

CAB230 Web Computing

Getting Started with Node

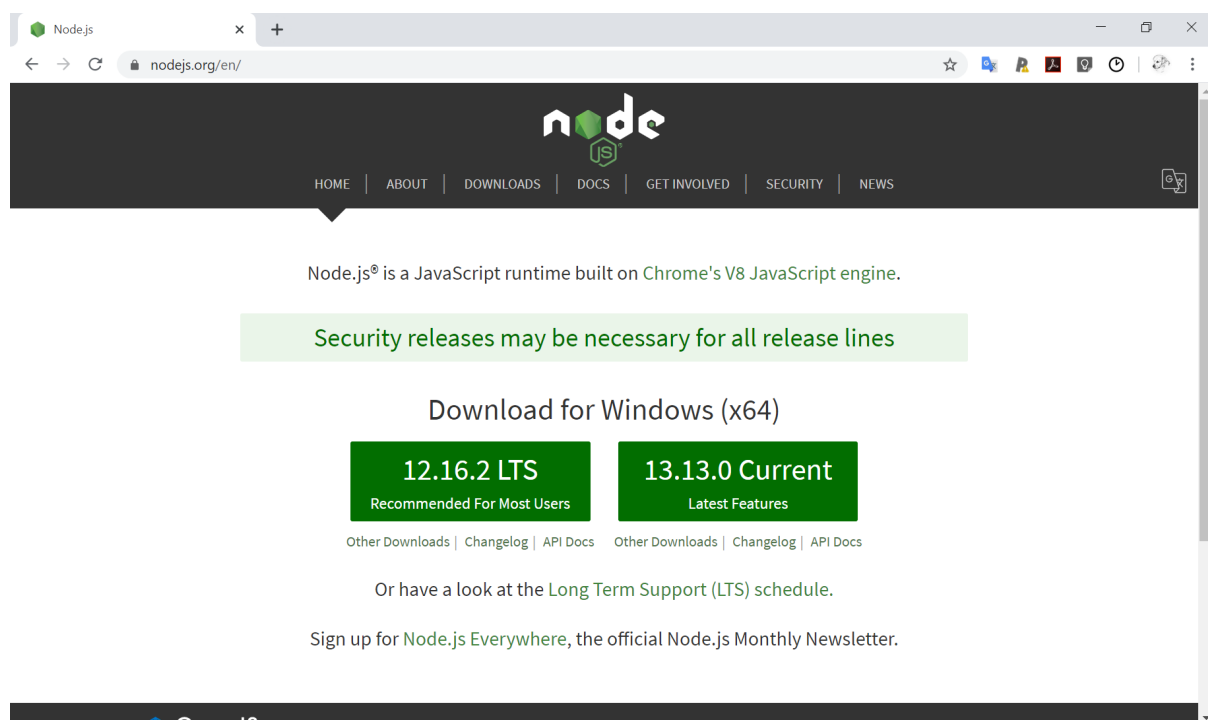
This prac is an extensive introduction to node and express. We will release this prac in week 7, with the expectation that you will have completed it by (say) the end of week 8. There is a mix of material here. We will begin with some discussion of Node, which builds on some of the examples in the lectures. The main goal here is to provide a proper introduction to Node applications and to the package manager npm. Some of you will find this aspect of the prac completely unnecessary and so you should feel free to skip over this material.

The most important aspect of this prac relates to the use of node and express.

Introducing Node and NPM

Getting started with Node:

Node is available for use in the labs, or you can easily set it up on your own machine. Installation instructions are available at the site: <https://nodejs.org/>.



For Windows, use the download button and run the installer, accepting the defaults. There is a link here to “Other Downloads”, and this is probably the best way of dealing with the Mac OS X install. For Linux, I think you are probably better to use your platform’s package manager.

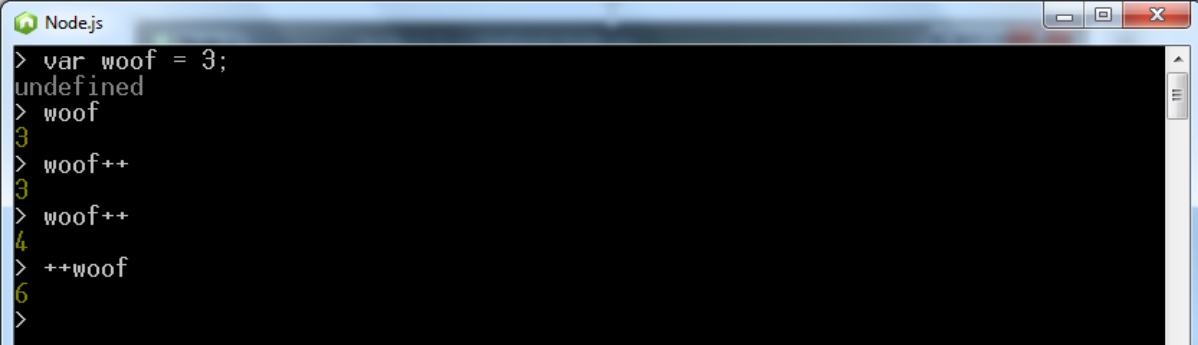
The Node nomenclature is different from the usual – you will see the v12.x.x LTS and the v13.x.x Current. The LTS stands for Long Term Support, and the promise is that v12 will be supported over a defined cycle, with the subsequent release transitioning to be the next LTS.

The features of node required for this unit have not changed much over the past few years and you may use whichever version you like.

Working with Node under Windows is very straightforward. For convenience, I normally work with the bash shell provided by Git or the Windows Subsystem for Linux. I can then use the Node commands at the command line as needed. If you like, you can also keep an interpreter window open to check out some of your basic JS. To do this, select the node.js option with the green icon as shown



As you have seen earlier in the unit, the resulting window allows you to enter fragments of code and to see what might result. This shows (entirely correct) interactions with a variable called `woof`, that we initially define to have a value of 3. We will work with Node as a convenient and flexible server-side technology, but you may also see it as an interactive interpreter to allow you to deal with the basics.

A screenshot of a Node.js REPL window. The window has a title bar with the text "Node.js" and standard Windows window controls. The main area is a black terminal with white text showing the following interactions:

```
> var woof = 3;
undefined
> woof
3
> woof++
3
> woof++
4
> ++woof
6
>
```

Working through some Node examples:

We assume at this point that you have successfully installed node and that you have chosen an editor and made it talk to the Node installation (this happens out of the box for VS Code, and is a pretty simple configuration option for most others). You are now ready to proceed. Choose a Node working directory. From within this directory, download the basic server Node examples provided with this prac, which you will find as usual in the `starter-files.zip` archive, which includes examples from the lectures. The key issue is that none of these examples require the use of packages outside the core. The server files are:

- `simple.js` – the basic HelloWorld example (port 2798)
- `events.js` – the alternative which handles events explicitly (port 2799)
- `logheaders.js` – HelloWorld server, but logging the headers (port 2800)
- `asyncfile.js` – the server which serves `simple.js` on request (port 2801)
- `client.js/server.js` – the tcp connection applications, linked on port 2802.

In each case, you may run the server using the command:

```
node <server-name>
```

where `<server-name>` is replaced by the relevant file name, and you may then connect by opening your browser at the address: <http://localhost:{port}>. Note that we are not here making any use of the application running facilities provided by npm.

In the case of `client.js` and `server.js` this will connect without the need for a browser but the launch is the same. Please work through each of these and confirm that you are able to make sense of what is happening in each case.

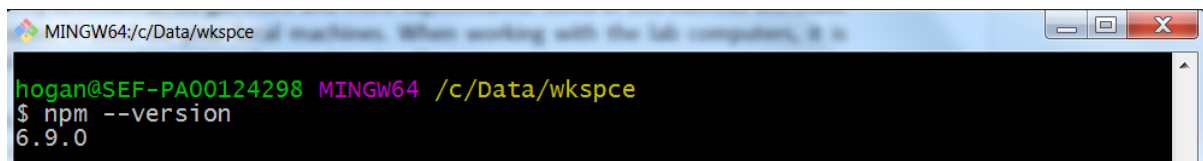
Introducing NPM and more sophisticated Node:

The Node Package Manager NPM allows us to manage the module dependencies for a Node application. This is an essential part of non-trivial Node development as otherwise we would do little else but manage package dependencies. In what follows we will step through some basic NPM and use it in the context of some simple Node applications before leaving it for a while, only to return as we get more and more sophisticated. Some of this exercise will make more sense if used on your local machines. When working with the lab computers, it is probably more sensible to install everything locally.

If you have sufficient privileges, the first step is to ensure that npm – installed with Node – is in fact up to date. To do this, run:

```
npm install npm -g
```

This shows the NPM command `install`, here applied to the NPM package itself. The `-g` flag indicates that the package should be *global* and hence available to all your Node applications. The package manager itself is located as part of the Node installation and so is available on the application path. After the update, you should see a version something like this: TM

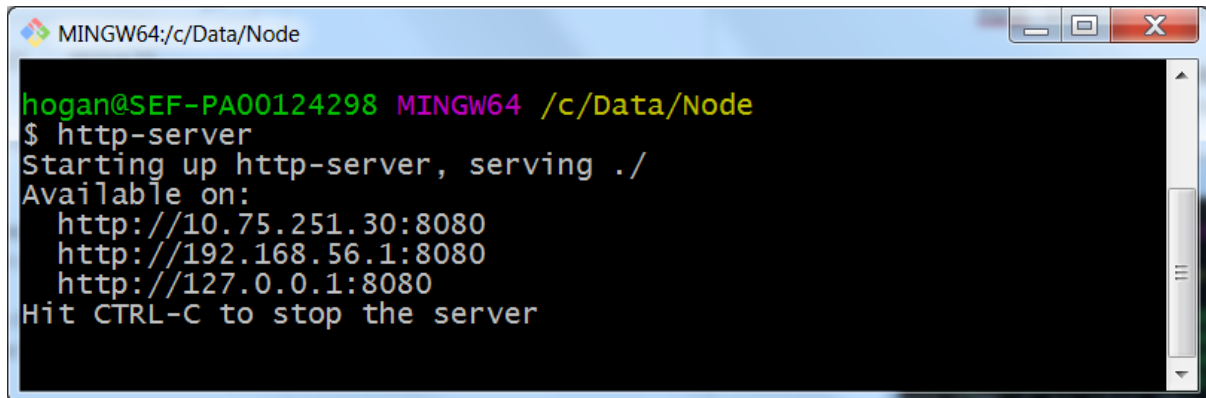
A screenshot of a terminal window titled 'MINGW64/c:/Data/wkspce'. The prompt is 'hogan@SEF-PA00124298 MINGW64 /c:/Data/wkspce'. The command entered is '\$ npm --version' and the output is '6.9.0'.

OPTIONAL: If you want to explore a more sophisticated server, we will now consider a Node http server for static web pages which builds on top of the simple servers we have played with in the lecture and in the earlier exercises. We are not going to do too much with this – this is an exercise in using npm, and in showing you that node too has a basic server that can be used for quick tests – just like jetty and Apache under XAMPP and the basic Python webserver.

We begin by installing the Node http-server package: <https://www.npmjs.com/package/http-server>. The subsequent application may be handled in a number of ways, but we will find it most useful to work with this Node server as an application, in much the same way as we might with Python's simple server. The advantage is that we can then serve pages from the working directory. The easiest approach is to install the package globally, but you can get a local install by leaving out the `-g` flag:

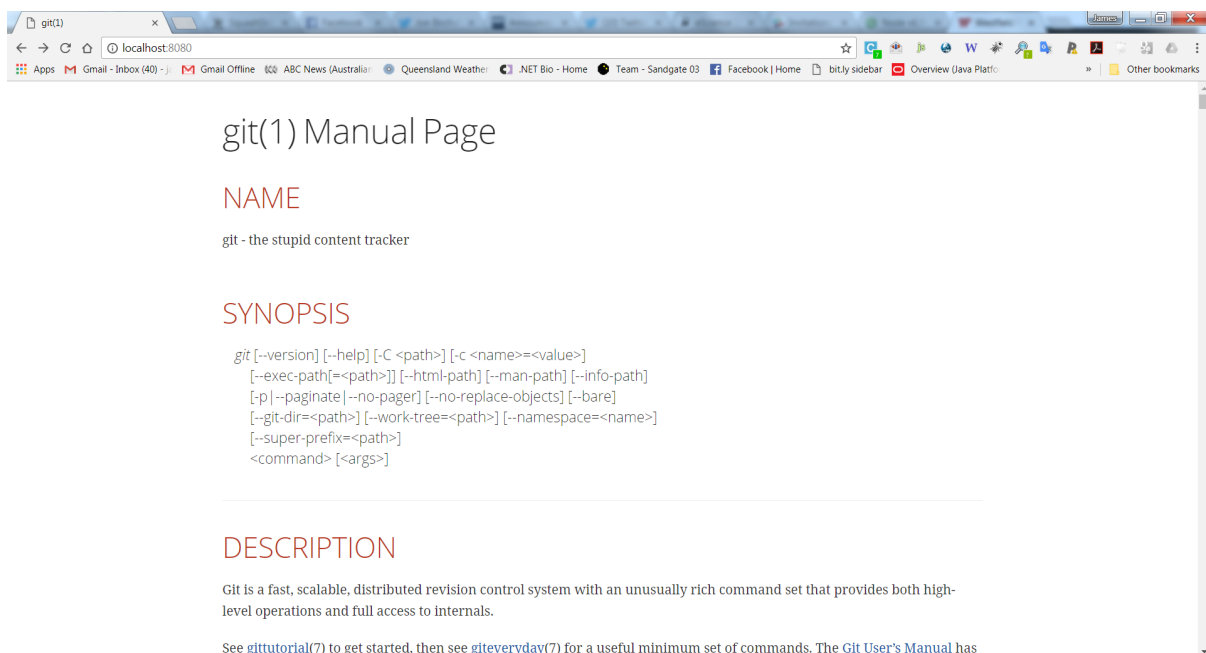
```
npm install http-server -g
```

Once the package is installed, we can start the server straightforwardly as shown. As it is an application, we do not need to use Node to invoke it:



```
MINGW64:/c/Data/Node
hogan@SEF-PA00124298 MINGW64 /c/Data/Node
$ http-server
Starting up http-server, serving ./
Available on:
  http://10.75.251.30:8080
  http://192.168.56.1:8080
  http://127.0.0.1:8080
Hit CTRL-C to stop the server
```

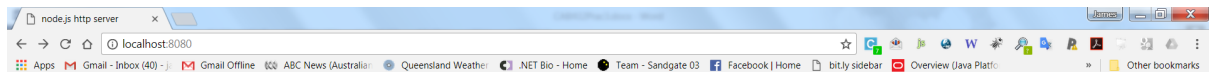
By default, the server will serve from the working directory. In the absence of any sensible content, you will see a simple empty directory listing from the server. In my case, I have grabbed a simple `index.html` file from the git docs (included in the starter files), and the resulting image is as follows:



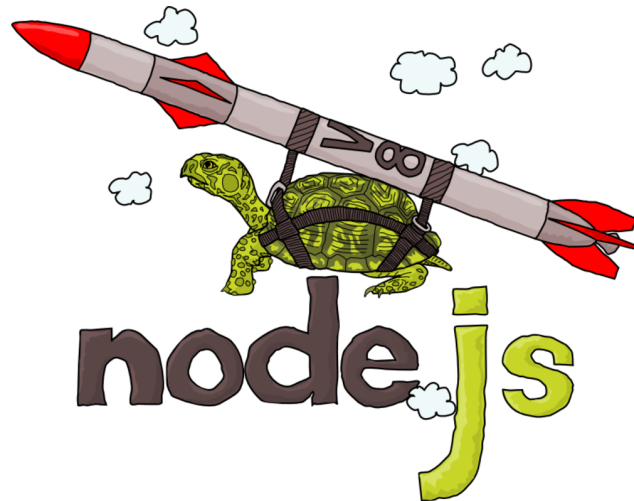
If you are unable to install the package globally or the application does not work as above, install locally (without the `-g` flag) and then create subdirectories and work locally within the `node_modules/http-server` directory. This time you will need to use Node to invoke the server using the command: `node bin/http-server`:

```
MINGW64/c:/Data/wkspce/node_modules/http-server
hogan@SEF-PA00124298 MINGW64 /c:/Data/wkspce/node_modules/http-server
$ node bin/http-server
Starting up http-server, serving ./
Available on:
  http://192.168.0.7:8080
  http://192.168.56.1:8080
  http://127.0.0.1:8080
Hit CTRL-C to stop the server
```

and the pages will be served from `node_modules/http-server`. The default response at <http://localhost:8080> used to be the page shown on the next page. It seems not to ship with the package anymore, so be grateful for some things at least.



Serving up static files like they were turtles strapped to rockets.



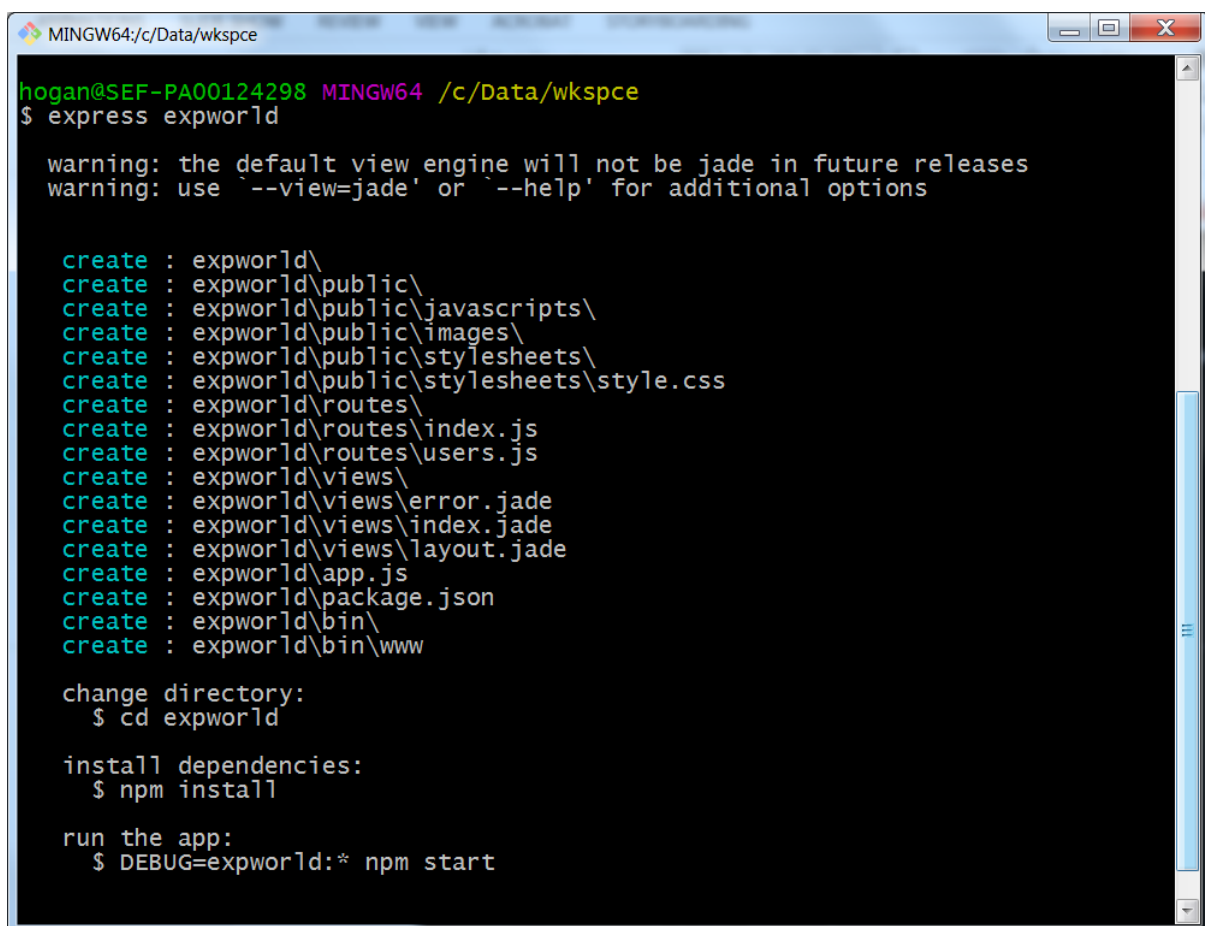
For the rest of this prac, we will spend our time creating a node application using express, and then ultimately exposing an API based on the world database example.

Using Express and MySQL

An Express API for the World database.

This exercise builds heavily on the MySQL example from the lectures – please take a good look at the videos in the folder **Node and MySQL - demos and installation**. Your first task is to follow through this simple example and to make sure that you make it work. Create a local version of the database on the MySQL server on the VM or on your home machine. If you get stuck with this, talk to one of the tutors. Both the `app.js` and `world.sql` files are made available in the `mysqlworld.zip` file. The process of creating a simple Express app is covered in the lectures and there is a detailed walkthrough there. We will gloss over some aspects of that process as we want to focus on the routes.

We begin, though, by creating such an express app. Following the approach in the lectures, we see:



```
MINGW64/c:/Data/wkspce
hogan@SEF-PA00124298 MINGW64 /c/Data/wkspce
$ express expworld

warning: the default view engine will not be jade in future releases
warning: use '--view=jade' or '--help' for additional options

create : expworld\
create : expworld\public\
create : expworld\public\javascripts\
create : expworld\public\images\
create : expworld\public\stylesheets\
create : expworld\public\stylesheets\style.css
create : expworld\routes\
create : expworld\routes\index.js
create : expworld\routes\users.js
create : expworld\views\
create : expworld\views\error.jade
create : expworld\views\index.jade
create : expworld\views\layout.jade
create : expworld\app.js
create : expworld\package.json
create : expworld\bin\
create : expworld\bin\www

change directory:
$ cd expworld

install dependencies:
$ npm install

run the app:
$ DEBUG=expworld:* npm start
```

The command `npm install` will grab all the basic dependencies for the app as it stands. In this case you should also install `mysql` as we did in the lectures.

At the moment, we can see a number of default styles and routes. The application itself is specified in `app.js` and we will take a closer look at this soon. Overall, we need to manage the following functionality:

- Connection to the database
- Management of the server start and stop
- Route handling and query definition

Whether we manage these within a single application file or split them across a number of files depends a lot on the scale and sophistication of the app. In the world database example we are working with something pretty small, but the routes may get pretty complicated. In this first step, we will handle all of the routes simply from a single router. In subsequent work we might expect to package single routes or groups of related routes in their own individual router files.

Open up `app.js` and take a look at some of the route handling. First we see some imports:

```
var indexRouter = require('./routes/index');
var usersRouter = require('./routes/users');
```

Here we will ignore the user routes and focus on index. All of the routes for this app will be handled by a single express Router in that file. Further down in `app.js` we see:

```
app.use('/', indexRouter);
app.use('/users', usersRouter);
```

This means that all valid routes other than `/users` will be redirected to be handled by `indexRouter`. Much of the remaining code will be left unchanged: `app.js` does many things for free – like handling 404 errors – and we don't want to disrupt this. We will return later to add a few important lines of code, but first we will need to take a look at the router itself.

Let us now open up the file `index.js` which we will find in the `routes` subdirectory of the app. Initially this file is very basic, just handling the response for requests to the site root. You will see the `require` statement, which includes `express` so that we can grab the Router functionality. We will see this call in every router JavaScript file, and full applications can – and usually should – have many of them. We will begin more simply, by changing the default message to something more appropriate for our purposes:

```
const express = require('express');

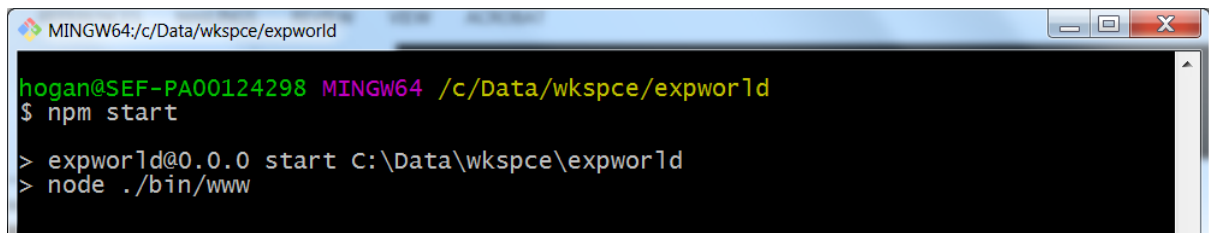
const router = express.Router();

/* GET home page. */
router.get('/', function(req, res, next) {
  res.render('index', { title: 'The World Database API' });
});
```

We will mimic this and add a route for the api itself. Normally we might decide to replace this simple message with some details of the endpoints, but here we shall be a bit lazy:

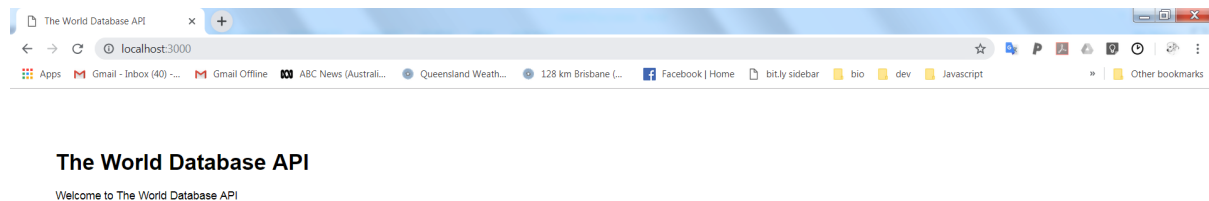
```
router.get('/api', function(req, res, next) {  
  res.render('index', { title: 'Lots of routes available' });  
});
```

We can save the files and we can launch the app from the top directory using `npm start`. We will then see something like this:

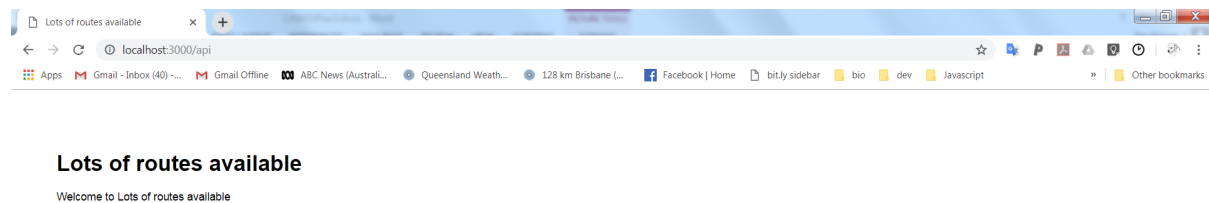


```
MINGW64/c:/Data/wkspce/expworld  
hogan@SEF-PA00124298 MINGW64 /c:/Data/wkspce/expworld  
$ npm start  
  
> expworld@0.0.0 start C:\Data\wkspce\expworld  
> node ./bin/www
```

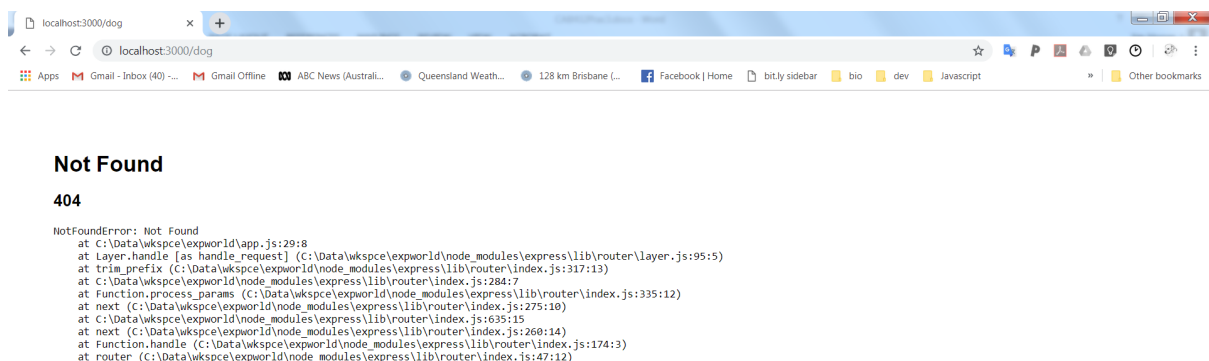
And when we hit the server we will see the following (at the root on port 3000):



At `localhost:3000/api`:



And at an obvious missing page (`localhost:3000/dog`) :



The Database and Middleware:

We now need to think much more about the database connection, but before we do that it is worthwhile considering the way Express looks at middleware. We have seen some rather odd call parameters. We are used to seeing the request and the response by now, but what do we make of this:

```
router.get('/', function(req, res, next) {  
  res.render('index', { title: 'The World Database API' });  
});
```

The key is found in the Express guides, which are really very good indeed. Some of them we have seen in the Week 6 lecture and there will be more in the Week 7 lecture. Here we are going to look at their guide for writing middleware:

<https://expressjs.com/en/guide/writing-middleware.html>

Here we see the reason for the third parameter, `next`. Middleware is best seen as a chain of complementary functions, each with a job to do to make sure that the overall task is successful. A simple route like the one we see here is not specific, and we may intend to send a chain of responses. As it happens, we don't, and the `next` function is not invoked, but this parameter gives us a way of telling express to execute the next task in the middleware chain. In the page we have just linked, there is a very good example based on logging. In reality we would have a more sophisticated log function, but here it is:

```
var express = require('express')  
var app = express()  
  
var myLogger = function (req, res, next) {  
  console.log('LOGGED')  
  next()  
}  
  
app.use(myLogger)  
  
app.get('/', function (req, res) {  
  res.send('Hello World!')  
})  
  
app.listen(3000)
```

Now the important thing is the placement of `app.use` and `app.get`. In this trivial application we do not use a router, but we see that the logger is *available* before the GET. So any time we have a request, we can expect that there will be a log available to us on the console. We might – and would – expand it to include the date and perhaps the method and the request path. But it would still happen each time. And importantly, it is understood that the chain continues, as there is an explicit call to `next()` as we leave the logger.

What we are going to do now is to use these same services to provide a database connection to the router for the API endpoints. In fact we are going to provide it to everything, which is not very good practice but we will keep it simple for now and fix everything next time.

So we are going to create a subdirectory at the top level of the app, and we will call this `database`. Inside this directory we will create a file called `db.js`. We will set up a database connection in much the same way as we did before (you may need to update the DB as before):

```
const mysql = require('mysql');

const connection = mysql.createConnection({
  host      : 'localhost',
  user      : 'root',
  password  : 'xxxx',
  database  : 'world'
});

connection.connect(function(err) {
  if (err) throw err;
});
```

Type this code into the file and make sure that you understand what is going on. Here once more we find the `mysql` module helping us, and I am connecting on localhost (the MySQL database is on the same machine) and I have a pretty lax attitude to security, with the `root` account and some pretty insecure password.

I then go ahead and create a connection. But before I use it, I am going to create a middleware function. At the bottom of this file we write:

```
module.exports = (req, res, next) => {
  req.db = connection;
  next()
}
```

What this means is that we have created a function that can tap into the infrastructure provided by express and node. We have a request and a response object – though we have nothing to call them with in this file – and we also have a `next` parameter. Most importantly, we realise that it would be pretty stupid to finish the chain of operations here, before we've actually done anything sensible. So we call `next()`.

The other important aspect of this function is that we attach a parameter to the request object and then pass this around when it comes time to handle the API routes. The function makes the database connection available to the application. The problem at the moment is that we will establish the connection too often, but for this week, we won't worry about that.

Make a quick return now to the file `app.js`, and take a look at the require statements near the top. We are going to include the database connection:

```
var logger = require('morgan');
const db = require('./database/db');

var indexRouter = require('./routes/index');
var usersRouter = require('./routes/users');
```

Note the addition of the identifier `db`. Further down, we make the connection available:

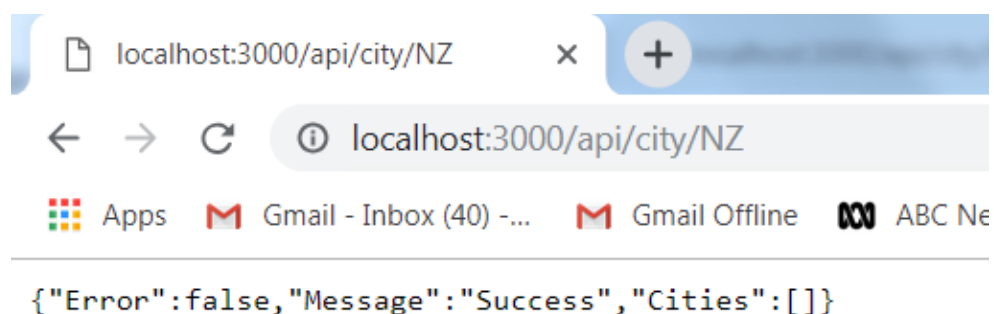
```
app.use(express.static(path.join(__dirname, 'public')));
app.use(db);

app.use('/', indexRouter);
app.use('/users', usersRouter);
```

At this point, we need to consider the API routes themselves. We are going to be generous and give you the SQL you need. You will be able to adapt much of this for use in the assignment. But we now need to look at the routes in the `indexRouter`. We are going to handle the following endpoints:

- `/api/city` – shows the list of all the cities in the database, together with their district (or state or province).
- `/api/city/country-code` – shows much more detail about each city, filtered according to a 3 letter country code.

So in the first case, the call is very simple, and the information returned is extensive. In the second case, we can filter, but we need also to think about some of the three letter country codes. The call <http://localhost:3000/api/city/nz> gives no results from the API as shown:



If we do it properly, using NZL, then the response is as follows:



We will now look at these in more detail. Before we do anything else, we must remember that we can't do much if we don't have the `mysql` object available to us. So at the very top of the file, we need to have the lines:

```
const express = require('express');
const mysql = require('mysql');

const router = express.Router();
```

The `/api/city` Call

The city API call grabs the name and the district associated with each city in the database. [Obvious extensions, for those thinking ahead: modify the API to accept a result limit, or modify the API to cause the results to be sorted in some way]. In the `index.js` file, after the root and the `/api` routes, we insert the following code:

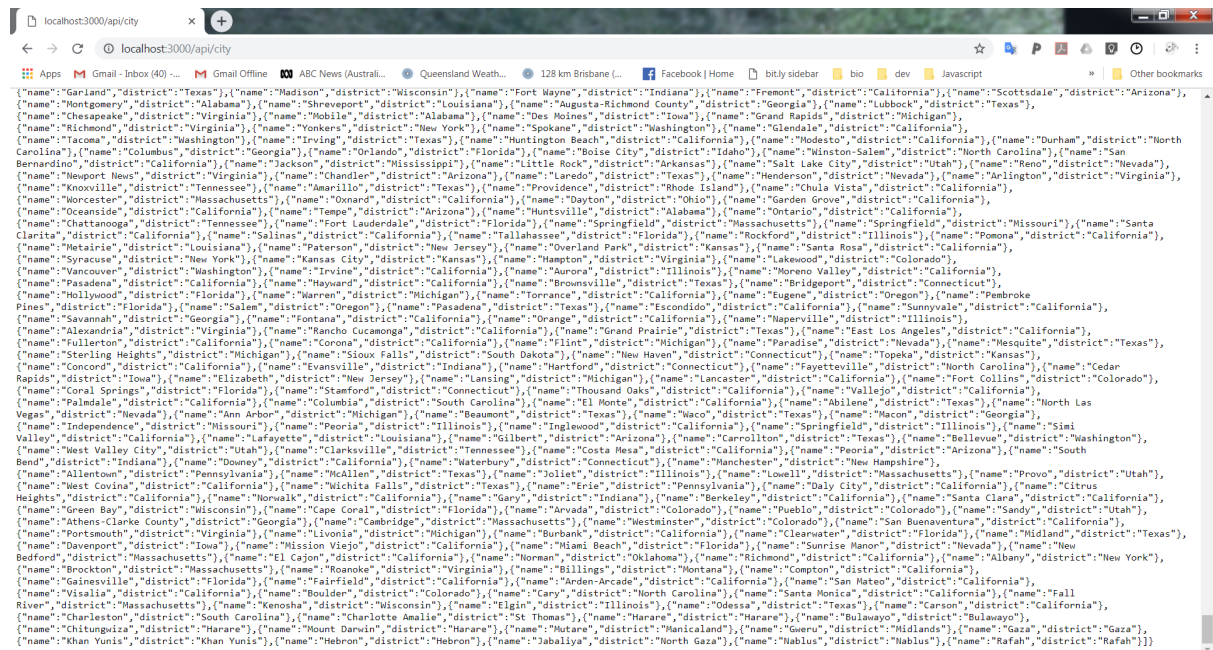
```
router.get("/api/city",function(req,res){
  var query = "SELECT name, district FROM ??";
  var table = ["city"];
  query = mysql.format(query,table);
  req.db.query(query,function(err,rows){
    if(err) {
      res.json({"Error" : true, "Message" : "Error executing MySQL query"});
    } else {
      res.json({"Error" : false, "Message" : "Success", "City" : rows});
    }
  });
});
```

Note the SQL query and the argument specification satisfied by the `table` parameter. The most unusual line is this one:

```
req.db.query(query,function(err,rows){
```

Here we see the effect of the middleware. We have the connection attached to the request object, and here we are able to access its query function without introducing additional parameters or other information passing.

The JSON method allows us to pass JSON back as the result, and we do this if there is an error in the query (not easy in this case) or a successful result. Here is result from the standard call: <http://localhost:3000/api/city>.



The `/api/city` Call

The final component mimics this structure but is more complicated, extending the API to grab all information about the cities from a particular country. The most important difference lies in the use of the URL parameters. The specification: `/city/:CountryCode` means that the `CountryCode` will be treated as a parameter, and made available to us via the `req.params.CountryCode` field. So our example queries might include:

- <http://localhost:3000/api/city/USA>
- <http://localhost:3000/api/city/AUS>

and of course, the example from earlier:

- <http://localhost:3000/api/city/NZL>

and so on. Once again the results can be extensive, but we are unlikely to retrieve some 4000 records as we did previously. The final code fragments are as follows:

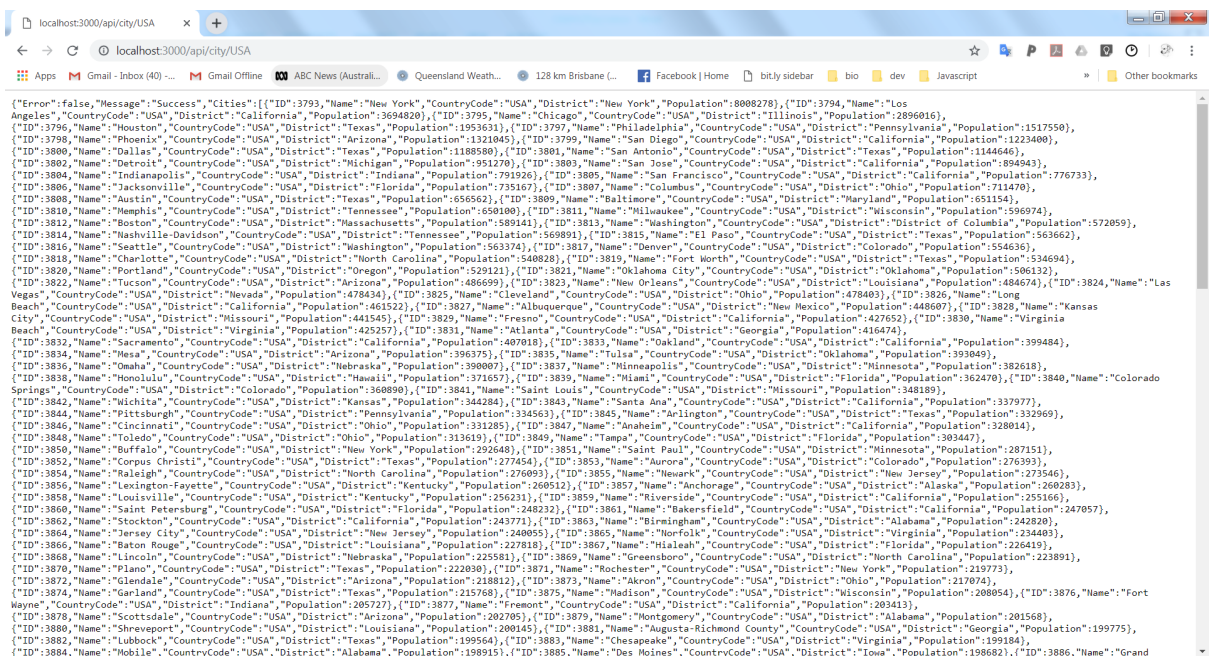
```
router.get("/api/city/:CountryCode",function(req,res){
  var query = "SELECT * FROM ?? WHERE ??=?";
  var table = ["city","CountryCode",req.params.CountryCode];
  query = mysql.format(query,table);
```

The main thing to notice here is the use of the specification of the `CountryCode` parameters through the `/api/city/:CountryCode` syntax, and the subsequent access via the request object parameters. The call proceeds in much the same way as before:

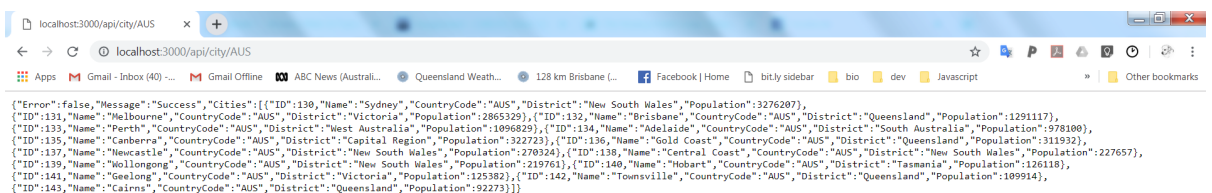
```
req.db.query(query,function(err,rows){
    if(err) {
        res.json({"Error" : true, "Message" : "Error executing MySQL
query"});
    } else {
        res.json({"Error" : false, "Message" : "Success", "Cities" : rows});
    }
});
});
```

Some example calls follow.

For the USA:



And for AUS:



Next time we will fix a few things up, and structure our application just a little better. But for now, you are getting ready to start thinking about the second, server-based assignment.