What is Node.js?

- Node is a framework.
- It has the ability to embed external functionality or extend functionality by making use of custom modules.
- These modules must be installed separately.
- An example of a module would be Postgres which would allow you to work with a Postgres database from your Node.js application.

How to install Node.js?

Go to https://nodejs.org/en/download to download the installer.

Hello world in Node.js

Let's start with displaying "Hello World" in a web browser using Node.

First, create a directory. Create a file named "helloword.js" and copy the following contents into the file:

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

http.createServer(function (req, res) {
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.end('Hello World!');
}).listen(8080);
```

"require" is a function. It takes one parameter. Here it is the string "http".

The basic functionality of the "require" function is that it reads a JavaScript file, executes it, and then returns and object. Using this object, we can then access the various functionalities available in the module called by the "require" function. In this case, we want to use the functionality of a module called "http". This returns an object and we assign this object to a variable.

In the second line of code, we are creating a server application. Remember, the “http” is the variable containing our imported http object. Objects can have methods associated with them. Here, we are calling “createServer” on the “http” object to instantiate a server application. We can do that because “createServer” is a method inside of the module “http”.

You might notice that there is a function being passed as a parameter. Any function passed to another function as an argument is called a callback function. This function is called whenever a request is made to our server application. A request comes from a client, the browser. So a request is made when someone accesses our page.

When a request is received, we are asking our function to return a “Hello World” response to that client. The writeHead function is used to send header data to the client. The end function just closes the connection to the client.

Lastly, we have a “server.listen()” function to make our server application listen to client requests on port number 8080.

Let’s go back to our callback function — the function that is being passed as a parameter to “createServer”.

In JavaScript all functions are objects. So we can pass them around as arguments. The purpose of a callback function is to execute after another function as finished executing. Why is this useful? JavaScript is both synchronous and asynchronous. It is an event driven language. When JavaScript makes a request to the file system, a database, or another external resource, it does not wait for a response to that request. Instead of waiting, JavaScript will keep executing other parts of the script. In the background, JavaScript is listening for any response to its requests. Whenever a response comes back, JavaScript will act on it and it will act on it within the callback function.

Let’s look at an example. Open up the Chrome Developer Console. Open Chrome and right click -> “Inspect”.

Type the following code in to your “Console”.

```
function first () {
  console.log(1);
}
```

```
function second () {
  console.log(2);
}

first ();
second ();
```

Output:
// 1
// 2

So what if the first function contains some code that can't be executed immediately? Say we have to query our database and wait for a response. We can simulate this by using the setTimeout function.

```
function first () {
  // Simulate a delay
  setTimeout( function() {
    console.log(1);
  }, 500);
}

function second () {
  console.log(2);
}

first();
second();
```

Now what we get is ...
// 2
// 1
… even though we invoked the first() function first.

It's not that JavaScript didn't execute our functions in the order we specified. JavaScript just didn't wait for a response from first() before moving on to execute second().

Let's see another example:

```
function takeANap(duration) {
  alert(`Starting my ${duration} nap.`);
}

takeANap('long');
```

Output:
// Alerts: Starting a long nap.

Now let's add in a callback function. We pass in a callback as a parameter into our takeANap function.

```
function takeANap(duration, callback) {
  alert(`Starting my ${duration} nap.`);
  callback();
}

takeANap('long', function() {
  alert('Done napping');
});

OR

function alertDone(){
  alert('Done napping');
}

takeANap('long', alertDone);
```

As you can see, we get our two alerts back to back.

Back to our Node.js example. Let's make this a bit more readable to drive home the point that everything is an object.

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

var server = http.createServer(function (req, res) {
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.end('Hello World!');
});
```

```
server.listen(8080);
```

Let’s run it. Enter the following command into your terminal:

$ node helloworld.js

If you open a browser and go to http://localhost:8080 you will see the message “Hello World!”

Handling a GET request in Node.js

Whenever people browse to our site, we get a request to send information back. This is GET request. Remember, HTTP is a data transfer protocol. It works by making requests and sending responses between client and server. HTTP has specific methods for this (e.g., GET and POST). GET is used to request data from a specified source and POST is used to send data to a server to create or update a resource.

Let’s have our server GET data from another site a.k.a. another server. There is a handy module that can help us with that. It’s called request.

Remember, we have to install modules in order to use them. Run the following command in your terminal:

$ npm install request

Let’s create a new file for this example. Call it “request.js”:

```
var request = require(“request”);
request(“http://www.google.com”, function(error, response,
body) {
  console.log(body);
});
```

We are making a GET request to Google and then calling our callback function when we receive a response. Our parameters in our callback function are a little bit different this time:

- Error - in case there is an error received it is recorded in this variable
- Response - this is the http headers which are sent back
- Body - this is the content of the actual response sent by Google.

What is Express.js?

There is a module called Express.js or just Express. It is a Node.js web application server framework, which is specifically designed for building web applications. It is the standard that most people use with Node.js. It makes it easier to develop an application which can handle multiple types of HTTP requests like GET and POST.

Again, let’s first install express with the following command:

$ npm install express

Let’s make another hello world example using express.

Create a file called “helloworld2.js” with the following content:

```
var express = require(“express”);
var app = express();

app.get(‘/’, function (req, res) {
  res.send(“Hello World”);
});

var server = app.listen(8080, function () {

});
```

Again, we have required the module so we can use it and created an express object stored in the variable called “app”.

Our callback function is simple. We didn’t have to explicitly write a header. All we said was “res.send()”.

The “res” parameter is used to send content back to the requesting client. “res” is short for response. Let’s run it using the following command:

$ node helloworld2.js

So far this doesn't look much different from our first hello world example. Let's add some stuff.

Remember, a client can make GET or POST requests to various URLs like these:

http://localhost:8080/Books
http://localhost:8080/Students

If a GET request is made for the first URL, then the response would, ideally, be a list of books. Based on the URL which is accessed by the client, a different functionality on the web server will be invoked and an appropriate response will be sent to the client. This is called routing.

Routing is a way to determine how an application responds to a client request. A route can have one or more handler functions which are executed when the route is matched.

The general syntax is: app.METHOD(PATH, HANDLER) where METHOD is an HTTP request method (i.e., GET or POST), PATH is a path on the server, also called the route and HANDLER is the function to execute when the route is matched. The HANDLER is our callback function.

Let's go back to our example and add some routes.

```
app.route('/hello').get(function (req, res) {
   res.send("hello");
});
app.route('/world').get(function (req, res) {
   res.send("world");
});
```

We are defining a route '/hello'. If the client accesses the URL http://localhost:8080/hello then we will send back "hello".

Note that a route is not a relative path of our file system. A route is a path on our server.

The callback function in this example has two parameters “req” and “res”. “req” has information about the request being made and “res” is the parameter we use to send information back.

Let’s try it. First, start the server with the following command:

```
$ node helloworld2.js
```

Try browsing to http://localhost:8080/hello.

From this example we can see how we can decide what to display based on routing. There is a second part to routing which is … templates.

Using templates with pug

Templates are a way to make developing applications faster. You can think of them as a replacement for HTML. They make it easier to insert data dynamically. We can send data from our back-end to our front-end and insert data into our page.

There are tons of templating engines (also called view engines) out there like Handlebars, EJS, Pug, etc.

Today we will use Pug (formally known as Jade). In lab, we will use EJS. Let’s install it with the following command:

```
$ npm install pug
```

Let’s create another example. Create a file called “index.js” and another file “index.pug”. Make a folder called “views” and place index.pug inside of it.

Let’s write our server-side code. Put the following content into your “index.js” file:

```
var express = require(“express”);
var app = express();
```

```
app.set('view engine', 'pug');

app.get('/', function (req, res) {
  res.render('index', { title: "Hello World",
                        message: "Welcome"})

});

var server = app.listen(8080, function () {

});
```

We have defined a view engine, pug. This means that our server now knows that all of our files are pug files and it will treat them accordingly.

The render function is used to render a view a.k.a. a web page. We are going to render index.pug. But, we don't need to put .pug because the server knows it's a pug file. In our call to render we are also passing a list as a parameter. This list contains some variables and their associated values. These will be passed along to the front-end with our response. Let's see what we can do with that.

Put the following content into your "index.pug" file:

```
html
  head
    title!=title

  body
    h1=message
```

As you can see, we can easily access the variables that we have passed in our render function.

Let's run it using the following commands:

```
$ node index.js
```

That's all folks.