**MERN Stack Front to Back**

**MongoDB Setup**

1. Go to **cloud.mongodb.com** and sign up for an account to use **MongoDB Atlas** cloud database services
2. Click on the **CONTEXT** dropdown button next to the Atlas logo and press the **New Project** button to create a new MongoDB project
3. Create a database by clicking on the **Build a Database** button
4. Select a database type in the **Path Selector** menu
   1. Use the **Shared** free **Cluster** database for learning and experimenting
   2. Choose AWS as the cloud provider and select the closest region
   3. Click on the **Cluster Name** tab and provide a name for the database
   4. Click on the **Create Cluster** button
5. Choose an authentication method and add a database user
   1. **\***The first user will have permission to read and write any data in the project
   2. Select the **Username and Password** option to add a new user with a username and password
      1. **\*\***Creds for course: **Admin/20notches**
   3. Select the **My Local Environment** option in the **Connect From** menu to allow IP address **Whitelisting**
      1. Only whitelisted IP addresses may access the database
   4. Add IP addresses to the whitelist to permit them access
      1. Click the **Add My Current IP Address** button to add the current IP you’re using
6. Once the database is created, press the **CONNECT** button (inside the database’s panel) to access the **Connection String** for the database
   1. Select **Connect your application** as the connection method
   2. Choose **Node.js** as the driver at the desired version
   3. Click on the **Browse Collections** button to view the database’s collections (data tables)

**Express Setup**

1. To start an Express project:
   1. Create a folder and name it the name of the project (i.e. “devconnector”)
   2. Type **npm init** in a bash window inside the project folder
      1. Follow the npm prompts and provide the necessary information
         1. Use **server.js** as the file name of the **entry point**
      2. This will create a **package.json** file for the project containing the settings entered in the init prompts
   3. Use **yarn add** to add the following packages:
      1. **express** – used to instantiate an express server
      2. **express-validator** – used to validate data in express
      3. **bcryptjs** – used for password encryption
      4. **config** – used to create global variables
      5. **gravatar** – used to access online (GitHub) user avatars
      6. **jsonwebtoken** – used for auth token validation
      7. **mongoose** – A document relational mapping framework for MongoDB
      8. **request** – Allows an express server to make HTTP requests to another API
   4. Use the command **yarn add -D** to add the following packages to the project’s **devDependencies**:
      1. **nodemon** – Used to refresh the express server with latest code changes upon save
      2. **concurrently** – Allows for multiple dev servers (i.e. express and react servers) to run at the same time with a single command
   5. Create a **server.js** file as the main entry point of the express server
   6. To setup an active express server in the server.js file:
      1. Require the ‘express’ library and assign it to a const
      2. Call **express()** and assign the returned object to an **app** constant
      3. Create a **PORT** constant and assign it to **process.env.PORT** OR a desired port number
      4. Call **app.listen()** to initiate the express server and listen for incoming requests
         1. Pass in the PORT variable as the first parameter
         2. Pass in a function to execute (i.e. console.log()) once the express server has started as the second parameter
      5. Ex.:

**const express = require('express');**

**const app = express();**

**const PORT = process.env.PORT || 5000**

**app.listen(PORT, () => console.log('Server started on port ${PORT}'));**

1. Call the command **node [entry\_point\_file],** where entry\_point\_file is the name of the server file (i.e. server.js), in command line to invoke the express server and have it listen for incoming requests
   1. The .js extension at the end of the file name is optional
2. In a dev environment, use the command **nodemon [entry\_point\_file]** to invoke a nodemon server that will update the server’s code every time a file has changed
3. In the **package.json** file, edit the **“scripts”** property adding:
   1. A **“start”** script that invokes the express server in a production environment
      1. Use the command line command node [entry\_point\_file], in quotes, for production
   2. A **“server”** script that invokes the express server in a development environment
      1. Use the command nodemon [entry\_point\_file], in quotes, for development

**Connecting to MongoDB with Mongoose**

1. Use the **config** npm package to create global values that can be used though out the express application:
   1. In the root folder of the project, create a **config** folder
   2. In the config folder, create a new file called **default.json**
      1. This will store all the default values for the app
      2. Create a new JSON object and add a property called **“mongoURI”**
         1. Set this property equal to the connection string of the MongoDB used for the application
         2. If using an Atlas connection string, replace **<password>** with the actual password of the connecting user
   3. Create another file named **db.js** to establish the MongoDB connection:
      1. Require **mongoose** – a **Document Relational Mapper** used to connect to a MongoDB database and create schemas
      2. Require **config** to retrieve configuration values
      3. Declare a const called **db** and use the config.**get()** function to retrieve values from the default.json file
         1. Use this function to get the mongoURI value from the default.json file
      4. Use the **mongoose.connect()** function to instantiate a connection to the database
         1. This method is asynchronous
         2. Pass in db as the first parameter
         3. See code for detailed connection notes

**Route Files With Express Router**

1. All express routes can be placed inside of the server.js file, but this becomes long and messy
2. Break the express routes down into multiple files
   1. See code for examples
3. Use the **app.use()** function, where app is an instance of express(), to assign parent route paths to a route file
   1. Pass in the parent route path as the first parameter (optional)
   2. Pass in the required file, using require(), as the second parameter
      1. **\*\***When referring to the parent route inside a route file, start the **path** string with ‘**/**’ then add additional child routes
         1. This will start the route path right behind the path passed into the app.use() method stated above