HANDS-ON

NODE.JS

The Node.js introduction and API reference

by Pedro Teixeira

Hands-on Node.js

Pedro Teixeira

This book is for sale at http://leanpub.com/hands-on-nodejs

This version was published on 2013-12-02

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Pedro Teixeira by spreading the word about this book on Twitter!

The suggested hashtag for this book is #hands-on-nodejs.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#hands-on-nodejs

Contents

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 combined Google's V8 JavaScript engine and an event loop to create a server-side platform programmable in JavaScript. The project 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'etre*, and the approach focussed on translating common practices from other platforms like Ruby, PERL and Python, into JavaScript. Node took a leap from this and said: "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 Server-side JavaScript attempts, unifying the client and server into a common language was not the primary goal for Node.

In my perspective 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 in any other existing platform.
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[1] 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[2]

or you can download it directly:

https://github.com/pgte/handson_nodejs_source_code/zipball/master[3]

[1] http://nodejs.org

[2] https://github.com/pgte/handson_nodejs_source_code

[3] 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!

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 for the database operation to complete.

Programming in this way does not allow you to parallelize I/O (such as performing another database query or communicating with a remote web service). 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 and modifying shared state; you never know when the thread you are running is going to be taken out of execution, which can lead to bugs that are strange and difficult to reproduce.

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 parallel "thread" of execution. This can relax the synchronization requirements but can become complex and error-prone, since the thread scheduling is left in the hands of the programmer.

Solution 2: Use event-driven I/O

Event-driven I/O is a scheme where you register callbacks to be invoked when an interesting I/O event happens.

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 in the previous example, you could change it to:

```
callback = function(post) {
  doSomethingWithPost(post); // this will only execute when the db.query function\
 returns.
};
db.query('SELECT * FROM posts where id = 1', callback);
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 database operation is complete, then passing this function as a callback argument to the db query operation. The db operation becomes responsible for executing the callback when the result is available.

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

```
db.query('SELECT * FROM posts where id = 1',
  function(post) {
    doSomethingWithPost(post); // this will only execute when the db.query functi\
on returns.
  }
);
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 some others.

Tip: For more info about event-driven server implementations, see http://en.wikipedia.org/wiki/Reactor_-pattern[4].

Implementing an application using one of these event-driven frameworks requires framework-specific knowledge and framework-specific libraries. For example: when using Event Machine, you must avoid using all the synchronous libraries available in Ruby-land (i.e. most libraries). To gain the benefit of not blocking, you are limited to using libraries that are specific for Event Machine. If you use blocking libraries, your program will not be able to scale optimally because the event loop is constantly being blocked, which may delay the processing of I/O events and makes your application slow, defeating the original purpose of using event-driven I/O.

Node has been devised as a non-blocking I/O server platform from day one, which means that you should expect everything built on top of it to be non-blocking (with some specific and very explicit exceptions). 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 like C, Ruby or Python), Node has a clean slate to build upon.

Why JavaScript?

Ryan Dahl began this pet project of his by building a platform that was programmable in C, but he soon realized that maintaining the context between callbacks was too complicated and could lead to poorly structured code. He then turned to Lua.

Lua already has several blocking I/O libraries and the 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. Let's see a little bit of this goodness in action:

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    clickCount ++;
4    alert('Clicked ' + clickCount + ' times.');
5  };
```

or, using jQuery:

[4] http://en.wikipedia.org/wiki/Reactor_pattern

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

In both examples we assign or pass a function as an argument. This function will then be executed later once the relevant event (button clicking in this case) happens. The click handling function has access to 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 *clickCount* 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.

Here you don't have to worry about synchronization: your callback function will not be interrupted until it returns - you have that guarantee.

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 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 planet Earth – and growing.

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:

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

Here we declare a function, but it's not of much use, because we 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:

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

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:

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

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

```
myFunction();
```

and also within inner scopes:

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

(function() {
  myFunction();
})();
```

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

```
var myFunc = function() {
  console.log('hello');
}
```

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:

```
myFunc();
myFunc2();
```

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

```
var myFunc2 = function myFunc() {
  console.log('hello');
}
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:

```
var myFunc = function() {
  console.log('hello');
}

console.log(myFunc);
```

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

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

Functions are first-class objects

In fact, there are no second-class objects in JavaScript. JavaScript is the ultimate object-oriented language, where (almost) 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.

Example:

```
var scheduler = function(timeout, callbackfunction) {
  return function() {
    setTimeout(callbackfunction, timeout)
  }
};

(function() {
  var timeout = 1000; // 1 second
  var count = 0;
  var schedule = scheduler(timeout, function doStuff() {
    console.log(++ count);
    schedule();
  });
  schedule()
})();

// "timeout" and "count" variables
// do not exist in this scope.
```

In this little example we create a function and store it in a variable called "scheduler" (starting on line 1). This function returns a function that sets up a timer that will execute a given function within a certain number of miliseconds (line 3). This timeout will schedule a callback function to be called after a time delay of 1 second, as specified by the *timeout* variable.

In line 9 we declare a function that will immediately be executed in line 15. This is a normal way to create new scopes in JavaScript. Inside this scope we create 2 variables: "timeout" (line 8) and "count" (line 9). Note that these variables will not be accessible to the outer scope.

Then, on line 10, we invoke the *scheduler* function, passing in the timeout value as first argument and a function called *doStuff* as second argument. This returns a function that we store in the local *schedule* variable, which we later invoke (line 14), provoking the *setTimeout* to be called. 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 form 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, you should learn it and embrace its good parts.

JSHint

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 me is JSHint. JSHint analyzes your JavaScript file and outputs a series of errors and warnings, including some known misuses of JavaScript, such as 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.

JHLint can be installed using

```
1 $ npm install -g jshint
```

If you don't have Node installed see section about NPM[5].

and can be run from the command line like this:

[5] index.html#npm

```
$ jshint myfile.js
```

You can also define what rules this tool should obey by defining and customizing a .jshintrc inside your home directory. For more information on JSHint plrease refer to the official documentation at http://www.jshint.com/docs/[6].

JavaScript versions

JavaScript is a standard with its 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/wi 5-Mozilla-Features-Implemented-in-V8[7]

References

Event Machine: http://rubyeventmachine.com/[8]

Twisted: http://twistedmatrix.com/trac/[9]

AnyEvent: http://software.schmorp.de/pkg/AnyEvent.html[10]

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

JSHint http://www.jshint.com/[12]

[6] http://www.jshint.com/docs/

[7] https://github.com/joyent/node/wiki/ECMA-5-Mozilla-Features-Implemented-in-V8

[8] http://rubyeventmachine.com/

[9] http://twistedmatrix.com/trac/

[10] http://software.schmorp.de/pkg/AnyEvent.html

[11] http://www.amazon.com/exec/obidos/ASIN/0596517742/wrrrldwideweb

[12] http://www.jshint.com/

Starting up

Install Node

If you don't have the latest version of Node.js installed in your local machine you should do that now: Head out to http://nodejs.org[13] and click on the "Install" button.

Depending on your platform, you will get a package installer downloaded that you should execute to install Node. This works if you have a MacOS or a Windows machine. If you're on a Linux box, your distribution probably supports the latest stable version of Node.js, but you can always download and build from the source code.

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

```
$ node -v
v0.10.22
```

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

To launch the CLI, just type, in the command line:

```
$ 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.

Let's then create our first Hello World HTTP server in Node.js inside a file named `hello_world.js`:

hello_world.js:

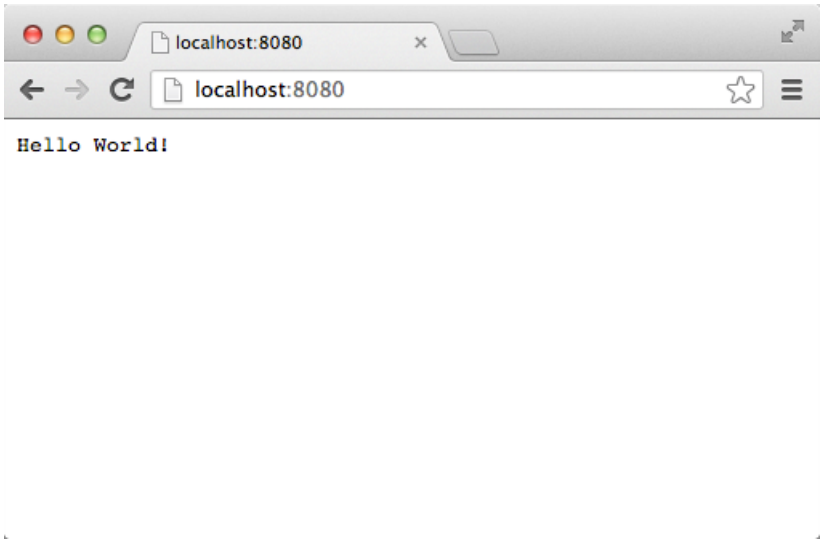
[13] http://nodejs.org

```
var http = require('http');
var server = http.createServer();
server.on('request', function(req, res) {
  res.end('Hello World!');
});
server.listen(8080);
```

You can launch Node on a file like this:

```
$ node hello_world.js
```

Now, open a web browser and point it to http://localhost:8080[14]:



Hello World on the browser

[14] http://localhost:8080

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 to be processed. 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 of 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

```
setTimeout(function() {
  console.log('World!');
}, 2000);
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 after 2000 milliseconds have passed. 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 and exit. This little script will just keep printing "Hello World".

Don't block!

Node's 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 returns. 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 queue. 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 (more about that later). 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. You'll just be increasing the response latency, decreasing the responsiveness of your service or application.

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. Do all your require'ing during the initialization phase of Node and not inside any callback (later addressed).

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 CPU intensive, time consuming operations in memory. There are several techniques that can be used to work around that, which I'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 led 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!

Modules and NPM

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:

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

This will fetch a module that was installed by NPM (more about NPM later). If you want to author modules (as you should when building an application), you can also use the relative notation like this:

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

This will fetch the module from a path relative to the current file we are executing. We will cover creating modules in 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 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.

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. NPM is also installed when you install Node.

Global vs. Local

NPM has two fundamentally different ways of working: local and global.

In the global mode, all packages are installed inside a shared folder, and you can keep only one version of each package.

In local mode you can have different installed packages per directory: In this mode NPM keeps a local directory named “node_modules” where it keeps the local modules installed. By switching directories you are inside different local contexts that can have different modules installed.

NPM works in local mode by default. To enable global mode you must explicitly use the “-g” switch to any of the following commands.

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

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

All the above commands install packages into the local directory.To install a pac\
kage globally, use the "-g " switch like this:

$ npm install -g express
```

[

package_name[@version] ...]]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
```

If you wish to remove a package that was installed globally you need to explicitly use the “-g” switch:

```
$ npm rm -g express
```

[

[.]...]]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.

The Package.json Manifest

Each application or module can (and should) have a manifest file named `package.json` at it's root. This files tells, amongst other things, the name of the module, the current version of it and what it's dependencies are.

If a module or application depends on other modules it should have a `package.json` file at it's root stating which modules it depends on.

For instance, if my application is named "myapp" and depends on third-party modules `request` and `async` it should contain a package.json file like this:

```
{
  "name": "myapp",
  "version": 1.0.0,
  "dependencies": {
    "request": "*",
    "async": "*"
  }
}
```

This manifest states that this application depends on *any* version of the modules `request` and `async`, denoted by the `*` as the version specification. Should the application rely on specific versions of any of these modules, it should specify the version like this:

```
{
  "name": "myapp",
  "version": 1.0.0,
  "dependencies": {
    "request": "2.27.0",
    "async": "0.2.9"
  }
}
```

Here we're tying the application to these two specific versions, but we can set the version specification to be more loose:

```
{
  "name": "myapp",
  "version": 1.0.0,
  "dependencies": {
    "request": "2.27.x",
    "async": "0.2.x"
  }
}
```

In this case we're saying that the application depends on request version 2 (major) . 27 (minor) and that any patch version is acceptable. The request version 2.27.3, should it exist, satisfies this requirement, as does version 2.27.0.

It's common practise to depend on a specific major and minor version and not specify the patch version, since patch versions should thoeretically only solve issues.

If you're going to deploy your application into several machines, consider specifying the exact version: pin down the major, minor and patch versions of each module. This prevents having two distinct Node processes running different versions of modules that may cause different behaviour, preventing bugs from being easily reproduced and traced.

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:

```
var a = {1: true, 2: false};
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:

```
var a = {1: true, 2: false};
console.log('This is a number: %d, and this is a string: %s, and this is an objec\
t outputted as JSON: %j', 42, 'Hello', a);
```

Which outputs:

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

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

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

and to print a stack trace:

```
console.trace();

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

util

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

```
var util = require('util');
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:

```
var util = require('util');
var a = {1: true, 2: false};
console.log(util.inspect(a));
// => { '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, which will be covered in a more appropriate chapter.

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 UTF-8 string like this:

```
1  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:

```
1  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:

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

and you can manipulate it:

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

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

```
1  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:

```
var buffer = new Buffer('this is the string in my buffer');
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:

```
var buffer = new Buffer('this is the string in my buffer');
var slice = new Buffer(10);
var targetStart = 0,
    sourceStart = 10,
    sourceEnd = 20;
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 the “on” method, which is simply an alias for “addListener”:

```
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:

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

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:

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

Here we are using *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 in 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:

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

// Here is the MyClass constructor:
var MyClass = function(option1, option2) {
  this.option1 = option1;
  this.option2 = option2;
}

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:

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

Here we are emiting an event named “custom event”, sending also some data (“some arguments” in this case).

Now clients of MyClass instances can listen to “custom event” events like this:

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

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:

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

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 whose 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's *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:

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

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 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

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

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

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

setImmediate

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

```
setImmediate(function() {
  // this runs on the next event loop
  console.log('yay!');
});
```

As we saw, this method is prefered 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 an increased latency in our application or service. 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 setImmediate() like this:

```
setImmediate(function() {
  // do something
});
```

You can use this to delay processing that does not have to run immediately, until 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:

```
stream.on('data', function(data) {
  stream.end('my response');
  setImmediate(function() {
    fs.unlink('path/to/file');
  });
});
```

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 regardless of whether the function has finished its duty or not.

Supposing there is an asynchronous function called "async" that performs some I/O and gets a callback to be invoked when finished. We want to call it every second, so:

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

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

```
var interval = 1000; // 1 second
(function schedule() {
  setTimeout(function() {
    async(function() {
      console.log('async is done!');
      schedule();
    });
  }, interval)
})();
```

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:

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

fs.stat('/etc/passwd', function(err, stats) {
  if (err)  {console.log(err.message); return; }
  console.log(stats);
  //console.log('this file is ' + stats.size + ' bytes long.');
});
```

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

```
{ dev: 234881026,
  ino: 24606,
  mode: 33188,
  nlink: 1,
  uid: 0,
  gid: 0,
  rdev: 0,
  size: 3667,
  blksize: 4096,
  blocks: 0,
  atime: Thu, 17 Mar 2011 09:14:12 GMT,
  mtime: Tue, 23 Jun 2009 06:19:47 GMT,
  ctime: Fri, 14 Aug 2009 20:48:15 GMT
}
```

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

```
stats.isFile()
stats.isDirectory()
stats.isBlockDevice()
stats.isCharacterDevice()
stats.isSymbolicLink()
stats.isFIFO()
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:

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

The first argument to *fs.open* is the file path. The second argument contains the flags, indicating the mode in which the file is to be opened. The flags can be ‘r’, ‘r+’, ‘w’, ‘w+’, ‘a’, or ‘a+’.

Here follow the semantics for 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 exits.

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

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 UTF-8-encoded string "ABCDEFGHIJLKLMNOPQRSTU-VXYZ0123456789abcdefghijklmnopqrstuvxyz".

Exercise 5 - append to a file

Having a file named a.txt, append UTF-8-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 UTF-8 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);
```

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

We then listen for 'request' type events, passing in a callback function that takes 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 (ContentType: text/plain) and the HTTP status 200 (OK).
On line 6 we reply with 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

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

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

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

require('http').createServer(function(req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end(util.inspect(req.headers));
}).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 names are 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

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

require('http').createServer(function(req, res) {
  res.writeHead(200, {
    'Content-Type': 'text/plain',
    'Cache-Control': 'max-age=3600'
  });
  res.end('Hello World!');
}).listen(4000);
```

In 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:

```
$ curl -i http://localhost:4000
HTTP/1.1 200 OK
Content-Type: text/plain
Cache-Control: max-age=3600
Connection: keep-alive
Transfer-Encoding: chunked

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() or res.end().

Write a piece of the response body You can write a string:

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

or an existing buffer:

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

This method can, as expected, be used to reply with dynamically generated strings or a 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

```
1  var http = require('http');
2
3  var options = {
4    host: 'www.google.com',
5    port: 80,
6    path: '/index.html'
7  };
8
9  http.get(options, function(res) {
10   console.log('got response: ' + res.statusCode);
11 }).on('error', function(err) {
12   console.log('got error: ' + err.message)
13 });
```

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  got response: 302
```

http.request()

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

```
1  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

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.

Streams

Node has a useful abstraction: Streams. More specifically, two very useful abstractions: Read Streams and Write Streams. They are implemented throughout several Node objects, and they represent inbound (ReadStream) or outbound (WriteStream) flow of data. We have already come across some of them, but here we will try to introduce them in a more formal way.

ReadStream

A ReadStream is like a faucet of data. After you have created one (and the method of creating them depends on the type of stream), you can:

Wait for data

By binding to the "data" event you can be notified every time there is a chunk being delivered by that stream. It can be delivered as a buffer or as a string.

If you use stream.setEncoding(encoding), the "data" events pass in strings. If you don't set an encoding, the "data" events pass in buffers. So here is an example:

```
var readStream = ...
readStream.on('data', function(data) {
  // data is a buffer;
});

var readStream = ...
readStream.setEncoding('utf8');
readStream.on('data', function(data) {
  // data is a UTF-8-encoded string;
});
```

So here data passed in on the first example is a buffer, and the one passed on the second is a string because we are informing the stream about the encoding we are expecting.

The size of each chunk may vary; it may depend on buffer size or on the amount of available data.

Know when it ends

A stream can end, and you can know when that happens by binding to the "end" event like this: