

# CS 572 Modern Web Applications

Najeeb Najeeb, PhD ([najeeb@miu.edu](mailto:najeeb@miu.edu))

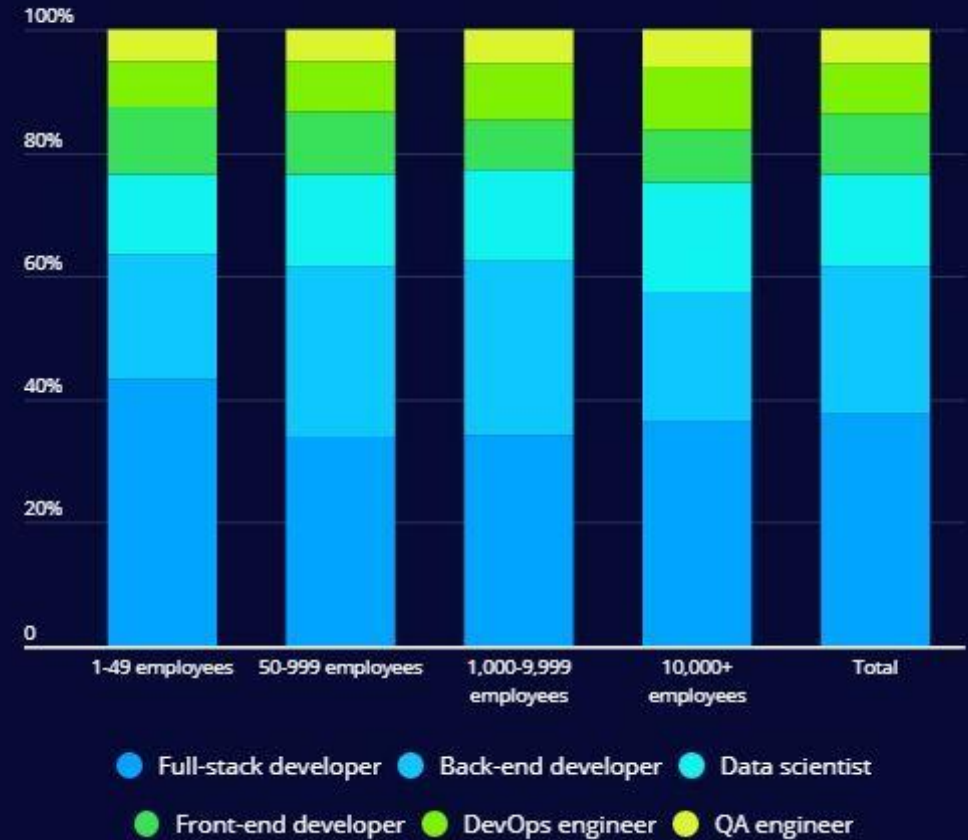
Copyright © 2022 Maharishi International  
University. All Rights Reserved.  
V3.0.0



# Why Full Stack Development?

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

What's the most important role you're looking to fill in 2020?



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](https://nodejs.org).
- Versions jumped from 0.x to 16.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](https://nodejs.org)

`node -v` (or `node --version`)

v16.15.0

Check node package manager (npm)

`npm -v`

8.5.5

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](https://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)):
  - Search 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

s

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

s

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.

# 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?



# http Module

# Modules

Written

Built-in

External



We can write our own modules, like talk module.

Built in modules are modules that come with Node and are available for use. A built-in module can be used by simply requiring the module using `"require(module_name)"`. The fs module is an example of built-in module.

External modules need to be downloaded then they can be used like built-in modules.

We will use the http Module to create a web application.

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

Is http a user, or built-in, or external module?

http

Setup

HelloWorld

HTML

JSON



app09.js file

```
const http= require("http");  
const server= http.createServer();  
server.listen(8080, "localhost", function() {  
    console.log("Server is running on  
http://localhost:8080");  
});
```

Run file

```
node app09.js
```

Server is running on http://localhost:8080

Open your browser and enter <http://localhost:8080>

Is the server running?

http

Setup

HelloWorld

HTML

JSON



app10.js file

```
...  
const helloWorld= function(req, res) {  
  res.writeHead(200);  
  res.end("Hello World!");  
}  
const server= http.createServer(helloWorld);  
...
```

Run file

```
node app10.js
```

Server is running on <http://localhost:8080>

Open your browser and enter <http://localhost:8080>

Do you get a response?



http

Setup

HelloWorld

HTML

JSON



app11.js file

```
...  
const helloWorldHtml= function(req, res) {  
  res.setHeader("Content-Type", "text/html");  
  res.writeHead(200);  
  res.end("<HTML><BODY><H1>Hello  
World!</H1></BODY></HTML>");  
}  
const server= http.createServer(helloWorld);  
...
```

Run file

```
node app11.js
```

Server is running on <http://localhost:8080>

Open your browser and enter <http://localhost:8080>

What is this response? How can you prove it?

http

Setup

HelloWorld

HTML

JSON



app12.js file

```
...  
const helloWorldJson= function(req, res) {  
  res.setHeader("Content-Type", "application/json");  
  res.writeHead(200);  
  res.end("{\"message\" : 'Hello World!' }");  
}  
const server= http.createServer(helloWorld);  
...
```

Run file

```
node app12.js
```

Server is running on <http://localhost:8080>

Open your browser and enter <http://localhost:8080>

What is this response? How can you prove it?

# Web App

## Serve Files

### Better Serve

### Error Handling

### Routing



Create index.html (an html file)

app13.js file

```
const http= require("http");
const fs= require("fs");
const readIndexAndServe= function(req, res) {
  fs.readFile(__dirname + "\\index.html", function(err, buffer) {
    res.setHeader("Content-Type", "text/html");
    res.writeHead(200);
    res.end(buffer);
  });
}
const server= http.createServer(readIndexAndServe);
```

Run file

```
node app13.js
```

Server is running on <http://localhost:8080>

Open your browser and enter <http://localhost:8080>

What type of server have we just created? Performance issue?

# Web App

## Serve Files

## Better Serve

## Error Handling

## Routing



app14.js file

```
const http= require("http");
const fs= require("fs");
let indexFileBuffer;
const serveIndex= function(req, res) {
  res.setHeader("Content-Type", "text/html");
  res.writeHead(200);
  res.end(indexFileBuffer);
}
fs.readFile(__dirname + "\\index.html", function(err, buffer){
  indexFileBuffer= buffer;
  server.listen(3000, "localhost", function(){
    console.log("Server is running on http://localhost:3000");
  });
});
const server= http.createServer(serveIndex);
```

Run file

```
node app14.js
```

Server is running on <http://localhost:8080>

Open your browser and enter <http://localhost:8080>

# Group Activity

## Error Handling & Routing

- Split up in groups of 4 (at least one student should speak a different language than the rest of the group, more is better). Then attempt to answer the questions assigned to your group. After 5 minutes make sure you have 4 answers to share with the class.
- Even Group Number
  - What kind of errors should be handled?
  - How do we deal with each error?
- Odd Group Number
  - What is routing? What do we mean by routing in a web application?
  - Give two examples of different routings.

# Students Answers

## ERROR HANDLING

- File Not Found
  - Handle provide error page
- File Too Large
  - Handle by reading in chunks (not error)
- Access denied
  - Do not handle (treat like File not found)

## ROUTING

- Routing is a way to present a response based on a client's request
- Examples:
  - lolcahost:3333\student
  - localhost:3333\student
    - POST

# Web App

## Serve Files

## Better Serve

## Error Handling

## Routing



app15.js file

```
...
let statusCode;
...
res.writeHead(statusCode);
...
fs.readFile(__dirname + "\\index.html", function(err, buffer) {
  if (err) {
    indexFileBuffer= "File not found";
    statusCode= 404;
  } else {
    indexFileBuffer= buffer;
    statusCode= 200;
  }
  server.listen(3000, "localhost", function() {
    console.log("Server is running on http://localhost:3000");
  });
});
```

Run file

```
node app15.js
```

Server is running on <http://localhost:8080>

Open your browser and enter <http://localhost:8080>

Change the name of inde.html and restart the application.

# Web App

## Serve Files

## Better Serve

## Error Handling

## Routing



```
app16.js file

const http= require("http");
const fs= require("fs");
const serveAllRequests= function(req, res) {
  switch(req.url) {
    case "/json" :
      res.setHeader("Content-Type", "application/json");
      res.writeHead(200);
      res.end("{\"message\" : 'Hello World!'");
      break;
    case "/":
      res.setHeader("Content-Type", "text/html");
      let statusCode;
      let fileBuffer;
      fs.readFile(__dirname + "\\index.html",
function(err, buffer) {
  if (err) {
    statusCode= 404;
    fileBuffer= "File not Found";
  } else {
    statusCode= 200;
    fileBuffer= buffer;
  }
  res.writeHead(statusCode);
  res.end(fileBuffer);
});
      break;
  }
}
```

```
const server=
http.createServer(serveAllReq
uests);
server.listen(3000, "localhost",
function() {
  console.log("Server is
running on
http://localhost:3000");
});
```

...

Run file

```
node app16.js
```

Server is running on  
<http://localhost:8080>

Open your browser and enter  
<http://localhost:8080>



# 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.
2. We can write web applications with NodeJS using the built-in http module. The http module abstracts the web protocols and networking issues, enabling us to create a request-response-based application (page serving and routing). Science and Technology of Consciousness: The laws of nature govern all aspects of our life. We do not need to know all the details of all the laws to be able to gain the benefits. We focus on what is important to us and gain benefits. Water the root and enjoy the fruit.



Express

# Introducing Node & Express

## Do Less Accomplish More

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

# Node Package Management (npm)

- Define and manage dependencies using npm.
- Using packages enables code reuse, and not writing things from scratch.
- Move code around and use latest versions of dependencies.

# Using npm

- Creating package.json can be done with `npm init`
- Follow the steps npm gives you.
- Entry point: this is the file that will contain the application starting point (the file to run).
  - We use (app.js)
- This creates package.json having all the information you provided.
- Use it to add dependencies, installing packages, development vs testing dependencies, run scripts.
- Ignoring dependencies when uploading to git.

**npm**

Create

Add

Development

Install

Scripts



How to create package.json file

```
npm init
```

```
package name: (app17)
```

```
version: (1.0.0)
```

```
description: This is my first npm project
```

```
entry point: (index.js) app17.js
```

```
test command:
```

```
git repository:
```

```
keywords: MEAN
```

```
author: Najeeb Najeeb
```

```
license: (ISC)
```

```
Is this OK? (yes)
```

```
npm create package.json
```

```
package.json
```

**npm**

Create

Add

Development

Install

Scripts



Add dependency on Express (using npm command line)

```
npm install express
```

```
+ express@4.17.3
```

npm added express to package.json

```
ls or dir
```

```
node_modules
```

```
"license": "ISC",  
"dependencies": {  
  "express": "^4.17.3"  
}
```

**npm**

Create

Add

Development

Install

Scripts



Add dependency on Express (using npm command line)

```
npm install mocha --save-dev  
+ express@9.2.2
```

npm added mocha to package.json

...

```
"license": "ISC",  
"dependencies": {  
  "express": "^4.17.3"  
},  
"devDependencies": {  
  "mocha": "^9.2.2"  
}
```

^x.y.z: use x major and the latest minor and patch.



**npm**

Create

Add

Development

Install

Scripts



Dependencies are not uploaded to git

Dependencies should be installed after fetching code from git

```
npm install
```

Install only production dependencies (on production server)

```
npm install --production
```

Create readme.md

```
"This repo contains the MEAN stack application that is built in  
CS572 Modern Web Applications course."
```

Ignore node\_modules when pushing to git.

Create .gitignore file and fill it with

```
node_modules
```

**npm**

Create

Add

Development

Install

Scripts



Start script; shortcut to start your application.

```
"scripts": {  
  "start": "node app17.js",  
  "test": "echo \"Error: no test specified\" && exit 1"  
}
```

Create file app17.js

To start the application:

```
npm start
```

```
> app17@1.0.0 start
```

```
> node app17.js
```

# What is Express

- Web framework for MEAN stack.
- Listen to incoming requests and respond.
- Deliver static html files.
- Compile and deliver html.
- Return JSON.

# Express Application

- Add dependency on Express.
- Require Express.
- Listen to requests (port) at URLs.
- Return HTTP status codes.
- Response HTML or JSON.

**Express**

Add

Listen

Application

Variables

Callback



Create package.json

```
npm init
```

Add dependency on Express (using npm command line)

```
npm install express
```

app18.js file

```
const express= require("express");  
const app= express();
```

Run the application:

```
npm start
```

The server terminates before we send a request!

# Express

Add

Listen

Application

Variables

Callback



app18.js file

```
const express= require("express");  
const app= express();  
app.listen(3000); // Hardcoded more than one place :(  
console.log("Listening to port 3000"); // Another place :(
```

Run the application

```
npm start
```

Check the browser (<http://localhost:3000>)

Nothing interesting, but we do have a server.

# Express

## Add Listen Application Variables Callback



app19.js file

```
const express= require("express");  
const app= express();  
app.set("port", 3000); // In one place  
app.listen(app.get("port");  
console.log("Listening to port "+ app.get("port");
```

Run the application

```
npm start
```

Check the browser (<http://localhost:3000>)

Same results but better software engineering, right? Why?

# Express

## Add Listen Application Variables Callback



app20.js file

```
const express= require("express");  
const app= express();  
app.set("port", 3000);  
const server= app.listen(app.get("port"), function(){  
  const port= server.address().port; // Get port from app  
  console.log("Listening to port "+ port);  
});
```

Run the application

`npm start`

Check the browser (<http://localhost:3000>)

Is this really a callback?



# Environment Variables

- Hard coding values in code is bad.
  - Why?
- Best to have constants outside the program.
  - In Java we use property files.
  - In NodeJS we use environment variables.
- To read environment variables we use a package (dotenv).

**dotenv**

Install

Add

Use

Install dotenv package

```
npm install dotenv
```

```
+ dotenv@16.0.0
```



# dotenv

Install

Add

Use

app21.js file

```
require("dotenv").config();  
const express= require("express");  
...
```

Create file .env and fill it with

```
PORT = 3000
```



# dotenv

## Install

## Add

## Use



app21.js file

```
require("dotenv").config();  
const express= require("express");  
const app= express();  
app.set("port", process.env.PORT);  
const server= app.listen(app.get("port"), function() {  
  console.log("Listening to port", server.address().port);  
});
```

Now you can change the port number outside of your program.

Don't forget to exclude the file from git (for security)

Update .gitignore (good idea to add lock file)

```
.env  
package-lock.json
```

# Routing using Express

- Routing is listening to requests on certain URLs and doing something on the server side then sending a response back.
- Route definition
  - HTTP method
  - Path
  - Function to run when route is matched

# Routing

Define

HTTP Status

Data Response

File Response



app22.js file

```
require("dotenv").config();  
const express= require("express");  
const app= express();  
app.get("/", function(req, res) {  
  console.log("GET received");  
});  
const server= app.listen(process.env.PORT, function() {  
  const port= server.address().port();  
  console.log(process.env.MSG_SERVER_START, port);  
});
```

Run the application

`npm start`

Check the browser (<http://localhost:3000>)

Are you getting a response? Is the server getting the request?

# Routing

Define

HTTP Status

Data Response

File Response



app22.js file

```
require("dotenv").config();  
const express= require("express");  
const app= express();  
app.get("/", function(req, res) {  
  console.log("GET received");  
  res.send("Received your GET request.");  
});  
const server= app.listen(process.env.PORT, function() {  
  const port= server.address().port();  
  console.log(process.env.MSG_SERVER_START, port);  
});
```

Run the application

`npm start`

Check the browser (<http://localhost:3000>)

# Routing

Define

HTTP Status

Data Response

File Response



app22.js file

```
require("dotenv").config();
const express= require("express");
const app= express();
app.get("/", function(req, res) {
  console.log("GET received");
  res.status(404).send("Received your GET request.");
});
const server= app.listen(process.env.PORT, function() {
  const port= server.address().port();
  console.log(process.env.MSG_SERVER_START, port);
});
```

Run the application

`npm start`

Check the browser (<http://localhost:3000>)



# Routing

Define

HTTP Status

Data Response

File Response



app22.js file

...

```
app.get("/json", function(req, res) {  
  console.log("JSON request received");  
  res.status(200).json({"JSON_Data": true});  
})
```

...

Run the application

`npm start`

Check the browser (<http://localhost:3000/json>)

# Routing

Define

HTTP Status

Data Response

File Response



app22.js file

```
const path= require("path");  
...  
app.get("/file", function(req, res) {  
  console.log("File request received");  
  res.status(200).sendFile(path.join(__dirname,  
  "app22.js"));  
});  
...
```

Run the application

```
npm start
```

Check the browser (<http://localhost:3000/file>)

# Express Serving Static Files

- Applications require foundations
  - HTML pages
  - CSS files
  - Images
  - JS libraries
- Easier to deliver static pages through Express directly.

# Static Pages

Folder

Subset of routes

CSS

JQuery

IMG



app23.js file, after port definition and before routes we define the static folder (introduce middleware)

```
app.use(express.static(path.join(__dirname, "public")));
```

Create a public folder and add index.html into it.

Run the application

```
npm start
```

Check the browser (<http://localhost:3000/index.html>)

Check the browser (<http://localhost:3000>)

**Static Pages**

**Folder**

Subset of routes

CSS

JQuery

IMG

app23.js file

```
app.use("/public", express.static(path.join(__dirname,  
"public")));
```

Run the application

```
npm start
```

Check the browser

(<http://localhost:3000/public/index.html>)



**Static Pages**

**Folder**

Subset of routes

CSS

jQuery

IMG



CSS bootstrap Theme available from  
[www.bootswatch.com/superhero](http://www.bootswatch.com/superhero) (bootstrap.min.css)

Add the downloaded file to /public/css folder

Link CSS file to index.html file header section

```
<link href="css/bootstrap.min.css" rel="stylesheet" />
```

Run the application

```
npm start
```

Check the browser (<http://localhost:3000>)

**Static Pages**

**Folder**

Subset of routes

CSS

jQuery

IMG

jQuery from [www.jquery.com/download/](http://www.jquery.com/download/) (jquery-3.5.1.min.js)

Reference jquery in the page

```
<script src="jquery/jquery-3.5.1.min.js"/>
```

Run the application

```
npm start
```

Check the browser (<http://localhost:3000>)



# Static Pages

## Folder

Subset of routes

CSS

JQuery

IMG



Create images folder

Go to (<https://compro.miu.edu/>) and obtain a copy of MIU logo, copy the image to the images folder

Add image to index.html

```
<a href="https://compro.miu.edu" target="_blank"></a>
```

Run the application

```
npm start
```

Check the browser (<http://localhost:3000>)



# In class Exercise

## Start Creating MEAN Games

Based on what we learned so far.

- Create a MEAN Games application (using npm init)
- Add dotenv
- Add Express
- Create a homepage (index.html), CSS, and image
  - Find index.html (in the resources folder)
  - Find custom.css (in the resources folder)

# Express & Middleware

- What is middleware?
- Create logging function
- When and how to use middleware

# Express & Middleware

- Example: `app.use`
  - Interact with request before response
  - Make the response, or passes it through
- Define a function that will process something in the request, do something, then follow through to the response.
- Order is important, they will run in the order defined.

# Middleware

## Log Requests

### Order

### Subsets



app27.js file, middleware (explicit), make sure this is the first app.use function

```
app.use(function(req, res, next) {  
  console.log(req.method, req.url);  
  next();  
});
```

Run the application

```
npm start
```

Check the browser (<http://localhost:3000/>)

GET /

GET /css/bootstrap.min.css

GET /css/custom.css

GET /images/xompro-web-logo-442x112.webp

GET /favicon.ico

# Middleware

Log Requests

Order

Subsets



Switch the order of middleware and observe the effect.

```
app.use(express.static(path.join(__dirname, process.env.PUBLIC_FOLDER)));
```

```
app.use(function(req, res, next) {  
  console.log(req.method, req.url);  
  next();  
});
```

Run the application

```
npm start
```

Check the browser (<http://localhost:3000/>)

Why are we not seeing the logs even though we have a logging middleware?

# Middleware

Log Requests

Order

Subsets



Middleware for only paths starting with "css"

```
app.use("/css", function(req, res, next) {  
  console.log(req.method, req.url);  
  next();  
});
```

Run the application

```
npm start
```

Check the browser (<http://localhost:3000/>)

GET /bootstrap.min.css

GET /custom.css

# Express Application

- So far do you find any problems with this code?
- Hint
  - Where is the setup code (port, logging)?
  - Where is the routing code?
- Why is this problematic?

# Express Router

- Separation of concerns
- Instantiating the router
- Applying router to subset of routes
- Testing routes using REST plugins



# Router

Separate routes

Subset routes

REST Test



app28.js file, this is what we have (everything in one place)

```
require("dotenv").config();
const path= require("path");
const express= require("express");
const app= express();
app.use(function(req, res, next) {
  console.log(req.method, req.url);
  next();
});
app.use(express.static(path.join(__dirname, process.env.PUBLIC_FOLDER)));
app.get("/json", function(req, res) {
  console.log("JSON request received");
  res.status(200).json({"JSON_Data": true});
});
const server= app.listen(process.env.PORT, function() {
  console.log(process.env.MSG_SERVER_START, server.address().port);
});
```

# Router

Separate routes

Subset routes

REST Test



Create routes folder, and inside it index.js

```
const express= require("express");
const router= express.Router();
router.route("/json")
  .get(function(req, res) {
    console.log("GET JSON request received");
    res.status(200).json({"JSON_Data": "GET"});
  })
  .post(function(req, res) {
    console.log("POST JSON request");
    res.status(200).json({"JSON_Data": "POST"});
  });
module.exports = router;
```

app28.js file

```
require("dotenv").config();
const path= require("path");
const express= require("express");
const routes= require("./routes");
const app= express();
app.use(function(req, res, next) {
  console.log(req.method, req.url);
  next();
});
app.use(express.static(path.join(__dirname, process.env.PUBLIC_FOLDER)));
app.use("/", routes);
const server= app.listen(process.env.PORT, function() {
  console.log(process.env.MSG_SERVER_START , server.address().port);
});
```

# Router

Separate routes

Subset routes

REST Test



Create routes folder, and inside it index.js

```
const express= require("express");
const router= express.Router();
router.route("/json")
  .get(function(req, res) {
    console.log("GET JSON request received");
    res.status(200).json({"JSON_Data": "GET"});
  })
  .post(function(req, res) {
    console.log("POST JSON request");
    res.status(200).json({"JSON_Data": "POST"});
  });
module.exports = router;
```

app29.js file

```
require("dotenv").config();
const path= require("path");
const express= require("express");
const routes= require("./routes");
const app= express();
app.use(function(req, res, next) {
  console.log(req.method, req.url);
  next();
});
app.use(express.static(path.join(__dirname, process.env.PUBLIC_FOLDER)));
app.use("/api", routes);
const server= app.listen(process.env.PORT, function() {
  console.log(process.env.MSG_SERVER_START, server.address().port);
});
```

# Router

Separate routes

Subset routes

REST Test

Add a Chrome REST extension

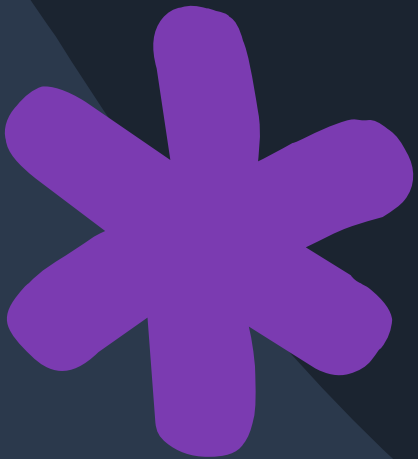
I picked "Advanced REST client" SOAP & REST Client

Make GET request from browser (<http://localhost:3000/>)

Use URL: <http://localhost:3000/api/json>

Make GET request from REST Client

Make POST request from REST Client



# Express Router

- Keep app.js clean and clear
  - Easy to read and understand
  - Easy to maintain and debug
- Don't put too much code of different types in one file.
- Move different codes to different places and keep code separate.

# Express Controller

- Separation of Concerns
- Creating API (REST API)
- What are controllers and their functionality
  - Controls what happens when a route is visited
  - Separate logic from routing from UI code
- Map controllers to routes

# Controller Setup Static Data



Create api folder, move routes folder inside it.

api/routes/index.js file

```
const express= require("express");  
const router= express.Router();  
const gamesController= require("../controllers/games.controllers");  
router.route("/games")  
  .get(gamesController.getAll);  
module.exports = router;
```

api/controllers/games.controllers.js

```
module.exports.getAll= function(req, res) {  
  console.log("JSON GET received");  
  res.status(200).json({"jsonData": "GET"});  
};
```

Run the application

`npm start`

Check the browser (<http://localhost:3000/api/games>)

GET api/games  
JSON GET received

# Controller Setup Static Data



Create data folder inside api, create json data file.

Get games.json file from Sakai (JSON representation of DB)

games.controllers.js

```
const gamesData= require("../data/games.json");  
module.exports.gamesGetAll= function(req, res) {  
  console.log("GET all games");  
  res.status(200).json(gamesData);  
  
}
```

Run the application

```
npm start
```

Check the browser (<http://localhost:3000/api/games>)

GET api/games

GET all games



# URL parameters in Express

- What are URL parameters?
  - How can you get information about one game?
    - You need to know the game of interest (user input).
  - Get user input through the URL (localhost:3000/api/games/2021).
    - Create a route for each id? :(
    - Parametrize it :)
- How to define URL parameters in routes.
  - `.route("/games/:gameId")`
- Use URL parameters in controllers.

URL  
parameter  
Router  
Controller

app32 define the route and connect to controller

api/routes/index.js add

...

```
router.route("/games/:gameId")  
  .get(gamesController.getOne);
```

...



# URL parameter Router Controller



app32 define the controller and run  
api/controllers/games.controllers.js add

```
...  
module.exports.getOne= function(req, res) {  
  const gameId= req.params.gameId;  
  const theGame= gameData[gameId];  
  console.log("GET game with gameId ", gameId);  
  res.status(200).json(theGame);  
};
```

Run the application

```
npm start
```

Check the browser (<http://localhost:3000/api/games/3>)

GET api/games/3

GET game with gameId 3

# Other Ways to get Input

- How to pass data from client to server?
  - URL parameter (Express native support)
  - Query string (GET method, Express native support)
  - Form body (POST method, Express no native support)
- Getting queryString data in Express controllers.
- Middleware for parsing forms.
- Getting form data in Express controllers.

# Client Data

## Query string

## Form data



Get certain number of games, for pagination, start from an offset and get a certain number of games

Browser (<http://localhost:3000/api/games?offset=3&count=2>)

Games.controller.js

```
module.exports.getAll= function(req, res){
  console.log("GET all games");
  console.log(req.query);
  let offset= 0;
  let count= 5;
  if (req.query && req.query.offset) {
    offset= parseInt(req.query.offset, 10);
  }
  if (req.query && req.query.count) {
    count= parseInt(req.query.count, 10);
  }
  const pageGames= gamesData.slice(offset, offset+count);
  res.status(200).json(pageGames);
};
```

Run the application

`npm start`

Check the browser (<http://localhost:3000/api/games?offset=3&count=2>)

GET /api/games?count=2&offset=3

GET all games

{ count: '2', offset: '3' }

# Client Data

## Query string

## Form data



Form body parsing is not natively supported by Express. We need to activate the parse of a form body.

Add new route, api/routes/index.js

```
router.route("/games")
  .get(gamesController.getAll)
  .post(gamesController.addOne);
```

Add the controller, api/controllers/gamesController.js

```
module.exports.addOne= function(req, res) {
  console.log("POST new game");
  console.log(req.body);
  res.status(200).json(req.body);
}
```

app34.js add the followings

```
...
app.use(express.static(path.join(__dirname, process.env.PUBLIC_FOLDER)));
app.use(express.json());
app.use(express.urlencoded({extended: true}));
...
```

Use ARC (<http://localhost:3000/api/games> POST)

# Nodemon

- Development tool, not for production system.
- Improve development experience and provide information.
- Install Nodemon globally (not related to an application).
- Use Nodemon.
- Configure Nodemon.

# Nodemon

Install

Run

Configure

Code and tests without having to always stop and start application.

Install nodemon

```
sudo npm install --g nodemon
```





# Nodemon

Install

Run

Configure

Run nodemon, run the start command in package.json

`nodemon`

Change something in app34.js and notice how nodemon restarts the application.



# Nodemon

Install

Run

Configure



Nodemon monitors everything, including static files. But we want them served as is.

Configure nodemon to ignore changes made in the public directory.

Create nodemon.json

```
{  
  "ignore" : ["public/*"],  
  "verbose" : true  
}
```

Change something in public folder and see how nodemon doesn't restarts the application.

Shows the file that triggered the change.

# 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.
2. We can write web applications with NodeJS using the built-in http module. The http module abstracts the web protocols and networking issues, enabling us to create a request-response-based application (page serving and routing). Science and Technology of Consciousness: The laws of nature govern all aspects of our life. We do not need to know all the details of all the laws to be able to gain the benefits. We focus on what is important to us and gain benefits. Water the root and enjoy the fruit.
3. When using the http module, if we are not careful, we could end up with a maintenance nightmare. Express enables us to easily implement separation of concerns (MVC), separating the routes from the application setup from the business logic (controllers). Routes and Controllers enable easier understanding and debugging of applications. Science and Technology of Consciousness: Trying to solve all the problems of life one by one is hard and very time-consuming (if not impossible). By focusing on the most important aspects of life first (Highest First Principle) we gain all the benefits, and everything else naturally falls into place.