Table of Contents

[**part 01 – Angular Project Setup** 2](#_Toc161399268)

[**part 02 – Terminal Window Operations** 6](#_Toc161399269)

[**part 03 – Adding a file to the project** 8](#_Toc161399270)

[**part 04 – Form Support** 11](#_Toc161399271)

[**part 05 – Adding Tailwind CSS** 16](#_Toc161399272)

[**part 06 – API Support** 19](#_Toc161399273)

[**part 07 – MongoDB Interaction** 23](#_Toc161399274)

[**part 08 – POSTing a new resource** 26](#_Toc161399275)

[**part 09 – Testing with Jest and Supertest** 29](#_Toc161399276)

Introduction to GitHub Copilot

## part 01 – Angular Project Setup

1. Unzip the starter file and make sure the folder is re-named to skills-ai
2. Open a terminal inside of your folder and run the command **npm install.** In linux just right click on the folder and choose *Open In Terminal*. It is important that you run this command against the skills-ai folder and not Documents or the root folder of your system.  
   A close up of a logo

   Description automatically generated
3. After running the command you should end up with a file structure like this:

|  |
| --- |
| A screenshot of a computer  Description automatically generated |

This is a typical Angular 16 file structure.

1. All of our changes will be in the app folder which is in the src folder. Open VS Code and then open the skills-ai folder as an application. Once VS Code is opened, you would be prompted to sign in to Copilot. Refer to #9 below to see how we start an Angular app.
2. Take a look at the app.component.html file. This is the template that displays it’s content in the browser. Create a blank line above the **<span>** tag and one below. You should see the sparkle icon from Copilot, if you have it installed:

|  |
| --- |
| A screenshot of a computer program  Description automatically generated |

If you do not see this icon, there are other ways of using Copilot, in fact I hardly ever use the sparkle icon.

1. If you do see the icon, just click on it. It would then invite you to ask it questions:

|  |
| --- |
| A screenshot of a computer program  Description automatically generated |

If you do not see the icon, just press CTRL-I or see #7.

1. If the icon did not show for you, just highlight the entire pair of **<span>** tags and right click. You can now choose to *Start inline Chat*:

|  |
| --- |
| A screenshot of a computer  Description automatically generated |

1. Take a look at the first prompt I sent to the tool. At this point we can accept by clicking the arrow  or amend the prompt and try again by clicking the refresh add icon . I will in this case, accept this suggestion by Copilot:

|  |
| --- |
| A screenshot of a computer  Description automatically generated |

Since dark brown did not work, try dark red. This is the actual prompt I used:

|  |
| --- |
| add css styles to make the content text two times as large and add dark red color to the text |

You could always *undo* the changes and try again.

1. To start this app in the terminal window type **ng serve.** This is what the result looks like on my Firefox browser:

|  |
| --- |
| A screenshot of a computer  Description automatically generated |

Notice the URL.

1. I am not satisfied with this so I want to center the text and add a border around the text:

|  |
| --- |
| add a border around the text and center the entire span tag |

1. I am still not satisfied with this, I want to reduce the width to 80% of the browser width:

|  |
| --- |
| center the span tag on the page and reduce its width to 80% of the width of the browser |

1. This is the result so far:

|  |
| --- |
| **<span style="font-size: 2em; color: darkred; border: 1px solid black; display: flex; justify-content: center; width: 80%; margin: 0 auto;">{{ title }} app is running!</span>** |
| **A screenshot of a computer  Description automatically generated** |

1. Copy the entire pair of **<span>** tags and paste it in the app.component.css file. Once there highlight the entire line and enter this prompt:

|  |
| --- |
| extract the css styles and make them css classes instead |

1. This is what it looks like on my VS Code:

|  |
| --- |
| **A screenshot of a computer  Description automatically generated** |

I will accept this code

1. Over in the template file, enter the prompt you see below. Remember you can always amend your prompt. In my case it did not recognize the class name from #13 so I changed it manually:

|  |
| --- |
| A screen shot of a computer  Description automatically generated |

16. This is the final code and browser output for this section:

|  |
| --- |
| **<span class="title-span">{{ title }} app is running!</span>**  **<!---->**  **<router-outlet></router-outlet>** |
| **A screenshot of a computer  Description automatically generated** |

**------end of part 01-------**

## part 02 – Terminal Window Operations

Since we are working with an Angular project we should build a new component. Angular is all about components. Usually each new component generated by the Angular CLI will consist of three to four files depending on version and options passed into the command. Since we are working in version 16, the **generate component** command will generate four files if not told to do otherwise.

A screenshot of a computer

Description automatically generated

1. If you look closely at the VS Code interface, on the left column there is an icon that looks like the one in the red square.

This will bring up a Copilot *chat* window that can then be used to be more specific about context. We will use it to send code to the terminal window. So the commands would be Linux based terminal commands. Windows users will have a similar experience.

1. Here is the Copilot interface on my VS Code editor running inside of a Linux VM:

|  |
| --- |
| **A screenshot of a computer program  Description automatically generated** |

As you can see, the prompts in blue are just suggestions

1. Put your cursor inside of that Ask Copilot search bar and type the @ character, choose **@terminal**:

|  |
| --- |
| **A screenshot of a computer  Description automatically generated** |

Notice how we can contextualize the prompt.

1. Expand that column in VS Code so that you have room to work and also make sure you have the terminal window opened as the result will be sent to the terminal window in VS Code. Enter the prompt shown below:

|  |
| --- |
| **A screenshot of a computer  Description automatically generated** |

Click on the Submit text.

1. In my case, it generated the CLI command but the flags are incorrect. We will remove those in the terminal window itself. Pay attention to the three icons above the generated command text, hover over them to see what they do. For now, choose the **Insert into Terminal** icon and the command will be sent to the terminal window for you to execute manually:

|  |
| --- |
| **A screenshot of a computer  Description automatically generated** |

1. Here is my window with the Copilot generated command inside. I removed the extra flags manually. You may now execute this code by hitting the Enter key:

|  |
| --- |
| **A screenshot of a computer  Description automatically generated** |

When you press the Enter key, make sure that the terminal window is the focus of your actions. It should generate four files and update one of them.

1. You may now clear the terminal window and click the **File Explorer** icon in VS Code and you will see a new folder called **home**. That folder contains all the files necessary for our home component in Angular 16.

**------end of part 02-------**

## part 03 – Adding a file to the project

Lets say we wanted users to register to our site. We will need a register component but also we should have a separate file that describes our data, so an interface. Sometimes you may hear *shape of the data* or *data abstraction.* For the register component, just add the code directly in the terminal window. We already did this for the home component, see Part2. The terminal window command will be **ng g c register**. You don’t need the AI tool for this.

If you completed the code above you should now have two folders home and register in your app folder. Any of the files starting with *app* can be considered the parent component for the rest of your application.

1. Since all users will have a username or email and a password we already know where to start. We can always change this later. Once again open the Copilot chat window on the left column and this time hit the forward slash (/). This is what shows up on my VS Code:

|  |
| --- |
| **A screenshot of a computer program  Description automatically generated** |

A lot more options appeared, but the old ones like @terminal is there also.

1. Choose the @terminal option and begin to type something like this:

|  |
| --- |
|  |

1. Click the Submit arrow  and watch the tool generate the entire file code for you:

|  |
| --- |
| A screenshot of a computer  Description automatically generated |

1. All you have to do now is click the *Insert into Terminal* icon, and execute this command from the terminal window:

|  |
| --- |
| A screenshot of a computer  Description automatically generated |

There is just one problem, the path is incorrect, just change that according to your project and execute the command. In my case it ws > ./src/app/user.ts

1. You should now have an entire file written for you. This is what it looks like on my system:

|  |
| --- |
| A screenshot of a computer screen  Description automatically generated |

This image does not show the correct path, there should be a src folder involved.

1. Move to the register.component.html file and ask Copilot to create a form for you. You may follow the prompt shown below:

|  |
| --- |
| A screenshot of a computer program  Description automatically generated |

In my case, the tool added the Angular ngModel directive to the code, you may remove that for now, **[(ngModel)]="user.name".** In order to see the form you would need to import the **register** component into the routing module and then add it to the **routes** array. Refer to part 04 #11 for this.

**------end of part 03-------**

## part 04 – Form Support

At the time of this boot camp, I could not find an easy way to get Copilot to import a module and add the module to the appropriate array. I got it to work by highlighting all the lines of code in the entire app.module.ts file, then right clicking brought up a different icon, see below.

1. Notice the blue icon on line 2. If you hover over it the pop up message is to Show Code Actions.

|  |
| --- |
| A screen shot of a computer program  Description automatically generated |

1. If you right click on this icon however you get an option to Modify using Copilot. Choose that option

|  |
| --- |
| A screenshot of a computer  Description automatically generated |

1. You may now proceed to enter the prompt below:

|  |
| --- |
| **A screen shot of a computer program  Description automatically generated** |

Note, you may have to accept the two different statements.

1. Back to the register.component.ts file. Since importing the user.ts file is very simple, just do it manually:

|  |
| --- |
| **import { Component } from '@angular/core';**  **import { User } from "./../user";**  **@Component({**  **selector: 'app-register',** |

The highlighted line was imported.

1. Now we use comments to get help from Copilot. Notice the comments on line 11 and the suggested code from Copilot. This is not exactly what I wanted, see #6:

|  |
| --- |
| **A screen shot of a computer  Description automatically generated** |

1. Since the comment did not work lets go back to Copilot inline chat. This is more like what I wanted, a FormGroup with the three existing fields. If you get this you may accept and make any changes afterwards:

|  |
| --- |
| **A screen shot of a computer program  Description automatically generated** |

In my case, it added a couple of extra lines that were already there. It also added a property that TypeScript is now complaining about. Just add the ! character at the end of RegisterForm, so the line should be: **registerForm!: FormGroup;**

1. Highlight the entire registerForm and ask Copilot to add the **required** validation to each FormControl:

|  |
| --- |
| **A screenshot of a computer program  Description automatically generated** |

From the image above it looks good to me, so I accepted. You may have to import the Validators class from @angular/forms or ask Copilot to do it.

1. Since we need a method to handle form submission and it’s a simple function, use a comment to get Copilot to do this for us:

|  |
| --- |
| **//add a method to handle the form submission** |

1. You may also want to send the form values to the console window to check if it is working properly. This is simple so I will just do it manually. This is the resulting onSubmit() function:

|  |
| --- |
| **onSubmit() {**  **let user: User = {**  **username: this.registerForm.value.username,**  **email: this.registerForm.value.email,**  **password: this.registerForm.value.password**  **};**  **console.log(user);**  **};** |

1. Move to the template file and hook up the **registerForm** to the HTML form:

|  |
| --- |
| **A screen shot of a computer program  Description automatically generated** |

In this case, the code worked out perfectly

1. At this point we can run the app and check our work so far but first we must add the **register** component to our app-routing module. Again we can ask Copilot but this is very simple code:

|  |
| --- |
| **A screenshot of a computer program  Description automatically generated** |

You will have to fix the path for **register** component, but it looks good.

1. To run the application go to the terminal window and enter the following command:

|  |
| --- |
| **ng serve** |

In my case, on the template I had to change name to username and it worked well

1. Once the app is running, open a browser and go to the port shown in the terminal window. Once there navigate manually to /register:

|  |
| --- |
| **A screenshot of a computer  Description automatically generated** |

**------end of part 04-------**

## part 05 – Adding Tailwind CSS

From the image in Part04, we might be able to ask Copilot to generate some CSS styles to make the form look better. Since we already used plain CSS in Part01 8-13, in this section I will try something different. We will install tailwind css and work with the classes defined there. Tailwind is similar to Bootstrap and StyleX.

1. First install Tailwind and add a configuration file, do the two steps in sequence:

**npm install -D tailwindcss  
npx tailwindcss init**

That last command will add a tailwind.config.js file at the root of your application

1. Open the config file in VS Code and add this line to the content key:value section:

|  |
| --- |
| **module.exports = {**  **content: [**  **"./src/\*\*/\*.{html,ts}",**  **],**  **theme: {**  **extend: {},**  **},**  **plugins: [],**  **}** |

This just means to consider all .ts and .html files

1. The final step is to add a few lines of code to the styles.css file which can be found at the root of your app folder:

|  |
| --- |
| **@tailwind base;**  **@tailwind components;**  **@tailwind utilities;** |

1. Run the app using ng serve and you will notice that all the previous styles are gone and even the form seems to be missing.

|  |
| --- |
| A white background with black text  Description automatically generated |

1. This is my first attempt at organizing the form and making it present better on the browser:

|  |
| --- |
| **A screenshot of a computer program  Description automatically generated** |

The result of this code is below, but we will try again.

1. The image below is from that firt attempt in #5. Lets try again this time to get the form narrower and also to start the form a few pixels from the top:

|  |
| --- |
| A screenshot of a computer  Description automatically generated |

1. Below is the prompt I used and the resulting form on the browser:

|  |
| --- |
| use Tailwind css to make this form narrower on the browser window and have it start 20px from the top |
| A screenshot of a computer  Description automatically generated |

1. (Optional)

|  |
| --- |
| Use Tailwind CSS to create hover effect for each form control and the button. This is the resulting code: |
| **<form [formGroup]="registerForm" (ngSubmit)="onSubmit()" class="border p-4 space-y-4 mx-auto mt-20" style="width: 60%;">**  **<div>**  **<label for="name">Name:</label>**  **<input type="text" id="username" name="username" formControlName="username" class="w-full px-2 py-1 border rounded hover:bg-gray-100">**  **</div>**  **<div>**  **<label for="email">Email:</label>**  **<input type="email" id="email" name="email" formControlName="email" class="w-full px-2 py-1 border rounded hover:bg-gray-100">**  **</div>**  **<div>**  **<label for="password">Password:</label>**  **<input type="password" id="password" name="password" formControlName="password" class="w-full px-2 py-1 border rounded hover:bg-gray-100">**  **</div>**  **<button type="submit" class="bg-blue-500 text-white px-4 py-2 rounded hover:bg-blue-600">Submit</button>**  **</form>** |

**------end of part 05-------**

## part 06 – API Support

We will now add data interaction to our current application. We will add a JSON file with dummy data in it and then display that data using another component.

1. Here I am using the chat window again to tell Copilot exactly what I want. Other than the path, Copilot seems to get it right:

|  |  |
| --- | --- |
| **A screenshot of a computer program  Description automatically generated** | **@terminal create a directory called data under my app directory** |

This is the window accessible on the left side of VS Code.

1. After entering the following prompt from you will use the arrow icon to push the command into the terminal window. Make any changes and execute the command:

|  |  |
| --- | --- |
| A screenshot of a computer screen  Description automatically generated | In the directory ./src/app/data add a file called db.json. In that file add three documents in JSON format each wiht three fields and dummy data. The three fields will match the three fields in the User interface. |

1. This is what the Linux command looks like in my terminal. All I have to do now is press the Enter key to execute this command and get a file with three dummy users:

|  |
| --- |
| **A screen shot of a computer  Description automatically generated** |

1. Below is an image showing the db.json file and the content. Note, I did ask Copilot to structure the code in the data file vertically for me:

|  |
| --- |
| A screen shot of a computer screen  Description automatically generated |

1. Lets see if Copilot can help us install software and configure that software:

|  |
| --- |
| **A screenshot of a computer program  Description automatically generated** |

1. After moving the commands that Copilot generated to the terminal window and executing that command, this is what resulted:

|  |
| --- |
| A screen shot of a computer  Description automatically generated |

As you can see, the server software was installed and now it is serving up the db.json file at localhost:3000. Open a browser and point it to this URL and see if you can see the three dummy users.

1. The json-server is now being processed by a daemon. The output is not exactly what I wanted so let us stop this process using CTRL-C and make some changes:

|  |
| --- |
| **A screenshot of a computer screen  Description automatically generated** |

After accepting, execute the json-server from a different terminal window

1. Remember the second command from #5 above that Copilot generated? Copy that command from the chat window and add it as a script to the package.json file:

|  |
| --- |
| **"build": "ng build",**  **"watch": "ng build --watch --configuration development",**  **"server":"json-server --watch ./src/app/data/db.json",**  **"test": "ng test"**  **},**  **"private": true,** |

So now, when we go to a fresh terminal and point it to the folder of your app, we can run the command **npm run server** and the json-server will start up.

1. Open a terminal window and point it to the skills-ai folder using your CD command. Once there run the command **npm run server**:

|  |
| --- |
| A screen shot of a computer  Description automatically generated |

Notice that last line, it is where you can now point your browser to and see all three dummy users.

**------end of part 06-------**

## part 07 – MongoDB Interaction

There is a different starter file for this section called nodejs-starter-07.zip. Just like before, unzip that file then rename the folder to just nodejs. After that open the resulting folder in your code editor.

In this part, I will try to show how the AI tool can help with database actions. Since I am familiar with no-sql and MongoDB, this is the example I will use here. However, these techniques can be applied to traditional relational databases like MySQL, as long as the tool was trained on these technologies.

You do not need to be familiar with NodeJS or MongoDB to follow along with this example. Just realize the idea of working with the tool in any technology. But just for some background, web developers follow the three tier system of presentation layer, logic layer and database layer. However in the web world, the logic layer involves JavaScript files and databases are usually replaced with API interaction.

So, in this example we will use a JavaScript file to add an employee to the database and also to retrieve that document (record). We will be using a software application called Postman for the POST request. Postman is a free download.

The application is setup with four routes but we will be working with just two of them. The **/addemployee** endpoint will add a new resource and **/getemployees** will return all the employees in the collection (table). In this sample application, the routes will pass execution to a matching function in a controller file. That function will do the main processing to get a result, either add a document or get all documents. This is the famous MVC pattern.

1. Open the employee.js file in an editor. This file will contain a schema which is the shape of our data. It will also connect to the database which is in the cloud so it has a URI instead of a local connection string. The connection string is hidden away in a file called .env but we can still use its key/value pairs. Use any of the prompt methods here but I will use a comment:

|  |
| --- |
| **A computer screen with text and images  Description automatically generated** |

From the image above, it tried to connect me to a local database

1. You can accept the suggestion but then erase most of it. It will also prompt you for a second parameter which it should already know:

|  |
| --- |
| **const mongoose = require('mongoose');**  **mongoose.set('strictQuery', false);**  **if(process.env.NODE\_ENV !== 'production'){**  **require('dotenv').config();**  **};**  **CONNECTION = process.env.CONNECTION; //connect to the database**  **mongoose.connect(CONNECTION, { useNewUrlParser: true, useUnifiedTopology: true });** |

1. Since this is an asynchronous function call, we need to use the then() method. In the GitHub code, I show the async/away method. But for our purposes here, make the code vertical and chain the then() method:

|  |
| --- |
| **A black background with white text  Description automatically generated** |

Once again the tool will offer a suggestion, accept it.

1. At this point the tool should also offer error handling. All these offerings are just basic code but it saves time:*ref*

|  |
| --- |
| **mongoose.connect(**  **CONNECTION,**  **{ useNewUrlParser: true, useUnifiedTopology: true }**  **).then(**  **() => console.log('Connected to MongoDB...')**  **).catch(**  **err => console.error('Could not connect to MongoDB...', err)**  **);** |

1. Since this database connection is in a separate file (not index), we need to define a schema. Here I even told the tool what names I wanted and it suggested this which I will accept:

|  |
| --- |
| **A screenshot of a computer  Description automatically generated** |

Now the last line of code in this file will have the correct parameters. You can export the model here if you want. That line would be the last line in the file something like this:  
**module.exports = mongoose.model('Employee', empSchema);**

1. We will use the /getemployees route since it calls the **exports.getemployees()** method in our controller file:

|  |
| --- |
| **exports.getemployees= function(req, res){**  **Employee.find();**  **};** |

Even though this code does nothing, it at least tests the connection. If you do not get an error, them move to the next step. You will not get a result here, the browser may just hang. This is because we did not engage the **res** parameter.

1. Lets continue in the **getemployees** function and add the **then()** method:

|  |
| --- |
| **exports.getemployees= function(req, res){**  **Employee.find()**  **.then(function(employee){**  **res.send(employee);**  **}).catch(function(err){**  **res.send (err);**  **});**  **};** |

This is what it suggested and I accepted. The browser call to this path /**getemployees** should now return an empty array. Obviously we do not have any data in the database. Also in my case it did not use arrow functions.

**------end of part 07-------**

## part 08 – POSTing a new resource

We will continue with the /**addemployee** endpoint and the addemployee function in the controller.js file. Once again, if you begin the code, Copilot will start offering suggestions.

1. Posting a new employee works slightly differently. We need to first create an object of the **Employee** file then we get access to the **save()** method. This is done by using the normal const keyword and creating an object with the new keyword:

|  |
| --- |
| A computer screen shot of a program code  Description automatically generated |

As you can see, this suggested code is mostly correct. However I will do this in steps, first create the object.

1. I ignored the suggestion from #1 and just created the object first:

|  |
| --- |
| **exports.addemployee=function(req, res){**  **const employee = new Employee();**  **};** |

1. Now we can save an actual employee but hardcoded:

|  |
| --- |
| **const employee = new Employee({**  **empName:"Axle",**  **empPass:"axle"**  **});** |

1. The last thing to do here is fire the **save()** method and return some message back to the developer:

|  |
| --- |
| **const employee = new Employee({**  **empName:"Axle",**  **empPass:"axle"**  **});**  **employee.save();**  **res.send(employee);** |

Copilot suggested both of these lines, but I would add a message to the **send()** method.

1. We could also add error handling:

|  |
| --- |
| A screen shot of a computer program  Description automatically generated |

I just added the **then()** method and watched Copilot fire off these suggestions which I will accept.

1. You may have to do some adjustments, but this is the final result:

|  |
| --- |
| **exports.addemployee=function(req, res){**  **const employee = new Employee({**  **empName:"Axle",**  **empPass:"axle"**  **});**  **employee.save().then(function(){**  **res.send("Employee added: " + employee);**  **})**  **.catch(function(err){**  **res.send(err);**  **});**  **};** |

Most of this code was suggested by Copilot.

1. To test the **save()** method, I used MongoDB Atlas, a cloud-based MongoDB database. Once the addemployee path is pointed to, that invokes the addemployee function and the one employee will be added to the database.
2. Since we cannot make POST requests directly in the browser, I am using Postman here to show the result:

|  |
| --- |
|  |

As you can see, we made a POST request to match our routes file and the **addemployee()** method returned to us the employee it stored in the database including an id field that MongoDB added.

1. We can also look at the Atlas end of the transaction:

|  |
| --- |
| A screenshot of a computer  Description automatically generated |

Note, **department** was hardcoded into the connection string.

**------end of part 08-------**

## part 09 – Testing with Jest and Supertest

Use the provided starter file and unzip in your folder of choice. Rename the folder to nodejs-jest. This is just a very basic node-express application with just one endpoint /home. Open the folder with your favorite editor. The package.json file already has the required dependencies for testing, so just run **npm install** to re-create the app.

Once you have the app restored you can type into the terminal window **npm start** and the app will be served from port 8000. Test and verify that you can see the home page at the location <http://localhost:8000/home> before continuing.

1. The first task is to add a test file to the project. I will use Copilot and @terminal for most of the tasks:

|  |  |
| --- | --- |
| A screen shot of a computer program  Description automatically generated | add a file to this project that will test the /home path served by the app file. the name of the test file is app.test.js. in that app.test.js file require the supertest package and the app.js file |

As you can see, Copilot gave us code for the file and even told us how to install jest and run the test file. I will change it up a bit. It did not however gave us the command to add the file to the project, but this is easy **touch app.test.js**

1. If you have the app.test.js file, open it and use the Insert at cursor icon on the Copilot chat window to copy the suggested code to that file:

|  |
| --- |
| **A screenshot of a computer screen  Description automatically generated** |

You may now close the chat window.

1. Now you could use the command that Copilot suggested and test the code we got so far:

|  |
| --- |
| **A screen shot of a computer screen  Description automatically generated** |

As you can see the test passed.

1. If you think that this task was too easy, change the path in the app.js file to something different and watch the test fail:

|  |
| --- |
| **const app = express();**  **app.get('/homesick', (req, res) => {**  **res.send('Welcome Home!')**  **});** |

1. I received several failed messages but this is the one that I thought was appropriate in this case:

|  |
| --- |
| A screen shot of a computer program  Description automatically generated |

It failed at the expect(200) line. If the site was serving properly it would respond with a 200 message. Instead it responded with a 404 as expected.

**------end of part 09-------**

**Appendix A – Mocha Chai Setup**

Install with this command: npm install mocha chai --save-dev

Add the configuration file to the root: .mocharc.json

Content for the above can be found here: <https://github.com/mochajs/mocha/blob/master/example/config/.mocharc.json>

Create a folder called test and a file called aboutus.test.js

Invoke the test environment with this change in the scripts section of package.json

|  |
| --- |
| "scripts": {  "start": "nodemon index.js",  "test":"mocha --exit"  }, |

**Appendix B – Install dotenv**

Install with this command: npm install dotenv

Change the password for the connection string

This line is required in the employee.js file: require('dotenv').config();