**Section 1 – Creating web servers with Node.js & Express**

* MongoDB stores JSON docs, so easier to use JavaScript without having to translate any information.
* Node is a single threaded. Fast and scalable. Helped JS mature into a robust language.

**Lecture 4**

* Splitting Node applications into multiple files
* Using require to load in files
  + require(‘ ‘);
* Exposing functionality using **module.exports**
  + module.exports = function ( ) { }
  + Then, require it in app.js and store it in a variable, so it can be called on demand.
  + module.exports can also expose select methods by declaring it as an object which will ultimately return those methods to be used elsewhere. Call them using dot notation. Similar to javascript **return** functionality.
  + We can also chain methods directly to module.exports using dot notation, which can utilize private variables and functions within the given file.
    - e.g., module.exports.ask = function(question) { console.log(question); return answer };
* **index.js**
  + **Index.js** is a special file name.
  + If you have an index.js file in a folder, you don’t need to specify the name of the file itself, just the name of the folder.
  + Require it in app.js, store it in a variable. It will look for a file specified in the same folder of app.js, and if it doesn’t find that, it will look for a folder of that name, and inside that a file called ‘index.js’.
  + One of the reasons why its generally best practice to forego .js at the end of the require statement.
  + If, in the future, a file gets big enough to where it warrents its own folder, we don’t need to remove the .js from the require statement.

**Lecture 5**

* **Node.js** is single-threaded, fast, and scalable. It helped JS mature into a robust language.
* Node.jsis designed to address I/O scalability, *not* computational scalability.
  + **I/O** – interacting with databases, reading/writing files. Single threaded, so when we start a web server, theres only **one process** that deals with one request from all visitors. *So it is crucial that I/O doesn’t slow anyone down.*
  + **Computational scalability** – If app has very complicated functions/uses a lot of resources, we need to find out how to run these functions asynchronously so they don’t slow anyone down. Node has ways that allows us to make the most of this process so as to not slow down the main process.
* Why Node needs **asynchronous operations**:
  + Node has lots of asynchronous methods to prevent the main single thread to be blocked from I/O operations. When these asynchronous methods complete, they call callbacks.
  + To deal with computationally heavy functions, we use child processes as well as the appropriate hardware.
* **fs** – special module in Node that gives access to the file system.
  + readFileSync – method that reads a file synchronously.
  + readFile(‘file’, function(err, file) { } – method that reads a file asynchronously. It takes a callback so it does not block the main process. By giving the callback a name, is easier to read and easier to test, since you cannot do unit testing on an anonymous function.
* **Computational Blocker**
  + A process where the code takes a while to execute and can block the main process.
  + To avoid this we can use the **child\_process** module along with the **spawn** method.
  + Ultimately this will address heavy computational processes by spawning them into separate node processes so that we don’t block the main single thread. *Not blocking the main process is* ***key*** *to good Node development. If you delay one visitor, you delay ALL the visitors.*

**Lecture 6**

* **npm** – a package manager for Node.
  + Use tested and proven code from others in the Node community.
  + Allows the easy movement of code between environments. E.g., development to production.
* **package.json**
  + main point is to manage dependencies.
* **version control**
  + ^4.13.3 (major.minor.patch). ^ shows that it has installed the most recent version of the package.
* **‘npm install**’
  + Command to install npm packages.
  + ‘npm install mocha --save-dev‘ / ‘npm install mocha --save-production’ – installs dependency in “dev” or “production” dependencies.
  + If we *only* want to install either dev or production dependencies, use ‘npm install –production’ / ‘npm install --dev'.
  + It’s generally the best idea to let npm manage installing packages because, especially when moving from one operating to another, the folder structures may be different. npm will handle these changes consistently and accurately. e.g., Unix based environments to Windows.

**MEAN Application**

* **Express** : The web application framework in the MEAN stack. Listens for requests and responds. Can serve static files, compiles and deliver html, return json data.
* **Paramaterized Routes:** 
  + In routes (index.js), define a parameterized route like this:
    - /api/hotels/:hotelId
  + hotelId parameter then be accessed in controller via the request’s params object using req.params.hotelId.
* **Query Strings:** 
  + Often used when forms have a GET method. Get form info and send it as query strings on the URL.
  + By adding an query of offset and count, we can allow paginate our data to only return a certain amount of hotels at a time.
  + **Count** – number of hotels we want to return at once.
  + **Offset** – starting position of json data.
  + e.g., /api/hotels?offset=2&count=2
* **Body-parser:** 
  + Deals with **POST** requests of forms.
  + When a form is posted, the field is added on the body of the request.
  + Unlike query strings, Express can’t deal with posted form natively, so we must install body-parser middlewear.
  + ‘urlencoded’ - method by which html forms are sent. extended: false - only need strings and arrays from our form body. If true, we can have access to other data types, but generally not useful.
  + Body Parser will store form data that it parses out onto **req.body**. To view this we can use a console log and to test it we can use Postman.

**Relatonal vs. Non-Relational Databases**

* ***Relational****/****SQL***:
  + Scales **vertically**.
  + Every different type of information will be held in a different table.
    - E.g., Posts, Comments, Users, Tags, Categories
  + Uses a row/column structure. Creates a placeholder for every possible data type regardless of whether or not the information exists.
  + *Highly structured*. Must define exact schema before writing data into it.
  + E.g., in a comment section, you would have one table for each post, and then another table which references the post ID from the first table to store each comment. “Get me the data for the post with the post ID of 2, and also, get all of the comments with a post ID of 2, and display them on the page.
  + A **primary goal** when designing relational databases is to *eliminate the duplication of data*. Therefore, each each piece of data should only be stored **once**. If any piece of information needs to be updated it should only need to be updated **once** in **one place**. In data architecture this is known as *normalization*.
* ***Non-Relational****/****NoSQL***:
  + - Types of NoSQL database:
      * Key-value store
      * **Document store** **(e.g., MongoDB)**
      * Graph databases
    - Designed to handle high levels of reads/writes while scaling **horizontally**.
    - Allows for *duplication of data*.
    - Allows you to apply a structure to the data, but doesn’t require it up front. Can store data even if there’s not a logical category for it yet.
    - Much easier to scale compared to SQL. Faster.
  + In a document/non-relational database and in NoSQL in general, there is no concept of a table to group data, but there still needs to be a way to group documents together logically – this is done using **collections.**
    - *If the data doesn’t exist for a particular attribute in a collection, then the attribute does not need to exist in the document.* E.g., if there is no ‘comment’ in a post document, then it does not store an empty comment container waiting to be populated.
    - ***Benefits***:
      * Database doesn’t waste space for missing data.
      * Flexibility. Two documents in a collection can have two completely different sets of data.

**MongoDB**/**Document Store**

* A database server. Utilizes a process that listens for applications that want to connect to a specific database stored on database server. Can have multiple users connected to multiple database at any given time.
* ***Three******key components*** *(from lowest to highest level)* **:**

1. **Document:** 
   * + *Describes the content of a collection*. Like a row in a spreadsheet. Made up of sets of key-value pairs containing all of the information you need.
     + A single entry made up of one or more properties – analgous to objects. What the user can interact with when pulling information.
     + {Name: Bill}, {Age: 26}, {Height: 233}
   * Each document/sub-document has its own **unique** **ID** that MongoDB automatically creates in a property called **“\_id”** with a unique value of **ObjectId(“23b017582o8a”)**.
2. **Collection:**

* *A collection of documents.*  Logical way of grouping data together.
* Describes a group of records/documents. Like a spreadsheet with rows and columns.
* An array of objects. Contains each document/s.
  + User : [ {Name: Bill}, {Age: 26}, {Height: 233} ]
* E.g., A collection of posts on a message board. Each separate post would be it own document within the collection, and this document would contain the PostId, Title, Content, and Comments. The comments itself would be assigned to the posts as nested objects.

1. **Database:**
   * 1. A group of collections. Like a worksheet with multiple spreadsheet tabs.
        1. Amazon

* **Mongoose**
  + A powerful way to define data schemas within the application which comes with many helper methods to help build a robust API.