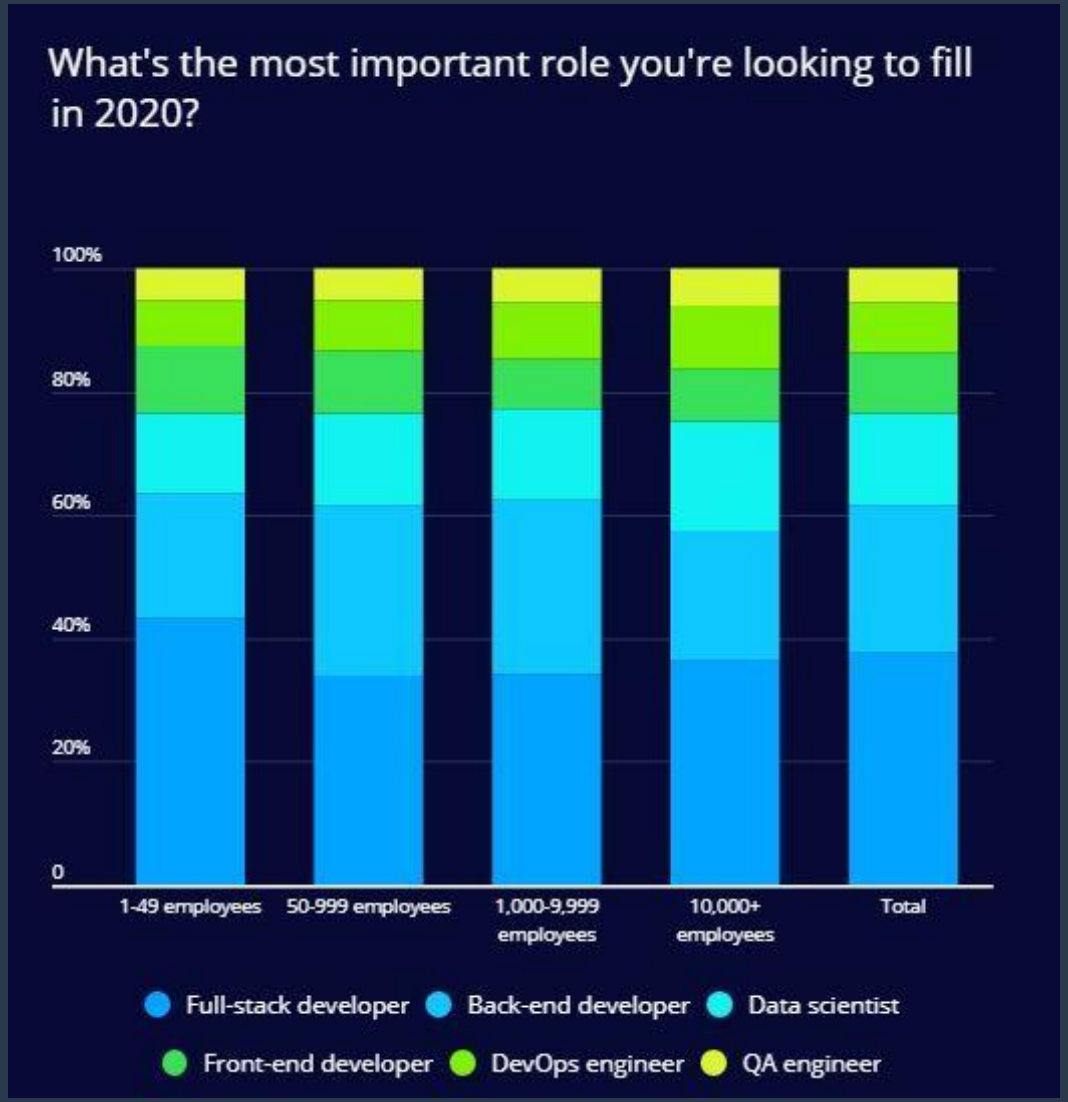
CS 572 Modern Web Applications

Najeeb Najeeb, PhD (najeeb@miu.edu)

V3.0.0

Why Full Stack Development?

- The HackerRank Developers skills report 2020[1].
- Small companies hiring priority, 38% of hiring managers state full-stack as the number 1.



1- https://research.hackerrank.com/developer-skills/2020?

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.

JavaScript Full Stack Development

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

Introducing NodeJS & Express
Do Less Accomplish More

Wholeness

Writing everything from scratch is difficult and time-consuming. When you use a platform or a framework it performs most of the heavy lifting. You may only fully utilize a framework if you write code that is aligned with the framework expectations. You get the support of nature when your actions are aligned with the laws of nature, this results in actions being correct the first time, and there is no need to waste time correcting things.

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.
- Angular: Investigate Angular and the architecture of an Angular application. Build a single-page application.
- MEAN application: Learn by example. We will create a MEAN Games application.

Demo MEAN Games

Introducing Node & Express
Do Less Accomplish More

1. How to write a Node application?
2. How to write a web application in Node?
3. How to write an Express application?

Introducing Node & Express
Do Less Accomplish More

1. How to write a Node application?
2. How to write a web application in Node?
3. How to write an Express application?

NodeJS

NodeJS and History

- Install Node from nodejs.org.
- Versions jumped from 0.x to 17.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)

v16.14.2

Check node package manager (npm)

npm -v

8.5.0

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
```

Write some JS

```
> var name = "Jack";
> console.log("Hello  "+ name);
Hello Jack
> name = 5;
> console.log("Hello  "+ name);
Hello 5
> .exit
```

NodeJS

Check version

Run Node

Create and run node file

vsCode (code.visualstudio.com has several MEAN plugins)

Create a file (instantHello.js)

```
let userName = "Jack";

console.log("Hello", userName);
```

Run file

```
node hello.js
```

```
Hello Jack
```

Modular Programming

- Best practice to have building blocks
  - You do not want everything running from a single file (hard to maintain).
- 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 talk.js file

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

app01.js file

```
require("./instantHello");
let goodbye = require("./talk");
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("Goodbye");
}
```

app02.js file

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

Run file

```
node app02.js
```

```
Hello Jack
Goodbye
```

Module.exports

Single function

Multi functions

Return values

Create talk/index.js file

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

app02.js file

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

Run file

```
node app02.js
```

```
Hello Jack
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");
const buffer= fs.readFileSync("largeFile.txt");
console.log("2: Got the file", buffer.toString().substring(0, 21));
console.log("3: App continues...");
```

Run file, you notice a short delay between 1: ... and 2: ...

```
node app04.js
```

```
1: Get a file
2: Got the file This is a long 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, buffer) {
   console.log("Got the file", buffer.toString().substring(0,
21));
});
console.log("App continues...");
```

Run file

```
node app05.js
```

```
Going to get a file
App continues...
Got the file This is a long file.
```

Async

setTimeout
readFileSync
readFileAsync
Named callback

app06.js file

```
const fs= require("fs");
const printFileFirstLine= function(err, file) {
   console.log("Got the file", buffer.toString().substring(0,
21));
}
console.log("1: Get a file");
fs.readFile("longFile.txt", printFileFirstLine);
console.log("3: App continues...");
```

Run file

```
node app06.js
```

```
Got the file
App continues...
Got the file This is a long 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

Nonblocking

fibonacci.js file

```
const fibonacci= function(number) {
if (number <= 2) {
    return 1;
} else {
    return fibonacci(number-1) + fibonacci(number-2);
} } ;
console.log("Fibonacci of 42 is "+ fibonacci(42));
```

Run file, you will notice a delay (right)

```
node fibonacci.js
```

Fibonacci of 42 is 267914296

Computation
Fibonacci
Blocking
Nonblocking

app07.js file

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

Run file

```
node app07.js
```

```
1: Start
Fibonacci of 42 is 267914296
2: End
```

Why is the dangerous and not a good idea?

Computation

Fibonacci

Blocking

Nonblocking

app08.js file

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

Run file

```
node app08.js
```

```
1: Start
2: End
Fibonacci of 42 is 267914296
```

Main Points
Introducing NodeJS & Express
Do Less Accomplish More

1. NodeJS is a single-threaded server-side JavaScript platform. We use modules in Node to write testable and maintainable code. We should be careful not to have computationally intense code blocking the Node platform. Science and Technology of Consciousness: The Unified Field is the ultimate platform. It is possible to experience it by anyone through the regular practice of Transcendental Meditation. Also, the most complex expressions in life do not block any other aspect of nature.