# An introduction to NodeJS

## 1. Introduction

These instructions present a gentle introduction to NodeJS. Throughout this lab you are going to develop a simple Web application with the following features:

* User can see the application in a browser
* The user sees a welcome page when they navigate to the following URL on your own PC http://localhost:3000/welcome
* The user is asked to enter name and password into a form
* The form is submitted to a server application which returns an acknowledgment

In order to do that, we need the following components

* An HTTP server
* A router to direct requests to processes depending on the URL from which they originate
* Code to handle the requests
* Handlers for the data which the user submits from the form
* Views which construct the pages we return to the user

The idea is to gradually build this Web application while exploring the different ways in which these components can be organised.

## 2. Getting set up

The first thing you must do is to make sure node is installed and ready to use in your system.

*If you are using the Azure Labs VM the VM is fully configured and ready to go (you don’t need to download anything).*

*If you are using a SHU pc then you need to activate node via AppsAnywhere.*

Here are some general instructions (remember to adapt those for you particular scenario).

1. Create a folder to save your work. For example, <some drive>:\sad-js01.
2. Download the latest version of the server from <http://nodejs.org/> (only if you are installing in your own machine).
3. Using a plain text editor such as VS Code, Atom, Notepad++ or Sublime Text create a file called hello.js in <some drive>:\sad-js01.
4. Put this code in the file and save it

* console.log("Hello World");

1. Open a terminal (or Windows Powershell) and go to <some drive>:\sad-js01
2. Run your code with this instruction if you have node in the same directory as your code

* .\node.exe hello.js
* Or this if node is installed on your system path
* node .\hello.js
* Or this if you are using a mac
* node ./hello.js
* Did it give you the output that you expected?

Note that you can use JavaScript engines to execute this type of code without having node.js

## 3. The Server

Create a file called server.js containing the following code

var http = require("http");  
  
http.createServer( function( request, response ) {  
  
 response.writeHead(200, {"Content-Type": "text/plain"});  
 response.write("Hello World");  
 response.end();  
  
}).listen(3000);

Run the code by typing node server.js in a terminal or shell. Now open the following URL in your browser: <http://localhost:3000>.

## 4. The Module

We should organise our code into distinct modules so we can call functions from other parts of our program. In order to convert server.js into a module you need to export those functions which you want to call from outside the module.

There are many design patterns and idioms for creating JavaScript and node modules. Here is one example.

*You’ll need to stop node (using Ctl-C) and restart it to run your new code*

Refactor server.js following these instructions.

var http = require("http");  
function start() {  
 function onRequest(request, response) {   
 response.writeHead(200, {"Content-Type": "text/plain"});  
 response.write("Hello World");  
 response.end();  
 }  
 http.createServer(onRequest).listen(3000);  
 console.log("Server started");  
}  
exports.start = start;

Now you’ll need a way of calling the code inside the module. Create a new file called index.js and put this code into it.

var server = require("./server");  
server.start();

Run the server with

node.exe index.js

View the result in the browser (<http://localhost:3000>).

Now let’s understand what is happening there. Try adding some console logging to your server.js.

var http = require("http");  
  
function onRequest(request, response) {  
 console.log("Request received");  
 response.writeHead(200, {"Content-Type": "text/plain"});  
 response.write("Hello World");  
 response.end();  
}  
  
http.createServer(onRequest).listen(3000);  
console.log("Server started");

Before moving on to the next task reflect on the following question:

**Is that really asynchronous and event-driven?**

Make some notes based on your understanding of what is happening when you open the page in the browser. Think about the different function calls (tip: add log statements in different places of your code: console.log("something happened")).

## 5. Routing requests

One of the powers of Web development with node is that each http request can be manipulated in a number of ways. For example,

* Requests from different URLs can be routed to different functions.
* POST and GET parameters can be extracted and made available in the code.

Everything we need is in the request object we are using as a parameter to onRequest().

Node has two modules which will help us handle these data items - url - querystring

Let’s modify our server.js module to start manipulating the request object.

var http = require("http");  
var url = require("url");  
function start() {  
 function onRequest(request, response) {  
 const baseURL = request.protocol + '://' + request.headers.host + '/';  
 const theURL = new URL(request.url, baseURL);  
 var pathname = theURL.pathname;  
 console.log("Request from "+ pathname);  
 response.writeHead(200, {"Content-Type": "text/plain"});  
 response.write("Hello World");  
 response.end();  
 }  
 http.createServer(onRequest).listen(3000);  
 console.log("Server started");  
}  
exports.start = start;

Now try calling different URLs such as:

* <http://localhost:3000>
* <http://localhost:3000/blah>
* <http://localhost:3000/blah?val=23>

Write down what happens when you call each of the URLs.

## 6. The router

Now let’s start using parts of the URL to route the request to the function we want.

In a file called router.js put

function route(pathname) {  
 console.log("Routing to "+ pathname);  
}  
exports.route = route;

And change the start() function in the server.js module to be

function start(route) {  
 function onRequest(request, response) {  
 var pathname = url.parse(request.url).pathname;  
 route(pathname);  
 }  
 // rest of the file omitted.

*This is an example of dependency injection.*

We now need to modify index.js so that the dependency is injected

var server = require("./server");  
var router = require("./router");  
  
server.start(router.route);

Restart the server and try the code.

* What output do you get?
* Do you understand what is happening? Write down the path a request goes through in your code.

## 7. Request handlers

The router needs somewhere to route to. Each URL (endpoint) is going to map onto a function.

These are called *request handlers*.

Create a new module named requestHandlers.js with this code (remember to include code for )

function start() {  
 console.log("Request handler start called");  
}  
function submit() {  
 console.log("Request handler submit called");  
}  
exports.start = start;  
exports.submit = submit;

Now you need to tell the router about the handlers. This has to be done in a way which is flexible and extensible. We’ll do this using a JavaScript object in index.js. Change that module to look like this

var server = require("./server");  
var router = require("./router");  
var requestHandlers = require("./requestHandlers");  
var handle = {}  
  
handle["/"] = requestHandlers.start;  
handle["/start"] = requestHandlers.start;  
handle["/submit"] = requestHandlers.submit;  
server.start(handle, router.route);

Now you need to change the server.js module to

* Accept the handler at start up
* Pass the handler to the router

function start(handle, route) {  
 function onRequest(request, response) {  
 var pathname = url.parse(request.url).pathname;  
 route(handle, pathname);  
 }  
 ...

The route() function needs some more drastic changes such as make decisions about where to send requests. It is acting as a *controller*.

Modify the router to be

function route(handle, pathname) {  
 console.log("Routing to "+ pathname);  
 if(typeof handle[pathname] === 'function') {  
 handle[pathname]();  
 } else {  
 console.log("No request handler found for "+ pathname);  
 }  
}  
exports.route = route;

Notice the following: - use of typeof() - use of === - how the function call is constructed dynamically

Reflect and make some notes on it. Can you explain why we need those elements and what is happening?

### Returning results *the wrong way*

You can get data back to the client by following the incoming route. Remember how server.js writes a response? You can return the result from the requestHandler to server.start() and then have server.start() write it out.

Try and do this for yourself…

You’ll need to edit server.js, requestHandlers.js and router.js.

Test all possible routes through your solution

* <http://localhost:3000>
* <http://localhost:3000/start>
* <http://localhost:3000/submit>
* <http://localhost:3000/blah>
* <http://localhost:3000/blah?val=23>

Can you think of any potential problems? What if the request handler is blocked waiting for data? - Accessing another system - Accessing a database server

Let’s explore this further. Modify requestHandlers.start() to be

function start() {  
 console.log("Request handler start called");  
 function sleep(mSeconds) {  
 var startTime = new Date().getTime();  
 while (new Date().getTime() < startTime + mSeconds);  
 }  
 sleep(10000);  
 return "In start()";  
}

*Add code to print the times when the function is called and when it finishes.*

Try running concurrent requests from two browsers, one for start and one for submit.

### Removing the block

We need to re-engineer our code so that it doesn’t block. We’ll move the response phase from the server and put it further down the pipeline.

Modify server.start() to begin with.

function start(handle, route) {  
 function onRequest(request, response) {  
 var pathname = url.parse(request.url).pathname;  
 route(handle, pathname, response);  
 }

Notice we’re now going to use the response object.

Change router.route() to be

function route(handle, pathname, response) {  
 console.log("Routing to "+ pathname);  
 if(typeof handle[pathname] === 'function') {  
 return handle[pathname](response);  
 } else {  
 console.log("No request handler found for "+ pathname);  
 response.writeHead(404, {"Content-Type": "text/plain"} );  
 response.write("404 page not found");  
 response.end()  
 }  
}  
exports.route = route;

Finally change the request handlers to

function start(response) {  
 console.log("Request handler start called");  
 response.writeHead(200, {"Content-Type": "text/plain"} );  
 response.write("Request Handler start called");  
 response.end();  
}  
function submit(response) {  
 response.writeHead(200, {"Content-Type": "text/plain"} );  
 response.write("Request Handler SUBMIT called");  
 response.end();  
}  
exports.start = start;  
exports.submit = submit;

Try testing your code now (repeat all the previous tests, including the concurrent requests). Reflect and make some notes on what is happening. Can you see a design pattern emerging from this example? Can you add code for dealing with a new path (e.g., query)?

## 8. Submitting data

Now that we have a basic Web application we can start writing code for manipulate submitted data. Data can be sent using either HTTP GET or POST.

The next steps are to: - Extract the submitted data - Put that data into a JavaScript object for further processing - Create a form from which the data can be sent

Change server.start.onRequest to be

function onRequest(request, response) {  
 var pathname = url.parse(request.url).pathname;  
 var query = url.parse(request.url).query;  
 route(handle, pathname, response, query);  
}

Notice the addition of the query parameter? That holds the query string. E.g., name=Hallam.

Change router.route() to be

function route(handle, pathname, response, query) {  
 console.log("Routing to "+ pathname);  
 if(typeof handle[pathname] === 'function') {  
 return handle[pathname](response, query);  
 } else {  
 console.log("No request handler found for "+ pathname);  
 response.writeHead(404, {"Content-Type": "text/plain"} );  
 response.write("404 page not found");  
 response.end()  
 }  
}  
exports.route = route;

Change requestHandlers.submit() to be

function submit(response, query) {  
 console.log("In submit with query string " + query);  
 var queryAsObject = querystring.parse(query);  
  
 response.writeHead(200, {"Content-Type": "text/plain"} );  
 response.write("Request Handler SUBMIT called with value "  
 + queryAsObject['a']);  
 response.end();  
}

Notice: - You’ll need to require the querystring module. - This code uses a JavaScript object as a dictionary.

Test your code.

* <http://localhost:3000/submit?name=hallam>
* <http://localhost:3000/blah?val=23>

## 9. Exercises

1. Modify the code such that a request object is passed to the handler which then extracts the parameters itself rather than having the server extract them.
2. Change requestHandler.start() so that it returns an HTML form
   * You’ll use a series of calls to response.write().
   * The form accepts three data items: name, degree course, age.
   * The form should send its data to your submit() method using HTTP GET.
3. Modify submit() to extract the three data items and return a table in which they are neatly formatted.

## 10. POST

HTTP POST is a little different to GET. - Data is sent in the body of the reply. - Data volumes can be very large. - Data arrives across many packets.

The server has to be non-blocking as data can take a long time to arrive, and the response handler has to reassemble the data.

When dealing with POST requests node will pass the packets of data to our client code. We must then implement listeners and callbacks to handle data as it arrives. The arrival of data triggers an event.

We’ll add the listeners to the request object in the server. The server is logically the place which provides data to the rest of the application.

First modify your form to use POST instead of GET.

Now modify the server.start() function to handle incoming data.

function start(handle, route) {  
 function onRequest(request, response) {  
 const baseURL = request.protocol + '://' + request.headers.host + '/';  
 const theURL = new URL(request.url, baseURL);  
 var pathname = theURL.pathname;  
 var query = theURL.query;  
 var postData = "";  
 request.addListener("data", function(postDataChunk) {  
 postData += postDataChunk;  
 console.log("Received more data: "+ postDataChunk);  
 });  
 request.addListener("end", function() {  
 route(handle, pathname, response, postData);  
 });  
 }

We will need two listeners: - One listens for incoming data packets and adds them to the data which has already arrived. - The second listens for the end of the data and calls route().

Notice that everything here is asynchronous. The order of the parameters have been changed so that the difference from GET is clearer.

We’ll need to change router.route() to pass the posted data onto the request handlers.

function route(handle, pathname, response, postData) {  
 console.log("Routing to "+ pathname);  
 if(typeof handle[pathname] === 'function') {  
 handle[pathname](response, postData);  
 } else {  
 console.log("No request handler found for "+ pathname);  
 response.writeHead(404, {"Content-Type": "text/plain"} );  
 response.write("404 page not found");  
 response.end()  
 }  
}  
exports.route = route;

The submit() handler just has to be tweaked to manage the different order and naming of the parameters.

function submit(response, postData) {  
 console.log("In submit with data "+ postData);  
  
 response.writeHead(200, {"Content-Type": "text/plain"} );  
 response.write("Request Handler SUBMIT called with value "+ postData);  
 response.end();  
}

And this is the code for the requestHandler.start().

function start(response, postData) {  
 var body = "<html>"+  
 "<head><title>Index</title></head>"+  
 "<body>"+  
 "<form action='/submit' method='post'>"+  
 "Name: <input type='text' name='name'/>"+  
 "Degree: <input type='text' name='degree'/>"+  
 "Age: <input type='text' name='age'/>"+  
 "<input type='submit' value='Submit it' />"+  
 "</body></html>";  
 response.writeHead(200, {"Content-Type": "text/html"} );  
 response.write(body);  
 response.end();  
}

## 11. Exercises

1. Extract the data from the body of the POST and
   * Output it console.log.
   * Return it the browser alongside the form that lets users submit more data.
2. Add a text area to the form and try passing large volumes of data around.
3. Change the requestHandler for POSTed data such that it will put the submitted data into a JavaScript object
4. Unify the code so that you have:
   * A method called question which accepts GET parameters.
   * This could be used to query a database.
   * The submit method we just worked on which handles POST data.