReadStream.<anonymous> (readline.js:73:12)
9.   at ReadStream.emit (events.js:81:20)
10.  at ReadStream._emitKey (tty_posix.js:307:10)
11.  at ReadStream.onData (tty_posix.js:70:12)
12.
```

util

Node has an `util` module which which bundles some functions like:

```
1. var util = require('util');
2. util.log('Hello');
```

which outputs a the current timestamp and the given string like this:

```
14 Mar 16:38:31 - Hello
```

The `inspect` function is a nice utility which can aid in quick debugging by inspecting and printing an object properties like this:

```
1. var util = require('util');
2. var a = {1: true, 2: false};
3. console.log(util.inspect(a));
4. // => { '1': true, '2': false }
```

`util.inspect` accepts more arguments, which are:

```
util.inspect(object, showHidden, depth = 2, showColors);
```

the second argument, `showHidden` should be turned on if you wish `inspect` to show you non-enumerable properties, which are properties that belong to the object prototype chain, not the object itself. `depth`, the third argument, is the default depth on the object graph it should show. This is useful for inspecting large objects. To recurse indefinitely, pass a `null` value.



Tip: `util.inspect` keeps track of the visited objects, so circular dependencies are no problem, and will appear as "[Circular]" on the outputted string.

The `util` module has some other niceties, such as inheritance setup and stream pumping, but they belong on more appropriate chapters.

Buffers

Natively, Javascript is not very good at handling binary data. So Node adds a native buffer implementation with a Javascript way of manipulating it. It's the standard way in Node to transport data.



Generally, you can pass buffers on every Node API requiring data to be sent. Also, when receiving data on a callback, you get a buffer (except when you specify a stream encoding, in which case you get a String). This will be covered later.

You can create a Buffer from an UTF8 string like this:

```
var buf = new Buffer('Hello World!');
```

You can also create a buffer from strings with other encodings, as long as you pass it as the second argument:

```
var buf = new Buffer('8b76fde713ce', 'base64');
```

Accepted encodings are: "ascii", "utf8" and "base64".

or you can create a new empty buffer with a specific size:

```
var buf = new Buffer(1024);
```

and you can manipulate it:

```
buf[20] = 56; // set byte 20 to 56
```

You can also convert it to a UTF-8-encoded string:

```
var str = buf.toString();
```

or into a string with an alternative encoding:

```
var str = buf.toString('base64');
```



UTF-8 is the default encoding for Node, so, in a general way, if you omit it as we did on the `buffer.toString()` call, UTF-8 will be assumed.

Slice a buffer

A buffer can be sliced into a smaller buffer by using the appropriately named `slice()` method like this:

1. `var buffer = new Buffer('this is the string in my buffer');`
2. `var slice = buffer.slice(10, 20);`

Here we are slicing the original buffer that has 31 bytes into a new buffer that has 10 bytes equal to the 10th to 20th bytes on the original buffer.

Note that the slice function does not create new buffer memory: it uses the original untouched buffer underneath.



Tip: If you are afraid you will be wasting precious memory by keeping the old buffer around when slicing it, you can copy it into another like this:

Copy a buffer

You can copy a part of a buffer into another pre-allocated buffer like this:

1. `var buffer = new Buffer('this is the string in my buffer');`
2. `var slice = new Buffer(10);`
3. `var targetStart = 0,`
4. `sourceStart = 10,`
5. `sourceEnd = 20;`
6. `buffer.copy(slice, targetStart, sourceStart, sourceEnd);`

Here we are copying part of `buffer` into `slice`, but only positions 10 through 20.

Buffer Exercises

Exercise 1

Create an uninitialized buffer with 100 bytes length and fill it with bytes with values starting from 0 to 99. And then print its contents.

Exercise 2

Do what is asked on the previous exercise and then slice the buffer with bytes ranging 40 to 60. And then print it.

Exercise 3

Do what is asked on exercise 1 and then copy bytes ranging 40 to 60 into a new buffer. And then print it.

Event Emitter

On Node many objects can emit events. For instance, a TCP server can emit a 'connect' event every time a client connects. Or a file stream request can emit a 'data' event.

.addListener

You can listen for these events by calling one of these objects "addListener" method, passing in a callback function. For instance, a file ReadStream can emit a "data" event every time there is some data available to read.

Instead of using the "addListener" function, you can also use "on", which is exactly the same thing:

```
1. var fs = require('fs'); // get the fs module
2. var readStream = fs.createReadStream('/etc/passwd');
3. readStream.on('data', function(data) {
4.   console.log(data);
5. });
6. readStream.on('end', function() {
7.   console.log('file ended');
8. });
```

Here we are binding to the `readStream`'s "data" and "end" events, passing in callback functions to handle each of these cases. When one of these events happens, the `readStream` will call the callback function we pass in.

You can either pass in an anonymous function as we are doing here, or you can pass a function name for a function available on the current scope, or even a variable containing a function.

.once

You may also want to listen for an event exactly once. For instance, if you want to listen to the first connection on a server, you should do something like this:

```
1. server.once('connection', function (stream) {
2.   console.log('Ah, we have our first user!');
3. });
```

This works exactly like our "on" example, except that our callback function will be called at most once. It has the same effect as the following code:

```
1. function connListener(stream) {
2.   console.log('Ah, we have our first user!');
3.   server.removeListener('connection', connListener);
4. }
5. server.on('connection', connListener);
```

Here we are using the `removeListener`, which also belongs to the EventEmitter pattern. It accepts the event name and the function it should remove.

.removeAllListeners

If you ever need to, you can also remove all listeners for an event from an Event Emitter by simply calling

```
server.removeAllListeners('connection');
```

Creating an Event Emitter

If you are interested on using this Event Emitter pattern - and you should - throughout your application, you can. You can create a pseudo-class and make it inherit from the EventEmitter like this:

```
1. var EventEmitter = require('events').EventEmitter,
2.   util           = require('util');
3.
4. // Here is the MyClass constructor:
5. var MyClass = function(option1, option2) {
6.   this.option1 = option1;
7.   this.option2 = option2;
8. }
9.
10. util.inherits(MyClass, EventEmitter);
```



`util.inherits` is setting up the prototype chain so that you get the `EventEmitter` prototype methods available to your `MyClass` instances.

This way instances of `MyClass` can emit events:

```
1. MyClass.prototype.someMethod = function() {
2.   this.emit('custom event', 'some arguments');
3. }
4.
```

Here we are emitting an event named "custom event", sending also some data ("some arguments" in this case);

Now clients of MyClass instances can listen to "custom events" events like this:

```
1. var myInstance = new MyClass(1, 2);
2. myInstance.on('custom event', function() {
3.   console.log('got a custom event!');
4. });
```



Tip: The Event Emitter is a nice way of enforcing the decoupling of interfaces, a software design technique that improves the independence from specific interfaces, making your code more flexible.

Event Emitter Exercises

Exercise 1

Build a pseudo-class named "Ticker" that emits a "tick" event every 1 second.

Exercise 2

Build a script that instantiates one Ticker and bind to the "tick" event, printing "TICK" every time it gets one.

Timers

Node implements the timers API also found in web browsers. The original API is a bit quirky, but it hasn't been changed for the sake of consistency.

setTimeout

`setTimeout` lets you schedule an arbitrary function to be executed in the future. An example:

```
1. var timeout = 2000; // 2 seconds
2. setTimeout(function() {
3.   console.log('timed out!');
4. }, timeout);
5.
```

This code will register a function to be called when the timeout expires. Again, as in any place in Javascript, you can pass in an inline function, the name of a function or a variable which value is a function.



You can use `setTimeout` with a timeout value of 0 so that the function you pass gets executed some time after the stack clears, but with no waiting. This can be used to, for instance schedule a function that does not need to be executed immediately.

This was a trick sometimes used on browser Javascript, but, as we will see, Node `process.nextTick()` can be used instead of this, and it's more efficient.

clearTimeout

`setTimeout` returns a timeout handle that you can use to disable it like this:

```
1. var timeoutHandle = setTimeout(function() {
   console.log('yehaa!'); }, 1000);
2. clearTimeout(timeoutHandle);
```

Here the timeout will never execute because we clear it right after we set it.

Another example:

Source code in
chapters/timers/timers_1.js

```
1. var timeoutA = setTimeout(function() {
2.   console.log('timeout A');
3. }, 2000);
4.
5. var timeoutB = setTimeout(function() {
6.   console.log('timeout B');
7.   clearTimeout(timeoutA);
8. }, 1000);
9.
```

Here we are starting two timers: one with 1 second (timeoutB) and the other with 2 seconds (timeoutA). But timeoutB (which fires first) unschedules timeoutA on line 7, so timeoutA is never executes - and the program exits right after line 7 is executed.

setInterval

Set interval is similar to set timeout, but schedules a given function to run every X seconds like this:

Source code in
chapters/timers/timers_2.js

```
1. var period = 1000; // 1 second
2. var interval = setInterval(function() {
3.   console.log('tick');
4. }, period);
```

This will indefinitely keep the console logging "tick" unless you terminate Node. You can unschedule an interval by calling:

clearInterval

clearInterval unschedules a running interval (previous scheduled with **setInterval**).

```
1. var interval = setInterval(...);
2. clearInterval(interval);
3.
```

Here we are using the setInterval return value stored on the **interval** variable to unschedule it on line 2.

process.nextTick

You can also schedule a callback function to run on the next run of the event loop. You can use it like this:

```
1. process.nextTick(function() {  
2.   // this runs on the next event loop  
3.   console.log('yay!');  
4. });
```



As we saw, this method is preferred to `setTimeout(fn, 0)` because it is more efficient.

Escaping the event loop

On each loop, the event loop executes the queued I/O events sequentially by calling the associated callbacks. If, on any of the callbacks you take too long, the event loop won't be processing other pending I/O events meanwhile. This can lead to waiting customers or tasks. When executing something that may take too long, you can delay the execution until the next event loop, so waiting events will be processed meanwhile. It's like going to the back of the line on a waiting line.

To escape the current event loop you can use `process.nextTick()` like this:

```
1. process.nextTick(function() {  
2.   // do something  
3. });
```

You can use this to delay processing that is not necessary to do immediately to the next event loop.

For instance, you may need to remove a file, but perhaps you don't need to do it before replying to the client. So, you could do something like this:

```
1. stream.on('data', function(data) {  
2.   stream.end('my response');  
3.   process.nextTick(function() {  
4.     fs.unlink('path/to/file');  
5.   })  
6. });
```



A note on tail recursion

Let's say you want to schedule a function that does some I/O - like parsing a log file - to execute periodically, and you want to guarantee that no two of those functions are executing at the same time. The best way is not to use a `setInterval`, since you don't have that guarantee. The interval will fire no matter if the function has finished its duty or not.

Supposing there is an asynchronous function called "async" that performs some IO and that gets a callback to be invoked when finished, and you want to call it every second:

```
1. var interval = 1000; // 1 second
2. setInterval(function() {
3.     async(function() {
4.         console.log('async is done!');
5.     });
6. }, interval);
```

If any two `async()` calls can't overlap, you are better off using tail recursion like this:

```
1. var interval = 1000; // 1 second
2. (function schedule() {
3.     setTimeout(function() {
4.         async(function() {
5.             console.log('async is done!');
6.             schedule();
7.         });
8.     }, interval)
9. })();
```

Here we are declaring a function named `schedule` (line 2) and we are invoking it immediately after we are declaring it (line 9).

This function schedules another function to execute within one second (line 3 to 8). This other function will then call `async()` (line 4), and only when **async** is done we schedule a new one by calling `schedule()` again (line 6), this time inside the `schedule` function. This way we can be sure that no two calls to **async** execute simultaneously in this context.

The difference is that we probably won't have **async** called every second (unless `async` takes no time to execute), but we will have it called 1 second after the last one finished.

Low-level file-system

Node has a nice streaming API for dealing with files in an abstract way, as if they were network streams, but sometimes you might need to go down a level and deal with the filesystem itself.

First, a nice set of utilities:

fs.stat and fs.fstat

You can query some meta-info on a file (or dir) by using `fs.stat` like this:

```
1. var fs = require('fs');
2.
3. fs.stat('/etc/passwd', function(err, stats) {
4.   if (err) {console.log(err.message); return; }
5.   console.log(stats);
6.   //console.log('this file is ' + stats.size + ' bytes
   long. ');
7. });
```

If you print the stats object it will be something like:

```
1. { dev: 234881026,
2.   ino: 24606,
3.   mode: 33188,
4.   nlink: 1,
5.   uid: 0,
6.   gid: 0,
7.   rdev: 0,
8.   size: 3667,
9.   blksize: 4096,
10.  blocks: 0,
11.  atime: Thu, 17 Mar 2011 09:14:12 GMT,
12.  mtime: Tue, 23 Jun 2009 06:19:47 GMT,
13.  ctime: Fri, 14 Aug 2009 20:48:15 GMT
14. }
```

`stats` is a Stats instance, with which you can call:

1. `stats.isFile()`
2. `stats.isDirectory()`
3. `stats.isBlockDevice()`
4. `stats.isCharacterDevice()`
5. `stats.isSymbolicLink()`
6. `stats.isFIFO()`
7. `stats.isSocket()`



If you have a plain file descriptor you can use `fs.fstat(fileDescriptor, callback)` instead.

More about file descriptors later.



If you are using the low-level filesystem API in Node, you will get file descriptors as a way to represent files. These file descriptors are plain integer numbers that represent a file in your Node process, much like in C POSIX APIs.

Open a file

You can open a file by using `fs.open` like this:

```
1. var fs = require('fs');
2. fs.open('/path/to/file', 'r', function(err, fd) {
3.   // got fd
4. });
```

The first argument to `fs.open` is the file path. The second argument is the flags, which indicate the mode with which the file is to be open. The flags can be 'r', 'r+', 'w', 'w+', 'a', or 'a+'.

Here is the semantics of each flag, taken from the `fopen` man page:

- **r** - Open text file for reading. The stream is positioned at the beginning of the file.
- **r+** - Open for reading and writing. The stream is positioned at the beginning of the file.
- **w** - Truncate file to zero length or create text file for writing. The stream is positioned at the beginning of the file.
- **w+** - Open for reading and writing. The file is created if it does not exist, otherwise it is truncated. The stream is positioned at the beginning of the file.
- **a** - Open for writing. The file is created if it does not exist. The stream is positioned at the end of the file. Subsequent writes to the file will always end up at the then current end of file.
- **a+** - Open for reading and writing. The file is created if it does not exist. The stream is positioned at the end of the file. Subsequent writes to the file will always end up at the then current end of file.

On the callback function, you get a second argument (`fd`), which is a file descriptor- nothing more than an integer that identifies the open file, which you can use like a handler to read and write from.

Read from a file

Once it's open, you can also read from a file like this:

Source code in
chapters/fs/read.js

```
1. var fs = require('fs');
2. fs.open('/var/log/system.log', 'r', function(err, fd) {
3.   if (err) { throw err }
4.   var readBuffer = new Buffer(1024),
5.     bufferOffset = 0,
6.     bufferLength = readBuffer.length,
7.     filePosition = 100;
8.
9.   fs.read(fd, readBuffer, bufferOffset, bufferLength,
    filePosition,
10.    function(err, readBytes) {
11.      if (err) { throw err; }
12.      console.log('just read ' + readBytes + ' bytes');
13.      if (readBytes > 0) {
14.        console.log(readBuffer.slice(0, readBytes));
15.      }
16.    });
17. });
```

Here we are opening the file, and when it's opened we are asking to read a chunk of 1024 bytes from it, starting at position 100 (line 9).

The last argument to the **fs.read** call is a callback function (line 10) which will be invoked when one of the following 3 happens:

- there is an error,
- something has been read or
- nothing could be read.

On the first argument, this callback gets an error if there was an one, or null.

On the second argument (**readBytes**) it gets the number of bytes read into the buffer. If the read bytes is zero, the file has reached the end.

Write into a file

To write into a file descriptor you can use **fs.write** like this:

```
1. var fs = require('fs');
2.
3. fs.open('/var/log/system.log', 'a', function(err, fd) {
4.   var writeBuffer = new Buffer('writing this string'),
5.     bufferOffset = 0,
6.     bufferLength = writeBuffer.length,
7.     filePosition = null;
8.
9.   fs.write(
10.    fd,
11.    writeBuffer,
12.    bufferOffset,
13.    bufferLength,
14.    filePosition,
15.    function(err, written) {
16.      if (err) { throw err; }
17.      console.log('wrote ' + written + ' bytes');
18.    }
19.  );
20. });
```

Here we are opening the file in append-mode ('a') on line 3, and then we are writing into it (line 8), passing in a buffer with the data we want written, an offset inside the buffer where we want to start writing from, the length of what we want to write, the file position and a callback. In this case we are passing in a file position of null, which is to say that he writes at the current file position. Here we are also opening in append-mode, so the file cursor is positioned at the end of the file.



Close Your files

On all these examples we did not close the files. This is because these are small simple examples destined to be run and returned. All open files will be closed once the process exists.

In real applications you should keep track of those file descriptors and eventually close them using `fs.close(fd[, callback])` when no longer needed.



Advanced Tip: careful when appending concurrently

If you are using these low-level file-system functions to append into a file, and concurrent writes will be happening, opening it in append-mode will not be enough to ensure there will be no overlap. Instead, you should keep track of the last written position before you write, doing something like this:

```
1. // Appender
2. var fs = require('fs');
3. var startAppender = function(fd, startPos) {
4.   var pos = startPos;
5.   return {
6.     append: function(buffer, callback) {
7.       var oldPos = pos;
8.       pos += buffer.length;
9.       fs.write(fd, buffer, 0, buffer.length, oldPos,
10.        callback);
11.     };
12.   };
13.
14.
```

Here we declare a function stored on a variable named "startAppender". This function starts the appender state (position and file descriptor) and then returns an object with an append function.

Now let's do a script that uses this Appender:

```
1. // start appender
2. fs.open('/tmp/test.txt', 'w', function(err, fd) {
3.   if (err) {throw err; }
4.   var appender = startAppender(fd, 0);
5.   appender.append(new Buffer('append this!'),
6.     function(err) {
7.       console.log('appended');
8.     });
9. });
```

And here we are using the appender to safely append into a file.

This function can then be invoked to append, and this appender will keep track of the last position (line 4 on the Appender), and increments it according to the buffer length that was passed in.

Actually, there is a problem with this code: `fs.write()` may not write all the data we asked it to, so we need to do something a little bit smarter here:

Source code in
chapters/fs/appender.js

```
1. // Appender
2. var fs = require('fs');
3. var startAppender = function(fd, startPos) {
4.     var pos = startPos;
5.     return {
6.         append: function(buffer, callback) {
7.             var written = 0;
8.             var oldPos = pos;
9.             pos += buffer.length;
10.            (function tryWriting() {
11.                if (written < buffer.length) {
12.                    fs.write(fd, buffer, written, buffer.length
13.                        - written, oldPos + written,
14.                        function(err, bytesWritten) {
15.                            if (err) { callback(err); return; }
16.                            written += bytesWritten;
17.                            tryWriting();
18.                        });
19.                } else {
20.                    // we have finished
21.                    callback(null);
22.                }
23.            })();
24.        }
25.    };
26. };
```

Here we use a function named "tryWriting" that will try to write, call `fs.write`, calculate how many bytes have already been written and call itself if needed. When it detects it has finished (`written == buffer.length`) it calls `callback` to notify the caller, ending the loop.

Don't be frustrated if you don't grasp this on the first time around. Give it some time to sink in and come back here after you have finished reading this book.

Also, the appending client is opening the file with mode "w", which truncates the file, and it's telling appender to start appending on position 0. This will overwrite the file if it has content. So, a wiser version of the appender client would be:

```
1. // start appender
2. fs.open('/tmp/test.txt', 'a', function(err, fd) {
3.   if (err) {throw err; }
4.   fs.fstat(fd, function(err, stats) {
5.     if (err) {throw err; }
6.     console.log(stats);
7.     var appender = startAppender(fd, stats.size);
8.     appender.append(new Buffer('append this!'),
9.       function(err) {
10.        console.log('appended');
11.      });
12.    });
13.  });
```

File-system Exercises

You can check out the solutions at the end of this book.

Exercise 1 - get the size of a file

Having a file named a.txt, print the size of that files in bytes.

Exercise 2 - read a chunk from a file

Having a file named a.txt, print bytes 10 to 14.

Exercise 3 - read two chunks from a file

Having a file named a.txt, print bytes 5 to 9, and when done, read bytes 10 to 14.

Exercise 4 - Overwrite a file

Having a file named a.txt, Overwrite it with the UTF8-encoded string "ABCDEFGHijklmnopQRSTUVWXYZ0123456789abcdefghijklmnopqrstuvwxyz".

Exercise 5 - append to a file

Having a file named a.txt, append utf8-encoded string "abc" to file a.txt.

Exercise 6 - change the content of a file

Having a file named a.txt, change byte at pos 10 to the UTF8 value of "7".

HTTP

HTTP Server

You can easily create an HTTP server in Node. Here is the famous http server "Hello World" example:

Source in file:

http/http_server_1.js

```
1. var http = require('http');
2.
3. var server = http.createServer();
4. server.on('request', function(req, res) {
5.   res.writeHead(200, {'Content-Type': 'text/plain'});
6.   res.write('Hello World!');
7.   res.end();
8. });
9. server.listen(4000);
10.
```

On line 1 we get the 'http' module, to which we call `createServer()` (line 3) to create an HTTP server.

We then listen for 'request' type events, passing in a callback function that gets two arguments: the request object and the response object. We can then use the response object to write back to the client.

On line 5 we write a header (Content-Type: text/plain) and the HTTP status 200 (OK).

On line 6 we reply the string "Hello World!" and on line 7 we terminate the request.

On line 9 we bind the server to the port 4000.

So, if you run this script on node you can then point your browser to `http://localhost:4000` and you should see the "Hello World!" string on it.

This example can be shortened to:

Source in file:

http/http_server_2.js

```
1. require('http').createServer(function(req, res) {
2.   res.writeHead(200, {'Content-Type': 'text/plain'});
3.   res.end('Hello World!');
4. }).listen(4000);
```

Here we are giving up the intermediary variables for storing the http module (since we only need to call it once) and the server (since we only need to make it listen on port 4000). Also, as a shortcut, the `http.createServer` function accepts a callback function that will be invoked on every request.

There is one last shortcut here: the `response.end` function can accept a string or buffer which it will

send to the client before ending the request.

The `http.ServerRequest` object

When listening for "request" events, the callback gets one of these objects as the first argument. This object contains:

`req.url`

The URL of the request, as a string. It does not contain the schema, hostname or port, but it contains everything after that. You can try this to analyze the url:

Source in file:

http/http_server_3.js

```
1. require('http').createServer(function(req, res) {
2.   res.writeHead(200, {'Content-Type': 'text/plain'});
3.   res.end(req.url);
4. }).listen(4000);
```

and connect to port 4000 using a browser. Change the URL to see how it behaves.

`req.method`

This contains the HTTP method used on the request. It can be, for example, 'GET', 'POST', 'DELETE' or any other one.

`req.headers`

This contains an object with a property for every HTTP header on the request. To analyze it you can run this server:


Source in file:

http/http_server_4.js

```
1. var util = require('util');
2.
3. require('http').createServer(function(req, res) {
4.   res.writeHead(200, {'Content-Type': 'text/plain'});
5.   res.end(util.inspect(req.headers));
6. }).listen(4000);
```

and connect your browser to port 4000 to inspect the headers of your request.

Here we are using `util.inspect()`, an utility function that can be used to analyze the properties of any object.

 **req.headers** properties are on lower-case. For instance, if the browser sent a "Cache-Control: max-age: 0" header, **req.headers** will have a property named "cache-control" with the value "max-age: 0" (this last one is untouched).

The http.ServerResponse object

The response object (the second argument for the "request" event callback function) is used to reply to the client. With it you can:

Write a header

You can use **res.writeHead(status, headers)**, where headers is an object that contains a property for every header you want to send.

An example:

Source in file:
http/http_server_5.js

```
1. var util = require('util');
2.
3. require('http').createServer(function(req, res) {
4.   res.writeHead(200, {
5.     'Content-Type': 'text/plain',
6.     'Cache-Control': 'max-age=3600'
7.   });
8.   res.end('Hello World!');
9. }).listen(4000);
```

On this example we set 2 headers: one with "Content-Type: text/plain" and another with "Cache-Control: max-age=3600".

If you save the above source code into http_server_5.js and run it with:

```
$ node http_server_5.js
```

You can query it by using your browser or using a command-line HTTP client like curl:

```
1. $ curl -i http://localhost:4000
2. HTTP/1.1 200 OK
3. Content-Type: text/plain
4. Cache-Control: max-age=3600
5. Connection: keep-alive
6. Transfer-Encoding: chunked
7.
8. Hello World!
```

Change or set a header

You can change a header you already set or set a new one by using

```
res.setHeader(name, value);
```

This will only work if you haven't already sent a piece of the body by using `res.write()`.

Remove a header

You can remove a header you have already set by calling:

```
res.removeHeader(name, value);
```

Again, this will only work if you haven't already sent a piece of the body by using `res.write()`.

Write a piece of the response body

You can write a string:

```
res.write('Hello');
```

or a an existing buffer:

```
1. var buf = new Buffer('Hello World');
2. buf[0] = 45;
3. res.write(buffer);
```

This method can, as expected, be used to reply dynamically generated strings or binary file. Replying with binary data will be covered later.

HTTP Client

You can issue http requests using the "http" module. Node is specifically designed to be a server, but it can itself call other external services and act as a "glue" service. Or you can simply use it to run a simple http client script like this one:

http.get()

Source in file:
http/http_client_1.js

This example uses `http.get` to make an HTTP GET request to the url `http://www.google.com:80/index.html`.

You can try it by saving it to a file named `http_client_1.js` and running:

```
1. $ node http_client_1.js
2.
3. got response: 302
4.
```

http.request()

Using `http.request` you can make any type of HTTP request:

```
http.request(options, callback);
```

The options are:

- **host**: A domain name or IP address of the server to issue the request to.
- **port**: Port of remote server.
- **method**: A string specifying the HTTP request method. Possible values: 'GET' (default), 'POST', 'PUT', and 'DELETE'.
- **path**: Request path. Should include query string and fragments if any. E.G. `'/index.html?page=12'`
- **headers**: An object containing request headers.

The following method makes it easy to send body values (like when you are uploading a file or posting a form):

```
Source in file:
http/http_client_2.js
```

```
1. var options = {
2.   host: 'www.google.com',
3.   port: 80,
4.   path: '/upload',
5.   method: 'POST'
6. };
7.
8. var req = require('http').request(options, function(res) {
9.   console.log('STATUS: ' + res.statusCode);
10.  console.log('HEADERS: ' + JSON.stringify(res.headers));
11.  res.setEncoding('utf8');
12.  res.on('data', function (chunk) {
13.    console.log('BODY: ' + chunk);
14.  });
15. });
16.
17. // write data to request body
18. req.write("data\n");
19. req.write("data\n");
20. req.end();
```

On lines 18 and 19 we are writing the HTTP request body data (two lines with the "data" string) and on line 20 we are ending the request. Only then the server replies and the response callback gets activated (line 8).

Then we wait for the response. When it comes, we get a "response" event, which we are listening to on the callback function that starts on line 8. By then we only have the HTTP status and headers ready, which we print (lines 9 and 10).

Then we bind to "data" events (line 12). These happen when we get a chunk of the response body data (line 12).

This mechanism can be used to stream data from a server. As long as the server keeps sending body chunks, we keep receiving them.

HTTP Exercises

You can checkout the solutions at the end of this book.

Exercise 1

Make an HTTP server that serves files. The file path is provided in the URL like this:
`http://localhost:4000/path/to/my/file.txt`

Exercise 2

Make an HTTP server that outputs plain text with 100 new-line separated unix timestamps every second.

Exercise 3

Make an HTTP server that saves the request body into a file.

Exercise 4

Make a script that accepts a file name as first command line argument and uploads this file into the server built on the previous exercise.