HANDS-ON

NODE.JS

Hands-on Node.js

Pedro Teixeira

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

This version was published on 2012-11-07

This is a Leanpub book. Leanpub helps authors to self-publish in-progress ebooks. We call this idea Lean Publishing.

To learn more about Lean Publishing, go to http://leanpub.com/manifesto.

To learn more about Leanpub, go to http://leanpub.com.

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/#hands-on-nodejs

Contents

API quick tour 26

Utilities 28

Buffers 31

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'etre*, 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 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 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[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!

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 com\
3  pletes
4  doSomethingWithPost(post);
5  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 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 "thread". 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:

```
1  callback = function(post) {
2    doSomethingWithPost(post); // this will only execute when the db.query fu\
3  nction returns.
4  };
5  db.query('SELECT * FROM posts where id = 1', callback);
6  doSomethingElse(); // this will execute independent of the returned status \
7  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 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:

```
1  db.query('SELECT * FROM posts where id = 1',
2    function(post) {
3      doSomethingWithPost(post); // this will only execute when the db.query \
4  function returns.
5    }
6  );
7  doSomethingElse(); // this will execute independent of the returned status \
8  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 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 should 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 blocking, which may delay the processing of I/O events.

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 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), 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 lead to poorly structured code, so 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;document.getElementById('mybutton').onclick = function()\
2   {
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, which may be executed later. 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 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 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:

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

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:

```
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 in which it's declared

```
1  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  }
```

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

We can assign that function to another variable:

```
1  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 in 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 "callbackfunction" argument. 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 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 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

```
1  $ jshint myfile.js
```

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[6]

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 *myAsync-Function*, 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.

---

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

```
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/[7]

Twisted: http://twistedmatrix.com/trac/[8]

AnyEvent: http://software.schmorp.de/pkg/AnyEvent.html[9]

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

JSHint http://www.jshint.com/[11]

---

[7]http://rubyeventmachine.com/

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

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

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

[11]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[12] 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:

```
1  $ node -v
2  v0.8.12
```

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

To launch the CLI, just type

```
1  $ 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:

```
1  $ node myfile.js
```

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

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

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

---

[12]http://nodejs.org

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

You can then execute the file directly:

```
1   $ ./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. 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:

```
1   $ npm ls installed
```

List all stable packages:

```
1   $ npm ls stable
```

You can also combine filters:

```
1   $ npm ls installed stable
```

You can also use npm ls to search by name:

```
1   $ 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:

```
1   $ 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:

```
1   $ npm install package_name
```

Example:

```
1   $ npm install express
```

To install a specific version of a package do:

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

Example:

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

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

```
1   $ npm install express@">=0.1.0
2
3   You can also combine many filters to select a specific version, combining v\
4   ersion range and / or tags like this:
5
6   $ npm install sax@">=0.1.0
7
8   All the above commands install packages into the local directory.To install\
9    a package globally, use the "-g " switch like this:
10
11  $ 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:

```
1   $ npm rm sax
```

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

```
1   $ 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:

```
1   $ npm view connect
```

View information about a specific version:

```
1   $ 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 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

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

Run it using the node command line tool:

```
1  $ 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 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. 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!

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:

```
1    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 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.

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[13].

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 of the following modules:

fs

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

---

[13]http://nodejs.org.

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:

```
1  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\
3   object outputted as JSON: %j', 42, 'Hello', a);
```

Which outputs:

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

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

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

and to print a stack trace:

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

util

Node has an *util* module 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:

```
1   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:

```
1  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:

```
1   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 *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

```
1  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:

```
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  }
```

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:

```
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);
```

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:

```
1  var timeoutHandle = setTimeout(function() { console.log('yehaa!'); }, 1000)\
2  ;
3  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.

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 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 process.nextTick() like this:

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

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:

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

```
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');fs.open('/path/to/file', 'r', function(err, fd) {   /\
2  / 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.

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, callback);
10       }
11     };
12   };
```

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 write 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!'), function(err) {
 6       console.log('appended');
 7     });
 8   });
```

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 increment 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;
10        (function tryWriting() {
11          if (written < buffer.length) {
12            fs.write(fd, buffer, written, buffer.length - written, oldPos + w\
13  ritten,
14              function(err, bytesWritten) {
15                if (err) { callback(err); return; }
16                written += bytesWritten;
17                tryWriting();
18              }
19            );
20          } else {
21            // we have finished
22            callback(null);
23          }
24        })();
25      }
26    }
27  };
```

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 can'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!'), function(err) {
 9        console.log('appended');
10      });
11    })
12  });
```

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".

chapter_12

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

```
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 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

```
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);
```

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:

```
1  $ 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

```
1  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:

```
1  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:

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

or 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 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:

```
1  var readStream = ...
2  readStream.on('data', function(data) {
3    // data is a buffer;
4  });
5
6  var readStream = ...
7  readStream.setEncoding('utf8');
8  readStream.on('data', function(data) {
9    // data is a UTF-8-encoded string;
10 });
```

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:

```
1  var readStream = ...
2  readStream.on('end', function() {
3    console.log('the stream has ended');
4  });
```

Pause it

A read stream is like a faucet, and you can keep the data from coming in by pausing it like this:

```
1  readStream.pause();
```

Depending on the stream type, pausing a stream may not end the "data" events. For instance, a TCP connection may still emit some data events after you pause it (since some packets may still be in transit).

Resume it

If it's paused, the faucet can be reopened and the stream can start flowing again:

```
1  readStream.resume();
```

WriteStream

A WriteStream is an abstraction on somewhere you can send data to. It can be a file or a network connection or even an object that outputs data that was transformed - like when zipping a file. With a WriteStream you can:

Write

You can write a buffer or a string by calling write:

```
1  var writeStream = ...;
2  writeStream.write('this is an UTF-8 string');
```

Here Node assumes we are passing an UTF-8-encoded string.

Alternatively you can specify another encoding like this:

```
1  var writeStream = ...;
2  writeStream.write('7e3e4acde5ad240a8ef5e731e644fbd1', 'base64');
```

or you can simply write a buffer:

```
1  var writeStream = ...;
2  var buffer = new Buffer('this is a buffer with some string');
3  writeStream.write(buffer);
```

Wait for it to drain

Node does not block on I/O, so it does not block on read or write commands. On write commands, if Node is not able to flush the data into the kernel buffers, it will buffer that data for you, storing it in your process memory. Because of this, writeStream.write() returns a boolean. If write() manages to flush all data to the kernel buffer, it returns true. If not, it returns false.

When a writeStream manages to flush the data into the kernel buffers, it emits a "drain" event so you can listen to it like this:

```
1  var writeStream = ...;
2  writeStream.on('drain', function() {
3    console.log('drain emitted');
4  });
```

Later we will see how this draining notification combined with the pause and resume capabilities can come in handy when limiting the memory growth of your Node process.

Some stream examples

Here are some instances of Node streams.

Filesystem streams

You can create a read stream for a file path by doing something like:

```
1  var fs = require('fs');
2  var rs = fs.createReadStream('/path/to/file');
3  ...
```

Here you can pass a second argument to *fs.createReadStream* where you can specify the start and end position on your file, the encoding, the flags and the buffer size. Here are the defaults:

```
1  { flags: 'r',
2    encoding: null,
3    fd: null,
4    mode: 0666,
5    bufferSize: 64 * 1024
6  }
```

You can also create a write stream:

```
1  var fs = require('fs');
2  var rs = fs.createWriteStream('/path/to/file', options);
3  ...
```

Which also accepts a second argument with an options object. The *options* argument to *createWriteStream* has these default values:

```
1  { flags: 'w',  encoding: null,  mode: 0666 }
```

For instance, to create a file WriteStream that assumes UTF-8 encoding you can use:

```
1  var fs = require('fs');
2  var rs = fs.createWriteStream('/path/to/file', { encoding: 'utf8' });
```

Network streams

There are all kinds of streams on the networking API of Node.
For instance, a client TCP connection is a write and a read stream.
An http request object is a read stream. An http response object is a write stream. That is, each implements the *ReadStream* / *WriteStream* methods and events.

The Slow Client Problem and Back-pressure

As we said, Node does not block on writes, and it buffers the data for you if the write cannot be flushed into the kernel buffers. Imagine this scenario: you are pumping data into a write stream (like a TCP connection to a browser), and your source of data is a read stream (like a file ReadStream):

```
1  require('http').createServer(function(req, res) {
2    var rs = fs.createReadStream('/path/to/big/file');
3    rs.on('data', function(data) {
4      res.write(data);
5    });
6    rs.on('end', function() {
7      res.end();
8    });
9  });
```

If the file is local, the read stream should be fast. If the connection to the client is slow, the writeStream will be slow. So readStream "data" events will happen quickly, the data will be sent to the writeStream, but eventually Node will have to start buffering the data because the kernel buffers will be full.

What will happen then is that the */path/to/big/file* file will be buffered in memory for each request, and if you have many concurrent requests, Node memory consumption will inevitably increase, which may lead to other problems, like swapping, thrashing and memory exhaustion.

What can we do?

To address this problem you will have to make use of the pause and resume of the read stream, and pace it alongside your write stream so your memory does not fill up:

```
1  require('http').createServer(function(req, res) {
2    var rs = fs.createReadStream('/path/to/big/file');
3    rs.on('data', function(data) {
4      if (!res.write(data)) {
5        rs.pause();
6      }
7    });
8    res.on('drain', function() {
9      rs.resume();
10   });
11   rs.on('end', function() {
12     res.end();
13   });
14 });
```

On line 5 we are pausing the readStream if the write cannot flush it to the kernel, and we are resuming it (line 9) when the writeStream is drained.

Pipe

What was described here is a recurring pattern, and instead of this complicated chain of events, you can simply use *stream.pipe()*, which does exactly what we described:

```
1  var util = require('util');
2
3  require('http').createServer(function(req, res) {
4    var rs = fs.createReadStream('/path/to/big/file');
5    rs.pipe(res);
6  });
```

Much simpler, right? *readStream.pipe* accepts 1 argument containing the destination writable stream.

By default, *end()* is called on the destination when the read stream ends. You can prevent that behavior by passing in *end: false* on the second argument options object like this:

```
1  var fs = require('fs');
2  require('http').createServer(function(req, res) {
3    var rs = fs.createReadStream('/path/to/big/file');
4    rs.pipe(res, {end: false});
5    rs.on('end', function() {
6      res.end("And that's all folks!");
7    }).listen(4000);
8  });
```

TCP

Node has a first-class HTTP server implementation, but this server descends from the "bare-bones" TCP server. Being so, everything described here applies also to every class descending from the *net.Server*, like the *http.Server*.

You can create a TCP server using the "net" module like this:

```
1   require('net').createServer(function(socket) {
2
3     // new connection
4
5     socket.on('data', function(data) {
6       // got data
7     });
8
9     socket.on('end', function(data) {
10      // connection closed
11    });
12
13    socket.write('Some string');
14
15  }).listen(4001);
```

On line 1 we use the *createServer* method on the *net* package, which we bind to TCP port 4001 on line 15. We can pass in a function callback to *createServer* to be invoked every time there is a "connection" event.

On this socket object we can then listen to "data" events when we get a package of data and the "end" event when that connection is closed.

On a socket we can also:

Write a string or a buffer

We can pass in a string or a buffer to be sent through the socket. If a string is passed in, you can specify an encoding as a second argument like this:

```
1  flushed = socket.write('453d9ea499aa8247a54c951', 'base64');
```

If you don't specify the encoding, Node will assume it's UTF-8.

The socket object is an instance of *net.Socket*, which is a writeStream, so the *write* method returns a boolean, saying whether it flushed to the kernel or not.

You can also pass in a callback function to be invoked when the data is finally written out like this:

```
1  var flushed = connection.write('453d', 'base64', function() {
2    // flushed
3  });
```

or, assuming UTF-8:

```
1  var flushed = connection.write('I am UTF-8!', function() {
2    // flushed
3  });
```

end

You can end the connection by calling the *end* method. This sends the TCP FIN packet, notifying the other end that this end wants to close the connection.

But you can still get "data" events after you have issued this, simply because there still might be some data in transit, or the other end might be insisting on sending you some more data. Also, you can pass in some final data to be sent when invoking end:

```
1  socket.end('Bye bye!');
```

...and all the other methods

socket object is an instance of *net.Socket*, and it implements the writeStream and readStream interfaces, so all those methods are available, like pause() and resume(). Also you can bind to the "drain" events.

Idle sockets

You can also be notified when a socket has been idle for some time, i.e., there has been no data received. For that, you must define a timeout by calling *setTimeout()*:

```
1  var timeout = 60000; // 1 minute
2  socket.setTimeout(timeout);
3  socket.on('timeout', function() {
4    socket.write('idle timeout, disconnecting, bye!');
5    socket.end();
6  });
```

or, in a shorter form:

```
1  socket.setTimeout(60000, function() {
2    socket.end('idle timeout, disconnecting, bye!');
3  });
```

Keep-alive

In Node, a *net.Socket* can implement a keep-alive mechanism to prevent timeouts occurring on the network or on the peer. Node does that by sending an empty TCP packet with the ACK (Acknowledgement) flag turned on.

You can enable the keep-alive functionality by:

```
1  socket.keepAlive(true);
```

You can also specify the delay between the last packet received and the next keep-alive packet on the second argument to the keepAlive call like this:

```
1  socket.keepAlive(true, 10000); // 10 seconds
```

Delay or no delay

When sending off TCP packets, the kernel buffers data before sending it off, and uses Nagle's algorithm to determine when to send off the data. If you wish to turn this off and demand that the data gets sent immediately after write commands, use:

```
1  socket.setNoDelay(true);
```

server.close()

This method closes the server, preventing it from accepting new connections. This function is asynchronous, and the server will emit the "close" event when actually closed:

```
1  var server = ...
2  server.close();
3  server.on('close', function() {
4    console.log('server closed!');
5  });
```

Listening

As we saw, after the server is created, we can bind it to a specific TCP port like this:

```
1  var port = 4001;var host = '0.0.0.0';server.listen(port, host);
```

The second argument (host) is optional. If omitted, the server will accept connections directed to any IP address.

This method is asynchronous. To be notified when the server is really bound you have to pass a callback like this:

```
1  server.listen(port, host, function() {
2    console.log('server listening on port ' + port);
3  });
```

or without a host:

```
1  server.listen(port, function() {
2    console.log('server listening on port ' + port);
3  });
```

TCP client

You can connect to a TCP server using the "net" module like this:

```
1  var net = require('net');var port = 4001;var conn = net.createConnection(po\
2  rt);
```

Here we omitted the second argument for the *createConnection* function, which is the host name. If you omit it, it defaults to localhost.

Now with a host name:

```
1  var net = require('net');
2  var port = 80;
3  var host = 'www.google.com';
4  var conn = net.createConnection(port, host);
```

Then you can listen for data:

```
1  conn.on('data', function(data) {
2    console.log('some data has arrived')
3  });
```

or send some data:

```
1  conn.write('some string over to you!');
```

or close it:

```
1  conn.close();
```

and also listen to the "close" event:

```
1  conn.on('close', function() {
2    console.log('connection closed');
3  });
```

Socket conforms to the ReadStream and WriteStream interfaces, so you can use all of the previously described methods on it.

Error handling

When handling a socket on the client or the server you can (and should) handle the errors by listening to the "error" event like this:

```
1  require('net').createServer(function(socket) {
2    socket.on('error', function(error) {
3      // do something
4    });
5  });
```

If you don't choose to catch an error, Node will handle an uncaught exception and terminate the current process. Unless you want that, you should handle the errors.

Also, you can choose to catch uncaught exceptions - preventing your Node process to go down - by doing something like:

```
1  process.on('uncaughtException', function() {
2    // Do something with it
3  });
```

TCP Exercises

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

Exercise 1

Make a chat server that requires no authentication, just a TCP client connection. Each time the client sends some text, the server broadcasts it to the other clients.

Exercise 2

Make a chat client that accepts 2 command line arguments: host and port, and reads from *stdin*, sending data to the server on each new line.

UNIX sockets

Server

The *net.Server* class not only supports TCP sockets, it also supports UNIX domain sockets. Unix domain sockets are sockets that are bound to the same host operating system. To create a UNIX socket server you have to create a normal *net.Server*, as you would on a TCP server, but then make it listen to a file path instead of a port like this:

```
1  var server = net.createServer(function(socket) {
2    // got a client connection here...
3  });
4  server.listen('/path/to/socket');
```

Unix domain socket servers present the exact same API as a TCP server.

Tip: If you are doing inter-process communication that is local to your host, consider using UNIX domain sockets instead of TCP sockets, as they should perform much better. For instance, when connecting Node to a front-end web-server that stays on the same machine, choosing UNIX domain sockets is generally preferable.

Client

To connect to a UNIX socket server, you can also use *net.createConnection* as when connecting to a TCP Server, but pass in a socket path instead of a port - like this:

```
1  var net = require('net');
2  var conn = net.createConnection('/path/to/socket');
3  conn.on('connect', function() {
4    console.log('connected to unix socket server');
5  });
```

Here, on line 2, we are passing in a socket path instead of a port and a host name.

Datagrams (UDP)

UDP is a connection-less protocol that does not provide the delivery characteristics that TCP does. When sending UDP packets, you cannot guaranteed the order they might arrive in, nor whether they will even arrive at all.

On the other hand, UDP can be quite useful in certain cases, like when you want to broadcast data, when you don't need hard delivery guarantees and sequence or even when you don't know the addresses of your peers.

Datagram server

You can setup a server listening on a UDP port like this:

Code in
udp/udp_server_1.js

```
1   var dgram = require('dgram');
2
3   var server = dgram.createSocket('udp4');
4   server.on('message', function(message, rinfo) {
5     console.log('server got message: ' + message + ' from ' + rinfo.address +\
6     ':' + rinfo.port);
7   });
8
9   server.on('listening', function() {
10    var address = server.address();
11    console.log('server listening on ' + address.address + ':' + address.port\
12  );
13  });
14
15  server.bind(4000);
```

Here we are using the "dgram" module, which provides a way to create a UDP socket (line 3). The *createSocket* function accepts the socket type as the first argument , which can be either "udp4" (UDP over IPv4), "udp6" (UDP over IPv6) or "unix_dgram" (UDP over unix domain sockets).

You can save this in a file named "udp_server_1.js" and run it:

```
1  $ node udp_server_1.js
```

Which should output the server address, port and wait for messages.

You can test it using a tool like netcat (http://netcat.sourceforge.net/[14]) like this:

```
1  $ echo 'hello' | netcat -c -u -w 1 localhost 4000
```

This sends an UDP packet with "hello" to localhost port 4000. You should then get on the server output something like:

```
1  server got message: hello
2  from 127.0.0.1:54950
```

Datagram client

To create an UDP client to send UDP packets you can do something like:

Code in
udp/udp_client_1.js

```
1  var dgram = require('dgram');
2
3  var client = dgram.createSocket('udp4');
4
5  var message = new Buffer('this is a message');
6  client.send(message, 0, message.length, 4000, 'localhost');
7  client.close();
```

Here we are creating a client using the same *createSocket* function we did to create the client, with the difference that we don't bind.

You have to be careful not to change the buffer you pass on *client.send* before the message has been sent. If you need to know when your message has been flushed to the kernel, you should pass a last argument to *client.send* with a callback function to be invoked when the buffer may be reused like this:

---

[14]http://netcat.sourceforge.net/

```
1  client.send(message, 0, message.length, 4000, 'localhost', function() {
2    // you can reuse the buffer now
3  });
```

Since we are not binding, the message is sent from a UDP random port. If we wanted to send from a specfic port, we could have used *client.bind(port)* like this:

Code in
udp/udp_client_2.js

```
1  var dgram = require('dgram');
2
3  var client = dgram.createSocket('udp4');
4
5  var message = new Buffer('this is a message');
6  client.bind(4001);
7  client.send(message, 0, message.length, 4000, 'localhost');
8  client.close();
```

Here we are binding to the specific port 4001, and when saving this file and executing it, the running server should output something like:

```
1  server got message: this is a message from 127.0.0.1:4001
```

This port binding on the client really mixes what a server and a client are, and can be useful for maintaining conversations like this:

```
1   var dgram = require('dgram');
2
3   var client = dgram.createSocket('udp4');
4
5   var message = new Buffer('this is a message');
6   client.bind(4001);
7   client.send(message, 0, message.length, 4000, 'localhost');
8   client.on('message', function(message, rinfo) {
9     console.log('and got the response: ' + message);
10    client.close();
11  });
```

Here we are sending a message, and also listening to messages. When we receive one message we close the client.

Don't forget that UDP is unreliable, and whatever protocol you devise on top of it, account for the fact that messages can be lost or delivered out of order!

Datagram Multicast

One of the interesting uses of UDP is to distribute messages to several nodes using only one network message. This can have many uses like logging, cache cleaning and in the general case where you can afford to lose some messages.

Message multicasting can be useful when you don't need to know the address of all peers. Peers just have to "tune in" and listen to that channel.

Nodes can report their interest in listening to certain multicast channels by "tuning" into that channel. In IP addressing there is a space reserved for multicast addresses. In IPv4 the range is between *224.0.0.0* and *239.255.255.255*, but some of these are reserved. *224.0.0.0* through *224.0.0.255* is reserved for local purposes (as administrative and maintenance tasks) and the range *239.0.0.0* to *239.255.255.255* has also been reserved for "administrative scoping".

Receiving multicast messages

To join a multicast address like *230.1.2.3* you can do something like this:

Code in
udp/udp_multicast_listen.js

```
1  var server = require('dgram').createSocket('udp4');
2
3  server.on('message', function(message, rinfo) {
4    console.log('server got message: ' + message + ' from ' + rinfo.address +\
5    ':' + rinfo.port);
6  });
7
8  server.addMembership('230.1.2.3');
9  server.bind(4000);
```

On line 7 we are saying to the kernel that this UDP socket should receive multicast messages for the multicast address *230.1.2.3*. When calling *addMembership*, you can pass the listening interface as an optional second argument. If omitted, Node will try to listen on every public interface.

Then you can test the server using netcat like this:

```
1  $ echo 'hello' netcat -c -u -w 1 230.1.2.3 4000
```

Sending multicast messages

To send a multicast message you simply have to specify the multicast address like this:

```
1  var dgram = require('dgram');
2
3  var client = dgram.createSocket('udp4');
4
5  var message = new Buffer('this is a multicast message');
6  client.setMulticastTTL(10);
7  client.send(message, 0, message.length, 4000, '230.1.2.3');
8  client.close();
```

Here, besides sending the message, we previously set the Multicast time-to-live to 10 (an arbitrary value here). This TTL tells the network how many hops (routers) it can travel through before it is discarded. Every time a UDP packet travels through a hop, the TTL counter is decremented, and if 0 is reached, the packet is discarded.

What can be the datagram maximum size?

It really depends on the network it travels through. The UDP header allows up to 65535 bytes of data, but if you are sending a packet across an Ethernet network, for instance, the Ethernet MTU is 1500 bytes, limiting the maximum datagram size. Also, some routers will attempt to fragment a large UDP packet into 512 byte chunks.

UDP Exercises

Exercise 1

Create a UDP server that echoes the messages it receives back into the origin socket.

Child processes

On Node you can spawn child processes, which can be another Node process or any process you can launch from the command line. For that you will have to provide the command and arguments to execute it. You can either spawn and live along side the process (spawn), or you can wait until it exits (exec).

Executing commands

You can then launch another process and wait for it to finish like this:

```
1  var exec = require('child_process').exec;
2
3  exec('cat *.js wc -l', function(err, stdout, stderr) {
4    if (err) {
5      console.log('child process exited with error code ' + err.code);
6      return;
7    }
8    console.log(stdout);
9  });
```

Here on line 3 we are passing in "*cat *.js wc -l*" as the command, the first argument to the *exec* invokation. We are then passing as the second argument a callback function that will be invoked once the exec has finished.

If the child process returned an error code, the first argument of the callback will contain an instance of Error, with the code property set to the child exit code.

If not, the output of stdout and stderr will be collected and be offered to us as strings.

You can also pass an optional *options* argument between the command and the callback function like this:

```
1  var options = {timeout: 10000};exec('cat *.js wc -l', options, function(err\
2  , stdout, stderr) {  ...});
```

The available options are:

- *encoding*: the expected encoding for the child output. Defaults to 'utf8';

- *timeout*: the timeout in milliseconds for the execution of the command. Defaults to 0, which does not timeout;

- *maxBuffer*: specifies the maximum size of the output allowed on stdout or stderr. If exceeded, the child is killed. Defaults to 200 * 1024;

- *killSignal*: the signal to be sent to the child if it times out or exceeds the output buffers. Identified as a string;

- *cwd*: current working directory;

- *env*: environment variables to be passed into the child process. Defaults to null.

On the *killSignal* option you can pass a string identifying the name of the signal you wish to send to the target process. Signals are identified in node as strings. For a complete list of strings type on your shell:

```
1  $ man signal
```

Scroll down, and you will see a list of constants representing the signals. Those are the strings used in Node.

Spawning processes

You can spawn a new child process based on the *child_process.spawn* function like this:

```
1  var spawn = require('child_process').spawn;
2
3  var child = spawn('tail', ['-f', '/var/log/system.log']);
4  child.stdout.on('data', function(data) {
5    console.log('stdout: ' + data);
6  });
```

Here we are spawning a child process to run the "tail" command, passing in as arguments "-f" and "/var/log/system.log". This "tail" command will monitor the file "/var/log/system.log" - if it exists - and output every new data appended to it into the stdout.

On line 4 we are listening to the child stdout and printing its output. So here, in this case, we are piping the changes to the "/var/log/system.log" file into our Node application. Besides the stdout, we can also listen to the stderr child output stream like this:

```
1  child.stderr.on('data', function(data) {
2    console.log('stderr: ' + data);
3  });
```

Killing processes

You can (and should) eventually kill child processes by calling the *kill* method on the child object:

```
1  var spawn = require('child_process').spawn;
2  var child = spawn('tail', ['-f', '/var/log/system.log']);
3  child.stdout.on('data', function(data) {
4    console.log('stdout: ' + data);
5    child.kill();
6  });
```

This sends a *SIGTERM* signal to the child process.

You can also send another signal to the child process. You need to specify it inside the kill call like this:

```
1  child.kill('SIGKILL');
```

Child Processes Exercises

Exercise 1

Create a server that a) opens a file b) listens on a unix domain socket and c) spawns a client. This client opens the socket to the server and waits for a file descriptor. The server then passes in the file descriptor we opened in a). The client writes to the file and quits. When the client process quits, the server quits.

Streaming HTTP chunked responses

One of the great features of Node is to be extremely streamable, and since HTTP is a first-class protocol in Node, HTTP responses are no different.

HTTP chunked encoding allows a server to keep sending data to the client without ever sending the body size.

Unless you specify a "Content-Length" header, Node HTTP server sends the header

```
1  Transfer-Encoding: chunked
```

to the client, which makes it wait for a final chunk with length of 0 before giving the response as terminated.

This can be useful for streaming data - text, audio, video - or any other data to the HTTP client.

A streaming example

Here we are going to code an example that pipes the output of a child process into the client:

Source code in chapters/chunked/chunked.js

```
1  var spawn = require('child_process').spawn;
2
3  require('http').createServer(function(req, res) {
4    var child = spawn('tail', ['-f', '/var/log/system.log']);
5    child.stdout.pipe(res);
6    res.on('end', function() {
7      child.kill();
8    });
9  }).listen(4000);
```

Here we are creating an HTTP server (line 3) and binding it to port 4000 (line 9).
When there is a new request we launch a new child process by executing the command "tail -f /var/log/system.log" (line 4) whose output is being piped into the response (line 5).
When the response ends (because the browser window was closed, or the network connection was severed, for instance), we kill the child process so it does not hang around afterwards indefinitely.

So here, in 9 lines of code, we are making a Node streaming server that spawns, pipes the output of a process and then kills it as needed.

Streaming Exercises

Exercise 1

Create a mixed TCP and HTTP server that, for every HTTP request, streams all the TCP clients input into the request response.

TLS / SSL

TLS (Transport Layer Security) and SSL (Secure Socket Layer) allow client / server applications to communicate across a network in a way designed to prevent eavesdropping (others looking into your messages) and tampering (others changing your message). TLS and SSL encrypt the segments of network connections above the Transport layer, enabling both privacy and message authentication.

TLS is a standard based on the earlier SSL specifications developed by Netscape. In fact, TLS 1.0 is also known as SSL 3.1 , and the latest version (TLS 1.2) is also known as SSL 3.3. So, from hereon, we will be using "TLS" instead of the deprecated "SSL".

Public / private keys

Node TLS implementation is based on the OpenSSL library. Chances are you have the library installed, also as the openssl command-line utility. If it's not installed, you should be looking for a package named "openssl".

Private key

TLS is a public / private key infrastructure. Each client and server must have a private key. A private key can be created by the openssl utility on the command line like this:

```
1  $ openssl genrsa -out my.pem 1024
```

This should create a file named *my.pem* with your private key.

Public key

All servers and some clients need to have a certificate. Certificates are public keys signed by a Certificate Authority or self-signed. The first step to getting a certificate is to create a "Certificate Signing Request" (CSR) file. This can be done with:

```
1  $ openssl req -new -key my_key.pem -out my_csr.pem
```

This will create a CSR file named *my_csr.pem.*

To create a self-signed certificate with the CSR, you can do this:

```
1  openssl x509 -req -in my_csr.pem -signkey my_key.pem -out my_cert.pem
```

This will create a self-signed certificate file named *my_cert.pem.*

Alternatively you can send the CSR to a Certificate Authority for signing.

TLS Client

You can connect to a TLS server using something like this:

```
1  var tls = require('tls'),
2      fs  = require('fs'),
3      port = 3000,
4      host = 'myhost.com',
5      options = {
6        key : fs.readFileSync('/path/to/my/private_key.pem'),
7        cert : fs.readFileSync('/path/to/my/certificate.pem')
8  };
9
10 var client = tls.connect(port, host, options, function() {
11   console.log('connected');
12   console.log('authorized: ' + client.authorized);
13   client.on('data', function(data) {
14     client.write(data); // just send data back to server
15   });
16 });
```

First we need to inform Node of the client private key and client certificate, which should be strings. We are then reading the pem files into memory using the synchronous version of *fs.readFile*, *fs.readFileSync*.

- Here we are using *fs.readFileSync*, a synchronous function. Won't this block the event loop?

No, this will just run on the initialization of our app. As long as you don't use blocking functions inside an event handler, you should be ok.

- Wait, what if this is a module we are requiring this inside a callback?

You shouldn't be requiring modules inside callbacks. They do synchronous file system access and will block your event loop.

Then, on line 10 we are connecting to the server. *tls.connect* returns a *CryptoStream* object, which you can use normally as a *ReadStream* and *WriteStream*. On line 13 we just wait for data from the server as we would on a *ReadStream*, and then we, in this case, send it back to the server on line 14.

TLS Server

A TLS server is a subclass of *net.Server*. With it you can make everything you can with a *net.Server*, except that you are doing over a secure connection.

Here is an example of a simple echo TLS server:

```
 1  var tls = require('tls');
 2      fs  = require('fs');
 3      options = {
 4        key : fs.readFileSync('/path/to/my/server_private_key.pem'),
 5        cert : fs.readFileSync('/path/to/my/server_certificate.pem')
 6  };
 7
 8  tls.createServer(options, function(s) {
 9    s.pipe(s);
10  }).listen(4000);
```

Besides the *key* and *cert* options, *tls.createServer* also accepts:

- *requestCert*: If *true* the server will request a certificate from clients that connect and attempt to verify that certificate. Default: *false*.

- *rejectUnauthorized*: If *true* the server will reject any connection which is not authorized with the list of supplied CAs. This option only has an effect if requestCert is *true*. Default: *false*.

Verification

On both the client and the server APIs, the stream has a property named *authorized*. This is a boolean indicating if the client was verified by one of the certificate authorities you are using, or one that they delegate to. If *s.authorized* is *false*, then *s.authorizationError* contains the description of how the authorization failed.

TLS Exercises

Exercise 1

Create a certificate authority. Create a client certificate signed by this new certificate authority.

Exercise 2

Create a TLS echo server that uses the default certificate authorities.

Exercise 3

Create a TLS client that reads from *stdin* and sends it to the echo TLS server created on exercise 2.

Exercise 4

Make the TLS server only accept connections if the client is certified. Verify that he does not let the client created on exercise 3 connect.

Exercise 5

Make the TLS Server use the same certificate authority you used to sign the client certificate with. Verify that the server now accepts connections from this client.

HTTPS

HTTPS is the HTTP protocol over TLS. In Node, HTTPS is implemented as a separate module. The HTTPS API is very similar to the HTTP one, with some honorable small differences.

HTTPS Server

To create a server, you can do something like this:

```
1   var https = require('https'),
2       fs    = require('fs');
3
4   var options = {
5     key: fs.readFileSync('/path/to/server/private_key.pem'),
6     cert: fs.readFileSync('/path/to/server/cert.pem')
7   };
8
9   https.createServer(options, function(req, res) {
10    res.writeHead(200, {'Content-Type': 'text/plain'});
11    res.end('Hello World!');
12  });
```

So here, the first argument to *https.createServer* is an options object that, much like in the TLS module, provides the private key and the certificate strings.

HTTPS Client

To make an HTTPS request you must also use the *https* module like this:

```
1   var https = require('https');
2   var options = {
3     host: 'encrypted.google.com',
4     port: 443,
5     path: '/',
6     method: 'GET'
7   };
8
9   var req = https.request(options, function(res) {
10    console.log("statusCode: ", res.statusCode);
11    console.log("headers: ", res.headers);
12
13    res.on('data', function(d) {
14      process.stdout.write(d);
15    });
16  });
17  req.end();
```

Here the options object, besides the *http.request* options, also accepts:

- port: port of host to request to. Defaults to 443.

- key: The client private key string to use for SSL. Defaults to *null*.

- cert: The client certificate to use. Defaults to *null*.

- ca: An authority certificate or array of authority certificates to check the remote host against.

You may want to use the *key* and *cert* options if the server needs to verify the client.

Also, you may pass the *ca* argument, which is a certificate authority certificate or an array of them with which you may verify the server.

Much like the *http* module, this module also offers a shortcut *https.get* method that can be used like this:

```
1   var https = require('https');
2   var options = { host: 'encrypted.google.com', path: '/' };
3
4   https.get(options, function(res) {
5     res.on('data', function(d) {
6     process.console.log(d.toString());
7   });
```

Making modules

CommonJS modules

When crafting your first Node app, you tend to cram everything into one file, but sooner or later you will need to expand.

The Node way to spread your app is to compartmentalise it into logic blocks called modules. These modules will have an interface, exposing module properties like functions or simple attributes.

One file module

To create a module you simply have to create a file somewhere in your app dir tree (*lib/my_module.js* is perhaps the appropriate place to start).

Inside each module you can use the global namespace without fear of stepping on another module's toes. And, at the end, you expose only what you wish to expose by assigning it to module.exports. Here is a quick example:

```
1   var counter = 0;
2   var onePrivateMethod = function() {
3     return counter;
4   } ;
5
6   var onePublicMethod = function() {
7     onePrivateMethod();
8     return 'you already called this module ' + counter + ' times';
9   };
10
11  module.exports = onePublicMethod;
```

Here we are exporting (on the last line) only one function. If we save this module in the current directory under "my_module.js":

```
1   var myModule = require('./my_module');
2   myModule(); // => 'you already called this module 1 times';
```

You can export any JavaScript object you wish, so, for instance, you can export an object that has a collection of functions like this:

```
1   var counter1 = 0;
2   var onePublicMethod = function() {
3     return 'you already called this function ' + (++ counter1) + ' times';
4   };
5
6   var counter2 = 0;
7   var anotherPublicMethod = function() {
8     return 'you already called this function ' + (++ counter2) + ' times';
9   }
10
11  module.exports = {
12    functionA: onePublicMethod,
13    functionB: anotherPublicMethod
14  };
```

A client using this module would look something like:

```
1   var myModule = require('./my_module');
2   myModule.functionA();
3   // => 'you already called this function 1 times';
4   myModule.functionA();
5   // => 'you already called this function 2 times';
6   myModule.functionB();
7   // => 'you already called this function 1 times';
```

An aggregating module

Also, modules can aggregate other modules and mix and expose them as they wish. For instance, such a module could look like:

```
1   var moduleA = require('./moduleA');
2   var moduleB = require('./moduleB');
3
4   var myFunc = function() {
5     return "doing some crazy stuff";
6   }
7
8   module.exports = {
9     funcA: moduleA.funcA,
10    funcB: moduleB.funcB,
11    funcC: myFunc
12  }
```

A pseudo-class

If you need to learn about JavaScript pseudo-classes and prototypical inheritance I can recommend that you read the book "Eloquent JavaScript"[15] or Douglas Crockford's "JavaScript - The Good Parts".

It is possible to implement a classlike(ish) behavior on your module using something like this:

```
1   var Line = function(x1, y1, x2, y2) {
2     this.x1 = x1;
3     this.y1 = y1;
4     this.x2 = x2;
5     this.y2 = x2;
6   };
7
8   Line.prototype.length = function() {
9     return Math.sqrt(
10      Math.pow(Math.abs(this.x1 - this.x2), 2) +
11      Math.pow(Math.abs(this.y1 - this.y2), 2)
12    );
13  };
14
15  module.exports.create = function(x1, y1, x2, y2) {
16    return new Line(x1, y1, x2, y2);
17  };
```

Here we are creating the Line pseudo-class, but we are not exporting its constructor directly. Instead, we are exporting a "create" function, which calls the constructor for us. We are doing this because people using this module may not remember that it would be necessary to use the "new" keyword when invoking the constructor function. If they forgot to do so, this would be bound to the global namespace, yielding very strange results. To prevent it we just export one create function, leading to clear module usage like:

```
1   var Line = require('./line');
2   var line = Line.create(2, 4, 10, 15);
3   console.log('this line length is ' + line.length());
```

A pseudo-class that inherits

Besides implementing a class behavior, you can also inherit it from another class. For instance, the EventEmitter class is a very useful to use like this:

---

[15]http://eloquentjavascript.net/

```
1   var util        = require('util'),
2       EventEmitter = require('events').EventEmitter;
3
4   var Line = function(x1, y1, x2, y2) {
5     this.x1 = x1;
6     this.y1 = y1;
7     this.x2 = x2;
8     this.y2 = x2;
9   };
10
11  util.inherits(Line, EventEmitter);
12
13  Line.prototype.length = function() {
14    return Math.sqrt(
15      Math.pow(Math.abs(this.x1 - this.x2), 2) +
16      Math.pow(Math.abs(this.y1 - this.y2), 2)
17    );
18  };
```

Note that you should call *util.inherits* before declaring the prototype properties on the pseudo-class (like the *Line.prototype.length* on the previous example). If you call *util.inherits* afterwards, they will be removed, since the prototype object is replaced on line 11.

Now you should be ready to create your own modules, so your app code doesn't have to live on a single file!

node_modules and npm bundle

As explained on the module loading chapter, Node tries to use the nearest "node_modules" directory by backtracking the current directory up to root. What this means is that, generally, you can put the external modules your application depends on into a node_modules folder inside your app root folder, thus bundling and "freezing" your application dependencies.

Fortunately npm can do that for you. For that to work you need to declare your application dependencies inside a package.json file like this:

Source code in chapters/packaging/app1

```
 1  { "name" : "app1"
 2  , "version" : "0.1.0"
 3  , "description" : "Hands-on packaging app example"
 4  , "main" : "app.js"
 5  , "dependencies" :
 6    {
 7      "express" : ">= 1.0"
 8      , "jade" : "0.8.5"
 9    }
10  }
```

This is a minimal package.json, you can add more information to it. You can type

```
 1  $ npm help json
```

to get a man page documenting the JSON document format.

On lines 7 and 8 we declare that this application depends on "express" and "jade" npm packages, specifying the version requirements.

Bundling

Having now the package.json package in your root directory you can bundle all your dependencies into a "node_modules" by typing into your console, in your app root dir:

```
 1  $ npm install
```

This will create a "node_modules" inside the app root dir – if it doesn't exist yet – and when your app is executed, node will first look into this directory to resolve modules.

Using this technique you can package an application so you don't run into cross-dependencies problems on co-existing applications and removing the need to install packages globally.

chapter_22

Debugging

If you find yourself in a situation where you need to inspect the inner workings of your Node app code, there are several tools that can come to your aid.

console.log

The simplest one is *console.log.* You can use it to inspect objects like this:

```
1  var obj = {a: 1, b: 2};
2  console.log(obj);
```

Which will print

```
1  { a: 1, b: 2 }
```

Node built-in debugger

If you need to halt execution to carefully inspect your app, you can use Node's (simple and basic) built-in debugger.

First, you have to insert an initial breakpoint in your app by inserting a *debugger* instruction like this:

```
1  var a = 1;
2  debugger;
3  var b = a + 1;
```

And then you can start the Node debuuger on your app like this:

```
1  $ node debug my_app.js
```

This will launch the Node debugger. You should now get the debugger prompt, but your app is not running just yet.

Inside the debugger prompt you will have to type:

```
1   $ run
```

And your app will start running, hitting your first *debugger* instruction.

It will then stop on the breakpoint you set as soon as it is encountered. If you type

```
1   $ list
```

you will see where you are inside your script.

You can inspect the local scope. You can print variables like this:

```
1   $ print a
```

If you type

```
1   $ next
```

you will step into the next line of your script.

If you type

```
1   $ continue
```

your app will resume, stopping if it passes another (or the same) breakpoint.

When inside the prompt you can kill your app by commanding:

```
1   $ kill
```

Node Inspector

Another debugging tool is Node Inspector. This debugger brings the full-fledged Chrome inspector to your Node app.

You can install Node Inspector like this:

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

Node Inspector runs as a daemon by default on port 8080. You can launch it like this:

```
1   $ node-inspector &
```

This will send the node-inspector process to the background.

Next you need to fire up your app, but using a *–debug* or *–debug-brk* option on the node executable like this:

```
1   $ node --debug-brk myapp.js
```

The *–debug-brk* option will make your app break on the first line, while the *–debug* option will simply enable debugging.

Tip: when debugging servers you will want to use *–debug*, and when debugging other scripts you may want to break on the first line by using *–debug-brk*.

Now you can open your browser and point it to http://localhost:8080, and you should get something like this:
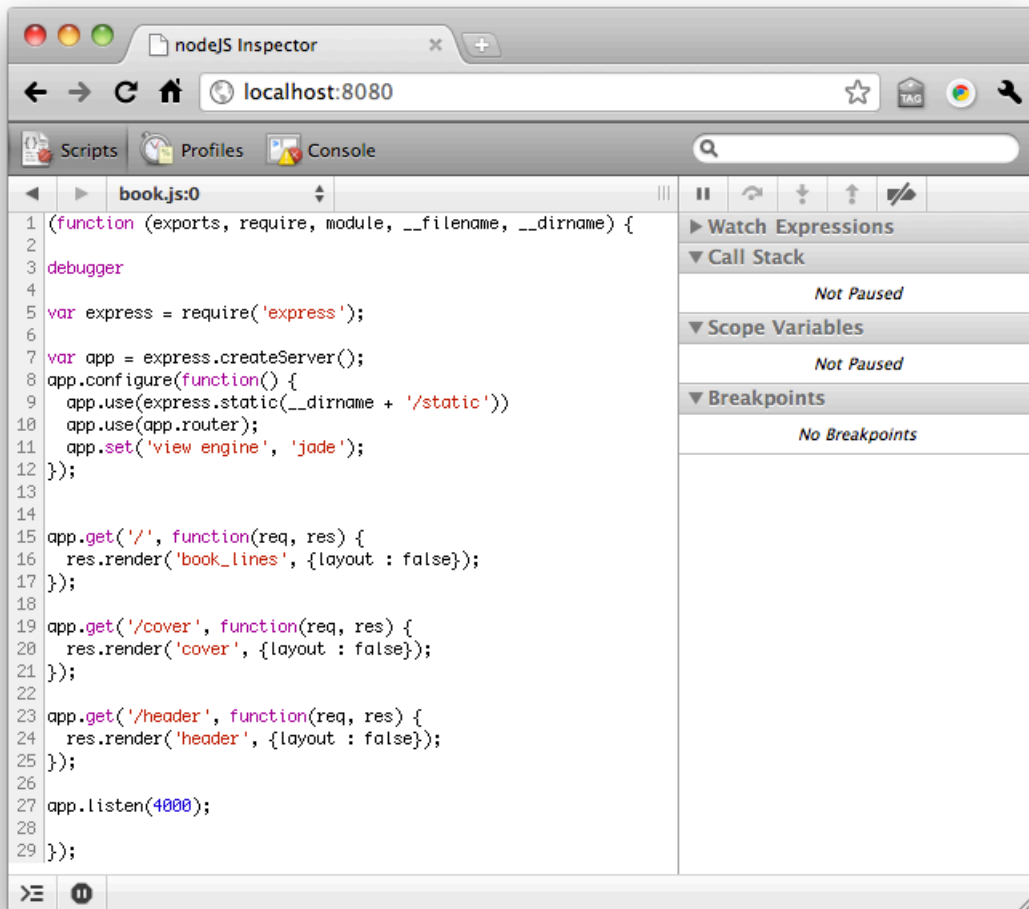
You can set and unset breakpoints by clicking on the line numbers.

When a breakpoint is reached, your app freezes and Node Inspector shows the context:
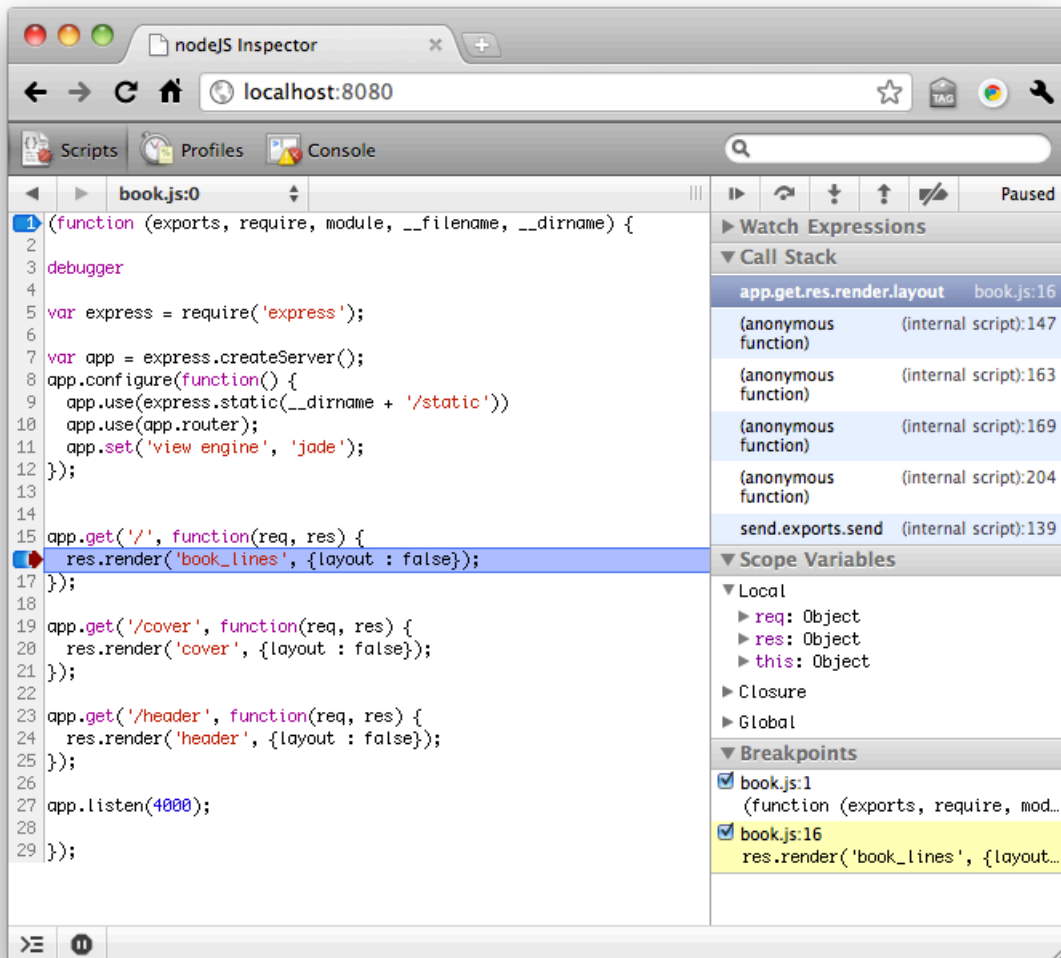
You can see the two most interesting context panes on the right:

The call stack, where you can see which functions were invoked to get to this point.

The scope variables, where you can inspect the local variables, the global variables and the closure variables, which are variables that are defined on a higher function scope.

Above those panes you can see some buttons which you can use to manipulate the executed instructions:

Continue execution up until a breakpoint is reached.

Execute this function and stop on the next line.

Step into this function.

⬆ Continue, and when this function ends return to the caller.

Live edit

You can change the code while you are debugging. For this you can double-click on a line of code and edit it. Change the code and hit the "Tab" key or click outside the edit area. And voila, you just changed running code!

Changing the code like this will not save save the changes - it should only be used to test quick fixes.

Automated Unit Testing

When building a module you should also create an automated way to test it. A set of tests that test a module are called unit tests.

Let's say you want to create the unit tests for a module that exports a function that sums two integers:

```
1  module.exports = function(a, b) {
2    return a + b;
3  };
```

For that you need two tools: a test runner and an assertion module.

A test runner

A test runner is a piece of software that loads your tests, runs them and then presents the test result. The result can be positive (test passed) or negative (test failed), generally accompanied by a description of the failure.

There are various test runners, and my favorite nowadays is *mocha*. You can install it by running:

```
1  $ npm install -g mocha
```

On your app you should create a "tests" dir under the root, and create a file module for each one of your modules.
So, we would create a test module under *tests/sum.js*. This module should then define description scopes and tests for any functionality you need testing.
Each one of these tests is a function that is launched asynchronously. When all are done or failed, mocha reports how many were run and how many failed / succeeded.

A test file for the "sum" module would be something like this:

```
 1  require('should');
 2
 3  var sum = require('../lib/sum');
 4
 5  describe("Sum Lib", function() {
 6
 7    it("should be able to sum 0 and 5", function() {
 8      sum(0, 5).should.equal(5);
 9    });
10
11    it("should be able to sum 2 and 5", function() {
12      sum(2, 5).should.equal(7);
13    });
14
15    it("should be able to sum do some sums", function() {
16      sum(1, 1).should.equal(2);
17      sum(1, 2).should.equal(3);
18       sum(2, 1).should.equal(3);
19      sum(10, 120).should.equal(130);
20    });
21  });
```

To run this test you'll also need the *should* package to be installed:

```
 1  $ npm install should
```

To run expresso on all files inside the *tests* directory you can invoke the *mocha* executable like this:

```
 1  $ mocha test/*.js
```

You can also specify asynchronous tests by accepting a callback function that should be invoked once the test is done:

```
 1  it("should be able to execute asynchronously", function(done) {
 2    setTimeout(function() {
 3      done();
 4    }, 2000);
 5  });
```

You can obtain more info about mocha here http://visionmedia.github.com/mocha/[16].

---

[16]http://visionmedia.github.com/mocha/

Assertion testing module

An assertion testing module is something that allows you to easily compare results to what you expect. For instance, a module that sums two integers should be compared against some known cases.

Node already comes with a basic assertion module named "assert". You can use the following module functions:

- *assert.equal(a, b, [message])* - test shallow equality (with *==*);

- *assert.deepEqual(a, b, [message])* - test for deep equality;

- *assert.notEqual(a, b, [message])* - test shallow inequality (as with *!=*);

- *assert.notDeepEqual(a, b, [message])* - test for deep inequality;

- *assert.strictEqual(a, b, [message])* - test strict equality (as with *===*);

- *assert.notStrictEqual(a, b, [message])* - test strict inequality (as with *!==*);

- *assert.throws(block, [error], [message])* - test if the block (given as function) throws an error. With the optional *error* argument you can pass in an Error instance to be compared with the thrown exception, a regular expression to be compared with the exception, or a validation function.

- *assert.doesNotThrow(block, [error], [message])* - the negation of the result of assert.throws() with the same arguments.

All these messages have a final optional argument where you can pass the message in case of failure.

should.js

Another useful assertion testing module is "should.js". Should.js provides a nice extension to the objects, into which you can perform tests using a nice API that you can chain like this:

```
1   value.should.be.a('object').and.have.property('name', 'Pedro');
```

You can install Should.js using npm like this;

```
1   $ npm install should
```

and then include the module

```
1   require('should');
```

There is no need to assign a variable to the module since should extends

With it you can:

Assert truthfulness:

```
1  true.should.be.ok;
2
3  'yay'.should.be.ok;
```

or untruthfulness:

```
1  false.should.not.be.ok;
2
3  ''.should.not.be.ok;
```

=== true

```
1  true.should.be.true;
2
3  '1'.should.not.be.true;
```

=== false

```
1  false.should.be.false;
2
3  ''.should.not.be.false;
```

emptiness

```
1  [].should.be.empty;
2
3  ''.should.be.empty;
```

equality

```
1  ({ foo: 'bar' }).should.eql({ foo: 'bar' });
2
3  [1,2,3].should.eql([1,2,3]);
```

equal (strict equality)

```
1  (4).should.equal(4);
2
3  'test'.should.equal('test');
4
5  [1,2,3].should.not.equal([1,2,3]);
```

assert numeric range (inclusive) with *within*

```
1   15.should.be.within(10, 20);
```

test numeric value is above given value:

```
1   10.should.be.above(5);
2
3   10.should.not.be.above(15);
```

test numeric value is below given value:

```
1   10.should.not.be.below(5);
2
3   10.should.be.below(15);
```

matching regular expressions

```
1   "562".should.match(/[0-9]{3}/);
```

test length

```
1   [1, 2, 3].should.have.length(3);
```

substring inclusion

```
1   "abcdef".should.include.string('bc');
```

assert typeof

```
1   {a:1, b:2}.should.be.a('object');
2
3   "test".should.be.a('string');
```

property existence

```
1   {a:1, b:2}.should.have.property('a');
2
3   {a:1, b:2}.should.not.have.property('c');
```

array containment

```
1  [1,2,3].should.contain(3);
2
3  [1,2,3].should.not.contain(4);
```

own object keys

```
1  var obj = { foo: 'bar', baz: 'raz' };
2  obj.should.have.keys('foo', 'bar');
3  obj.should.have.keys(['foo', 'bar']);
```

responds to, asserting that a given property is a function:

```
1  user.should.respondTo('email');
```

Putting it all together

The source code for this section can be found at chapters/testing

So now we should be able to make a test for our "sum" module:

```
1  require('should');
2
3  var sum = require('../lib/sum');
4
5  describe("Sum Lib", function() {
6
7    it("should be able to sum 0 to 5", function() {
8      sum(0, 5).should.equal(5);
9    });
10
11   it("should be able to sum 2 to 5", function() {
12     sum(2, 5).should.equal(7);
13   });
14
15   it("should be able to sum do some sums", function() {
16     sum(1, 1).should.equal(2);
17     sum(1, 2).should.equal(3);
18     sum(2, 1).should.equal(3);
19     sum(10, 120).should.equal(130);
20   });
21  });
```

And then we can run our test from the command line like this;

```
1  $ mocha tests/*.js
```

Which should print out:

```
1  ... â" 3 tests complete (4 ms)
```

In case you are testing callbacks, you can also use a first argument given to all testing functions, which we can name *beforeExit*.
This function can be used to attach callbacks before the tests end, so you can test if your callbacks have been called or not:

```
1  module.exports.testAsync = function(beforeExit) {
2    var n = 0;
3    setTimeout(function(){
4      ++n;
5      assert.ok(true);
6    }, 200);
7    setTimeout(function(){
8      ++n;
9      assert.ok(true);
10   }, 200);
11   beforeExit(function(){
12     assert.equal(2, n, 'Ensure both timeouts are called');
13   });
14 };
```

Callback flow

As you may have noticed, asynchronous programming does not rely on the stack to organize flow between caller and called function. Instead, it relies on callback functions that are usually passed as arguments.

Imagine that you would have to build a script that does the following:

Append bytes 10-20 from file a.txt into file b.txt. Both files already exist.

A solution may be something like this:

Source code in:
flow/exercise_1.js

```
1   var fs = require('fs');
2
3   var doWhatWasAsked = function(callback) {
4     fs.open(__dirname + '/a.txt', 'r', function(err, aFd) {
5       if (err) { callback(err); return; }
6       var buffer = new Buffer(10);
7       fs.read(aFd, buffer, 0, 10, 10, function(err, bytesRead) {
8         if (err) { callback(err); return; }
9
10        fs.open(__dirname + '/b.txt', 'a', function(err, bFd) {
11          if (err) { callback(err); return; }
12          fs.fstat(bFd, function(err, bStats) {
13            if (err) { callback(err); return; }
14            fs.write(bFd, buffer, 0, 10, bStats.size, callback);
15          })
16        })
17      });
18    })
19  };
20
21  console.log('starting...');
22  doWhatWasAsked(function(err) {
23    if (err) { throw err; }
24    console.log('done');
25  });
```

Here we devise a function sillily called "doWhatWasAsked", which receives a callback to be invoked when there is an error or when the task is done.
This function opens a.txt (line 4), and then reads 10 bytes starting at pos 10 (line 7).

103

Then it opens b.txt (line 10), checks its size using *fs.stat* (line 12) and then writes into the end of the file (line 14).

The boomerang effect

In this example you can see what can be called a "boomerang effect" of callback chaining, where the text indentation increases and then decreases along with function nesting.

This can turn your code into "callback spaghetti", making it visually hard to track which context you are in. This style also makes debugging your application difficult, reducing even more the maintainability of your code.

There is more than one way around this. One is: instead of using anonymous inline functions we can use named functions - like this:

Source code in
flow/exercise_2.js

```
1  var fs   = require('fs');
2
3  var doWhatWasAsked = function(callback) {
4    var aFd, bFd, buffer = new Buffer(10);
5
6    function openA() {
7      fs.open(__dirname + '/a.txt', 'r', readFromA);
8    }
9
10   function readFromA(err, fd) {
11     if (err) { callback(err); return; }
12     aFd = fd;
13     fs.read(aFd, buffer, 0, 10, 10, openB);
14   }
15
16   function openB(err) {
17     if (err) { callback(err); return; }
18     fs.open(__dirname + '/b.txt', 'a', statB);
19   }
20
21   function statB(err, fd) {
22     if (err) { callback(err); return; }
23     bFd = fd;
24     fs.fstat(bFd, writeB);
25   }
```

```
26
27     function writeB(err, bStats) {
28       if (err) { callback(err); return; }
29       fs.write(bFd, buffer, 0, 10, bStats.size, callback);
30     }
31
32     openA();
33   };
34
35   console.log('starting...');
36   doWhatWasAsked(function(err) {
37     if (err) { throw err; }
38     console.log('done');
39   });
```

This code does what the previous code did, but it's unarguably clearer. We are now declaring one named function for each callback all under the same scope, and using the function names to pass them as the next callback to be executed.

The downside of this technique is that we lose the closure scopes, so we need to store the application state on a common scope (line 4).

We could discuss which approach is more elegant, but this one is certainly more readable.

Using caolan/async

Async is a utility module which provides straight-forward, powerful functions for working with asynchronous JavaScript. Although originally designed for use with node.js, it can also be used directly in the browser.

Collections

Parallel Iterations

When you have a collection and you need to resolve them in an async manner, you can use async.forEach():

Here is a service that returns the square of a number:

Source code in
flow/squaring_server.js

```
1  require('http').createServer(function(req, res) {
2    var body = '';
3
4    req.setEncoding('utf8');
5    req.on('data', function(data) {
6      body += data;
7    });
8
9    req.on('end', function() {
10      var number = parseInt(body, 10);
11      var squared = Math.pow(number, 2);
12      res.end(squared.toString());
13    });
14
15  }).listen(4001);
```

You can save this snippet into a file named "squaring_server.js" and launch it:

```
1  $ node squaring_server.js
```

You can install the "async" package using NPM inside the localhost:

```
1  $ npm install async
```

For our purpose you should also install the "request" module using NPM:

```
1  $ npm install request
```

async.forEach

You can then use async to iterate asynchronously and query the square of a collection of numbers:

Source code in
flow/squaring_client.js

```
1   var async = require('async');
2   var request = require('request');
3
4   var collection = [1, 2, 3, 4];
5
6   function iterator(element, next) {
7     request({ uri: 'http://localhost:4001/',
8               body: element.toString() },
9            function(err, res, body) {
10              console.log('square of %d is %d', element, body);
11              next(err);
12            });
13  }
14
15  function callback() {
16    console.log('finished');
17  }
18
19  async.forEach(collection, iterator, callback);
```

async.map

You can also collect the results asynchronously using async.map like this:

Source code in
flow/squaring_client_map.js

```
1   var async = require('async');
2   var request = require('request');
3
4   var collection = [1, 2, 3, 4];
5
6   function iterator(element, next) {
7     request({ uri: 'http://localhost:4001/',
8               body: element.toString() },
9            function(err, res, body) {
10              next(err, parseInt(body, 10));
11            });
12  }
13
14  function callback(err, result) {
15    console.log('finished.');
16    for (var i in collection) {
```

```
17        console.log('the square of %d is %d', collection[i], result[i]);
18    }
19  }
20
21  async.map(collection, iterator, callback);
```

Notice that the results are in the same order as they were issued, even though they might have finished in a different order.

async.forEachLimit

You can limit the amount of parallelism by using async.forEachLimit like this:

Source code in
flow/squaring_client_limited.js

```
1  var async = require('async');
2  var request = require('request');
3
4  var collection = [];
5
6  for (var i = 0; i   collection.push(i);
7  }
8
9  var maxConcurrency = 5;
10
11 function iterator(element, next) {
12   request({ uri: 'http://localhost:4001/',
13           body: element.toString() },
14          function(err, res, body) {
15            console.log('square of %d is %d', element, body);
16            next(err);
17          });
18 }
19
20 function callback() {
21   console.log('finished');
22 }
23
24 async.forEachLimit(collection, maxConcurrency, iterator, callback);
```

async.filter

You can also filter a collection asynchronously:

```
1   var async = require('async');
2   var exists = require('path').exists;
3
4   var files = [
5     'filter.js',
6     'does_not_exist.js',
7     'squaring_client.js',
8     'also_does_not_exist.js',
9     'squaring_client_limited.js',
10    'squaring_client_map.js',
11    'also_does_not_exist_2.js',
12    'squaring_server.js'
13  ];
14
15  function callback(result) {
16    console.log('of files %j, these exist: %j', files, result);
17  }
18
19  async.filter(files, exists, callback);
```

Here we are using path.exists (which is asynchronous) to filter out a list of files.

async.reject

Reject is the opposite of filter:

```
1   var async = require('async');
2   var exists = require('path').exists;
3
4   var files = [
5     'filter.js',
6     'does_not_exist.js',
7     'squaring_client.js',
8     'also_does_not_exist.js',
9     'squaring_client_limited.js',
10    'squaring_client_map.js',
11    'also_does_not_exist_2.js',
12    'squaring_server.js'
13  ];
14
15  function callback(result) {
16    console.log('of files %j, these do not exist: %j', files, result);
```

```
17  }
18
19  async.reject(files, exists, callback);
```

async.reduce

This reduces a list of values using an asynchronous iterator to return each step.

For instance, if we want to calculate the 100th Fibbonacci number, we can use a web service that sums two numbers:

```
1  function sum(a, b) {
2    return a + b;
3  }
4
5  require('http').createServer(function(req, res) {
6    var body = '';
7
8    req.setEncoding('utf8');
9    req.on('data', function(data) {
10     body += data;
11   });
12
13   req.on('end', function() {
14     var numbers = body.split('&').map(function(arg){
15       return arg.split('=');
16     }).map(function(strs) {
17       return parseInt(strs[1], 10);
18     });
19     console.log(numbers);
20     var total = numbers.reduce(sum, 0);
21     res.end(total.toString());
22   });
23
24 }).listen(4001);
```

Now we are ready to make a client to calculate the 100th Fibbonacci number:

Source code in
flow/fibbonacci.js

```
 1   var async = require('async');
 2   var request = require('request');
 3
 4   var items = [];
 5   var order = process.argv[2] && parseInt(process.argv[2]) || 100;
 6   var orderMinus2 = order - 2;
 7
 8   for (var i = 0 ; i    items.push(i);
 9   }
10
11   var memo = [0, 1];
12
13   function iterator(memo, item, next) {
14     request({uri: 'http://localhost:4001/', form: {a: memo[0], b: memo[1]}}, \
15   function(err, res, body) {
16       next(err, [memo[1], parseInt(body, 10)]);
17     });
18   }
19
20   async.reduce(items, memo, iterator, function(err, result) {
21     if (err) { throw err; }
22     console.log('Fibbonacci of order %d: %d', order, result[1]);
23   });
```

async.detect

async.detect returns the first value in a list that passes an async truth test:

Source code in
flow/detect.js

```
 1   var async = require('async');
 2   var exists = require('path').exists;
 3
 4   var files = [
 5     'filter.js',
 6     'does_not_exist.js',
 7     'squaring_client.js',
 8     'also_does_not_exist.js',
 9     'squaring_client_limited.js',
10     'squaring_client_map.js',
11     'also_does_not_exist_2.js',
12     'squaring_server.js'
```

```
13  ];
14
15  function callback(result) {
16    console.log('this file exists: %s', result);
17  }
18
19  async.detect(files, exists, callback);
```

If you save this to a file named detect.js, and run it:

```
1  $ node detect.js
```

You should get the result:

```
1  file exists: filter.js
```

async.some

Returns true if at least one element in the array satisfies the asynchronous test.

Source code in
flow/some.js

```
1   var async = require('async');
2   var exists = require('path').exists;
3
4   var files = [
5     'does_not_exist.js',
6     'also_does_not_exist.js',
7     'also_does_not_exist_2.js',
8     'filter.js' ];
9
10  function callback(result) {
11    console.log('at least one of these files exists: %j', result);
12  }
13
14  async.some(files, exists, callback);
```

async.every

Returns true if all elements in the array satisfies the asynchronous test.

Source code in
flow/every.js

```
1   var async = require('async');
2   var exists = require('path').exists;
3
4   var files = [
5     'filter.js',
6     'does_not_exist.js',
7     'squaring_client.js',
8     'also_does_not_exist.js',
9     'squaring_client_limited.js',
10    'squaring_client_map.js',
11    'also_does_not_exist_2.js',
12    'squaring_server.js'
13  ];
14
15  function callback(result) {
16    console.log('all these files exist: %j', result);
17  }
18
19  async.every(files, exists, callback);
```

Running this example should result in:

```
1   all these files exist: false
```

Flow Control

async.series

Executes a bunch of tasks in series. Returns results to callback as an array. Stops if an error occurs.

Source code in
flow/series.js

```
1   var async = require('async');
2   var request = require('request');
3
4   var functions = [];
5
6   for ( var i = 0; i    functions.push(function(i) {
7
8       return function(done) {
9
10        request({
11          uri: 'http://localhost:4001'
12        , body: i.toString()}
13        , function(err, res, body) {
14            done(err, body);
15          });
16
17      }
18
19    }(i));
20  }
21
22  function callback(err, result) {
23    if (err) { throw err; }
24    console.log('done: %j', result);
25  }
26
27  async.series(functions, callback);
```

async.parallel

Run an array of functions in parallel without waiting for the previous function to have finished.

Fire up the squaring server:

```
1   $ node squaring_server.js
```

And then run this:

Source code in
flow/parallel.js

```
1   var async = require('async');
2   var request = require('request');
3
4   var functions = [];
5
6   for ( var i = 0; i    functions.push(function(i) {
7
8       return function(done) {
9
10        request({
11          uri: 'http://localhost:4001'
12        , body: i.toString()}
13        , function(err, res, body) {
14            done(err, body);
15          });
16
17      }
18
19    }(i));
20  }
21
22  function callback(err, result) {
23    if (err) { throw err; }
24    console.log('done: %j', result);
25  }
26
27  async.parallel(functions, callback);
```

async.whilst

Repeatedly call fn, while test returns true. Here is an example for calculating the biggest integer whose square is smaller than 1000:

```
1   var async = require('async');
2   var request = require('request');
3
4   var i = -1;
5   var n = i;
6   var lastResult = -1;
7
8   function test() {
9     var pass = lastResult    if (pass) { i = n; }
10    return pass;
11  }
12
13  function action(done) {
14    n = i + 1;
15    request({
16      uri: 'http://localhost:4001/'
17    , body: n.toString()}
18    , function(err, res, body) {
19        if (err) { return done(err); }
20        lastResult = parseInt(body, 10);
21        done();
22    });
23  }
24
25  function callback(err) {
26    if (err) { throw err; }
27    console.log('the biggest integer whose square is smaller than 1000: %d', \
28  i);
29  }
30
31  async.whilst(test, action, callback);
```

async.until

The same as async.whilst, except that the test is reversed.

async.waterfall

Runs an array of functions in series, each passing their results to the next in the array. Stops on the first error.

Here we're using our squaring server to calculate 5^4:

```
1   var async = require('async');
2   var request = require('request');
3
4   var functions = [
5
6     function(done) {
7       request(
8         { uri: 'http://localhost:4001'
9           , body: '5' }
10        , done);
11    },
12
13    function(res, body, done) {
14      request(
15          { uri: 'http://localhost:4001'
16          , body: body}
17        , done);
18    }
19
20  ];
21
22  function callback(err, res, result) {
23    if (err) { throw err; }
24    console.log('5^4 = 5^2^2 = %s', result);
25  }
26
27  async.waterfall(functions, callback);
```

async.queue

This creates a queue with a determined concurrency. You have to provide a worker function and push work to it. Example:

```
 1  var async = require('async');
 2  var request = require('request');
 3
 4  function worker(number, done) {
 5    request({uri: 'http://localhost:4001/', body: number.toString()}, functio\
 6  n(err, res, body) {
 7      done(err, number, parseInt(body, 10));
 8    });
 9  }
10
11  var queue = async.queue(worker, 5);
12
13  for ( var i = 0; i   queue.push(i, function(err, number, squared) {
14      if (err) {return console.log(err); }
15      console.log('the square of %d is %d', number, squared);
16    });
17  }
18
19  queue.drain = function() {
20    console.log('queue drained!');
21  };
```

Appendix - Exercise Results

Chapter: Buffers

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.

One Solution:

Source in
exercises/buffer/1.js

```
1   var buffer = new Buffer(100);
2
3   for(var i = 0; i < buffer.length; i ++) {
4     buffer[i] = i;
5   }
6   console.log(buffer);
```

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.

One Solution:

Source in
exercises/buffer/2.js

```
1   var buffer = new Buffer(100);
2
3   for(var i = 0; i < buffer.length; i ++) {
4     buffer[i] = i;
5   }
6   console.log(buffer);
7
8   var buffer2 = buffer.slice(40, 60);
9
10  console.log(buffer2);
```

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.

One Solution:

Source in
exercises/buffer/3.js

```
1   var buffer = new Buffer(100);
2
3   for(var i = 0; i < buffer.length; i ++) {
4     buffer[i] = i;
5   }
6   console.log(buffer);
7
8   var buffer2 = new Buffer(20);
9   buffer.copy(buffer2, 0, 40, 60);
10
11  console.log(buffer2);
```

Chapter: Event Emitter

Exercise 1

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

One Solution:

Source in
exercises/event_emitter/1.js

```
1   var util        = require('util'),
2       EventEmitter = require('events').EventEmitter;
3
4   var Ticker = function() {
5     var self = this;
6     setInterval(function() {
7       self.emit('tick');
8     }, 1000);
9   };
```

Exercise 2

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

One Solution:

Source in
exercises/event_emitter/2.js

```
1   var util        = require('util'),
2       EventEmitter = require('events').EventEmitter;
3
4   var Ticker = function() {
5     var self = this;
6     setInterval(function() {
7       self.emit('tick');
8     }, 1000);
9   };
10
11  util.inherits(Ticker, EventEmitter)
12
13  var ticker = new Ticker();
14  ticker.on('tick', function() {
15    console.log('TICK');
16  });
```

Chapter: Low-level File System

Exercise 1 - get the size of a file

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

One Solution:

Source code in
exercises/fs/1.1.js

```
1  var fs = require('fs');
2
3  fs.stat(__dirname + '/a.txt', function(err, stats) {
4    if (err) { throw err; }
5    console.log(stats.size);
6  });
```

Exercise 2 - read a chunk from a file

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

One Solution:

Source code in
exercises/fs/1.2.js

```
1  var fs = require('fs');
2
3  fs.open(__dirname + '/a.txt', 'r', function(err, fd) {
4    if (err) { throw err; }
5    var buffer = Buffer(5);
6    var readBytes = 0;
7    (function readIt() {
8      fs.read(fd, buffer, readBytes, buffer.length - readBytes, 10 + readByte\
9  s, function(err, bytesRead) {
10       if (err) { throw err; }
11       readBytes += bytesRead;
12       if (readBytes === buffer.length) {
13         console.log(buffer);
14       } else {
15         readIt();
16       }
17     });
18   })();
19  });
```

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.

One Solution:

Source code in
exercises/fs/1.3.js

```
1   var fs = require('fs');
2
3   fs.open(__dirname + '/a.txt', 'r', function(err, fd) {
4     if (err) { throw err; }
5     function readSome(startingAt, byteCount, callback) {
6       var buffer = Buffer(byteCount);
7       var readBytes = 0;
8       (function readIt() {
9         fs.read(fd, buffer, readBytes, buffer.length - readBytes, startingAt \
10  + readBytes, function(err, bytesRead) {
11          if (err) { throw err; }
12          readBytes += bytesRead;
13          if (readBytes === buffer.length) {
14            callback(buffer)
15          } else {
16            readIt();
17          }
18        });
19      })();
20    }
21    readSome(5, 4, function(buffer1) {
22      console.log(buffer1);
23      readSome(10, 4, function(buffer2) {
24        console.log(buffer2);
25      });
26    })
27  });
```

Exercise 4 - Overwrite a file

Having a file named a.txt, Overwrite it with the UTF-8-encoded string "ABCDEFGHIJLKLMNOPQRSTU-VXYZ0123456789abcdefghijklmnopqrstuvxyz".

One Solution:

Source code in
exercises/fs/1.4.js

```
1   var fs = require('fs');
2
3   fs.open(__dirname + '/a.txt', 'w', function(err, fd) {
4     if (err) { throw err; }
5     var buffer = new Buffer('ABCDEFGHIJLKLMNOPQRSTUVXYZ0123456789abcdefghijkl\
6   mnopqrstuvxyz');
7     var written = 0;
8     (function writeIt() {
9       fs.write(fd, buffer, 0 + written, buffer.length - written, 0 + written,\
10  function(err, bytesWritten) {
11        if (err) { throw err; }
12        written += bytesWritten;
13        if (written === buffer.length) {
14          console.log('done');
15        } else {
16          writeIt();
17        }
18      });
19    })();
20  });
```

Exercise 5 - append to a file

Having a file named a.txt, append UTF-8-encoded string "abc" to file a.txt.

One Solution:

Source code in
exercises/fs/1.5.js

```
1   var fs = require('fs');
2
3   fs.open(__dirname + '/a.txt', 'a', function(err, fd) {
4     if (err) { throw err; }
5     var buffer = new Buffer('abc');
6     var written = 0;
7     (function writeIt() {
8       fs.write(fd, buffer, 0 + written, buffer.length - written, null, functi\
9   on(err, bytesWritten) {
10        if (err) { throw err; }
11        written += bytesWritten;
12        if (written === buffer.length) {
13          console.log('done');
14        } else {
15          writeIt();
16        }
17      });
18    })();
19  });
```

Exercise 6 - change the content of a file

Having a file named a.txt, change byte at pos 10 to UTF-8 value of "7".

One Solution:

Source code in
exercises/fs/1.6.js

```
1   var fs = require('fs');
2
3   fs.open(__dirname + '/a.txt', 'a', function(err, fd) {
4     if (err) { throw err; }
5     var buffer = new Buffer('7');
6     var written = 0;
7     (function writeIt() {
8       fs.write(fd, buffer, 0 + written, buffer.length - written, 10, function\
9   (err, bytesWritten) {
10        if (err) { throw err; }
11        written += bytesWritten;
```

```
12        if (written === buffer.length) {
13          console.log('done');
14        } else {
15          writeIt();
16        }
17      });
18    })();
19  });
```

Chapter: HTTP

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

One Solution:

Source code in
exercises/http/exercise_1.js

```
1  var path = require('path'),
2      fs   = require('fs');
3
4  require('http').createServer(function(req, res) {
5    var file = path.normalize(req.url);
6    path.exists(file, function(exists) {
7      if (exists) {
8        fs.stat(file, function(err, stat) {
9          var rs;
10
11          if (err) { throw err; }
12
13          if (stat.isDirectory()) {
14            res.writeHead(403);
15            res.end('Forbidden');
16          } else {
17            rs = fs.createReadStream(file);
18            res.writeHead(200);
19            rs.pipe(res);
```

```
20           }
21         });
22       } else {
23         res.writeHead(404);
24         res.end('Not found');
25       }
26     })
27   }).listen(4000);
```

Exercise 2

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

One Solution:

Source code in
exercises/http/exercise_2.js

```
1   require('http').createServer(function(req, res) {
2
3     res.writeHead(200, {'Content-Type': 'text/plain'});
4     var left = 10;
5     var interval = setInterval(function() {
6       for(var i = 0; i      res.write(Date.now() + "\n");
7       }
8
9       if (-- left === 0) {
10        clearInterval(interval);
11        res.end();
12      }
13
14    }, 1000);
15
16  }).listen(4001);
```

Exercise 3

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

One Solution:

Source code in
exercises/http/exercise_3.js

```
1   var fs = require('fs');
2
3   var sequence = 0;
4   require('http').createServer(function(req, res) {
5     var fileName = '/tmp/' + sequence + '.bin';
6     console.log("writing " + fileName);
7     var writeStream  = fs.createWriteStream(fileName);
8
9     req.pipe(writeStream);
10    req.on('end', function() {
11      res.writeHead(200);
12      res.end();
13    });
14    sequence ++;
15  }).listen(3000);
```

Here we are creating a write stream (line number 7) every time there is a new request. Each file will have a sequential file name *n.bin* (*1.bin*, *2.bin*, etc.), saved to */tmp*.

After creating the write stream we pipe the request data into it (line 9). From line 10 to line 13 we are responding after the request is done.

You can test this by using curl from the command line and piping in a file like this:

```
1   $ curl http://localhost:3000 -T /var/log/mail.log
```

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.

One Solution:

Source code in
exercises/http/exercise_4.js

```
 1  var http = require('http'),
 2      fs   = require('fs');
 3
 4  if (process.argv.length < 5) {
 5    console.log('Usage: ' + process.argv[0] + ' ' + process.argv[1] + '    ');
 6    return;
 7  }
 8
 9  var options = {
10    host: process.argv[2],
11    port: parseInt(process.argv[3], 10),
12    path: '/',
13    method: 'PUT'
14  };
15
16  var req = http.request(options);
17
18  console.log('piping ' + process.argv[4]);
19  fs.createReadStream(process.argv[4]).pipe(req);
```

To test this you can try:

```
 1  $ node exercise_4.js localhost 3000 /var/log/mail.log
```

Here we are initializing an HTTP put request on line 16, with the host name and ports passed in as command line. Then, on line 19 we are creating a read stream from the file name and piping it to the request object. When the file read stream is finished it will call *end()* on the *request* object.

Chapter: Child processes

Exercise 1

Create a server that a) opens a file b) listens on a unix domain socket and c) spawns a client. This client opens the socket to the server and waits for a file descriptor. The server then passes in the file descriptor we opened on a). The client writes to the file and quits. When the client process quits, the server quits.

One Solution:

The server code:

Source code in
exercises/child_processes/exercise_1/server.js

```
1  var spawn = require('child_process').spawn;
2
3  require('fs').open(__dirname + '/example.txt', 'a', function(err, fileDesc)\
4   {
5
6    var server = require('net').createServer(function(socket) {
7
8      socket.write('Here you go', fileDesc);
9      socket.end();
10     server.close();
11   });
12
13   server.listen('/tmp/ho_child_exercise_1.sock', function() {
14
15     var child = spawn(process.argv[0], [__dirname + '/client.js']);
16     child.on('exit', function() {
17       console.log('child exited');
18     });
19
20   });
21 });
```

First we open the file on line 3. Once it is opened, we create the server (line 5) and bind it to a well-known socket path (line 12). When we start listening, we spawn the child process (line 14), which is a node process executing 'child.js' from the current directory. This server will simply write the file descriptor into the first connecting client, end the connection and close the server socket.

The client code:

Source code in
exercises/child_processes/exercise_1/client.js

```
1  var fs = require('fs');
2
3  var conn = require('net').createConnection('/tmp/ho_child_exercise_1.sock')\
4  ;
5  conn.on('fd', function(fileDesc) {
6    fs.write(fileDesc, "this is the child!\n", function() {
7      conn.end();
8    });
9  });
```

The client is very simple: it will connect to the server and wait for a file descriptor to be handed down (line 4). When that happens, it will append the string "this is the child!" into the file and end the server connection, exiting the process.

Chapter: Streaming HTTP Chunked responses

Exercise 1

Create a mixed TCP and HTTP server that, for every HTTP request, streams all the TCP clients input into the request response.

One Solution:

```
 1  var util = require('util'),
 2      EventEmitter = require('events').EventEmitter;
 3
 4  var Hose = function() {
 5    var self = this;
 6    require('net').createServer(function(socket) {
 7      socket.on('data', function(data) {
 8        self.emit('data', data);
 9      })
10    }).listen(4001);
11  };
12
13  util.inherits(Hose, EventEmitter);
14
15  var hoser = new Hose();
16
17  require('http').createServer(function(req, res) {
18    res.writeHead(200, {'Content-Type': 'text/plain'});
19    hoser.on('data', function(data) {
20      res.write(data);
21    });
22
23  }).listen(4002);
```

Here we are creating a pseudo-class named "Hose" that inherits from *EventEmitter*. An instance of this class (*hoser*), when created, starts the TCP server, and on every message it emits a "data" event.

Then, the HTTP server simply binds to that event on the hoser, and when the hoser emits it, that data is piped into the response.

Chapter: UDP

Exercise 1

Create a UDP server that echoes the messages ot receives back into the origin socket.

One Solution:

```
1  var dgram = require('dgram');
2
3  var socket = dgram.createSocket('udp4', function(message, rinfo) {
4    console.log(rinfo);
5    socket.send(message, 0, message.length, rinfo.port, rinfo.address);
6  });
7
8  socket.bind(4001);
```

You can test this using netcat like this:

```
1  $ netcat -u -w 1 localhost 4001
```

and then you can type a message and hit Return, and you should get the same message back.

Chapter: TCP

Exercise 1

Make a chat server that requires no authentication, just a TCP client connection. Each time the client sends some text, the server broadcasts it to the other clients.

One Solution:

Source code in
exercises/http/exercise_1.js

```
1  var sockets = [];
2
3  require('net').createServer(function(socket) {
4
5    sockets.push(socket);
6
7    socket.on('data', function(data) {
8      sockets.forEach(function(socket) {
9        socket.write(data);
10     });
11   });
12
13   socket.on('end', function() {
```

```
14        var pos = sockets.indexOf(socket);
15        if (pos > 0) {
16          sockets.splice(pos, 1);
17        }
18    });
19
20  }).listen(4001);
```

On line 5 we are adding every new connection to the *sockets* array. On line 8 and 9 we are broadcasting every message received to every client connection. On lines 14-17 we are removing the client socket from the *sockets* array if he disconnects.

Exercise 2

Make a chat client that accepts 2 command line arguments: host and port, and reads from *stdin*, sending data to the server on each new line.

One Solution:

Source code in
exercises/http/exercise_2.js

```
1   var net = require('net');
2
3   if (process.argv.length < 4) {
4    console.log('Usage: ' + process.argv[0] + ' ' + process.argv[1] + '  ');
5    return;
6   }
7
8   var host = process.argv[2],
9     port = process.argv[3];
10
11  var conn = net.createConnection(port, host);
12
13  process.stdin.resume();
14  process.stdin.pipe(conn);
15  conn.pipe(process.stdout, {end: false});
```

Here we are opening a connection to the chat server on line 11. Then, we pipe the process *stdin* into the socket (line 13). Also, to print what we get from the server, we pipe that socket into the process *stdout* (line 15).

Chapter: SSL / TLS

Exercise 1

Create a certificate authority. Create a client certificate signed by this new certificate authority.

One Solution:

Create the Certificate Authority (CA):

```
1  $ mkdir private
2  $ openssl req -new -x509 -days 3650 -extensions v3_ca -keyout private/cakey\
3  .pem -out cacert.pem
```

Here is an example output:

```
1  Generating a 1024 bit RSA private key
2  ...............................................++++++
3  ..............++++++
4  writing new private key to 'private/cakey.pem'
5  Enter PEM pass phrase:
6  Verifying - Enter PEM pass phrase:
7  -----
8  You are about to be asked to enter information that will be incorporated
9  into your certificate request.
10 What you are about to enter is what is called a Distinguished Name or a DN.
11 There are quite a few fields but you can leave some blank
12 For some fields there will be a default value,
13 If you enter '.', the field will be left blank.
14 -----
15 Country Name (2 letter code) [AU]:PT
16 State or Province Name (full name) [Some-State]:
17 Locality Name (eg, city) []:Lisbon
18 Organization Name (eg, company) [Internet Widgits Pty Ltd]:Test CA
19 Organizational Unit Name (eg, section) []:
20 Common Name (eg, YOUR name) []:Pedro Teixeira
21 Email Address []:pedro.teixeira@gmail.com
```

The CA should now be generated:

```
1  $ tree
2  .
3  -- cacert.pem
4  `-- private
5      `-- cakey.pem
```

Now we create the client private and public keys:

```
1  $ mkdir client1
2
3  $ cd client1
4
5  $ openssl genrsa -out client.pem 1024
```

Now we generate a Certificate Signing Request (CSR):

```
1  openssl req -new -key client.pem -out client_csr.pem
```

The output should be something like this:

```
1  You are about to be asked to enter information that will be incorporated
2  into your certificate request.
3  What you are about to enter is what is called a Distinguished Name or a DN.
4  There are quite a few fields but you can leave some blank
5  For some fields there will be a default value,
6  If you enter '.', the field will be left blank.
7  -----
8  Country Name (2 letter code) [AU]:PT
9  State or Province Name (full name) [Some-State]:
10 Locality Name (eg, city) []:Lisbon
11 Organization Name (eg, company) [Internet Widgits Pty Ltd]:Test Client
12 Organizational Unit Name (eg, section) []:
13 Common Name (eg, YOUR name) []:Pedro Teixeira
14 Email Address []:pedro.teixeira@gmail.com
15
16 Please enter the following 'extra' attributes
17 to be sent with your certificate request
18 A challenge password []:
19 An optional company name []:
```

Now we create a signed certificate:

```
1  $ openssl x509 -req -in client_csr.pem -signkey ../private/cakey.pem -out c\
2  lient_cert.pem
```

You will be prompted for the CA private key password, and he output will look something like this:

```
1  Signature ok
2  subject=/C=PT/ST=Some-State/L=Lisbon/O=Test Client/CN=Pedro Teixeira/emailA\
3  ddress=pedro.teixeira@gmail.com
4  Getting Private key
5  Enter pass phrase for ../private/cakey.pem:
```

Now you should have the client certificate in the file named *client_cert.pem*.

Exercise 2

Create a TLS echo server that uses the default certificate authorities.

One Solution:

Create a directory to store the keys named *exercise2*:

```
1  $ mkdir exercice2
2
3  $ cd exercise 2
```

Create the server private key:

```
1  $ openssl genrsa -out server-key.pem 1024
```

Create the server certificate sigining request (CSR):

```
1  $ openssl req -new -key server-key.pem -out server-csr.pem
```

Create a self-signed sertificate:

```
1  $ openssl x509 -req -in server-csr.pem -signkey server-key.pem -out server-\
2  cert.pem
```

Now go down a dir and type the server script:

```
1   $ cd ..
```

Source code in
exercise/ssl_tls/exercise_2.js

```
1   var fs = require('fs');
2   var options = {
3     key: fs.readFileSync(__dirname + '/exercise_2/server-key.pem'),
4     cert: fs.readFileSync(__dirname + '/exercise_2/server-cert.pem'),
5     ca: fs.readFileSync(__dirname + '/exercise_1/private/cakey.pem')
6   };
7   require('tls').createServer(options, function(socket) {
8     socket.pipe(socket);
9   }).listen(4001);
```

Exercise 3

Create a TLS client that reads from stdin and sends it to the echo TLS server created on exercise 2.

One Solution:

Source code in
exercise/ssl_tls/exercise_3.js

```
1   var fs = require('fs');
2   var client = require('tls').connect(4001, function(err) {
3     client.connected = true;
4     console.log('connected');
5     process.stdin.resume();
6     process.stdin.pipe(client);
7     client.pipe(process.stdout, {end: false});
8   });
```

Exercise 4

Make the TLS server only accept connections if the client is certified. Verify that he does not let the client created on exercise 3 connect.

One Solution:

Source code in
exercise/ssl_tls/exercise_4.js

```
1   var fs =require('fs');
2   var options = {
3     key: fs.readFileSync(__dirname + '/exercise_2/server-key.pem'),
4     cert: fs.readFileSync(__dirname + '/exercise_2/server-cert.pem'),
5     ca: fs.readFileSync(__dirname + '/exercise_1/private/cakey.pem'),
6     requestCert: true,
7     rejectUnauthorized: true
8   };
9   require('tls').createServer(options, function(socket) {
10    socket.on('data', function(data) {
11      console.log(data.toString());
12    });
13    socket.pipe(socket);
14  }).listen(4001);
```

Here we are passing in the *requestCert* and *rejectUnauthorized* options with a true value. Once our client connects the connection will be rejected since it is not using a certificate regognizable to the server.

Exercise 5

Make the TLS Server use the same certificate authority you used to sign the client certificate with. Verify that the server now accepts connections from this client.

One Solution:

Source code in
exercise/ssl_tls/exercise_5.js

```
1   var fs = require('fs');
2   var options = {
3     key: fs.readFileSync(__dirname + '/exercise_1/client1/client.pem'),
4     cert: fs.readFileSync(__dirname + '/exercise_1/client1/client_cert.pem')
5   };
6   var client = require('tls').connect(4001, options, function(err) {
7     client.connected = true;
8     console.log('connected');
9     process.stdin.resume();
10    process.stdin.pipe(client);
11    client.pipe(process.stdout, {end: false});
12  });
```

Here we are using the client key and certificate we generated on exercise 1, which should be recognizable by our server since it was signed by our Certificate Authority.