**NODEJS PROGRA EXAMPLES**

**NodeJS is Case Sensitive, Space Sensitive & Syntax Sensitive.**

<http://nicholasjohnson.com/node/course/exercises/node-console/>

The Node console

You can open up a NodeJS console from the command line with the node command, like so:

node

* .help

**.break** Sometimes you get stuck, this gets you out

**.clear**  Alias for .break

**.editor** Enter editor mode

**.exit**  Exit the repl

**.help**  Print this help message

**.load**  Load JS from a file into the REPL session

**.save**  Save all evaluated commands in this REPL session to a file

If all is well you should drop into a console where you can type arbitrary JavaScript commands. You can do maths, create functions, assign variables, everything you can do with JavaScript.

* .editor

var x=10;

var y=20;

var z=x+y;

consle.log (“Result Is :”+z);

Result Is : 30



## Exercise - Node Console

Let's try out the Node console and execute a command. Drop into the node console by typing node. Now try the following:

1. Add 10 + 10.
2. Use console.log('hello world') to output hello world.
3. **Find out how many seconds there are in a year. How many seconds there are in a century**

Use **ctrl+l** to clear the node prompt

## Running a program

You can also run saved programs using the node command. Node files have a .js (JavaScript) suffix, like this: program.js.

We can execute our JavaScript program using node like this:

**node program.js**

console.log("Hello World")

var x=new Date();

console.log(x);

## Running a program

You can also run saved programs using the node command. Node files have a .js (JavaScript) suffix, like this: program.js.

We can execute our JavaScript program using node like this:

node program.js

## Exercise - Executing a program

You can make Node output to the terminal using console.log(). Create a file called app.js and add a line that uses console.log() to tell Node write "Hello World" to the console.

Run your program from the console using the node command.

You should see hello world.

## Further exercise

You can find the current time using:

new Date();

Create a node program that tells you the current time and date.

You can get a date object for a specific date using:

new Date(2022,11,25);

You can subtract one date object from another to get the number of milliseconds between them.

dateOne - dateTwo;

Make a timeTillChristmas program that tells you how many seconds there are until Christmas.

Optionally make it tell you how many days there are.

# **Let's make a server!**

In which we create a simple Node server and ship out a website.

Node is not a web server, but it comes bundled with modules that let you create a web server very easily.

## Why Node makes a jolly good server

Node is single threaded and event driven. This is actually a very good architecture for a web server.

We define callback functions that listen out for events, for example network connections, file system events, API calls, etc. When the event occurs, the callback function is executed.

This means that Node does not need to spawn a new process for each connection. We can have many thousands or even millions of open connections with little impact on performance.

We also don't need to muck about with concurrency issues as much because there are no threads. We get most of the benefits of threads without the tricky bits.

## Let's create a server now.

## // \* Node has Modules

## // \* Require http module

## // \* Create a server with createServer

## // \* Write to the response with writeHead, write

## // \* return with end

## // \* Set the server listening with listen

## // \* Log the request

We can create a server in Node with 6 or 7 lines of code, like so:

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

**http.createServer(function (request, response) {**

**response.writeHead(200);**

**response.write('<h1>Hello Node!!!!</h1>\n');**

**response.end();**

**}).listen(3000);**

**console.log('Server running at http://localhost:3000');**

This is all we need to serve a simple website. Yes, that's it.

Let's break it down a little:

### 1. Require http

First we need to require the http module. This comes built in to Node. We require this module and save the object that the module returns in a variable. We can then use this object to create a server.

var http = require('http');

### 2. Create the server

Now we need to create a server and set it listening.

The createServer function receives a callback function. This is the function that will be called when a network request comes in.

http.createServer(function (request, response) {

// You have access to the request and response objects here.

}).listen(3000);

### 3. Write to the response

The callback function receives two objects, which we typically call request and response (or req and res if you prefer)

Whenever someone hits our website, Our callback function will receive request and response objects. We can write whatever we like to the response head and body, and finally return the result:

response.writeHead(200); // write a 200 OK header

response.write('<h1>Hello Node!</h1>

'); // write to the body

response.end(); // return the response to the user

We call response.end() to return the response. If you forget this your server will never return any content. This is by design as it allows us to create long lasting connections that only return a result when something interesting happens.

## Exercise - Create a server

Create a file containing the code above.

In a console, navigate to the directory in which you have saved your file. Start the server with:

node app.js

Visit [http://localhost:3000](http://localhost:3000/) to ensure you have achieved success.

### Extension

* Modify your server to make it return a string of your choosing.
* Modify your server to make it listen to port 5000.

## Request and Response objects

You will remember from the previous exercise that our callback function receives two objects which we commonly call request and response.

The request object contains information about the request. The response object is where we write our server response.

## Exercise - Log the request

Modify your server. Use console.log() to output the entire request object to the console. Have a look around inside, everything is there.

Have a look through the request object. Now try to console.log just the current URL.

Now modify your code. Try to write the current URL as a string to the response. If I visit <http://localhost:3000/nacktschnecke> I should see a web page containing the word nacktschnecke, ideally wrapped in an h1 tag.

## Exercise - A simple router

Given that we can gain access to the URL, write a simple 2 page website that responds to <http://localhost:3000/nacktschnecke> and <http://localhost:3000/about> and serves content appropriately. Do this using if, else if and else.

If no route matches, have your server return a 404 file not found status code and page.

## Reading

Read up on the NodeJS http object (including the request and response objects) here: <http://nodejs.org/api/http.html>

**FILE SYSTEM**

In which we interact with the file system, and serve a web page from a file.

# **Filing**

We can access the file system using the fs module. This gives us methods to read from and write to a file.

IO operations are by their very nature slow and error prone. The fs module allows us to specify a callback function which will have access to our file. The file will be opened for you, then the callback will be invoked and the file passed in to it. You don't need to worry about opening and closing the file, you just need to write code to talk to the file object which you will receive.

Node is not blocked during IO, because your code is not invoked until the file is ready for writing.

## First require fs

We will need the fs module, so first require it, like this:

var fs = require('fs');

## \_\_dirname

\_\_dirname is a special variable that always points to the current working directory.

If we console.log \_\_dirname it will give us a string showing us our current directory.

## Read a file with fs.readFile().

We can read the file using fs.readFile():

fs.readFile(path, function (error,data) {

// data contains the file contents

// error is null if no error occurred

});

## Exercise - get the current directory

Create a little program that outputs the current working directory.

Extend your Node server so that it can output the value of \_\_dirname.

## Exercise - Read a file

Make a file containing some content. Write a program which can read the file from the current directory and output it to to console.

## Node is Asynchronous with Callback Functions

JavaScript has an unusual way of dealing with potentially slow operations such as reading from the file system, using a callback function. Say we have code like this.

console.log('Before');

fs.readFile(path, function (error,data) {

console.log('\*\* During! \*\*');

});

console.log('After');

If our code was executed synchronously (i.e. in order), we would expect this little program to output something like this:

Before

\*\* During! \*\*

After

If we actually run our code, we find instead we most likely see something like this:

Before

After

\*\* During! \*\*

This is because the callback function is not actually executed until the filesystem has opened the file for reading, which will take more time than it takes to log the word "After". This will be a familiar pattern if you have previously used DOM scripting or JQuery.

## Exercise - Extend your node server so that it can serve out a file

Create a folder called views (there's nothing special about this folder name, it's just a common name for one part of a standard MVC pattern). In this folder make a file called about.html. Add some content to it.

Use fs.readFile to get hold of the about.html file. Remember you can use \_\_dirname to get the current directory. In the fs callback use response.write() to write the file contents to the response. Finally call response.end to send the about content to the user.

Remember callbacks are asynchronous, you don't know how long it will take to start reading the file, and Node will not wait. **This means you will need to call response.end in the fs callback or the response will be returned before the file has been read.** Note this.

Now make it so!

## Further Exercise - Re-implement Apache

Use the path portion of the URL to decide which file to show. You can get this by inspecting the request object.

There are obvious security concerns here. Don't do this on a live server please!!!

## Farther further exercise - 404

If the file doesn't exist, the error object will be non-null. Detect this case and return a 404.

Congratulations! You've implemented a rudimentary Apache! Of course there are limitations here, there's no mime-type handling and security in non existent. In the next section we'll see how we can use express to create static file serving middleware.

## [Node Modules](http://nicholasjohnson.com/node/course/exercises/modules)

In which we learn about Node modules, how they work, and how to make one.

# **Modules**

One of the problems with JavaScript in a browser is the global scope. Any variables you create in the global scope are shared between all scripts. They can easily clash. Also, global variables are generally bad news when it comes to making any sort of sensibly architected system.

Node solves all this with a module system.

## A node module

A node module is a simple JavaScript file that saves an object in a variable called module.exports

For example, we could create a cat module like this

var Cat = {

legs: 4,

head: 2,

ears: 2,

sayHello: function() {

console.log('meow');

}

};

module.exports = cat;

In another file we can now require('./cat'), like so:

var cat = require('./cat');

console.log(cat.legs)

cat.sayHello();

## How require works

The require function simply creates an object called module which has an attribute called exports. It then loads your file and executes it with the module object available to it. Once your code has run it returns module.exports.

There is no magic in Node. This is by design.

## Further Reading

[nodejs.org/api/modules.html](http://nodejs.org/api/modules.html)

## Returning a newable object

If we want to mimic a class, we could might create a newable object using prototypical inheritance.

Our cat.js file might look like this:

var Cat = function() {}

Cat.prototype = {

legs: 4,

head: 2,

ears: 2,

sayHello: function() {

console.log('meow');

}

};

module.exports = Cat;

When we require it using we can treat it like a class, like so:

var Cat = require('./cat');

var tigger = new Cat();

console.log(tigger.legs)

tigger.sayHello();

## Exercise - Create a Date Module

Revisit your code from earlier. Create an ultra-simple date module that can tell you the current date and time.

Create some code that requires this module and have it output the time.

## Further Exercise - Days Till Module

Extend your code to create a daysTill module. It should be able to tell you the number of days till Christmas and the number of days till mothers day. Pick any other significant dates you see fit to include.

Integrate this module into your Node server so you can hit a URL and find the number of days till different dates.

## Harder Exercise - A Router

This is a bit tougher. Only tackle this if you feel able.

Create a router module. It should receive a URL. It should return the path of a static file which you can then serve out, or else return nil.

Your node app can now var router = require('./router');

You can pass the url to the router and get back the file that should be returned.

## Much Harder Exercise - A controller

Create a controller module. It should have methods in it which can write to the response object. Now have your router send messages to your controller. I'll leave the implementation details up to you here.

## [NPM (Node Package Manager)](http://nicholasjohnson.com/node/course/exercises/npm)

In which we learn about Node modules, how they work, and how to make one.

# **NPM (Node Package Manager)**

NPM is a tool that allows us to manage node modules. We can install many thousands of modules from [www.npmjs.com/](https://www.npmjs.com/).

We can install modules in two ways, globally or locally Global modules are available to any project. Local modules are onle available in your current project, allowing you to have different version of modules for different projects.

Local modules are the default, and are what we need most of the time.

## Global install

We can install node modules globally using npm install -g

For example, we might want to install yeoman globally to allow us to run generators in any folder. We could do this with:

npm install -g yo

On my mac this saves the yeoman module in:

/usr/local/lib/node\_modules

## Local install

The default behaviour is to install node modules locally. Say you want to use the express module. You would type:

npm install express

Npm will create a directory called node\_modules in your current directory and save the module in here. Node will look inside this directory first when requiring a module.

## package.json

NPM will optionally read from and write to a file called package.json. This file stores a list of all your dependencies. You can recreate your entire node\_modules directory on a new computer or server using just by typing:

npm install

We check our package.json into version control, then a new team member can install all the correct dependencies with a single command. When we deploy our app to production, our deployment script can run npm install to get all the dependencies it needs.

## Saving dependencies —save

We can save modules to our package.json file. When we install a module use the –save flag, like so:

npm install express --save

This will add the express dependency to the package.json file

## Development dependencies —save-dev

We can also save development dependencies. Say we want to use gulp to automate our build process. We don't want gulp on our live server, just our development box. We can run:

npm install gulp --save-dev

Gulp will be added as a development dependency.

## Installing production dependencies —production

We can install only production dependencies with the –production flag like so:

npm install --production

## Install a dependency

We're going to be writing some Jade templates in the next section, so go on ahead and install Jade as a dependency.

Have a look inside the pacakage.json file which has been created for you. Notice the dependency and specific version number. Notice also some metadata about your project which has been automatically seeded for you.

Have a look in the node\_modules directory. See Jade there?

## Install a development dependency

We're going to be using Gulp in a bit to help us with our code. Install gulp using the –save-dev flag.

Have another look inside the package.json file. See the dependency there listed as a development dependency. This lets us keep development helpers off of the live site.

## [Jade](http://nicholasjohnson.com/node/course/exercises/jade)

Jade is a popular Node templating language. You write less code which makes you more productive. Once you use it, you'll wonder how you ever lived without it. Never close a tag again.

First Install with NPM

To use Jade you must install the package. Do this with NPM.

npm install jade

Elements

The first word on any line is the html element:

h1 Hello there

Compiles to:

<h1>Hello there</h1>

Nesting elements

We can nest elements inside each other using indentation:

article

h1 Hello

p Hey!

Compiles to:

<article>

<h1>Hello</h1>

<p>Hey!</h1>

</article>

Attributes

We can create attributes:

a(href='http://www.lolcats.com/') Lol cats

Compiles to:

<a href="http://www.lolcats.com/">Lol cats</a>

Variables

We can include variables in our template using an equals sign like this if we just have one value:

article

h1= titleContent

or using curly brace interpolation like this if we want to embed content into text:

html

head

title myWebsite.com - #{titleContent}

Compiling Jade

We compile Jade using the jade package. First we install it with npm:

npm install jade

Now we can compile our template. This gives us a template function:

var jade = require('jade');

var template = jade.compile('h1 hello #{name}');

Or from a file:

var template = jade.compileFile('./path/to/template.jade');

Finally we can compile our template into HTML, passing it an object full of include variables, like so:

template({name: "Davie Skitch Mulldoon"});

Exercise - Using Jade

Modify your Node server to use Jade. Instead of outputting a file directly, use Jade to compile a template and return the result.

Layout files

We can create layout files that will wrap our content in a standard wrapper. This saves typing.

Assume an index.jade file like this. Notice that we are declaring a block called content:

extends ../layout

block content

h1= title

article= content

We can create a layout file like this. Note here that we are including a block called content:

html(lang='en')

head

title= title

body

header

h1 My Cool Site

block content

Exercise - Layout

Create a layout file for your application. Have all your views use the same layout.

Partials

We can also include partials in our layout, for example we might split our layout into several smaller parts to make it easier to work on. Smaller files are generally easier to work with than large ones.

A layout file might look more like this:

html(lang='en')

head

include ./partials/meta

include ./partials/css

body

include ./partials/header

block content

include ./partials/footer

include ./partials/javascripts

The includes are always relative to the current directory.

Exercise - Create a partial

Split your layout file into partials. Create a header partial and a footer partial.

Further Exercise - Content module (Model?)

Tackle this if you are ahead of the group. We'll look at MVC more later.

It's good practice to separate our content from the code that renders our content. Create a module that holds your content. We should be able to require the module, then get an object containing the content out of it and pass this to the view.

I'll leave the details up to your discretion.

## [Automation with Gulp](http://nicholasjohnson.com/node/course/exercises/gulp)

Gulp is a tool for automating your workflow, the successor to Grunt.

# **Gulp**

Gulp is a development automation tool. It's the successor to Grunt.

## Gulp streams

Gulp has a concept of streams. A stream is a set of files that can be transformed.

We create a stream using gulp.src, then pipe it through different functions which can transform the stream in a variety of ways. We can optionally pipe our stream back out onto the filesystem at any point using gulp.dest.

We use gulp for:

* Validation
* Compilation
* Concatenation
* Reformatting
* Renaming
* Wrapping content

## Gulp modules

We extend the capabilities of Gulp using modules, which are installed using npm. Here are some useful ones:

* jshint - JavaScript validation
* sass - CSS precompilation
* Uglify - JavaScript minification
* Autoprefixer - Automatically add vendor prefixes to CSS
* Header - Adds a header to the file
* Size - Outputs the size of a minified file

## Validating code

A common requirement is to validate our JavaScript. We can do this with a simple Gulp task:

var gulp = require('gulp'),

jshint = require('gulp-jshint');

gulp.task('assets:js', function () {

return gulp.src(components.js)

.pipe(jshint())

.pipe(jshint.reporter('default'))

}

We execute this task with:

gulp assets:js

## Automatic execution

We can tell gulp to watch our filesystem for changes, and execute a task whenever a file is modified.

gulp.task('watch', function() {

gulp.watch(components.js, ['assets:js']);

Now we might create a default gulp task:

gulp.task('default', [

'watch'

]);

We can now set our gulp task running simply by typing gulp at a command line.

gulp

## Exercise - Validate your code

Set up Gulp to automatically validate the code in your node server. When you save the file, it should give you an error if you have made a mistake.

## Exercise - Validate your gulpfile

Slightly meta this one. Use the gulp-jsonlint package to automatically validate your gulpfile.

# **SASS**

SASS is Syntactically Awesome StyleSheets. It's similar to LESS, and most of what we learn here will also apply to LESS.

Refer to the documentation here:

<http://sass-lang.com/documentation/file.SASS_REFERENCE.html>

## Exercise - Nested Sass

Write nested queries to style the header with a horizontal nav bar, nicely positioned h1, and pretty background colour.

## Exercise - Variables

1. Create a $header\_colour variable. Use it to set the background-color of your header.
2. Create a $font\_size variable (12px) and a $font\_scale variable (1.5).
3. Set the font-size to be the $font\_size variable.
4. Set the h2 font-size to be $font\_size \* $font\_scale
5. Set the h1 font-size to be $font\_size \* $font\_scale \*\* 2
6. Adjust the $font\_size. See how everything updates. Woot!

## Exercise - Mixins

1. Create a transition mixin which applies a css-transition with vendor prefixes.
2. Create a button mixin which sets a width, a height, a background-color, a padding and display:inline-block. Apply it to input type="submit", button, and a class="button"

## [Jasmine](http://nicholasjohnson.com/node/course/exercises/jasmine)

We see how Jasmine can be used to test a Node project.

# **Jasmine**

Jasmine is a unit testing framework for JavaScript. It was originally developed for the browser, but it works equally well in Node using the jasmine-node module

First we must install it globally:

npm install jasmine-node -g

We can now run specs using:

jasmine-node spec

## Creating something to test

Let's return to our daysUntil module. Mine looks something like this:

const MS\_IN\_DAY = 24\*60\*60\*1000;

var day = {

today: function() {

return new Date();

},

year: function() {

return this.today().getFullYear();

},

christmas: function() {

return new Date(this.year(), 11, 25);

},

};

// console.log(day.christmas());

module.exports = {

christmas: function(date) {

var today = date || day.today();

return (day.christmas() - today) / MS\_IN\_DAY;

}

}

Notice how my christmas function receives an optional date parameter. Parametrising our functions like this helps a lot with testing.

## Creating a spec

create a folder called specs (nothing special about this name, it's just traditional)

Now create a file called say daysuntil-spec.js.

Now we simply include the module, and describe its features using Jasmine's easy to use

var daysUntil = require("../days\_until");

describe('daysUntil', function() {

it('Calculate a non-zero number of days', function() {

expect(daysUntil.christmas()).toBeGreaterThan(0);

expect(daysUntil.christmas()).toBeLessThan(366);

});

it('Calculates 10 days from December 15th', function() {

var year = new Date().getFullYear();

var decFifteenth = new Date(year, 11, 15);

expect(daysUntil.christmas(decFifteenth)).toBe(10);

})

})

Read more on Jasmine syntax here: [jasmine.github.io/2.0/introduction.html](http://jasmine.github.io/2.0/introduction.html)

## Integration with Gulp

We can run our Jasmine right from Gulp.

jasmine = require('gulp-jasmine')

gulp.task('jasmin', function () {

return gulp.src('spec/\*.js')

.pipe(jasmine());

});

## Exercise - Test your Days till module

Write a test for each method in your Days till module

## Exercise - Gulp-Jasmine

Extend your Gulpfile to run your tests automatically.

Set a watcher which watches the js files in your project. When they update execute the jasmine task.

## Exercise - Mocha

Mocha is a project very similar to Jasmine, but with better support for asynchronous tasks. Refactor your code to use Mocha

## [Promises](http://nicholasjohnson.com/node/course/exercises/promises)

We see how we can combine the best of asynchronous and synchronous code with Promises.

# **Promises**

A promise is a way of having asyncronous Javascript functions execute synchronously without blocking the execution, and without passing a callback. A call to your method call returns a promise object which has a then method which receives a function. This function will be called when the promise is resolved.

This gives us a nice method chaining syntax.

## Promises in Node

We can get promises in Node using the promise package:

npm install promise

Documentation here: [www.npmjs.com/package/promise](https://www.npmjs.com/package/promise)

## Promises give us an API for creating callbacks

Using callbacks we might write code like this:

var read = fs.readfile;

read(path, function(err, data) {

console.log(data);

});

Our read function receives a path, and also a callback. This is acceptable, but we can do better.

Method chaining give us a nicer syntax.

First we must create the promise. We can convert fs.readFile into a function that will return a Promise like this:

var read = Promise.denodeify(fs.readFile);

read(path).then(function(str) {

console.log('' + str);

});

Rather than a nested callback, which may potentially have further callbacks in it, we use a method chaining syntax. If we return a new promise we can chain these promises together, creating a set of functions which will execute when the previous function completes

## Creating a promise

We can create a promise manually should we need to. A Promise accepts a function. This function will receive resolve or reject. If we call resolve, any functions which are listening for promise resolution will be called. If we call reject, the promise will not resolve and the listeners will not be called.

var Promise = require('promise');

new Promise(function(resolve, reject) {

setTimeout(resolve, 5000)

}).then(function(){

console.log('done');

});

console.log('waiting');

## Multiple listeners

Unlike callbacks, any number of functions can listen to a single promise:

var Promise = require('promise');

promise = new Promise(function(resolve, reject) {

setTimeout(resolve, 5000)

})

promise.then(function() {

console.log('done');

})

promise.then(function() {

console.log('and some');

})

## Chaining promises

Nested callbacks become unmanageable quickly. Instead, we can chain promises together one after another by having our promise callback return a new promise, like so:

var Promise = require('promise');

new Promise(function(resolve, reject) {

setTimeout(resolve, 5000)

}).then(function(){

console.log('done');

return new Promise(function(resolve, reject) {

setTimeout(resolve, 5000)

})

}).then(function(){

console.log('done');

});

console.log('waiting ( this gets called right away');

This is fairly ugly though. We can improve things a with a little syntax.

var Promise = require('promise');

var asyncronousTask = function() {

return new Promise(function(resolve, reject) {

console.log('long lived asynchronous task, perhaps accessing an API');

setTimeout(resolve, 5000);

})

}

asyncronousTask()

.then(asyncronousTask)

.then(asyncronousTask)

.then(asyncronousTask)

## Exercise - Promises with a file

You earlier wrote code which would serve out a file in response to an http request. Modify this code to use a promise.

## Exercise - Request-Promise

Use the [request-promise](https://github.com/tyabonil/request-promise) module. Write a node route that pulls content from an open weather map URL:

<http://api.openweathermap.org/data/2.5/weather?q=London,uk>

When you have got the content (the promise is fulfilled), return the data to the user.

## [Process](http://nicholasjohnson.com/node/course/exercises/process)

Pull information about the current running process.

# **Process**

The process object holds information about the current running process.

If we open up our process object, we will find a slew of useful information in it, notably our environment variables, and passed in parameters.

## Creating a config module

Most applications will require configuration options, things like database connection strings and log level. We likely want to change these values depending on whether we are running in a production or development environment.

env = process.env.NODE\_ENV || 'development';

var config = {

development: {

port: 3000,

db: 'mongodb://localhost/mydb',

prettyHtml: true

},

production: {

port: process.env.PORT || 5000,

db: 'mongodb://mongolab.com/mydb',

prettyHtml: false

}

};

module.exports = config[env];

Now we can simply require('config') wherever we need it, and we'll get back a configuration object that's specific to our environment.

## Exercise - Investigate the process object

Open up a node console. Now grab the process object and have a glance through. Useful.

## Process.argv

The process.argv object contains the current path of the file, and values passed in from the command line.

console.log(process.argv)

If we call the program like this:

node demo.js hello goodbye

We will get something like this:

[

'node',

'/path/to/file/process.js',

'hello',

'goodbye'

]

## Exercise - Configure your node app

Create a configuration object for your Node server

## [Middleware with Express](http://nicholasjohnson.com/node/course/exercises/express)

Express is a Node module that gives us middleware. Execution flows through middleware, and a response drops out the bottom. Let's have a play.

# **Express**

Express is a node module that which provides us with middleware. Middleware is like a pipeline. Requests are passed through each middleware function in turn top to bottom. Each middleware function receives the request and response objects, and can modify them.

Middleware functions can do things like:

* Log a request
* Check authorisation
* Serve a static file
* Inspect the URL, parse out URL parameters, and save them in the request
* Compile a SASS file and serve the result as CSS
* Serve a web page
* Serve an error or 404

We can build an entire site out of middleware.

## Exercise 1 - Install express locally using NPM

We're going to use NPM to manage our packages.

Create a directory to hold your express app. We'll use the NPM generator to make a package file. This will define our dependencies. Run the following:

npm init

Npm init is smart. If you're in a Git repo, it will use the details from that.

Now install the express dependency:

npm install express --save

Check out your directory. You have gained an node\_modules directory that contains express and all it's dependencies. Now check out your package.json file. The express dependency has been saved in it.

You can now run:

npm install

…to install these dependencies on any machine.

You might be interested in the other npm command line methods. You can find them here: <https://www.npmjs.org/doc/>

## A Server

We create a simple server in Express by wiring together middleware functions.

var express = require('express');

var http = require('http');

var app = express();

// create a route

app.get('/', function(req, res){

res.writeHead(200);

res.write('Hello Express!!!');

res.end();

});

// Create a server

http.createServer(app).listen(3000);

## Exercise - Create a server

Use the code above to create a simple Express server.

Add another route. Make your server respond to a request for "/yo".

Change the port. Make your server listen on port 4567

## Custom Middleware

We can tell express to use a piece of middleware using app.use. This function receives 3 parameters, request, response and next.

Next is a function. If you call it, control will pass to the next item of middleware.

A piece of custom middleware might look like this:

app.use(function(req, res, next) {

console.log('hello from the middleware!');

next();

});

## Logging Middleware

Create a custom middleware function that outputs the current URL to the console.

## Static file handling

We can use built in functions for middleware. Here we will use express to serve some static files.

Insert the following line before your routes:

app.use('/public', express.static(\_\_dirname + '/public'));

Now create directory called public. Express will try to serve static files from the public directory, or subdirectories before moving down the middleware stack. You can put JavaScript files in here, CSS files, even JavaScript templates.

Exercise - static files

Add static files to your website. You can now serve out CSS and JavaScript files.

Have your '/' route serve out a complete web page that pulls in a CSS file with Jade and a layout file.

For bonus points, generate your CSS file from SASS using Gulp.

You see how this is all starting to come together?

## Further Reading

<http://evanhahn.com/understanding-express/>

<http://docs.nodejitsu.com/articles/getting-started/what-is-require>

## [REST](http://nicholasjohnson.com/node/course/exercises/rest)

In which we access a RESTful API

# **REST**

REST stands for REpresentational State Transfer. It's a complex term for a simple idea, that you should get a different result if you send a POST request to a URL than if you send a GET request to that same URL.

Typically POST requests are used for sending data to the server, and GET requests for retrieving data. We also have PUT, PATCH and DELETE requests available to us.

## Parsing Post Parameters with Body Parser

Body Parser is a piece of Express middleware that you can use to parse post requests. You add it to your middleware stack and it will automatically parse post data and add it to the the request object.

require it like this:

var bodyParser = require('body-parser');

We can then add the middleware somewhere near the top of our stack, like so:

app.use(bodyParser.json());

app.use(bodyParser.urlencoded());

We compose s form that submits data as a post request like this.

form(url='/myurl' method='post')

input(name="testValue")

button Submit

We will then find that we have a params object in our request which we can read values from.

request.params.testValue

## Exercise - Create a calculator

Create a simple form that can send a POST request to an Express endpoint containing 2 numbers.

Have Express return a simple page that shows the sum of the two numbers.

## A Restful API

We are going to build a proxy onto a RESTful API. We will build a little module that uses http.get and http.post to hit the API and return the result. This is the foundation of a Service Oriented Architecture (SOA).

The following is a simple API spec.

### Pull a list of all articles

GET <http://simple-api.herokuapp.com/api/v1/articles>

### Pull a single article by id

GET <http://simple-api.herokuapp.com/api/v1/articles/1>

### Create an article

POST <http://simple-api.herokuapp.com/api/v1/articles>

with params:

* title
* intro
* content
* author\_id

### Update an article

POST <http://simple-api.herokuapp.com/api/v1/articles/1/update>

with params:

* title
* intro
* content
* author\_id

## GET a list of articles

Create a module that can use http.get to hit the API and pull a list of articles. For bonus points, have it return a promise.

Create a Node URL that hits this endpoint and returns the data.

## CREATE an article

Extend your module so it can POST to the API to create an article. Again, have it return a promise.

Bonus points t othe first one to create an Article.

## [Publishing an API with Mongo](http://nicholasjohnson.com/node/course/exercises/mongo)

Mongo is a JSON based document storage engine. Use it to publish an API

# **Mongo plus Node - perfect together**

To talk to Mongo we use Mongoose. Mongoose gives us a simple wrapper around Mongo with data validation and access to the Mongo commands.

## A schema

A simple Mongoose schema might look something like this:

var mongoose = require('mongoose'),

Schema = mongoose.Schema,

message;

if (mongoose.models.Message) {

message = mongoose.model('Message');

} else {

var MessageSchema = new Schema({

name: String,

email: String,

message: String

});

message = mongoose.model('Message', MessageSchema);

}

module.exports = message;

You would save this in a file called message.js, and require it using require('./message.js')

## Exercise - Create a schema

* Install mongoose with:

npm install mongoose --save

* Create a Person schema for based on the message schema above.
* Import it into your express app using require('./Person')
* Restart express and verify that everything is still working.

## Connecting to the database

To talk to Mongo, we need to connect to our database.

Start the mongo daemon running by typing mongod at the command line:

mongod

Leave mongod running in a shell somewhere.

Now connect by adding something like the following near the start of your express app:

mongoose.connect('mongodb://localhost/database-name');

var db = mongoose.connection;

db.on('error', function () {

throw new Error('unable to connect to database');

});

## Exercise - Connect to Mongo

* Start mongod and connect to mongo.
* Restart your app. Verify all is well.

## Create a route

Finally we need a route which will get all the people.

app.get('/people', function (req, res, next) {

Person.find({}, function(err, people) {

res.json(people);

});

});

## Exercise - Make your route

* Create a route to show all of the people.

Wow, you jut made an API!

## Exercise - Create a show page

Create a route which shows a person with a particular ID. You will need to receive this id using somethig like:

router.get('/people/:id')

You can get hold of the parameter in the route function using:

req.params.id

Your code will look something like this:

app.get('/people/:id', function (req, res, next) {

Product.findOne({\_id: req.params.id}, function(err, product) {

res.json(product);

});

};

## [Angular plus Node](http://nicholasjohnson.com/node/course/exercises/angular)

Angular and Node are perfect together. Node provides the API, and Angular presents the front end.

# **Creating a Single Page App (SPA)**

Create a catch all route which will return index.html no matter what URL is requested.

You will need a router, something like this. You can modify one of your existing routers:

var express = require('express');

var router = express.Router();

/\* GET index. \*/

router.get('/\*', function(req, res) {

res.render('index', { title: 'Marks and Spencer' });

});

module.exports = router;

Now add it to the middleware stack.

You might want also want to create a route to serve template files from, or you could create a templates folder in the public directory and serve them from there.

## Exercise - Tabs

Now let's make some tabs. Remember, everything in the public directory is shipped out as is. We can put static templates here, and Angular will find them. Alternately we can create a middleware route and serve Jade.

Create an Angular router and try to wire together some tabs.

## Further exercise

Using the simple API, create a view to add an article, a view to show an article with comments, and another to show a list of articles. Wire these together with an Angular router. Woot, a real website!

## [Yeoman](http://nicholasjohnson.com/node/course/exercises/yeoman)

Yeoman is a code generator built for Node. Let's use it now to create a better express app.

# **Yo Yeoman**

Yeoman is a code generator. You can use it to bootstrap an entire project, or you can use subgenerators to make individual units of code, perhaps a single controller, JavaScript file or view.

You can read more about Yeoman here: <http://yeoman.io/>

In this section we're going to use the Yeoman Express generator to build an Express project.

## Installing Yo

First install Yeoman. As you would expect, we install it with npm. We're going to use the -g global flag because we want to call yeoman from outside of a project.

npm install -g yo

## Installing a generator

Next we'll need a generator. You can find a list of them here: [yeoman.io/generators/](http://yeoman.io/generators/). Yeoman generators all start with the word generator, for example express-generator or angular-generator.

Create a project directory to hold your code. Now install the express generator. We'll make this local to our project in case we need a sub-generator again later:

mkdir myproject

cd myproject

npm install generator-express

## Build your Express application

Create an app using the express generator. At a command line type:

yo

Choose the Express generator. Accept the defaults. Choose Jade for templates. As a build tool, choose gulp (or grunt if you prefer an old friend). For CSS choose SASS. Optionally choose MVC.

## Using SASS

If you choose SASS you'll probably need the compass Ruby gem. Install Ruby if you don't already have it. For Windows this is a double click install:

<http://rubyinstaller.org/downloads/>

test if from a terminal using:

ruby -v

Now install the gem. From a terminal:

gem install compass

## Starting the server

We start the server using gulp:

gulp

## View in the browser

We can now visit [http://localhost:3000](http://localhost:3000/) to view our app in all it's glory.

## Exercise - Create an Express scaffold with Yeoman

Use yo to generate a simple express site. Fire it up with by running Gulp.

* Try to add another route

## Exercise - Gulp

Gulp is watching your project. Every time you change a view file, your browser will refresh.

Make a change to the SASS file and watch the browser reload automatically. This is a massive productivity gain, especially with multiple screens.

You ay need to install the livereload Chrome plugin.

## Exercise - Modify a view

Look in your views folder. You'll find a file called layout.jade. Jade is an html preprocessor with semantic indentation. You don't need to use angled braces or close your tags, it handles that for you.

Read more on Jade here: <http://jade-lang.com/>

1. Try to create a header and footer in your layout.jade file that will appear on every page.
2. Create an about us page url with content.

## [SocketIO](http://nicholasjohnson.com/node/course/exercises/socket-io)

The jewel in the crown of the Node stack.

# **Socket.io**

Socket.io has been described as the jewel in the crown of the NPM toolbox. It allows you to create an object that is apparently shared by the server and all connected clients. An event fired on a client can be recieved on the server, and vice versa.

Events can contain payloads allowing us to pass messages among clients easily. We can build realtime chat applications, multiplayer games, virtual remote controls, anything requiring realtime communication with a server.

Because Node is event driven rather than multi-threaded, adding a client involves very little overhead. You can connect as many sockets as you like (within reason).

## Exercise - create an Express app

Use the Yeoman generator to create a quick express app.

## Exercise - Installation

Installation is with NPM. Use –save to add the dependencies to your package.js file.

npm install socket.io --save

## Exercise - setup

We need to create an io object. Do it like this:

var http = require('http').Server(app);

var io = require('socket.io')(http);

## Exercise Create a socket on connection

We can now create a socket when a browser connects. We can have as many open sockets as we like. Sockets can listen to and emit events.

io.on('connection', function(socket){

socket.on('chat', function(msg){

io.emit('chat', msg);

});

});

## Exercise - Create the socket clientside

In your index.html file, include the following script tag:

<script src="/socket.io/socket.io.js"></script>

Sockets.io has created this route for you. You don't need to expose this file yoursellf.

Create a socket like this:

var socket = io();

You can emit an event like this:

socket.emit('chat', "hello, 123");

You can listen for an event like this:

socket.on('chat', function(msg){

// Do something here

});

Given our ability to emit events clientside and hear them serverside, and vice versa, create a simple instant chat client.

## Further exercise

We can hit an API from within Node using something like the folowing:

var api = http.createClient(80, 'simple-api.herokuapp.com');

var request = api.request('POST', '/api/v1/comments', { /\* comment object goes here \*/ });

Integrate socket.io into your comments list in your simple-api website. Make it so that when a comment is made, everyone is notified, and the comment is persisted in the api.

## [Scaling Node](http://nicholasjohnson.com/node/course/exercises/scaling)

Node scales nicely. Here are some pointers.

# **Scaling Node**

## Accessing the OS

We can get information about our machine, including the number of cores using the os module.

var os = require('os'); os.cpus();

## Cluster Module

We can use the cluster module to spark up a new server on each CPU. The CPUs will not share state, but will share a port. A simple round robin router will direct traffic between the cores

read more about the cluster module here:

[https://nodejs.org/api/cluster.html](http://nicholasjohnson.com/node/course/exercises/scaling/nodejs.org/api/cluster.html)

var cluster = require("cluster"),

http = require("http"),

os = require("os"),

cpus = os.cpus().length;

console.log(cluster)

if (cluster.isMaster) {

for (var i = 0; i < cpus; i++) {

cluster.fork();

}

cluster.on("exit", function(worker, code, signal) {

cluster.fork();

});

console.log('listening on port 3001')

} else {

http.createServer(function(req, res) {

res.writeHead(200);

res.write("Hello Cluster!!");

res.end

}).listen(3001);

console.log('Making child process')

}

## Exercise - Scaling

Find the number of cores on your machine.

Scale your app across all of your cores.

## Downloads