CS 572 Modern Web Applications

Najeeb Najeeb, PhD (najeeb@miu.edu)

V1.1.0

Syllabus

- Course Goal
- Course Schedule
- Grading
- Exam Objection Policy
- Your Personal Goals During any Course

JavaScript Full Stack Development

- MongoDB
  - NoSQL database (document store)
  - Stores JSON documents
- Express
  - JavaScript web framework
  - On top of Node
- Angular
  - JavaScript UI framework
  - Single Page Applications
- Node
  - JavaScript server-side platform
  - Single threaded, fast and scalable

Full Stack Development

- Build the front end and back end of a website or web application.
- Front end: Interaction with browser.
- Back end: Interaction with database and server.
- Database driver application.

No Frameworks

- We will start with nothing and build up.
- No opinionated frameworks (you are advised to investigate these in the future)
  - MEAN.io
  - MEANjs
  - Express Generator
  - Yeoman
- Frameworks are good for complex projects and for advanced users not good for learning and understanding for beginners.

Roadmap and Outcomes

- Node.js: write asynchronous (non-blocking) code. Understand node platform to start a project.
- Express: setup express and get requests and send back responses. REST API.
- MongoDB: what NoSQL DB looks like. Full API interacting with DB.
- AngularJS: Investigate AngularJS and architect it. A single page application.
- MEAN application: Learn by example. We will create a MEAN Games application.

Demo MEAN Games

Installation

- NodeJS
  - nodejs.org
- Mongo DB
  - mongodb.com
- IDE
  - vscode.com

NodeJS

NodeJS and History

- Install Node from nodejs.org.
- Versions jumped from 0.x to 14.x
  - Due to the merge back from io.js to Node.js
  - Some original Node.js developers forked io.js why
    - community-driven development
    - Active release cycles
    - Use of semver for releases.
  - Node.js owned by Joyent had slow development, advisory board

Joyent Advisory Board

- Centralize Node.js to make development and future features faster.
- Board of large companies that use Node.js
- It moved Node.js from mailing lists and GitHub issues and developer's contribution to the power of the "big shots".
- Companies like Walmart, Yahoo, IBM, Microsoft, Joyent, Netflix, and PayPal were controlling things not the developer.
- The advisory board resulted in slower development and feature releases.

SEMVER

- Semantic Versioning
- MAJOR.MINOR.PATCH
- Major: incompatible API changes
- Minor: add backward compatible functionality
- Patch: add backward compatible bug fixes.

NodeJS

Check version

Run Node

Create and run node file

Install node from nodejs.org

node -v (or node --version)

v14.16.1

Check node package manager (npm)

npm -v

6.14.12

Start node

node

Print "Hello World!" from node

> console.log("Hello World!");

Hello World!

NodeJS

Check version

Run Node

Create and run node file

Start node

```
node
```

Print "Hello World!" from node

```
> console.log("Hello World!");
Hello World!
```

Write some JS

```
> const name = "Jack";
> console.log("Hello "+ name);
Hello Jack
> .exit
```

NodeJS

Check version

Run Node

Create and run node file

I use vsCode (it has a lot of MEAN plugins)

Create a file (instantHello.js)

```
const username = "Jack";

console.log("Hello  "+ username );
```

Run file

```
node instantHello.js
```

Hello Jack

Modular Programming

- Best practice is to have building blocks
  - You do not want everything running from a single file (hard to maintain, test).
- Separate the main application file from the modules you build.
- Separate loading from invocation.
- Each module exposes some functionality for other modules to use.

Modular Node

Multi files Node application
Require to load file
Expose functionality using module.exports

Create app01.js file

```
require("./instantHello");
```

Run file

```
node app01.js
```

```
Hello Jack
```

Modular Node

Multi files Node application

Require to load file

Expose functionality using module.exports

Create goodbye.js file

```
module.exports = function(){
  console.log("Goodbye");
}
```

app01.js file

```
require("./instantHello");
const goodbye = require("./goodbye");
goodbye();
```

Run file

```
node app01.js
```

```
Hello Jack
Goodbye
```

Exports

- Export more than one function.
- Encapsulation; reducing side effects, improve code maintainability.
- Avoid using .js in require. This will enable changing the structure of your modules in the future. If a file becomes complex, we can put it in a folder by itself as a module and make index.js backwards compatible.
- When require searches (require(name)):
  - Serach for name.js, if not found
  - Search for index.js in folder name
- Three ways to export
  - Single function
  - Multi functions
  - Return value

Module.exports

Single function

Multi functions

Return values

Create talk/index.js file

```
module.exports = function(){
  console.log("Hi");
}
```

app02.js file

```
require("./instantHello");
const goodbye = require("./talk");
goodbye();
```

Run file

```
node app02.js
```

```
Hello Jack
Hi
```

Module.exports

Single function

Multi functions

Return values

Create talk/index.js file

```
const filename = "index.js";
consthello = function(name) {
  console.log("Hello "+ name);
}
const intro = function() {
   console.log("I'm a node file called "+ filename);
}
module.exports = {
   greeting : hello,
   intro : intro
}
```

app02.js file

```
const talk = require("./talk");
talk.greeting("Jim");
talk.intro();
```

Run file

```
node app02.js
```

```
Hello Jack
Hello Jim
I'm a node file called index.js
```

Module.exports

Single function

Multi functions

Return values

Create talk/question.js file

```
const answer = "This is a good question.";
module.exports.ask = function(question) {
    console.log(question);
    return answer;
}
```

app02.js file

```
const question= require("./talk/question");
const answer = question.ask("What is the meaning of life?");
console.log(answer);
```

Run file

```
node app02.js
```

```
What is the meaning of life?
That is a good question.
```

Single Threaded Node

- Node is single threaded.
  - One process to deal with all requests from all visitors.
- Node.js is designed to address I/O scalability (not computational scalability).
- I/O: reading files and working with DB.
- No user should wait for another users DB access.
- What if a user requests a computationally intense operation? (compute Fibonacci)
- Timers enable asynchronous code to run in separate threads. This enables scalable I/O operations. Perform file reading without everything else having to wait.

Async

setTimeout

readFileSync

readFileAsync

Named callback

app03.js file, setTimeout creates asynchronous code

```
console.log("1: Start app");
const laterWork = setTimeout( function(){
    console.log("2: In setTimeout");
}, 3000);
console.log("3: End app");
```

Run file

```
node app03.js
```

```
1: Start app
3: End app
2: In the setTimeout
```

Async

setTimeout

readFileSync

readFileAsync

Named callback

app04.js file

```
const fs= require("fs");
console.log("1: Get a file");
constfile= fs.readFileSync("shortFile.txt");
console.log("2: Got the file");
console.log("3: App continues...");
```

Run file

```
node app04.js
```

```
1: Get a file
2: Got the file
3: App continues...
```

Async

setTimeout

readFileSync

readFileAsync

Named callback

app05.js file

```
const fs= require("fs");
console.log("Going to get a file");
fs.readFile("shortFile.txt", function(err, file) {
    console.log("Got the file");
});
console.log("App continues...");
```

Run file

```
node app05.js
```

```
Going to get a file
App continues...
Got the file
```

Async

setTimeout
readFileSync
readFileAsync
Named callback

app06.js file

```
const fs= require("fs");
const onFileLoad= function(err, file) {
    console.log("Got the file");
}
console.log("Going to get a file");
fs.readFile("shortFile.txt", onFileLoad);
console.log("App continues...");
```

Run file

```
node app06.js
```

```
Going to get a file
App continues...
Got the file
```

Benefits of Named Callbacks

- Readability
- Testability
- Maintainability

Intense Computations

- Avoid delays in a single threaded application server.
- If someone performs a task that takes too long to finish, it should not delay everyone else on a webserver.
- Computation is not I/O operations. Computations need a process to perform the operation.
- Spawn a child process to perform the computation. This will consume resources, but it will not block the main server.

Computation

Fibonacci

Blocking

non-Blocking

./computation/_fibonacci.js file

```
const fib= function(number){
if (number <= 2){
    return 1;
} else {
    return fib(number-1) + fib(number-2);
}};
console.log("Fibonacci of 45 is "+ fib(45));
```

Run file

```
node _fibonacci.js
```

Fibonacci of 55 is 1134903170

Computation
Fibonacci
Blocking
non-Blocking

app07.js file

```
console.log("1: Start");
require("./computation/_fibonacci");
console.log("2: End");
```

Run file

```
node app07.js
```

```
Start
Fibonacci of 55 is 1134903170
End
```

Computation

Fibonacci

Blocking

non-Blocking

app08.js file

```
const child_process= require("child_process");
console.log("1: Start");
const newProcess= child_process.spawn("node",
["computation/_fibonacci.js"], {stdio : "inherit"});
console.log("2: End");
```

Run file

```
node app08.js
```

```
Start
End
Fibonacci of 55 is 1134903170
```