

Week 1 Workshop

NodeJS Express MongoDB





Agenda

Activity	Time
Get Prepared: Log in to Nucamp Learning Portal • Slack • Screenshare	10 minutes
Introductions & Check-In	10 minutes
Week Recap	60 minutes
Task 1 & 2	40 minutes
BREAK	15 minutes
Tasks 2 & 3	90 minutes
Check-Out	15 minutes



Introductions & Check-In

- Instructor introduction (and Instructor Lead, if present)
- Student introductions – students should know each other already, but please introduce yourself briefly to your instructor if you have not met.
- Check-In:
 - How was this week? Any particular challenges or accomplishments?
 - Did you understand the Exercises and were you able to complete them?
 - You must complete all Exercises before beginning the Workshop Assignment.



Welcome to Node, Express, MongoDB!

Week 1 Recap: New Concepts This Week

- | | |
|--|---|
| <ul style="list-style-type: none">• Node Modules<ul style="list-style-type: none">• Three types• <code>Module.exports/require</code>• Asynchronous Computation with Callbacks and Closures• Node Event Loop• Error Handling with Callbacks | <ul style="list-style-type: none">• Review: Networking & REST• Node HTTP module• Express Server Framework• NPM Packages<ul style="list-style-type: none">• Versioning• <code>Package.json</code>• Express Router |
|--|---|

Next slides will review these concepts



Node Modules

- Discuss (ask for a student or multiple students to answer this question): What are the **three types of Node modules** you learned about this week?
- Name them, and also discuss any details you remember about what you learned about each type.



Node Modules

Answer:

- 1. Node's core modules
 - Built into the Node binaries, do not need to be installed
 - Import them when needed with the CommonJS-style `require()` function or ES6 `import`, use module name without a path, e.g. `const http = require('http');`
 - By the way, it is not a requirement to name the const the same as the module name, it's just common convention and you should typically follow it, but it would still work if you used: `const somethingElse = require('http');`
 - Intentionally kept small to keep Node small & to encourage third-party innovation
 - Examples include: `fs` (filesystem), `http`, `path`
 - Can anyone name other core modules and what they're used for?

(continued next slide)



Node Modules

Answer (continued):

- 2. External third-party modules
 - Typically installed using **npm install** (or another Node package manager e.g. yarn/homebrew)
 - Import using `require` or ES `import`, use the module name without path, e.g. **`const express = require('express');`**
 - Again, you're not required to name the `const` exactly the same as the module name. They usually are with core modules, but you'll see more variance with third-party modules. **Ex.: `const app = express();`**
 - Installed into **node_modules** as packages (the package includes not just the modules but folders, `readme.md`, `package.json`, etc)
 - Come with **package.json** manifest files with information on version, dependencies, etc

(continued next slide)



Node Modules

Answer (continued):

- 3. File-based modules within your own application
 - You create these modules by exporting resources from them (functions or variables) which can then be used in other files
 - Export with **module.exports =** syntax
 - **exports** shorthand: if you are exporting a property/method into module.exports and not trying to assign to the entire module.exports object, you can use the exports shorthand, e.g.: **exports.perimeter() =** or **exports.name =**
 - Use in other files in your project with require() or import, you *do* need to give a path to the file, e.g. **const campgroundRouter = require('./routes/campsiteRouter');**



JavaScript Function Concepts

- **First-Class Functions**

- - A programming language has "first-class functions" when it supports treating functions like any other variable – JavaScript has first-class functions
- - With first-class functions, functions can be assigned to variables, passed around as arguments to other functions, used as the return value of another function

- **Higher-Order Function**

- A function that takes another function as an argument, or returns a function as its return value

- **Callback Function**

- A function that is passed to another function as an argument which is then run (called back) inside that function, often asynchronously



Closures

- Refers to the concept that an inner function has access to its enclosing scope
- When a function is defined inside another function, the inner function automatically gets access to the variables in the outer function – even if the inner function is called after the outer function has already completed.
- This allows asynchronous callbacks to work without losing the scope in which they were initially created.



Node.js Asynchronous I/O Handling

- Node is designed to use a **non-blocking, asynchronous I/O model** even though it runs on JavaScript, which is single-threaded
- It accomplishes this by handing off expensive I/O operations to the multi-threaded system kernel to complete without blocking Node's single threaded operations
- Then the kernel lets Node know when an operation is completed, and Node uses callbacks and the event loop to pick up where it left off



Node.js Event Loop

- Six Phases of the Node Event Loop:
 - Timer
 - Pending Callbacks
 - Idle, Prepare
 - Poll
 - Check
 - Close Callbacks
- Typically only the **Timer**, **Poll**, and **Check** phases will be relevant to a Node developer – the rest are handled by Node in the background



Node.js Event Loop

- **Timer phase**: Handles callbacks from `setTimeout()` and `setInterval()`
- **Poll phase**: Processes I/O callbacks in a queue, waits for more callbacks
- **Check phase**: Handles callbacks from `setImmediate()` which are run as soon as poll phase's callbacks queue is empty



Node Callback Pattern & Error Callback Convention

Node community uses this callback pattern for asynchronous functions:

```
function asyncOperation(a, b, c, callback) {  
  // ... lots of hard work ...  
  if ( /* an error occurs */ ) {  
    return callback(new Error("An error has occurred"));  
  }  
  // ... more work ...  
  callback(null, returnValues);  
}  
  
asyncOperation(argA, argB, argC, (err, returnValues) => {  
  // Code in this callback runs -after- asyncOperation function runs  
});|
```

An error object is used for first argument in callback, then returned with valid error object if an error occurred in the function, or returned as null if no error



Node Callback Pattern & Error Callback Convention

- Expect to use the same error callback convention when writing your own asynchronous functions
- Expect that other Node modules written by other people will use the same convention
- If you don't fully understand how it works/why this particular pattern is used yet, be patient – the more you work with Node, the more you will be exposed to this pattern, and the more it will sink in



Node Core Modules: http

- Once we start using Express, we will typically not use http core module – but it's good to be familiar with it as Express uses it under the hood
- `http.createServer()` is used to instantiate (create an instance of) a server object of the `http.Server` class built into Node, this object is able to handle fundamental low-level server operations
- `createServer()` requires a parameter: a callback function called a **request handler** - this function is run every time a client makes a server request, it handles parsing the request and sending the response
- *(cont next slide)*



Node Core Modules: http request handler

- The request handler callback function takes a request and response object as parameters – usually shortened to `req` and `res`
- These objects are a special type of object in Node called **Streams**, they represent data transmitted in smaller chunks rather than all at once – the request stream and response stream
- The request stream contains information such as request **headers**, **body**, etc
- Response stream also contains headers, body, as well as **statusCode** (404, 200, etc), you can use `setHeaders()` to set response headers
- To set response body: `res.write()`, or you can include the body as an argument to `res.end()`
- Use `res.end()` at the end of the response to close the response stream



Node Core Modules: http

- Once a server is created with `http.createServer()`, you can start it listening with the `.listen()` method, optionally providing a port and/or hostname
 - e.g. `server.listen(port, hostname)`



Node Core Modules: path, fs

- **path**: utilities for working with file and directory paths
- **fs**: utilities for interacting with local filesystem
- You can expect to work with these core modules often
- Some other core modules include:
 - **process**: provides information about, and control over, the current Node.js process. Process is a **global** module – you do not need to require it, you can use it anywhere.
 - **url**: provides utilities for URL resolution and parsing
 - **os**: provides operating system-related utility methods and properties



NPM Packages

- Semantic Versioning: Major Version.Minor Version.Patch
- Major versions often have breaking changes. Minor versions typically don't, but you never know. Same with patches.
- Use `npm install somePackage@~#.#.#` if you don't mind a higher patch than what you've specified being installed
- Use `npm install somePackage@^#.#.#` if you don't mind a higher patch or minor version than what you've specified being installed
- Use `npm install` on its own in a folder with a `package.json` or `package-lock.json` file and it will install all modules listed as dependencies or devDependencies in that file
 - This will default to using `package-lock.json` file if it exists, because it contains the dependency versions from the last time the `package.json` file or `node_modules` was updated for that project, useful in cases there are version conflicts



Express

- Express is a "Fast, unopinionated, minimalist web framework for [Node.js](https://nodejs.org/)" - de facto standard server framework for Node applications
- When you hear about the **MEAN** stack or **MERN** stack for web development, Express is the E in that acronym – that's how popular it is (MEAN is MongoDB Express Angular Node, MERN is MongoDB Express React Node)
- Example:

```
const express = require('express');  
const app = express();  
app.listen();
```

- The `app.listen()` method will both create an Express server and start listening on the default host/port for it, it combines `http.createServer().listen()` in one method



Express Middleware

- The core Express framework has a minimalist design – you're meant to extend it for your specific needs using the many available middleware libraries, both built into Express (such as `express.static` for handling static files) and third-party libraries you need to install (such as `morgan`)
- Install and require middleware as necessary, then use the `.use()` method to add middleware functions to your Express app, e.g.:

```
const express = require('express');
const morgan = require('morgan');

const app = express();

app.use(morgan('dev'));
app.use(express.json());
app.use(express.static(__dirname + '/public'));
```

- Notice in the above examples, you did *not* need to require `express.static` before using it, because it's a middleware built into Express



Express Routing Methods

- Express has routing methods for each HTTP verb such as **GET/POST/PUT**, as `.get()`, `post()`, `put()`, etc
- Each method takes a **path** and a **callback function** – Express calls it a "handler", more or less what Node docs call a "request handler" -- it handles the request that comes in via that particular HTTP verb
- Takes three arguments: **req**, **res**, and optionally, **next()** - a function that you can use to pass control to the next appropriate routing method
 - E.g. as you did in your exercises, when you set a default statusCode and headers for the response using the `.all()` method then passed control to the next routing method using `next()`
- **Endpoint**: Combination of an HTTP verb plus a resource location (path or URL) - a single path (such as `/campsites`) can have multiple endpoints on it (multiple points to which a server request can end up)



Express Route Parameters

- To use a route parameter in a routing method, you can use a **colon** in a path, followed by a string, similar to what you've done before in React
- Express will take any server request that matches that pattern, grab whatever string that the client sends in the same location as the **:<string>** in your route, and save that string to **req.params.<string>**
- e.g. if you have a routing method that uses the path **'/campsites/:campsiteld'** and a client sends in a request to **'/campsites/23'**, then inside that routing method's handler, **req.params.campsiteld** will be set to the string **'23'**
- If the client sent a request to **/campsites/foo**, then **req.params.campsiteld** would be set to **'foo'**.
 - You can test this with Postman and the Part 1 version of your server.js file from the Express Router exercise. Try sending a GET request from Postman to **localhost:3000/campsites/foo** and see what happens.



Express Router

- The Express Router is a built-in tool in Express
- Create an Express router using the `createRouter()` method
- Functions as a mini Express app that is focused on routing and can use middleware, has access to the `.use()` method
- The router itself functions as a middleware, so you can include it in your main Express app via `.use()`, providing a path for its root, e.g.:
 - `app.use('/campsites', campsiteRouter);`
- Helps you divide your routing into separate modules from which you can export the router then require/import it into your main app, easier to manage when you have many endpoints



Workshop Assignment

- It's time to start the workshop assignment!
- Break out into groups of 2-3. Sit near your workshop partner(s).
 - Your instructor may assign partners, or have you choose.
- Work closely with each other.
 - 10-minute rule does *not* apply to talking to your partner(s). You should consult each other throughout.
- Follow the workshop instructions very closely.
 - Both the video and written instructions. Pay careful attention to any screenshots in the written instructions.
- Talk to your instructor if any of the instructions are unclear to you.



Check-Out

- Submit to the learning portal one of the following options:
 - Either: a zip file of your entire **node-express** folder with your updated files, *excluding the node_modules* folder,
 - Or: a text file that contains the link to a public online Git repository for the **node-express** folder.
- Wrap up – Retrospective
 - What went well
 - What could improve
 - Action items
- Start Week 2 or work on your Portfolio Project.
- If everyone is done early, then take time to go over the Code Challenges and Challenge Questions from this week – for each one, a student volunteer who has completed the challenge may explain their answer to the class.