

1. Introduction

This is a web application used to simulate a health food store. This application utilizes mostly JavaScript as well as small samples of HTML and CSS.

2. Project Architecture

This application was created using the “MERN” stack, this is a web development framework used by developers to create full stack web applications it consists of the following components

- **MongoDB Atlas:** A document-oriented, No-SQL database used to store the application data in the cloud.
- **NodeJS:** The JavaScript runtime environment. It is used to run JavaScript on a machine rather than in a browser
- **ExpressJS:** A framework layered on top of NodeJS, used to build the backend of a site using NodeJS functions and structures. Since NodeJS was not developed to make websites but rather run JavaScript on a machine, ExpressJS was developed.
- **ReactJS:** A library created by Facebook. It is used to build UI components that create the user interface of the single page web application.



the user interacts with the ReactJS UI components at the application front-end residing in the browser. This frontend is served by the application backend residing in a server, through ExpressJS running on top of NodeJS.

Any interaction that causes a data change request is sent to the NodeJS based Express server, which grabs data from the MongoDB database if required, and returns the data to the frontend of the application, which is then presented to the user.

I chose this framework as it allowed me to quickly develop this application and is very flexible, also the fact that everything is done in JavaScript helped me develop faster. There are plenty of third party plugins that can be used with this stack in order to create web applications, for example “ReactStrap” which make use of the “Bootstrap” CSS library to create reusable React Components.

3. Description of Functionality

This application simulates an online health food store, it has 2 types of user Administrator and Customer. Both customer and admin users are able to login, view items and search and sort the items in both ascending and descending order based on an attribute of the item. Both customer and admin users can edit the stock levels, customers can purchase items which affects stock levels and admins can add or remove stock.

This application makes use of ExpressJs's router object, this allowed me to create routes that would allow me to create routes that could be called in a sense my own API. In ReactJS I used a third party library called axios.js. Axios is a Javascript library used to make HTTP requests from node.js or XMLHttpRequests from the browser that also supports the ES6 Promise API.

Axios HTTP request made from react

```
removeStock={()=>
{
  const Item = {
    Title: this.state.title,
    Q: this.state.quantity
  }

  axios.post("/Admin/RemoveStock" , {Item})
    .then(this.getItems()).then(this.getItems()).then(this.setState({removeStock:false , title:"", quantity:0}))
}
```

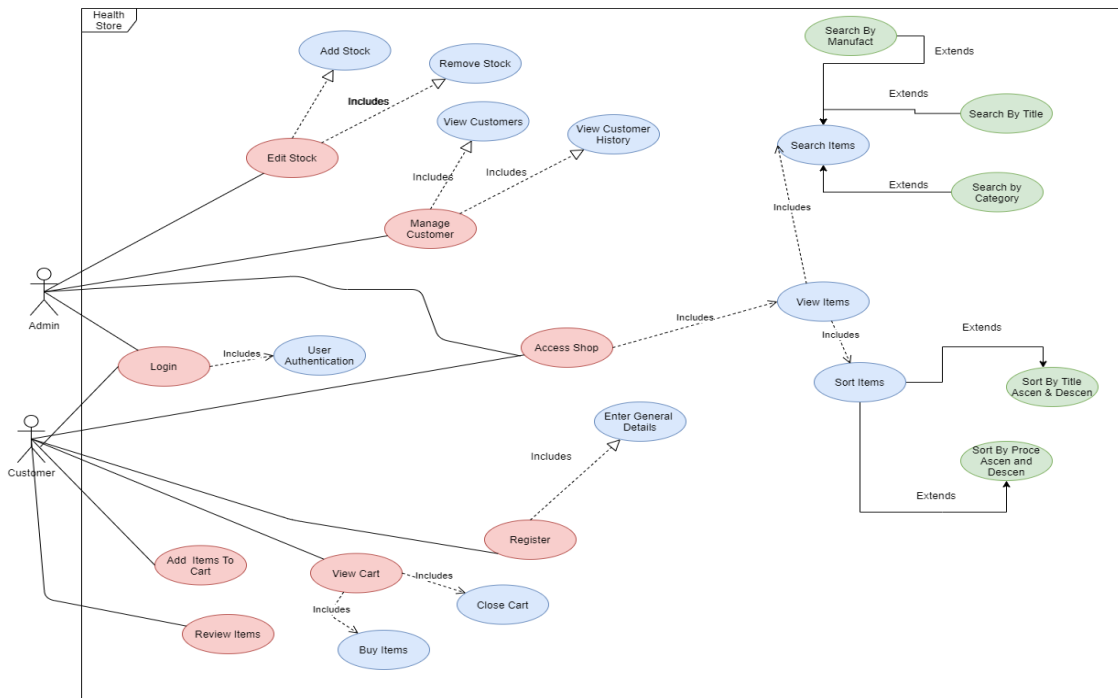
Here a JSON object is created that takes in the current components state of title and quantity , this object is then passed into the axios request ,after the request is completed the list is re rendered and the title, quantity and Dialog visibility values are reset.

Express route example:

```
router.post("/Admin/RemoveStock" , (req , res)=>
{
  Item.findOne({Title:req.body.Item.Title})
    .then(item =>
    {
      item.Stock = item.Stock - req.body.Item.Q
      item.save()
    })
})
```

Here Item class searches for the item that has the name that matches the title that is passed in from the request body made from the front end when this item is found the stock is adjusted to remove the quantity also passed in from the request body the item then records this change by saving

- Use Case Diagram



4. Use Cases

a. Shared Functionality

i. Login

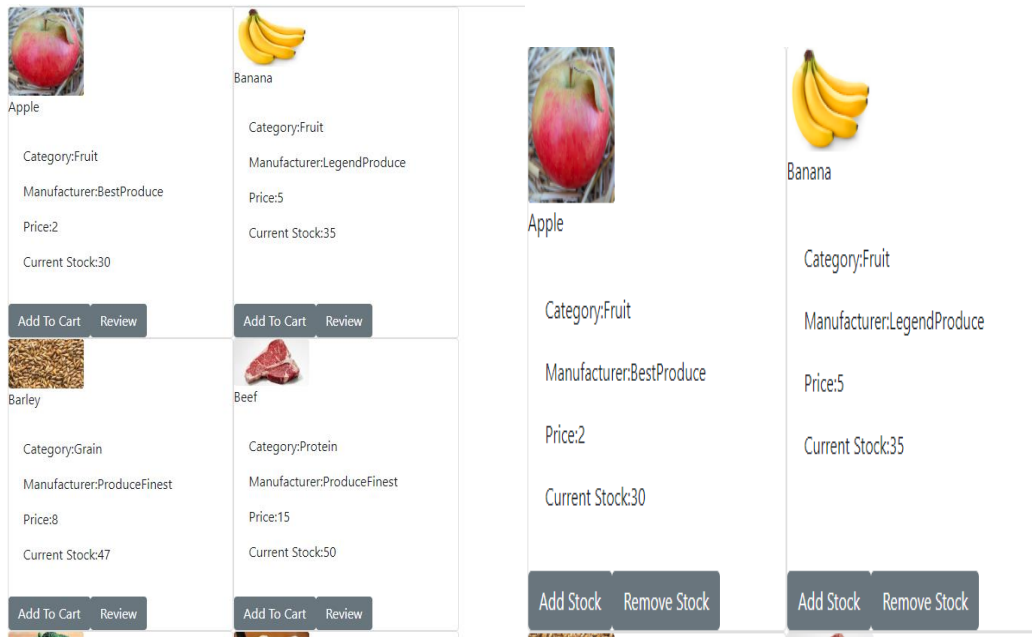
Enter the following details to Login

Email:

Password:

Both customer and admin can Login by entering their email and password if any fields are empty an error message is displayed or if the user does not exist.

ii. View Items



When users have successfully logged in, they can view all the items in the store however depending on if it's a Customer or Admin user the buttons will be different customers will see "Add To Cart" and "Review Buttons" whereas Admin users will see "Add Stock" and "Remove Stock"

iii. Search Items

Search By...

Title

Search bar and dropdown

ch

Title

Title - Ascending

Chicken

Category:Protein

Manufacturer:BestProduce

Price:18

Current Stock:48

Add Stock Remove Stock

Mozzerella Cheese

Category:Dairy

Manufacturer:BestProduce





Price:10

Current Stock:50

Add Stock Remove Stock

Title search results

Category
Title - Ascending



 <p>Beef</p> <p>Category:Protein</p> <p>Manufacturer:ProduceFinest</p> <p>Price:15</p> <p>Current Stock:50</p> <div>Add Stock Remove Stock</div> 	 <p>Chicken</p> <p>Category:Protein</p> <p>Manufacturer:BestProduce</p> <p>Price:18</p> <p>Current Stock:48</p> <div>Add Stock Remove Stock</div> 
---	--

Category Search results

Users can search for specific items using a search bar and a dropdown, the dropdown allows users to select the type of attribute they would like to search items by as seen above three options are present title, category and manufacturer

iv. Sort Items

Title - Descending

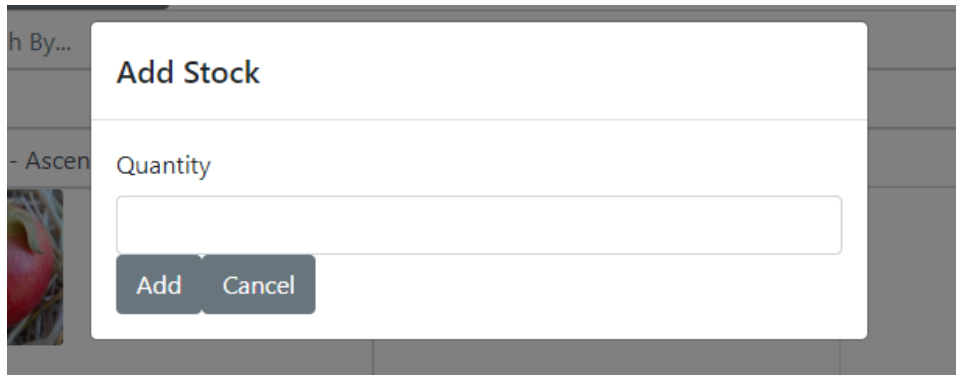
 <p>Yoghurt</p> <p>Category:Dairy</p> <p>Manufacturer:ProduceFinest</p> <p>Price:7</p> <p>Current Stock:49</p> <div>Add To Cart Review</div>	 <p>Wheat</p> <p>Category:Grain</p> <p>Manufacturer:BestProduce</p> <p>Price:1</p> <p>Current Stock:49</p> <div>Add To Cart Review</div>
--	--

Items in descending order by title

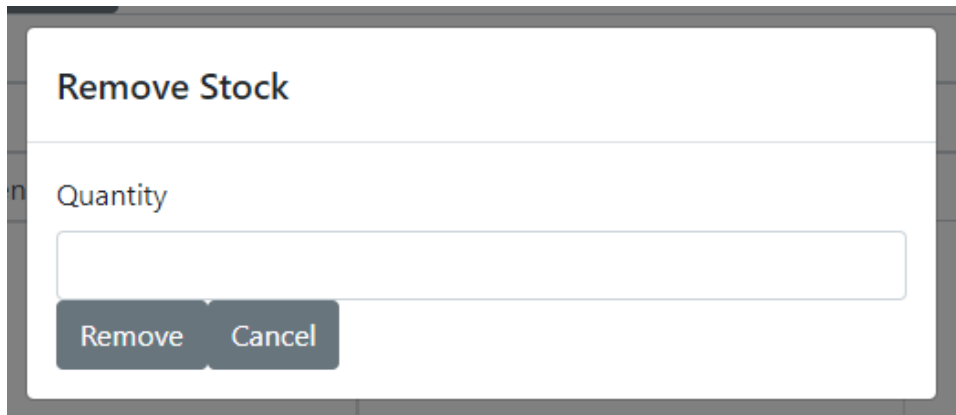
Using another dropdown users can sort items by title or price both ascending or descending in value

b. Admin Functionality

i. Add/Remove Stock



A dialog box titled "Add Stock" with a "Quantity" label and an input field. Below the input field are two buttons: "Add" and "Cancel".

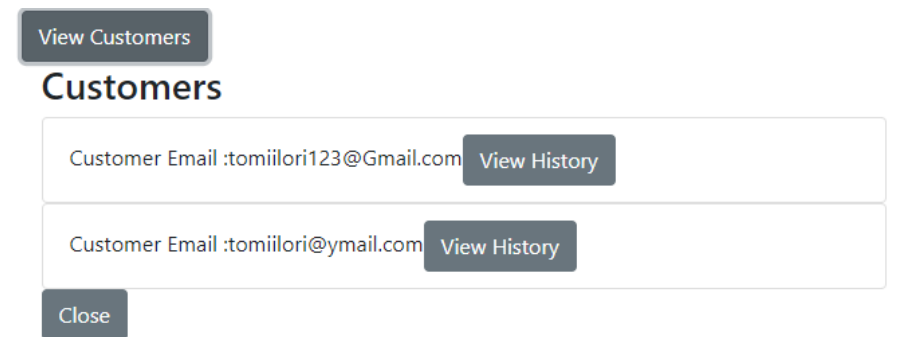


A dialog box titled "Remove Stock" with a "Quantity" label and an input field. Below the input field are two buttons: "Remove" and "Cancel".

Dialog windows

An admin has the ability to add and remove stock when they click on either button to add or remove stock a dialog window is open requesting the quantity the admin wants to add or remove. This then adds or removes to specified quantity from the database

ii. View Customers & Customer History



A dialog box titled "View Customers" containing a list of customer emails and a "View History" button for each. At the bottom is a "Close" button.

Customer Email	Action
Customer Email :tomilori123@Gmail.com	View History
Customer Email :tomilori@gmail.com	View History

List of customers

When the admin clicks the View Customers button a list of each customer email and a button to view their history is rendered. When the View History button is pressed a list containing the customers purchase history is rendered

Customer Email :tomiilori@ymail.com

View History

Close

History

Date Of Purchase:14/04/2020Customer Email :tomiilori@ymail.com

Close

List of customer history

c. Customer Functionality

i. Register

Welcome Please Enter the following details to register

Email:

Password:

Shipping Address:

Payment Option:

Submit

A customer enters the following details to register , If any fields are empty the user is asked to fill all fields, if they are all filled the user is created and a message is shown alerting the user of their successful registration.

ii. View Cart

Items in cart 1

View Cart

Cart

Apple

Buy Items

Close Cart

When customer clicks the view Cart button the cart is rendered with each item inside it

iii. Add To Cart



Banana

Category:Fruit

Manufacturer:LegendProduce

Price:5

Current Stock:35

Add To Cart

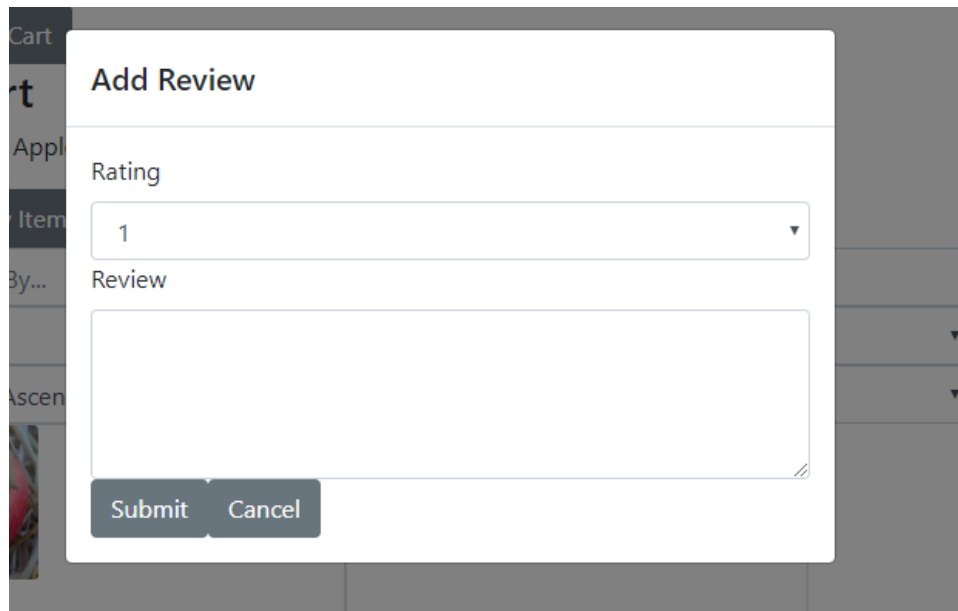
Review

Each Item has a button which allows the user to add an item to the cart.

iv. Buy Items

When the user clicks the buy item button each item in the carts stock level is adjusted and the cart is cleared

v. Add Review

A screenshot of a web application showing a modal dialog box titled "Add Review". The dialog has a white background and rounded corners. It contains a "Rating" section with a dropdown menu currently showing the value "1". Below the rating is a "Review" section with a large, empty text area. At the bottom of the dialog are two buttons: "Submit" and "Cancel". The background of the application is dark gray, and parts of a sidebar menu are visible on the left, including items like "Cart", "Appl", "Item", "By...", and "Ascen".

Dialog Window

When a customer clicks the review button a window which takes in a rating value between 1-5 and a text area which takes in any comments, when the user clicks the submit button the review is added to the database for the corresponding item.

5. Database Schema

Even though mongoDb is a NOSQL database, you can add structure to it using a third party library called mongoose. Mongoose is an Object Data Modeling (ODM) library for MongoDB and Node.js. It manages relationships between data, provides schema validation, and is used to translate between objects in code and the representation of those objects in MongoDB.

i. Customer class

```

//Requiring Mongoose
const mongoose = require("mongoose")

//Customer Schema for customer fields
const CustomerSchema = new mongoose.Schema(
  {
    Password:{
      type:String,
    },
    Email:{
      type:String,
    },
    Shipping:
    {
      type:String,
    },
    Payment:
    {
      type:String,
    },
  }
)

//Creating a user object based off of schema
let Customer = mongoose.model("Customer" , CustomerSchema)

//Making object exportable
module.exports = Customer

```

Email – Customer email

Password – Customer Password

Shipping – Customer Shipping address

Payment – Customer Payment Type

ii. Admin class

```

1  //Requiring Mongoose
2  const mongoose = require("mongoose")
3
4  //Admin Scehema for customer fields
5  const AdminSchema = new mongoose.Schema(
6
7      Password:{
8          type:String,
9      },
10     Email:{
11         type:String,
12     },
13 },
14
15 )
16
17
18 //Creating a admin object based off of schema
19 let Admin = mongoose.model("Admin" , AdminSchema)
20
21 //Making object exportable
22 module.exports = Admin

```

Email – Admin email

Password – Admin Password

iii. **Item class**

```

//Requiring Mongoose
const mongoose = require("mongoose")

//Customer Scehema for customer fields
const ItemSchema = new mongoose.Schema(
  {
    Title:{
      type:String,
    },
    Manufact:{
      type:String,
    },
    Price:
    {
      type:Number,
    },
    Category:
    {
      type:String,
    },
    ImageLink:
    {
      type:String,
    },
    Stock:
    {
      type:Number
    }
  }
)

//Creating a user object based off of schema
let Item = mongoose.model("Item" , ItemSchema)

//Making object exportable
module.exports = Item

```

Title - Item title

Category – Item Category

ImageLink – Link to Item image

Stock – value of the Item stock

Price – Item price

Manufact – Items manufacturer

iv. **Purchase Order class**

```

const poSchema = new mongoose.Schema(
  {
    Date: {
      type: String,
    },
    Email: {
      type: String,
    }
  }
)

//Creating a user object based off of schema
let PO = mongoose.model("PO" , poSchema)

//Making object exportable
module.exports = PO

```

Email – Customer email

Date – Date the order was made

v. Review class

```

const ReviewSchema = new mongoose.Schema(
  {
    ItemName: {
      type: String
    },
    Review: {
      type: String,
    },
    Rating: {
      type: Number,
    }
  }
)

//Creating a user object based off of schema
let Review = mongoose.model("Review" , ReviewSchema)

//Making object exportable
module.exports = Review

```

ItemName– Name of the item Reviewed

Rating- The item rating given by user

Review – The item Review

6. Software Patterns

Although I managed to get the application to function completely , it turned out to be my downfall in the sense that I was unfamiliar with how to implement software patterns in the application as I would with JavaScript. I could not see clearly where any of the usual GOF patterns would fit in with my application, however I did manage to identify a number of software patterns that are affiliated with this technology stack

i. MVC Pattern

This pattern is used by the MERN stack in order to organize and separate logic between the client side of the application and the logic side of the application

- Model

The files under the Model folder in this application would be the model files these JSON objects carry data and are able to change if the data inside the object changes.

- View

The files created using ReactJS would serve as the view of this stack, these files represent the user interface and any data that the user can interact with.

- Controller

The files under the Routes Folder would serve as the applications controller, I divide the applications logic into separate files in order to reduce the length of the files but it can be done in one file to serve as the central controller which handles the applications logic such as return items from the database or make changes to the items stock values

ii. Module Pattern

The Module Pattern is a commonly used Design Pattern which is used to wrap a set of variables and functions together in a single scope. In my application this used on multiple occasions. To use this pattern you must use the keyword “require” which allows you to bring in an object that represents a file with variables and functions attached to it.

An example would be the use of the database objects in my routes files, the actual object model must be brought into the route file using the keyword require this then gives the route file the ability to use any methods or access any variables that this database object may contain

```
const Customer = require("../Models/Customer")
const Item = require("../Models/Item")
const po = require("../Models/po")
const Review = require("../Models/Review")
```

For example in the Customers.js route file I require each one of the Database objects in order to be able to manipulate them in the API calls that I will write by storing them as local variables

```
router.post("/Customer/AddReview/" , (req , res)=>
{
    const newRev = new Review()

    newRev.ItemName = req.body.Item.Name
    newRev.Rating = req.body.Item.Rating
    newRev.Review = req.body.Item.Review
    newRev.save()
})
```

Here a variable is created that is set to represent a new Review database object, as this variable now represents one of these Review objects all the variables and the .save() method are accessible and can be used.

iii. State Pattern

React Components make use of the state pattern in order to make dynamic web applications , the state contains any values that may be changed by the user in the application and can be changed at any time throughout the application. If the user of the application changes a value in the state the behavior of the React component will change an example of this would be creating alerts to deal with user errors. One of the best features of React is that any change to the state of a component will trigger a re rendering of the page which can be used to automatically load changes to the web page.

```
export class Register extends Component {

    state = {
        email:"",
        password:"",
        pay:"Visa",
        shipping:"",
        EmptyError:null,
    }
}
```

Welcome Please Enter the following details to register

Email:

Password:

Shipping Address:

Payment Option:

Submit

This component represents the Web page that allows the user to register , it keeps track of the following values on the screen and has a value called EmptyError.

```
Register= async e =>
{
  e.preventDefault();
  const {email , password , shipping , pay} = this.state
  if(email =="" || password =="" || shipping =="" || pay =="" )
  {
    this.setState({EmptyError:true})
  }
}
```

Here is the register method we see that local values can be created to represent the values in the components state , if any of these values are empty a method called .setState() is called which allows the change of any value in the state. Here the EmptyError is set to true which will have an affect on what the webpage displays

```
</FormGroup>
{this.state.EmptyError==true ? <Alert color="danger">Enter All Fields</Alert> : null}
```

Here we see what is called conditional rendering which means certain parts of a web page are rendered when certain conditions are met for example the fields meant are empty this will result in an alert rendering

Welcome Please Enter the following details to register

Email:

Password:

Shipping Address:

Payment Option:

Submit

Enter All Fields

As we can see when the fields are empty the internal state of the component has been changed which in turn creates the alert due to the value of EmptyError now being true, this is an example of the State pattern as the behavior of the web page changes in accordance to a change in the state.

iv. Chain of Responsibility Pattern

I use this pattern multiple times in my application by chaining methods this allows me to execute multiple methods back to back, where each subsequent method starts when the previous operation succeeds. In JavaScript this makes use of the Promise object which represents the eventual completion or failure of an asynchronous method.

```
buyItems=()=>  
  
    const cart = this.state.cart  
  
    axios.post(`Customer/buyItems/${this.state.email}`, {cart})  
    .then(res =>  
        console.log("items Bought")  
        .then(this.setState({cart:[]}))  
        .then(this.getItems())  
    )
```

Here in the method called when the user buys items in their cart we can see an axios request being made, at the end of this method .then() is called, inside this method an es6 arrow function is used in order to log that the items have been bought, .then() is called again and in this chain the state of the cart array is set to an empty array signifying the items have been bought, this changes the stock of whatever item has been purchased so I needed to call the getItems method to get the updated items in the database and return them to the user.

7. Link to repository

https://github.com/Lordjiggyx/SP_4