CS615 Internet Solutions Engineering
NodeJS Tutorial
Part 1: Installation and Testing

Part 1: Install node

You should already have *node* installed on your computer from week 3's lab. If not, install it now: https://nodejs.org/en/

To check that you have *node* installed successfully, go to your command line & type in the command line:

```
node -v
```

Part 2: Install npm

You should also already have had *npm* (the node package manager) installed on your computer from week 3's lab; or if not you will have installed it just now using the link in Part 1 above.

To check that you have *npm* installed successfully, go to your command line & type in the command line:

```
npm -v
```

Part 3: Editor for creating your node code

A list of editors was given in week 1's lab activity sheet.

Part 4: Trying Node for the first time

Create a folder *cs615_nodejs* for all the code you will create in today's lab.

Create a file in your *cs615_nodejs* folder called *app.js*. Type the following in the *app.js* file:

```
console.log('Hello World!');
```

To run this *app.js* code, type the following on your command line:

`node app.js`   (or simply: `node app`)

Hello Node.js

The following example creates a web server that listens for any kind of HTTP request on the URL `http://127.0.0.1:8000/` — when a request is received, the script will respond with the string: "Hello World". If you have already installed node, you can follow these steps to try out the example:

1. Open Terminal (on Windows, open the command line utility)
2. Create the folder where you want to save the program, for example, `test-node` and then enter it by entering the following command into your terminal:

```
cd test-node
```

3. Using your favorite text editor, create a file called `hello.js` and paste the following code into it:

```
// Load HTTP module
const http = require("http");

const hostname = "127.0.0.1";
const port = 8000;

// Create HTTP server
const server = http.createServer(function(req, res) {

   // Set the response HTTP header with HTTP status and Content type
   res.writeHead(200, {'Content-Type': 'text/plain'});

   // Send the response body "Hello World"
   res.end('Hello World\n');
});

```

```
// Prints a log once the server starts listening

server.listen(port, hostname, function() {

   console.log(`Server running at http://${hostname}:${port}/`);

})
```

24. Save the file in the folder you created above.
25. Go back to the terminal and type the following command:

```
node hello.js
```

Finally, navigate to `http://localhost:8000` in your web browser; you should see the text "**Hello World**" in the upper left of an otherwise empty web page.

(Source: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/Introduction)

CS615 Internet Solutions Engineering
NodeJS Tutorial

Part 2: JS / NodeJS basics

Make sure you have the folder cs615_nodejs where you keep all your files that you create in this tutorial.

In this tutorial we will look at some basic JavaScript (that you already learned previously) and some very basic NodeJS features.

Javascript, Documentation and the Global Object

NodeJS has a comprehensive documentation describing the JS commands that NodeJS understands: https://nodejs.org/api/. Let us have a look.

Let us try some standard JS code with NodeJS. Create a new file and call it 1_basic.js. Let us try something from the Global object (see API link above):

```
setTimeout(function(){
  console.log('3 seconds have passed');
}, 3000);
```

What does this code do? Find the documentation of this function in the API. Next try:

```
var time = 0;
var timer = setInterval(function(){
  time += 2;
  console.log(time + ' seconds have passed!')
  if (time > 5){              // add this code later
    clearInterval(timer);     // (i.e. run code first without the lines
  }                           // in blue font to see what happens)
},
2000);
```

Check the functions we have been using with the documentation in the API and find out what they do. Also see:

https://www.w3schools.com/jsref/met_win_settimeout.asp

https://www.w3schools.com/jsref/met_win_setinterval.asp

The Global object has more on offer. Try for example the following two lines below by appending it to the file.

```
console.log(__dirname);
console.log(__filename);
```

What does this code display? Can you find more details about these methods? Try one or two more methods from the Global object.

As you can see, NodeJS executes standard JS just like in any website!

Functions and Function Expressions

Creating functions give us greater control and allow us to combine code into units. Create a new file 2_functions.js. A function is expressed and called like this:

```
function sayHi(){
  console.log('Hi');
}

sayHi();
```

An often used alternative form --- the function expression --- looks like this:

```
var sayBye = function(){
  console.log('Bye');
};

sayBye();
```

We can use this construct to pass a function into another function, like this:

```
function callFunction(fun){
  fun();
}

callFunction(sayBye);
```

What do you get as an output? Why?

Modules and require()

What if we want to go even further and separate code into different modules? Well, we can put JS code in any files and include them in others. This allows us to reuse code and modularise logic. For example, we want to create a function for counting elements in an array and place it in a new file called 3_math.js:

3_math.js
```
var counter = function(arr){
  return 'There are ' + arr.length + ' elements in this array.';
}

console.log(counter(['Shaun','Crystal', 'Ryu']));
```

Call 3_math.js and check the output. Does it work?

Let us call this method from app.js with the following code:

```
require(./3_math’); // note that you omit the .js extension
console.log(counter(['Shaun','Crystal', 'Ryu']));
```

When we call this, we get an error telling us that the method counter is unknown. Why?

We need to explicitly tell JS that we want to export the counter method and map the method to a variable before using it in app.js:

3_math.js
```
var counter = function(arr){
  return 'There are ' + arr.length + ' elements in this array.';
}
module.exports = counter;
```

app.js
```
var counter = require('./math');
console.log(counter(['apple', 'pear', 'cherry']));
```

The `modules.export` declares that we want to allow access to counter. The `var counter = ...` explicitly maps the exported method to a variable name that we then can use within app. Makes sense? Now you can write JS code and modularise it in as many files are you like.

What if we want to add more functions to our module?

Lets add more math functions to our math module.

3_math.js
```
var counter = function(arr){
  return 'There are ' + arr.length + ' elements in this array.';
}

var adder = function(a,b){
    return `The sum of the 2 numbers is ${a+b}`;
}

var pi = 3.1415;

module.exports.counter = counter;
module.exports.adder = adder;
module.exports.pi = pi;
```

We can now see that we have different exported functions at the bottom of the page. Also note that the adder function uses a template string (more about this more elegant way to format out can be found here: https://wesbos.com/javascript-template-strings).

Lets try it out

app.js
```
var math = require('./3_math');
console.log(math.counter(['Shaun','Crystal', 'Ryu']));
console.log(math.adder(math.pi, 10));
```

Run it. Does it work?

We can further simplify the module by getting rid of those odd module.export lines at the end and moving them closer to the function:

3_math.js
```
module.exports.counter = function(arr){
  return 'There are ' + arr.length + ' elements in this array.';
}

module.export.adder = function(a,b){
    return `The sum of the 2 numbers is ${a+b}`;
}

module.expoert.pi = 3.1415;
```

Looks easier and clearer. Doesn't it? Run your app again. Does it work?

Files and Directories

Now, we will see how we can read and write files in Node. Create a new file and call it 4_fs.js. Also create a file called 'readMe.txt' and add a paragraph of random text from here: https://www.lipsum.com (or just type something).

4_fs.js
```
var fs = require('fs');
```

Note that you do not need the extension for fs, because fs is a core module. Also note that you should always name the variable name the same way as the module.

```
var readMe = fs.readFileSync(...);
```

What does 'Sync' here mean? Find out, you know where...

We store this information in a variable called readMe. Lets add some specifics to this line ...

```
var readMe = fs.readFileSync('readMe.txt', 'utf8');
```

So, we are reading a txt file called 'readMe.txt' and we are reading it in utf8 format.

Now add some code to output the variable to the console and run the code.

You should get an output of all your random text.

Let us now change this further and write the file, after reading it, to another file.

Remove the code with the console output that you just wrote.

Let us write the file elsewhere, with this:

```
fs.writeFileSync('writeMe.txt', readMe);
```

Notice that we write to writeMe.txt and we do write our variable readMe.

Run the code and check if you have this new file 'writeMe.txt'. Check its contents. You should find your earlier text in there.

Let us now try to read and write asynchronously (a fancy word of doing something without waiting for something else to be finished first).

Comment out the earlier code, except the require.

fs.**readFile**('readMe.txt', 'utf8', **function(err, data){…}**);

Now, this is slightly different from before. First we call readFile (instead of readFileSync) meaning that this method does not block execution. Second, we can see that there is a third parameter with a co-called callback function. This is a function that is called when reading has finished. This function has two parameters: a error (when something goes wrong) and data.

Lets create an output inside the function. We also put a line of output after the readFile command.

```
fs.readFile('readMe.txt', 'utf8', function(err, data){
     console.log('File read!');
});
console.log('After Reading.');
```

Run this. You should get the 'After Reading' now before the 'File read!' output. What does this mean?

Let us extend this code further and instead write the text asynchronously to another file. Let modify our code into this:

```
fs.readFile('readMe.txt', 'utf8', function(err, data){
  console.log('File read!');
  fs.writeFile('writeMe.txt', data, function(err, data){
    console.log('File written!');
  });
});
console.log('This is some code after the read-write code');
```

We have a counterpart to the readFile method --- writeFile. Again, we name the file we write to, we tell what we write (‘data’) and we have a callback function that we again use for a console output.

Run this code. What does the output tell and why is it like this?

CS615 Internet Solutions Engineering
NodeJS Tutorial

Part 3: Servers and Streams of Data

Make sure you have a folder cs615_nodejs where you keep all your files that you create in this tutorial series.

In this tutorial we will look at server features and learn how to handle streaming data.

Basic Server

Now, we will see how we can create a basic HTTP server in Node.js and use it to make requests to from our browser-client. Create a new file and call it 5_httpserver.js.

5_httpserver.js

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

Now, we have access to http functions that we need to write server code. Let us now create a server from this package.

```
var server = http.createServer();
```

This creates a server and stores it in a variable, so that we can refer to it in the future. Let us now make that server listen to requests --- we choose the local address 127.0.0.1 and port 3000 (lower ports require special permissions so we stick with a port in a higher region).

```
server.listen(3000, '127.0.0.1');
console.log('Listening to port 3000 ...');
```

Run this script and connect to the address with your browser or using the curl command:

```
curl 127.0.0.1:3000
```

You will get a ERR_EMPTY_RESPONSE error (code 52) in your browser or the following error when you use curl:

```
curl: (52) Empty reply from server
```

Why is that?

Well, we have not told our server to do anything when a request is being made. We created it, we listened to requests but our server does not talk. It is a bit like listening to a conversation and every time people call our name and ask a question we just stay quiet. After a while, the client is frustrated of waiting and simply points out that we are not fun to stick around --- the error code for this case happens to be 52.

So, we need to do something more in our server script.

```
var server = http.createServer(function(req, res){
 console.log('Request was made: ' + req.url);
});
```

Let us add a call-back function that is executed every time a request is made to the listening server. This function has two parameter --- a request and a response. We use the reqeest (req) parameter to log the request URL. This function is called every time now. When we run this, nothing changes, but we can see in the log that the server receives a request and what URL is requested.

Lets change this some more and actually send something back to the client.

```
var server = http.createServer(function(req, res){
  console.log('Request was made: ' + req.url);
  res.writeHead(200, {'Content-Type' : 'text/html'});
  res.end('Hello students!\n');
});
```

We use the response (res) parameter to reply now. We first write a header with the code 200 (everything is fine), and the content is of the type text/html (remember the Mime types). This is the header that is usually hidden. As a body of the message, we return the welcome message.

Run the script and access the server again. You should see the message.

Open the inspector of the browser, select 'Network' and click on '127.0.0.1' to see the header information. Alternatively (if you have OSX or Linux), try `curl -i 127.0.0.1:3000` to see the header information:

```
HTTP/1.1 200 OK
Content-Type: text/html
Date: Tue, 12 Mar 2019 23:54:11 GMT
Connection: keep-alive
Transfer-Encoding: chunked

Hello students!
```

What does all this mean?

Ok, so now we have learnt how to create a server and handle simple requests and responses with Node.js and understand how this works mechanically.

Streams of Data

Now, we need to talk about streaming data before we can learn more about serving data via servers.

Imagine you discover a **big mountain of candy in the woods** and you decide to move it to your home to feed it to your kids. You are pretty strong, but even so, it will take you a long time to move the entire mountain at once. You would move really slowly and it would take forever. Luckily, you are living next to a river and you happen to have a 1000 water-proof floating boxes nearby. You figure that you can use this for moving the mountain of candy to your home. You break small bits of candy from the mountain, fill your boxes, and throw them in the river. Your kids on the other end fish our the boxes from the river passing by your house. Over time, you can use this method to move all the candy to your house. More importantly, while you are still breaking off candy from the mountain, your kids can already start enjoying the candy. Everybody is happy. You do not need all the candy at home for your kids to enjoy the candy and eventually you get the candy to your home.

Obviously, it is unlikely that you discover a big pile of candy somewhere. However, you may have to deal with a lot of data. You may have to move it elsewhere, e.g. from a server to a client. And while this is needed, we can do very well to just move it bit by bit and the client can start enjoying it while the rest of the data is still coming along.

A buffer is a temporary storage for a small chunk of data that is transferred from one place to another. A mountain of data is split into small chunks and send along a way and over time arrives in chunks at its destination. When the buffer is full (like the box in our example) it is send via the stream (the river in our example).

So, when you stream a movie on Netflix you can start watching even thought the entire movie has not yet reached your TV. Cool right?

Streams are important for moving data in Node.js. We have writeable streams, readable streams and streams that can do both. Lets see how we can do this.

Lets create a new file and call it 6_streams.js.

6_streams.js
```
var http = require('http');
var fs = require('fs');
```

The first module is our http module, the second module is for file systems, called fs.

```
var myReadStream = fs.createReadStream(__dirname + '/readMe.txt',
'utf8');
```

Make sure you have your file readMe.txt at your correct directory (we used it before, remember?). Note that we concatenate the current directory with the filename to create a ReadStream.

Let us define a function for an event every time a chunk of data arrives:

```
myReadStream.on('data', function(chunk){
  // do something
});
```

We listen to our ReadStream and define an event handler with 'on'. The name of the event is 'data' and we attach a function that is called every time a chunk of data arrives.

```
myReadStream.on('data', function(chunk){
  console.log('new chunk received:' + chunk);
});
```

Right now, we only log it to the screen. Run this and see if you can stream the data from the file to your screen.

Lets send this to the user with a writeable stream.

So, this is the code that we have so far:

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

var myReadStream = fs.createReadStream(__dirname + '/readMe.txt',
'utf8');

myReadStream.on('data', function(chunk){
  console.log('New chunk received:' + chunk);
});
```

Let us first create a WriteStream and place it right after where we create our ReadStream.

```
var myReadStream = fs.createReadStream(__dirname + '/readMe.txt',
'utf8');
var myWriteStream = fs.createWriteStream(__dirname +
'/writeMe.txt');
```

The WriteStream is created very similar to the ReadStream, we have a method called createWriteStream and point to where we want to write. We write to our folder in a file called 'writeMe.txt'.

Let us add some extra code to our event handler function from earlier:

```
myReadStream.on('data', function(chunk){
  console.log('New chunk received:' + chunk);
  myWriteStream.write(chunk);
});
```

Pretty simple, right? Run the code and check the file writeMe.txt. It should now contain the text from readMe.txt.

Pipes work on both ends and exist because in Node.js we have to do this a lot. They receive input from a ReadStream and forward it into a WriteStream. We can simplify the code above by substituting

```
myReadStream.on('data', function(chunk){
  console.log('New chunk received:' + chunk);
  myWriteStream.write(chunk);
});
```

with:

```
myReadStream.pipe(myWriteStream);
```

The code should now look like this:

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

var myReadStream = fs.createReadStream(__dirname + '/readMe.txt',
'utf8');
var myWriteStream = fs.createWriteStream(__dirname +
'/writeMe.txt');

myReadStream.pipe(myWriteStream);
```

Delete the file writeMe.txt and run this code. You should see the file re-appear and again, containing the same data from readMe.txt. Pretty cool, right?

Let us start a new file and copy the code above inside**:**

6_pipeServer.js
```
var http = require('http');
var fs = require('fs');

var server = http.createServer(function(req, res){
  console.log('Request was made: ' + req.url);
  res.writeHead(200, {'Content-Type' : 'text/html'});
  res.end('Hello students!\n');
});

server.listen(3000, '127.0.0.1');
console.log('Listening to port 3000...');
```

Notice that the server code is what we wrote at the beginning of the tutorial. We have a server that has a call-back function for every request that is being made. As a response (res) we write a header and a body with a text (‘Hello students!’).

Now, let us stream our readMe.txt file to the client instead:

We now substitude the ‘Hello Students!’ message with a ReadStream. We read our file, like before:

```
var server = http.createServer(function(req, res){
  console.log('Request was made: ' + req.url);
  res.writeHead(200, {'Content-Type' : 'text/html'});
  var myReadStream = fs.createReadStream(__dirname +
'/readMe.txt', 'utf8');
});
```

Next, we want send what we read directly to the client using the response (res). We use a pipe for that and pipe our file contains to res.

```
var server = http.createServer(function(req, res){
  console.log('request was made: ' + req.url);
  res.writeHead(200, {'Content-Type' : 'text/plain'});
  var myReadStream = fs.createReadStream(__dirname +
'/readMe.txt', 'utf8');
  myReadStream.pipe(res);
});
```

Now, we can stream file content via HTTP to clients. In the next section, we will expand this to HTML content.

Serving HTML

In real life, we will not serve a lot of plain text files to a browser. We will however serve HTML.

So, let us first create an index.html:

index.html
```
<html>
  <head>
    <style>
      body {background: skyblue; font-family: verdana; color:
#fff; padding: 30px;}
      h1 {font-size: 48px; text-transform: uppercase; text-align:
center;}
      p {font-size:16px; text-align: center;}
    </style>
  </head>
  <body>
    <h1>Welcome to my Website</h1>
    <p>Delivered by Node.js.</p>
  </body>
```

```
</html>
```

Lets create a new file 7_servingHTML.js where we re-use the code from the previous JS file:

7_servingHTML.js

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

var server = http.createServer(function(req, res){
  console.log('request was made: ' + req.url);
  res.writeHead(200, {'Content-Type' : 'text/plain'});
  var myReadStream = fs.createReadStream(__dirname +
'/readMe.txt', 'utf8');
  myReadStream.pipe(res);
});

server.listen(3000, '127.0.0.1');
console.log('Listening to port 3000...');
```

This script sent a text file (readMe.txt) to a client. This time we have a HTML file, so the first thing we have to change is the Mime type:

```
var server = http.createServer(function(req, res){
  console.log('request was made: ' + req.url);
  res.writeHead(200, {'Content-Type' : 'text/html'});
  var myReadStream = fs.createReadStream(__dirname +
'/readMe.txt', 'utf8');
  myReadStream.pipe(res);
});
```

Next, we have to point to our newly created HTML file.

```
var server = http.createServer(function(req, res){
  console.log('request was made: ' + req.url);
  res.writeHead(200, {'Content-Type' : 'text/html'});
  var myReadStream = fs.createReadStream(__dirname +
'/index.html', 'utf8');
  myReadStream.pipe(res);
});
```

Thats it! Restart the server and try it out.

So we learned how to deal with HTTP servers and handle streaming data, alone and in combination with client-server communication.

CS615 Internet Solutions Engineering NodeJS Tutorial

Part 4: Json, routing, npm and express

Make sure you have a folder cs615_nodejs where you keep all your files that you create in this tutorial series.

Serving JSON

Now, we will see how we can serve JSON to the client. This time, we will not use streams but instead write it directly via the response object. First, create a new file and call it 8_json.js.

8_json.js
```
var http = require('http');
var fs = require('fs');

var server = http.createServer(function(req, res){
  console.log('request was made: ' + req.url);
  res.writeHead(200, {'Content-Type' : 'application/json'});
  // some json here
});

server.listen(3000, '127.0.0.1');
console.log('Listening to port 3000...');
```

Like before we change our header to the mime-type for json. We also need some json data for the body of the message. So, let us create a json object:

```
var server = http.createServer(function(req, res){
  console.log('request was made: ' + req.url);
  res.writeHead(200, {'Content-Type' : 'application/json'});
  // some json here
  myObject = {
    name: 'Paul',
    job: 'student',
    age: 25
  };
});
...
```

We now send it to the client:

```
var server = http.createServer(function(req, res){
  console.log('request was made: ' + req.url);
  res.writeHead(200, {'Content-Type' : 'application/json'});
  // some json here
  myObject = {
    name: 'Paul',
    job: 'student',
    age: 25
  };
  // res.end(myObj); // does not work
  res.end(JSON.stringify(myObj));
});
...
```

Notice that we cannot simply return json with res.end(MyObj). We need to convert myObj into json format. We do that with JSON.stringify. Try both version and see what error you are getting.

Right-click in the browser and go to Network and inspect the header (or run curl -i localhost:3000 to see the header. Can you see the application/json mime-type?

Ok, now we know how to send json data to a client. We would obviously not do this from the browser, but for clients who need to access data.

Routing

Now, let us talk about a new concept --- routing. Let us first create a new folder ‘routing’ to combine all files that we create for this part of the tutorial. In this folder, we create new file with some basic server code (like before):

9_routing.js
```
var http = require('http');
var fs = require('fs');

var server = http.createServer(function(req, res){
  console.log('Request was made: ' + req.url);
  res.writeHead(200, {'Content-Type' : 'text/plain'});
  res.end('Hello Students!');
});

server.listen(3000, '127.0.0.1');
console.log('Listening to port 3000...');
```

Run the script and access the page with your browser. No matter what url you use (e.g. http://127.0.0.1:3000 or http://127.0.0.1:3000/api/item/1), the server responds always with the same message.

In real life, however, we want to use the url to serve different types of information. When we access http://localhost:3000/home we expect a home page; when we type http://localhost:3000/contact we want a contact page etc.

This requires a feature called 'routing' where a server responds by serving different types of content based on the url.

Let us amend the server above to provide some basic routing.

```
var server = http.createServer(function(req, res){
  console.log('Request was made: ' + req.url);
  if (req.url === '/home' || req.url === '/'){
    res.writeHead(200, {'Content-Type' : 'text/html'});
    fs.createReadStream(__dirname + '/index.html').pipe(res);
  }
});
```

We are now using the request parameter (req) to read the url. Based on the url we identify those cases where a client asks for either the home or the root folder. For those cases, we deliver a index.html.

Copy the index.html from the project folder in your routing folder to make this example work.

Access the server by pointing your browser to http://localhost:3000/home or http://127.0.0.1:3000/. You should see the homepage from earlier.

Now, access the server again with the url http://locahost:3000/anything. Now, the browser waits and eventually gives up again with error 52 (ERR_EMPTY_RESPONSE). There is nothing served for anything else than /home or /.

Duplicate the index.html page and change a few lines to differentiate it.

```
<body>
  <h1>About</h1>
  <p>This website is about routing.</p>
 </body>
```

Now, add a few more lines to the server code.

```
var server = http.createServer(function(req, res){
  console.log('Request was made: ' + req.url);
  if (req.url === '/home' || req.url === '/'){
    res.writeHead(200, {'Content-Type' : 'text/html'});
    fs.createReadStream(__dirname + '/index.html').pipe(res);
  } else if (req.url === '/contact'){
    res.writeHead(200, {'Content-Type' : 'text/html'});
    fs.createReadStream(__dirname + '/contact.html').pipe(res);
  }
});
```

The else-conditon identifies if the contact page is requested which then routes to the contact.html page. Try if you can access both pages from your browser.

Let us also try to add a json feature as a route where a client can extract data:

```
...
} else if (req.url === '/api'){
    res.writeHead(200, {'Content-Type' : 'application/json'});
    var data = {name: 'Charlie', job: 'lecturer', age: 29};
    res.end(JSON.stringify(data));
}
...
```

This is very similar to what we did at the beginning of the tutorial. Try to access this page as well and see if you get the json data.

But still, if we request a page that does not exist, we get an error after a timeout. Let us solve this by creating a 404 page that captures these cases.

First, create a 404 page by duplicating index.html and changing it a bit:

Also, let us add a else (catch everything else) part to our server code:

```
...
} else {
  res.writeHead(404, {'Content-Type' : 'text/html'});
  fs.createReadStream(__dirname + '/404.html').pipe(res);
}
...
```

Notice that we only changed the HTTP code to 404 and the name of the page. It is still a html page, and we still stream it to the response. Try it out.

Express

Express is a package that offers an easy and flexible routing system (what we did above, just a lot simpler), integrates with many template engines, and a middleware framework. In this part of the tutorial, we will have a look at all of that.

Create a new subfolder 'expressApp' for all files in this section.

Initialise your project with: `npm init` (and answer the questions). Have a look at the newly created `package.json`.

Install express locally and store the dependency: `npm install express -save`
Notice the added dependency in `package.json`.

Create you entry application file: app.js

```
var express = require('express');
var app = express();

//
// We will handle requests here
//

app.listen(3000);
```

We import the express module and use it in line 2 to create an app. This app is central to do all kinds of things in express. The last line listens to requests on port 3000. You can hopefully already notice that this is a lot shorter than what we wrote earlier.

This code does not yet do anything. We now want to respond to different HTTP requests:

```
app.get('/', function(req, res){
  res.send('This is the homepage.');
});
```

This code responds to the root route.

```
app.get('/contact', function(req, res){
  res.send('This is the contact page.');
});
```

This code responds to the /contact route.

```
app.get('/profile/:id', function(req, res){
  res.send('You requested to see profile with the id of ' +
req.params.id);
});
```

We can use the route information (via the req parameter) to find out if somebody is requesting a particular type of id. Express allows us to define route pattern that we can use to identify parts of data or more specific request pattern.

In this tutorial we learnt about how to deliver json data to a client who wants data rather than a website (e.g. a client connecting to an API). We learnt about how we can set up a project folder and generate the package.json file while adding and removing packages as dependencies. We learnt about how to use routing with standard Node.js and with the framework package Express.