Darren Dixon

darrenadixonpi@protonmail.com
+1(317)-209-6121

Grocers

April 21st, 2021 - April 30th, 2021

Overview

You work at XYZ solutions as a MEAN stack developer. Due to the Covid-19 situation every business is becoming online. You are asked by your management to lead and develop a dynamic website for the online purchase of groceries.

GitHub Repository Link: https://github.com/NTamez8/TCS-Capstone-Project/tree/darren

Note: From the GitHub Repository Link, the backend server is started with “cd backEnd” and then “nodemon index.js”. The frontend server is started with “cd capstone-grocers” and then “ng serve -o”. Also, when the backend server is started, a default Admin account is automatically created if one does not exist. This account’s information is: a_username: “admin”, a_password:”AdminCapstone”.

Goals

1. Create a rich front-end.
2. Create admin, employee, and user portals.
3. Create functionalities for the admin portal such as sign-in/logout, adding/updating/deleting an item, adding/deleting an employee, viewing/resolving requests and generating reports.
4. Create functionalities for the employee such as updating the order status, sending a request, sign-in/logout, edit profile and unlocking user accounts.

5. Create functionalities for the user portal such as registration, sign-in/logout, item-selection, adding to the cart, checking out, adding funds to their account, editing their profile, viewing order status and raising tickets.

6. Create a database for storing all application related information.

My Main Responsibilities

1. Backend DB-Table/Models, Frontend Classes

   Developed the backend DB design for each main object. This includes Requests, Products, Orders, Users, Tickets, Employees, Admins and Carts (Only Model/Class). Each of these required Controllers to alter the data and Routes for the backend server to connect to the Controllers.

   I also created classes for these main objects in the Angular frontend. These are used with Angular Services (one for each class) by their various Angular Components for frontend → backend communication.

2. Admin SignIn/Logout/Authentication

   Implemented authentication in the frontend using Angular AuthGuards with Angular Routing paths and in the backend using JWT and BCRYPT JS modules. The backend created a token based off of the admin signing in and sent it to the frontend. This token acted as complete authentication. Without it, none of the Admin Panel is accessible. All passwords are encrypted with BCRYPT before storage. Upon Logout, the token is cleared and the Admin is redirected to the homepage. I would like to note that only the Admin password encryption/validation is done by me. The rest of the group worked on Employee/User password encryption/validation.

3. Admin Panel

   Implemented an interface for the Admin to navigate to all of the necessary Admin-related Angular Components.

4. Product Management

   Implemented an interface to allow the Admin to View, Add, Delete, or Update Products. This is all done in a single Angular Component, but each CRUD Operation is handled in its own, separate component.

5. Request Management

   Implemented an interface to allow the Admin to View and Resolve Requests. Requests that are resolved could also be deleted. This is all done through one Angular Component.

Optional Responsibilities I Assisted With

1. User: Cart Management

   Assisted in troubleshooting with Users being unable to add items to a cart or to view their cart.

2. Employee: Panel

   Created the Employee Panel and helped set up the Employee Login page.

3. Authentication

   Assisted in implementing frontend/backend authentication for the User and the Employee. Follows similar procedures to the Admin Authentication.

4. Other

   Contributed to Git, Mongo, Node, and Angular troubleshooting. Contributed to modifying various services for proper frontend → backend communication. Also added additional functionality to allow Admins to Delete resolved Requests.

Methodologies/Algorithms

1. Main Project Concepts

I. Frontend

Angular Services/Routing/AuthGuard/Components/Classes, TypeScript ES5/6, Bootstrap

a. Services

Angular Services will always call the backend Controllers via the backend server URL and backend Routes. Please note this while you are reading ahead.

II. Backend

Node JS, MongoDB/Mongoose Models/Controllers/Routes

2. Backend DB-Table/Models, Frontend Classes

I. Backend Models

Each Model contained only the necessary properties for the project implementation. Uses Mongoose Schema/Model.

a. Admin

Properties:

- firstName:String
- lastName:String
- a_username:String
- a_password:String

Methods:

- encryptPassword: Encrypts Admin password and returns the corresponding hash.

- validPassword: Test the candidatePassword against the stored-encrypted password to indicate if the password is valid or not.

b. Cart

Properties:

- product:ProductModel _id
- quantity:Number

c. Employee

Properties:

- firstName:String
- lastName:String
- email_address:String
- e_password:String
- first_login:Boolean - Needed to force the Employee to change their password upon logging in for the first time.

Methods:

- encryptPassword: Encrypts Employee password and returns the corresponding hash.
- validPassword: Test the candidatePassword against the stored-encrypted password to indicate if the password is valid or not.

d. Order

Properties:

- user_ID:UserModel _id
- cart:CartModel Array - The items contained in the User’s order.
- datetime_requested:Date
- datetime_fulfilled:Date
- status:String - “in-progress”,”shipped”,”OutForDelivery”,”cancelled”

e. Product

Properties:

- name:String
- description:String
- price:Number
- quantity:Number

f. Request

Properties:

- employee_id:EmployeeModel _id - Employee who made the request.
- product_id:ProductModel _id - Product requesting to be changed.
- new_quantity:Number
- datetime_requested:Date
- datetime_resolved:Date
- status:String

g. Ticket

Properties:

- user_ID:UserModel _id - User who raised the ticket.
- reason:String
- datetime_raised:Date
- datetime_resolved:Date
- status:String - “in-progress”, “resolved”

h. User

Properties:

- firstName:String
- lastName:String
- user_id:String
- u_username:String
- u_password:String

- address:String
- phone_number:Number
- date_of_birth:String
- locked:Boolean
- funds:Number - Current funds in the User's account, separate from User's accountN balance.
- failedAttempts:Number
- currentCart:CartModel Array - To store the user's cart upon logout.
- accountN:Number - Account for funds to be retrieved from.
- balance:Number - Balance of the account.

Methods:

- encryptPassword: Encrypts User password and returns the corresponding hash.
- validPassword: Test the candidatePassword against the stored-encrypted password to indicate if the password is valid or not.

II. DB-Tables

The Tables are MongoDB collections and are created based off of all of the Backend Models except for the Cart. Collection implementation is as simple as creating the collection if it does not exist and then populating it with data based on its corresponding Mongoose Model/Schema. A default Admin is already populated into the backend when the server first starts. The information on this account that can be used for testing is listed just after the GitHub Repository link in "Overview".

III. Frontend Classes

These are essentially interfaces; classes with only properties and no methods. All methods are handled with Angular Services. To avoid repetition, I will only be including what is not in the Backend Models. Because Mongoose automatically generates _id's for newly inserted data in a collection, I will also not be including those. All Backend Models with CartModel Arrays are the same in the frontend, except they are Arrays of Cart Objects.

a. Cart

Properties:

- product:Product Object Reference
- Quantity:number

3. Admin SignIn/Logout/Authentication

I. Sign In

Upon signing in, the frontend Angular Admin Login Component would call the Admin Service, which would send the username and password to the backend. The username would be tested against all Admin a_username's, and if one is found, then the password sent would be encrypted and compared to the stored a_password. If they are the same, the Admin would be authorized and an adminToken would be sent to the frontend to confirm full stack authorization. This activates the Admin AuthGuard. This authentication would persist throughout the Admin's session.

II. Logout

Upon Logout, the adminToken would be cleared from the frontend. This would cause the frontend Angular Admin AuthGuard to deactivate and the client would be sent back to the homepage. In this case, no interaction with the backend is necessary.

4. Admin Panel

This is a simple Angular Navbar loaded as a Component at the top of the Admin's session. I am not responsible for Employee/Report management, therefore those sections will be brief. When an Admin is signed in, there is an indicator on the right-side of the Navbar next to the Logout button that indicates which Admin is currently signed in. This remains consistent throughout the Admin's session.

Navigation Items:

I. Product Management

A Component which contained all of the Components necessary for Product CRUD Operations. Within the main Component there is a Selector that the Admin can use to determine which CRUD Operation they want to perform on Products. This selection determines which Product Component is displayed and the View Products Component is always displayed regardless of the selection. This allows for easy Product referencing on the client's side.

II. View Requests

A Component which allowed the Admin to View/Resolve/Update Requests. These Requests are generated by Employees and are generated when it is necessary to change the quantity of a Product.

III. Add Employee

A Component allowing the Admin to add new Employees to the system.

IV. Delete Employee

A Component allowing the Admin to delete Employees from the system.

V. Generate Reports

A Component allowing the Admin to generate Reports. The Reports are loaded as Orders and could be sorted by various means.

5. Product CRUD Operations

I. View Products

Contains a Template-Driven Form which the Admin can use to either view all Products or a Product based on its index in a Products Array. The Products are displayed in a table which is automatically generated whenever the Component is initialized. If the Admin tries to view a Product of an index that does not exist, an alert is thrown. From the Admin's viewpoint, it seems as though the index itself is tested, however in Angular the _id of the Product is retrieved and is tested against the Products Array. If the _id is not within the Array, then the alert is thrown telling the Admin to use a valid _id.

II. Add Product

Contains a Template-Driven Form which the Admin can use to add a new Product to the system. The form is not allowed to be submitted until all of the required Product fields are populated. These are the Name, Description, Price and Quantity. The minimum of the Price and the Quantity are 0 and if the Admin attempts to submit a number less than this, an error is thrown. If no errors occur, the Product is then added to the backend DB through the Product Service. Once the Product is successfully added, the Products Table is refreshed.

III. Update Product

Contains a Template-Driven Form which the Admin can use to update Product Quantities. The form cannot be submitted until all fields are filled in. These are Product _id and the new_quantity. The Product _id is tested for existence, and if it exists then the Product quantity is tested against the new_quantity. If they are the same, the quantity is not updated. If they are not the same, then the quantity is changed. Minimum new_quantity is 0 and an alert will be thrown if it is less than that.

IV. Delete Product

Contains a Template-Driven Form which the Admin can use to delete Products. The form cannot be submitted until the Product_id field is populated. If the Product exists, then it will be deleted. If not, then an alert is thrown telling the Admin that the Product they are trying to delete does not exist and no Product will be deleted.

6. Request Management

I. View Request

Contains a Template-Driven Form which the Admin can use to either view all Requests or a Request based on its index in a Requests Array. The Requests are displayed in a table which is automatically generated whenever the Component is initialized. If the Admin tries to view a Request of an index that does not exist, an alert is thrown. From the Admin's viewpoint, it seems as though the index itself is tested, however in Angular the _id of the Request is retrieved and is tested against the Requests Array. If the _id is not within the Array, then the alert is thrown telling the Admin to use a valid _id.

II. Resolve Request

This is done via buttons generated by the View Requests Table. Once the button is clicked to resolve a Request, the Product quantity is retrieved from the Products Array and is tested against the Request new_quantity. The Product is also tested for existence. If the Product does not exist, then all Requests with nonexistent Products are automatically removed from the View Requests Table and the Admin is notified. If the Product exists and if the quantities match, then the Component calls the Request service to update the Request status to "resolved" and the Resolve button disappears on the table. At this point, a new Delete (X) button appears on that Request in the View Requests Table. The View Requests Table is regenerated to refresh the Requests.

III. Delete

This is done via buttons generated by the View Requests Table. The button is only visible for Requests that have been resolved (status == "resolved"). When the button is clicked, the Request's Product is tested for existence. If the Product does not exist, then all Requests with nonexistent Products are automatically removed from the View Requests Table and the Admin is notified. If it does exist and the Request status is resolved, then the Request service is called and the Request is deleted.

Test Cases

Most test cases were manual. My implementation was tested by adding products through its respective Component and using the other Product Management Components to C(RUD) the newly added Product. I manually inserted Requests into the backend for testing my View/Resolve/Delete Requests implementation because I was not responsible for adding

Requests. I tested adding a Product without filling in all of the fields, which is impossible because the submit button is disabled in that case. I tested adding a Product with negative quantities/prices and an error was thrown to the Admin and the Product was not added. I tried viewing a Product that does not exist and an alert was thrown to the Admin. I tried updating a Product that does not exist and an alert was thrown to the Admin. I tried updating a Product with a new_quantity that was less than 0 and it was unsuccessful. I attempted to delete a Product that does not exist and an alert was thrown to the Admin.

As far as the Requests go, I tried viewing a Request that did not exist and an alert was thrown to the Admin. I tried resolving a Request whose Product did not have a matching quantity and an alert was thrown to the Admin. I tried resolving a Request whose Product did not exist, an alert was thrown to the Admin and all Requests with nonexistent Products were deleted. I tried deleting a Request with a nonexistent Product, an alert was thrown indicating that more Requests with nonexistent Products existed and all Requests with nonexistent Products were deleted.

As far as Admin Signin/Logout goes, I tried signing in with an admin username that did not exist and an alert was thrown. I tried signing in with a valid admin username but an invalid password and an alert was thrown. I tried manually navigating to Admin Components through the URL without being signed in and I was immediately redirected to the homepage.

Conclusion

Future Work

There are some occasional HTTP Request errors that are thrown from the backend when interacting with the Request and Product Services. This is because, in some cases, the HTTP is being sent a “text” ResponseType when it is expecting a JSON ResponseType. All it would take to fix is a few small alterations of res.send/res.write → res.json in the Request and Product backend Controllers.

I would also like to allow dynamically updating Product quantity based on a Request. I would like for the Admin to be able to click the Resolve button in the View Request Table and then be sent to the Update Products Component page with the Product _id and the new_quantity fields populated with the Request’s product_id and new_quantity respectively. The Admin could then simply click the Update button and be sent back to the View Requests Component with the Request resolved and the View Requests Table updated to match the changes.

USPs

- Proper Authentication/Security - This is necessary for any product or service to go public. Nobody wants to partake in a service that could leak their personal information. This is why my project has password encryption before it is stored. That way, even if the backend DB is hacked, then the hacker will only know the passwords if they know how to exactly reverse-engineer what was implemented in the backend with the BCRYPT package.
- Clean Frontend - The frontend of my part of the project is not overly complicated, is easy to read and navigate, and is clean. This makes website navigation easy for a client and is always essential to a popular website/web service.
- Scalable and Easily Modifiable - This project is very easily scalable because every functionality of the project is separated by Components, Services, Controllers, Models, and Classes. Adding additional functionality is as simple as adding a new function in a Service/Controller, changing a few properties in a Model/Class, or generating new Components and connecting them with the Angular Router. The entire Employee section of the project could be removed and it would not affect the User/Admin sections of the project at all and vise-versa.