IMY 220 ● Lecture 17 ● 16 September

What is node.js?

Node.js is a **cross-platform runtime environment** that can be used to create simple **web server applications with JavaScript**.

cross-platform: platform independent i.e., it runs on Microsoft Windows, Linux, and Mac OS X with no changes.

runtime environment: A testing environment for software developers that allow them to track program instructions as it is being executed at runtime.

webserver applications with JavaScript: JavaScript is a client-side scripting language which means it normally executes in the browser, but node.js is not executed in the browser.

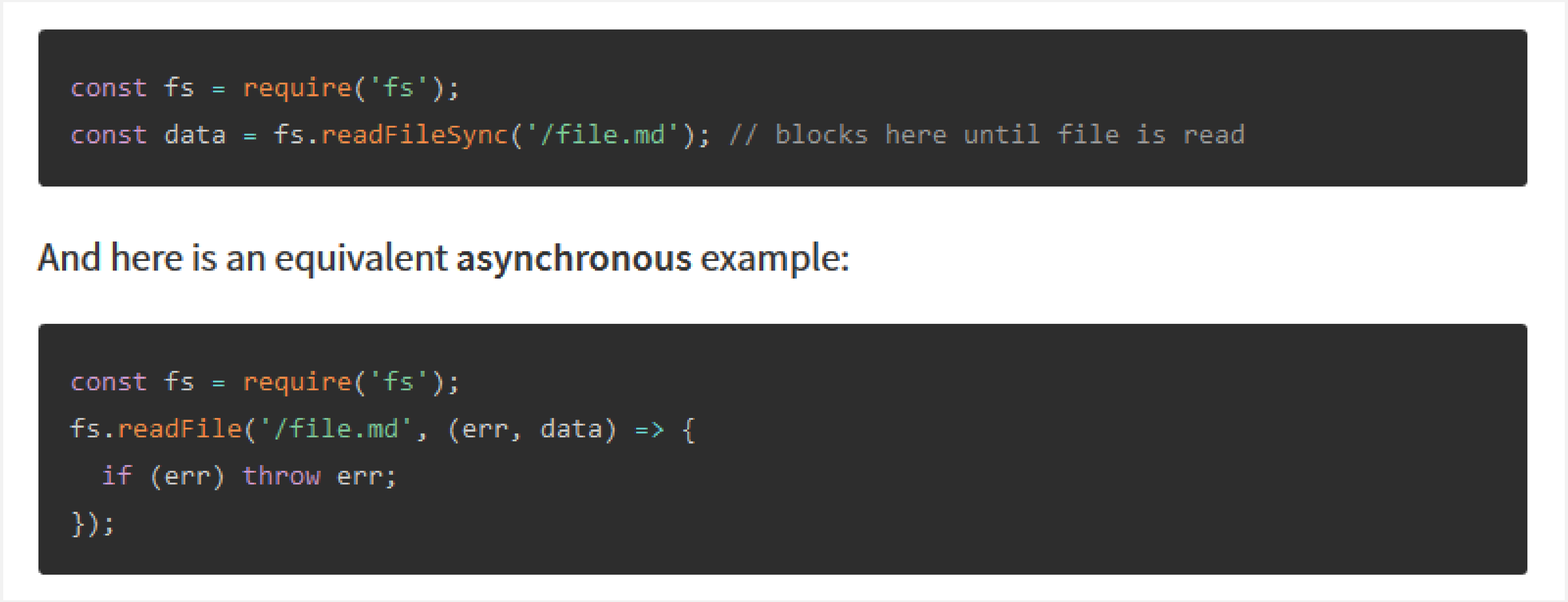
Why use node.js?

Node.js applications are designed to maximize throughput and efficiency, using non-blocking I/O and asynchronous events.

It is commonly used for real-time applications due to its asynchronous nature.

For a good discussion of blocking vs non-blocking:
https://nodejs.org/en/docs/guides/blocking-vs-non-blocking/

Why use node.js?



```
const fs = require('fs');
const data = fs.readFileSync('/file.md'); // blocks here until file is read
```

And here is an equivalent **asynchronous** example:

```
const fs = require('fs');
fs.readFile('/file.md', (err, data) => {
  if (err) throw err;
});
```

Why use node.js?

Node.js internally uses the **Google V8 JavaScript engine** to execute code, and a large percentage of the basic modules are written in JavaScript.

Node.js contains a built-in asynchronous I/O library for file, socket, and HTTP communication, which allows applications to act as a Web server without software such as Apache HTTP Server or IIS.

Who uses node.js?

node.js is gaining adoption as a high-performance server-side platform and is notably used by:

- Groupon
- SAP
- LinkedIn
- Microsoft
- Yahoo!
- Walmart
- PayPal
- Netflix
- Oracle.

Getting started with node.js in Windows

1. Install node with the .msi installer from https://nodejs.org/en/download/

    This will install Node.js and NPM (Node Packaged Modules)

2. Open the Windows Command Prompt (cmd)

Getting started with node.js in Windows

3.To test if node is working, type **node –v** (this will show the version of node, if it doesn't then node is not properly installed).

Current LTS (long term support) version: 10.16.3

Also type “npm -v” to see the Node Packaged Managed version installed

Getting started with node.js in Windows

4.Create a JavaScript file.

5. Navigate to the directory where your .js file is
e.g. **cd C:\Users\YourName\Desktop\folder**.

6. Run the server by typing the command: **node test.js**

7. Press Ctrl-C to quit the running Node app.

Creating a simple Hello World application

Create a JavaScript file (*.js)

Create a server:

```
const http = require('http');
http.createServer(
      …
).listen(1337, '127.0.0.1');
```

To use the HTTP server and client one must require('http').

- http.createServer([requestListener]) returns a new web server object.
- server.listen(PORT, HOST); The server listens to port 1337 on localhost.

Creating a simple Hello World application

http.createServer([requestListener])

- The requestListener is a function which is automatically added to the 'request' event.

```
const http = require('http');
http.createServer((request, response) => {
        …
}).listen(1337, '127.0.0.1');
```

Event: 'request'

- function (request, response) { }
  - Emitted each time there is a request.
  - Note that there may be multiple requests per connection (in the case of keep-alive connections).

Creating a simple Hello World application

response.writeHead(statusCode, [reasonPhrase], [headers])

- Sends a response header to the request.
- The status code is a 3-digit HTTP status code, like 404 or 200.
- The last argument, headers, are the response headers.
- Optionally one can give a human-readable reasonPhrase as the second argument.
- This method must only be called once on a message and it must be called before response.end() is called.

```
const http = require('http');
http.createServer((request, response) => {
        response.writeHead(200, {'Content-Type': 'text/html'});
        …
}).listen(1337, '127.0.0.1');
```

Creating a simple Hello World application

response.write(chunk, [encoding])

- This sends a chunk of the response body. This method may be called multiple times to provide successive parts of the body.
- chunk can be a string or a buffer. If chunk is a string, the second parameter specifies how to encode it into a byte stream.
  - By default the encoding is 'utf8'.

```
const http = require('http');
http.createServer((request, response) => {
        response.writeHead(200, {'Content-Type': 'text/html'});
        response.write('Hello');
        …
}).listen(1337, '127.0.0.1');
```

Creating a simple Hello World application

response.end([data], [encoding])

- This method signals to the server that all of the response headers and body have been sent; that server should consider this message complete.
- The method, response.end(), MUST be called on each response.
- If data is specified, it is equivalent to calling response.write(data, encoding) followed by response.end().

```
const http = require('http');
http.createServer((request, response) => {
       response.writeHead(200, {'Content-Type': 'text/html'});
       response.write('Hello');
       response.end('World');
}).listen(1337, '127.0.0.1');
```

Serving a page

Create an HTML file that you want to run on the server.

Create a JavaScript file that will be used to insert node code.

Insert the node code to create the server.

```
const http = require('http');
http.createServer((request, response) => {
      …
}).listen(1337, '127.0.0.1');
```

Serving a page

File System

In order to serve a page, your code will have to read the file.

For file I/O use require('fs').

```
const http = require('http');
const fs = require('fs');
http.createServer((request, response) => {
      …
}).listen(1337, '127.0.0.1');
```

Serving a page

fs.readFile(filename, [options], callback)

- filename = String
- options = Object
- callback= Function

- Asynchronously reads the entire contents of a file.
- The callback is passed two arguments (err, data), where data is the contents of the file.

```
const http = require('http');
const fs = require('fs');
http.createServer((request, response) => {
      fs.readFile('index.html', (err, data) => {
            …
      });
}).listen(1337, '127.0.0.1');
```

Serving a page

Now you can just add the same code that was used to write data to the page in the callback, but write the data argument to the page.

```
const http = require('http');
const fs = require('fs');
http.createServer((request, response) => {
      fs.readFile('index.html', (err, data) => {
            response.writeHead(200, {
                  'Content-Type': 'text/html',
                  'Content-Length': data.length
            });
            response.write(data);
            response.end();
      });
}).listen(1337, '127.0.0.1');
```

Show status messages in the terminal

```
http.createServer((request, response) => {
    …
}).listen(1337, '127.0.0.1', () => {
    console.log('Server running at http://127.0.0.1:1337');
});
```

Node modules

Node.js ships with a lot of useful modules.

You don't have to write everything from scratch.

node.js is thus two things:

- a runtime environment and
- a library.

Separating code into modules

It's relatively easy to keep the different concerns of your code separated, by putting them in modules.

This allows you to have a clean main file, which you execute with Node.js, and clean modules that can be used by the main file and among each other.

Modules in node.js

Separating code into modules relies on the ”exports” keyword

For example:

- greeter.js:

```
module.exports.greet = name => {
        return 'Hello ' + name;
}
```

- index.js:

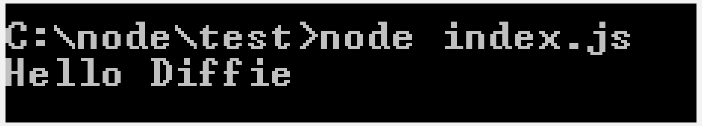
```
let greeter = require('./greeter');

console.log(greeter.greet('Diffie'));
```

Looks for a file called greeter.js in the same directory as this file

Executes the “greet”-function in our newly defined package

- Run:



```
C:\node\test>node index.js
Hello Diffie
```

Modules in node.js

You can export functionality a number of different ways:

- math.js

```
module.exports.add = (x, y) => x + y;

module.exports.subtract = (x, y) => x - y;

module.exports.multiply = (x, y) => x * y;

module.exports.divide = (x, y) => x / y;
```

Modules in node.js

You can export functionality a number of different ways:

- math.js

```
module.exports = {
   add(x, y){
      return x + y;
   },
   subtract(x, y){
      return x - y;
   },
   multiply(x, y){
      return x * y;
   },
   divide(x, y){
      return x / y;
   }
}
```

Modules in node.js

You can export functionality a number of different ways:

- math.js

```
const Math = function(){};

Math.prototype.add = (x, y) => x + y;

Math.prototype.subtract = (x, y) =>  x - y;

Math.prototype.multiply = (x, y) =>  x * y;

Math.prototype.divide = (x, y) =>  x / y;

const math = new Math();
module.exports = math;
```

Modules in node.js

You can use any of the above implementations as follows (if *index.js* and *math.js* are in the same directory):

- index.js

```
const math = require('./math');

console.log(math.add(1, 2));
// Prints out 3

console.log(math.multiply(5, 5));
// Prints out 25
```

Modules in node.js

You can also export ES6 classes. To do this, you must either export the entire class and create an instance of it to use it…

```
const MathClass = require('./math');

const math = new MathClass();

console.log(math.add(1, 2));
```

```
module.exports = class Math{
	add(x, y){
		return x + y;
	}

	subtract(x, y){
		return x - y;
	}

	multiply(x, y){
		return x * y;
	}

	divide(x, y){
		return x / y;
	}
}
```

Modules in node.js

…or you must export an instance of the newly created class

```
const math = require('./math');

console.log(math.add(1, 2));
```

```
class Math{
        add(x, y){
                return x + y;
        }

        subtract(x, y){
                return x - y;
        }

        multiply(x, y){
                return x * y;
        }

        divide(x, y){
                return x / y;
        }
}
module.exports = new Math();
```

Internal node.js modules

Modules you have already seen:

- http
- fs

How to make use of internal Node.js modules:

- `const http = require("http");`
- `const fs= require('fs');`

How to organize your http server

Create a **main** file which we use to start our application, and a **module** file where our HTTP server code lives.

It's a standard to name your main file *index.js* and to put our server module into a file named *server.js*.

Insert the code for a very basic HTTP server in the file server.js

Create a main file called index.js which is used to start the application by making use of the other modules of the application (like the HTTP server module that lives in server.js).

How to make server.js a real node.js module

Making some code a **module** means we need to *export* those parts of its functionality that we want to provide to scripts that require the module

Basic http server module

The functionality our HTTP server needs to export is simple: scripts requiring our server module simply need to *start the server*.

To make this possible, put the **server code** into a function named **start**, and **export** this function.

Basic http server module

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

const start = () => {
    const onRequest = (request, response)  => {
        response.writeHead(200, {'Content-Type': 'text/html'});
        response.write('Hello World');
        response.end();
    }

    http.createServer(onRequest).listen(8888);
    console.log('Server has started');
}

exports.start = start;
```

Main file

Create a main file *index.js*, and start the HTTP there, although the code for the server is in *server.js*.

Create a file *index.js* with the following content:

```
const server = require('./server');

server.start();
```

Using the server module

You can use the server module just like any internal module:

- by requiring its file and
- assigning it to a variable

Its exported functions become available to the programmer.

Now you can start the app via the main script, and it will work exactly the same:

- By typing ”node index.js” into the terminal

Other useful modules

You can install other modules using Node Package Manager.

ExpressJS

- It is a web framework that allows you to build single, multi-page, and hybrid web applications.

- (Next lecture)

References

http://blog.falafel.com/getting-started-with-nodejs-for-windows/

http://nodejs.org/api/http.html

http://nodejs.org/api/fs.html