Setting up the Mongoose Web Service

First open a command line terminal, navigate to the directory you wish to save the web server in, and type the following line in, and press enter to download the code:

git clone https://github.com/andrewrubinstein/Mongoose_Express_Tutorial.git

You should see the following project files if you type ls, or dir into the terminal
You can immediately type in the command

npm install

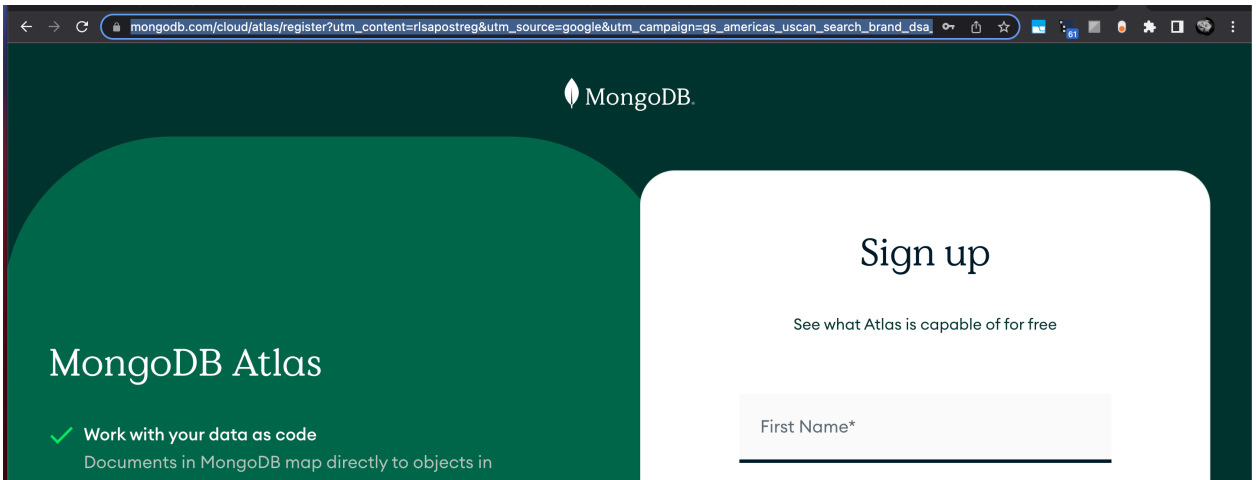
This will look in the package.json file, and see what dependencies your code has to have installed before it can be run

To start off let's talk about the three javascript files. These are the files with the code that has been written to demonstrate using mongoose with node.js.

The first file we will talk about is the "app.js" file this contains what is called boilerplate to setup the libraries we will be using.

```
//Parse .env file
    const dotenv = require('dotenv');
    dotenv.config();
//Include express classes
    const express = require('express');
    const expBodyParser = require('body-parser');
//Instantiate handlers for http and https
    const http = require('http');
//express app instantiation
    const app = express();
    const httpServer = http.createServer(app);
//require custom router implementation for db example app
    const routerGen = require('./router');
```

//



```
Middleware Definition
    app.use(express.json());
    routerGen.gen(app);
//End Middleware definition

//Start Server
console.log("Starting up server on ports: "+process.env.PORT+",
"+process.env.PORTSSL);
httpServer.listen(process.env.PORT,() => console.log('Server
started listening on port: '+process.env.PORT));
```

The above code is can be broken down into five sections as the comments denote.

First we have a section responsible for loading in a library in order to read the .env file which will need to setup with relevant data to connect to our mongodb instance, and which port we will be connecting on.

```
//Parse .env file
    const dotenv = require('dotenv');
    dotenv.config();
```

In node.js we use require to import modules, or code written in other javascript files into any other file we wish to use these modules in. Then we save the imported module to a variable so that we may access it later. Unlike in C++ when we include something requiring a module does not actually "copy and paste" the entire included code into the file you are using, instead only exported functions, and objects will be accessible through the variable we gave, in this case dotenv is the name of that variable.
Then after requiring the dotenv library we call ".config();" on the dotenv object which will call the library code to load the file ".env" that is in the same directory as the app.js file, and save it to variables
Below is a sample .env file

```
PORT=5080
MONGOUSER=testUser
MONGOPASS=testUserPass
MONGOURLTEMPLATE=mongodb+srv://
<user_name>:<password>@cluster0.pqym9.mongodb.net/?
retryWrites=true&w=majority
```

It is used to save all the variables on the left hand side of the assignment operators (=) and save the data on the right hand side so that they are accessible elsewhere in the program by those names.

The next boilerplate section we setup Express (a library to help us define our routes, and parse the data sent and received. As well as to setup the http server itself, a part of node which will listen for incoming http or hyper text transfer protocol messages, and forward them to express to use, it will also be how we send out data from our application to other applications like a client side graphical interface for example.
This is the code to do so:

```
    const express = require('express');
//Instantiate handlers for http and https
    const http = require('http');
//express app instantiation
    const app = express();
    const httpServer = http.createServer(app);
```

As you can see first we require express then we require the http server, then we create an express app on the following line:

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

The following like instantiates an http server that will use our express app for routing.

```
    const httpServer = http.createServer(app);
```

In the next section we will require the router setup module I have already created for the project where most of the custom code relating to connecting clients is handled. As well as code to tell our express app what middleware to use which includes a parser to parse the json data out of an http request, and then our router code which can be seen below:

```
//require custom router implementation for db example app
    const routerGen = require('./router');
//Middleware Definition
    app.use(express.json());
    routerGen.gen(app);
//End Middleware definition

//Start Server
console.log("Starting up server on ports: "+process.env.PORT);
httpServer.listen(process.env.PORT,() => console.log('Server
started listening on port: '+process.env.PORT));
```

Next let's dive into the code that define's our router's routes.

```
//Include Mongoose data models, and helper functions
const Models = require("./Models");
//End Mongoose data models
```

```
function gen(app)
{
    //Routes definition
    app.post('/',async (req, res, err) => {
        console.log('Request received to POST /');
    console.log(req.body);
        const user = new
Models.User({ userName:req.body.userName, password:
req.body.password, isAdmin:false });
        user.save();
        res.send("<h1>Record posted!</h1>");
    }
    );
    app.get('/',async (req, res, err) => {
          console.log("Get / called");
          const data = await Models.User.find()
        res.send(data);
    });
}
exports.gen = gen;
```

The above code are the contents of the router.js file. It can be broken up into three sections, first where we require our Models file, which is were we have define the types of data we wish to store in our mongo database. Which can be seen below:

```
//Include Mongoose data models, and helper functions
const Models = require("./Models");
//End Mongoose data models
```

The next section is a function where we take in an express application as a parameter, and create routes for it using its get, post, delete, and put methods in the function named gen.

Finally we write the line:

```
exports.gen = gen;
```

This line makes the function named gen available where we require this module by using <required_variable_name>.gen(<an instance of an express app>);

Just like we do in our app.js file:

```
    routerGen.gen(app);
```

Continuing let's analyze the call to app.get in the gen function:

```
    app.get('/',async (req, res, err) => {
          console.log("Get / called");
          const data = await Models.User.find()
          res.send(data);
    });
```

This is just about the simplest way to write a route, let's break it down first into two parts, then we'll go further, and break the second part into three more.

First let's notice the app.get method of an express app object takes two parameter, first a string, in our case just the string:

```
'/'
```

This is to specify the path in our website where this function will be called if we receive a get http request, for more on this see:
https://github.com/andrewrubinstein/CPPWebDevWorkshop/blob/main/PartOneWebDevWorkshop.pdf

The second parameter may seem a little strange to you, and it is in object oriented code and unusual type for a parameter, but in the functional paradigm it is incredibly common, and called a lambda function, this is the code that will be executed in when a client makes an HTTP get request to the root of our site.

Now let's break down what this lambda function is doing to understand what will happen when we make an HTTP get request to this route.

As you can see the lambda function is defined below:

```
async (req, res, err) => {
          console.log("Get / called");
          const data = await Models.User.find()
          res.send(data);
}
```

The first unusual thing to note is this keyword async we find at the beginning of this function. This is something we use due to the nature of node.js, to briefly explain what that is and why that is let's first understand that node.js is a runtime environment designed with the idea of allowing developers to be able to quickly write very fast, and efficient web servers using javascript, which is also what the client side uses making it easier to onboard new developers.

Now in computing anything to do with input and output, like reading from a Hard Disk, or even SSD is very slow compared to reading data from system memory (RAM) but transmitting data over the network is much slower, and to send a single byte can take 100x longer than reading from a single SSD taking on the order of thousandths of a second. Now traditional programming techniques where code is executed one line after the other would require us to wait while we send our network request to mongo db, then wait for a response back, and only then could we move on to handling other requests. What node.js does is says hey I don't really want to wait so I'm just gonna remember the functions I need to execute when I get back that response from the server, but before that happens it can go on to handle other code that has finished waiting for some input or output operation. This allows web servers written with express, and node.js to handle millions of requests per second by not waiting for the previous request to finish before starting a new one.

The "async" keyword is there to say that this is a function that may need to stop executing, wait for something to finish which is specified by the await keyword, and only once that is done go back, and finish executing this function.

The "async" keyword can be used before any javascript function definition, and lambda functions are just that a definition for a javascript object.

Next let's talk about the parameters our lambda function takes which are defined immediately following the async keyword:

```
(req, res , err)
```

The first is named "req" this is to represent the http request coming from the client, again refer to this link for more information of the client-server relationship, and http requests: https://github.com/andrewrubinstein/CPPWebDevWorkshop/blob/main/PartOneWebDevWorkshop.pdf

The second is named "res" this is where we will define the http response we wish to send back to the client. The last is any error message we may receive pertaining to the client's http request, which we will not be using now.

```
        console.log("Get / called");
        const data = await Models.User.find()
        res.send(data);
```

The above code is the function body of our lambda what will actually be run when a get request at our defined route is made.

First we call console.log("Get / Called")
console.log in javascript is equivalent to
cout<<"Get / called"; in c++, or

System.out.println(“Get / called”); in Java

And it will simply print the string to the console every-time a client makes a get request to this route.
The following line of code we actually ask Mongoose to query our database to get all the objects in the User collection of our database. The process of getting that data from our mongo database is one that takes a lot of time waiting for input output operations, so it is an operation that happens asynchronously, and we must use await before the call to say that we want to wait for that data to be retrieved before executing the rest of our function.

```
        const data = await Models.User.find()
```

Finally we use the response object to send the data we got from our mongo database back to the client that initially requested our route.

```
        res.send(data);
```

Remember this will actually be executing other code while we wait for the response from our database so we don’t create any bottlenecks in our code making a highly scalable project.

Mongoose Models

The following code includes boilerplate for connecting to our mongo database instance that has been defined in our .env file, as well as code defining our User model that we saw being used above to query the database.

```
//DB Connection
const wsUser =
{ id:process.env.MONGOUSER,password:process.env.MONGOPASS,
template_url:process.env.MONGOURLTEMPLATE };
const mongoURL = wsUser.template_url.replace('<password>',
wsUser.password).replace("<user_name>", wsUser.id);
//Mongoose instantiation
const mongoose = require('mongoose');
mongoose.connect(mongoURL,{ useNewUrlParser: true,
useUnifiedTopology: true });
const db = mongoose.connection;
db.on('error', ()=> {
    console.error.bind(console, 'connection error:');
    }
);
db.once('open', function() {
console.log('Succesfully connected to database server');
});

//Models, and schemas for Mongoose
    //User Model and schema
    const userSchema = new mongoose.Schema(
        {
            userName: String,
            password: String,
            isAdmin: Boolean
        }
    );
    const User = mongoose.model('User',userSchema);
    //End of Mongoose Model and Schema Definitions
//End of Mongoose Instantiation
```

```
//End Helper function to copy location data from regular JS
object to Mongoose location model
//
//Exports
exports.User = User;
//End of Exports
```

The first few lines of code in the file actually create a url that will be used by the Mongoose library to connect to our mongodb instance according to what we specified in the .env file

```
//DB Connection
const wsUser =
{ id:process.env.MONGOUSER,password:process.env.MONGOPASS,
template_url:process.env.MONGOURLTEMPLATE };
const mongoURL = wsUser.template_url.replace('<password>',
wsUser.password).replace("<user_name>", wsUser.id);
```

We assume the user, and password has not been added directly to the template url in our .env file, it instead assumed that we have substrings <password> and username in the places we want to put our user, and password, and replaces those substrings in the template with them.

This is to make changing the user, and password simpler to do as a user by putting the work on the application.

In the next section we have the boilerplate for connecting Mongoose to our mongodb instance, you can see some of the uglier side of event handling in javascript, where we define callback functions to be executed on particular events, like how on an error event we print out the connection error message, and on an open event we just print that we have connected successfully.

```
//Mongoose instantiation
const mongoose = require('mongoose');
mongoose.connect(mongoURL,{ useNewUrlParser: true,
useUnifiedTopology: true });
const db = mongoose.connection;
db.on('error', ()=> {
    console.error.bind(console, 'connection error:');
    }
);
db.once('open', function() {
console.log('Succesfully connected to database server');
});
```

Defining our Models

We only have one model defined in this example, it is a model for a user object

```
    //User Model and schema
    const userSchema = new mongoose.Schema(
        {
            userName: String,
            password: String,
            isAdmin: Boolean
        }
    );
    const User = mongoose.model('User',userSchema);
    //End of Mongoose Model and Schema Definitions
```

The above code first sets up a schema which is very similar to the idea of a class in C++, or Java. It is defining a data type.
The below code is where we actually define a new schema with the appropriate fields, in our case two strings, and username, a password, and a boolean value to denote whether the user is an administrator.

```
new mongoose.Schema(
        {
            userName: String,
            password: String,
            isAdmin: Boolean
        }
```

We take that schema, and save it to the variable userSchema so that we can use it to create a model that we can actually use to interact with the database from.

```
    const User = mongoose.model('User',userSchema);
```

```
//Exports
exports.User = User;
//End of Exports
```

Finally we export our user model so that it can be used in our router definition module, as well saw above.

Connecting to our mongo database instance

Now we must create our own .env file for our app to use, let's analyze the sample file to see how to fill ours out. We will need information we saved back during the first part.

```
PORT=5080
MONGOUSER=testUser
MONGOPASS=testUserPass
MONGOURLTEMPLATE=mongodb+srv://
<user_name>:<password>@cluster0.pqym9.mongodb.net/?
retryWrites=true&w=majority
```

Let's break this down line by line
First we have PORT=5080. This defines the port the server will listen for incoming connections you can see the. Last couple lines of app.js wee actually use this value to tell our http server what port to listen on. This is a part of the tcp/ip sockets level of the networking stack explained in this pdf:
https://github.com/andrewrubinstein/CPPWebDevWorkshop/blob/main/PartOneWebDevWorkshop.pdf

The next line MONGOUSER=testUser is where we define the username we will be logging into our database with. This is information you need to remember from part one setting up the environment.
You can see in Models.js on line 3 we read these values to create the wsUser object

```
//DB Connection
const wsUser =
{ id:process.env.MONGOUSER,password:process.env.MONGOPASS,
template_url:process.env.MONGOURLTEMPLATE };
const mongoURL = wsUser.template_url.replace('<password>',
wsUser.password).replace("<user_name>", wsUser.id);
//Mongoose instantiation
```

We also see the variables defined on the next two lines values used to build the mongoUrl

Make sure to save this .env file in the same directory as the app.js, and that it has no extension.