University of Missouri

Spring 2024 CMP_SC 4830/7830 Final Project

Ethan Mick, Aidan Puricelli, Spencer Berger, Matt Sibbit

Date: 05/04/2024

Table of Contents

1.0 Overview

Our group has built an online menu/website for a bar and restaurant establishment. The website will have a menu page, as well as separate pages for the bar’s events, and one for promotions for that week. We have appropriately implemented NodeJS, Express.js, MongoDB, and Routing.ts as they are needed.

2.0 How Requirements Are Met

2.1 Angular

1. Angular (points 40)You must build using Angular as your frontend. You will create at least 4

components

a. Header.component

b. 2 body.components

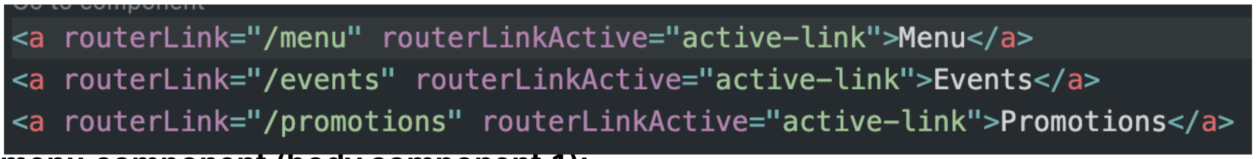
c. Footer.component

You will use the Header.component to store the routing for the application. You will use the body.components to send information between each other. Your footer only needs to contain the names of all the team members.

Our Project:

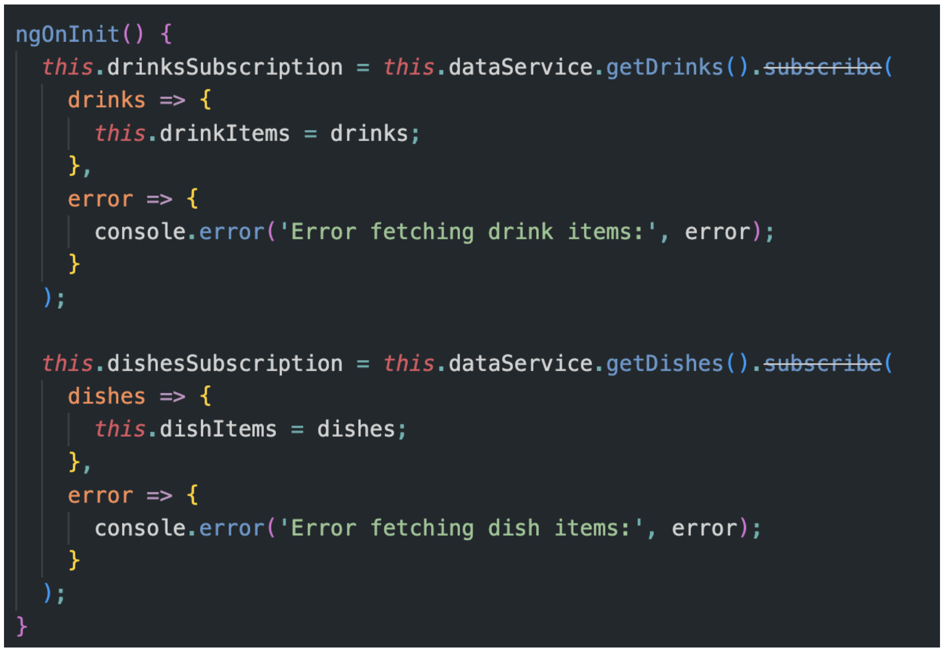
header.component:

Our header component consists of a simple navbar that allows the user to navigate between our pages (Menu, Events, Promotions). It uses Angulars ‘routerLink’ attribute to route the user to the desired page. It also uses Angulars builtin ‘routerLinkActive’ attribute to apply a specific CSS styling to whichever page is currently active.

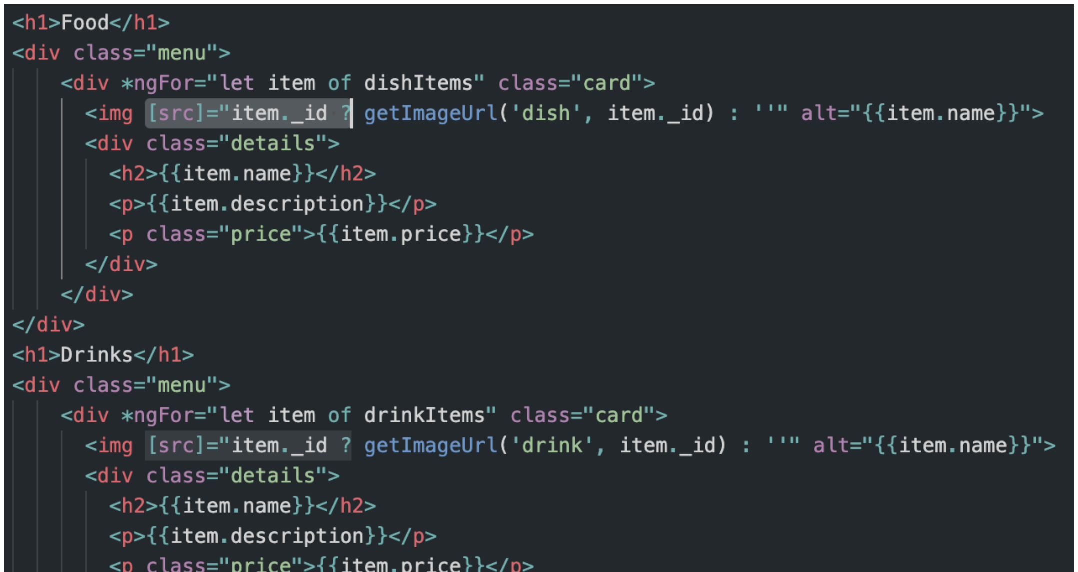

```
<a routerLink="/menu" routerLinkActive="active-link">Menu</a>
<a routerLink="/events" routerLinkActive="active-link">Events</a>
<a routerLink="/promotions" routerLinkActive="active-link">Promotions</a>
```

menu.component (body.component 1):

The menu component is the first body component in our application. It is responsible for displaying the 'dishes' and 'drinks' from our database. On initialization it retrieves the 'dish' and 'drink' models from our database by hitting the '/dishes' and '/drinks' endpoints in our api:


```
ngOnInit() {
  this.drinksSubscription = this.dataService.getDrinks().~~subscribe~~(
    drinks => {
      this.drinkItems = drinks;
    },
    error => {
      console.error('Error fetching drink items:', error);
    }
  );

  this.dishesSubscription = this.dataService.getDishes().~~subscribe~~(
    dishes => {
      this.dishItems = dishes;
    },
    error => {
      console.error('Error fetching dish items:', error);
    }
  );
}
```

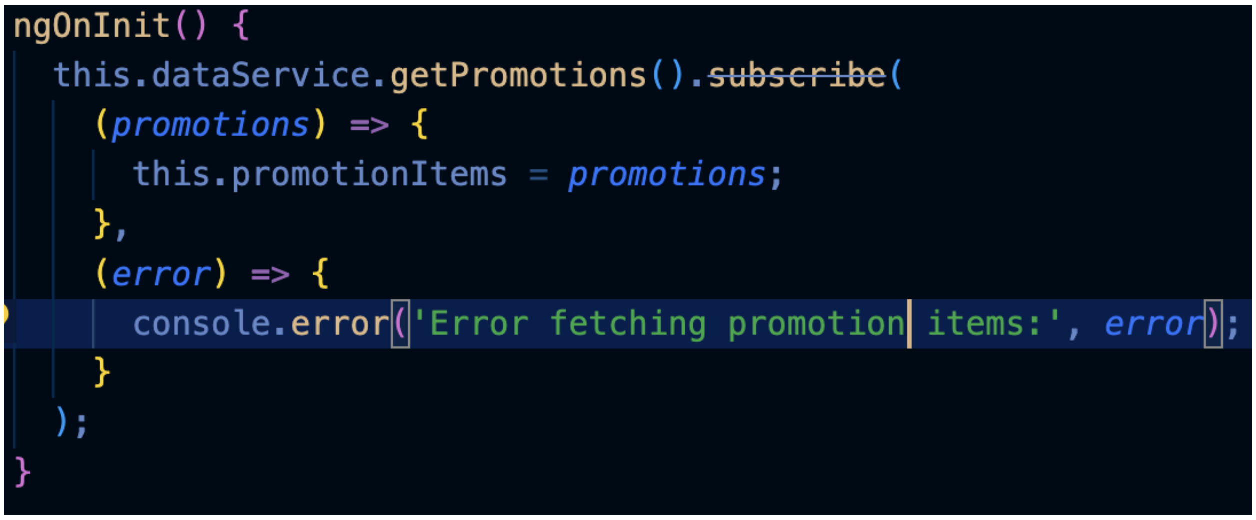
Then the data is displayed in the html of the page using the 'ngFor' directive. While being displayed the image for each drink or dish is retrieved from the '/images${type}/${itemId}' endpoint in our api and displayed on its respective card:


```
<h1>Food</h1>
<div class="menu">
    <div *ngFor="let item of dishItems" class="card">
      <img [src]="item._id ? getImageUrl('dish', item._id) : ''" alt="{{item.name}}">
      <div class="details">
        <h2>{{item.name}}</h2>
        <p>{{item.description}}</p>
        <p class="price">{{item.price}}</p>
      </div>
    </div>
</div>
<h1>Drinks</h1>
<div class="menu">
    <div *ngFor="let item of drinkItems" class="card">
      <img [src]="item._id ? getImageUrl('drink', item._id) : ''" alt="{{item.name}}">
      <div class="details">
        <h2>{{item.name}}</h2>
        <p>{{item.description}}</p>
        <p class="price">{{item.price}}</p>
```

Finally, after the page is destroyed the component unsubscribes from the drink and dish subscriptions.

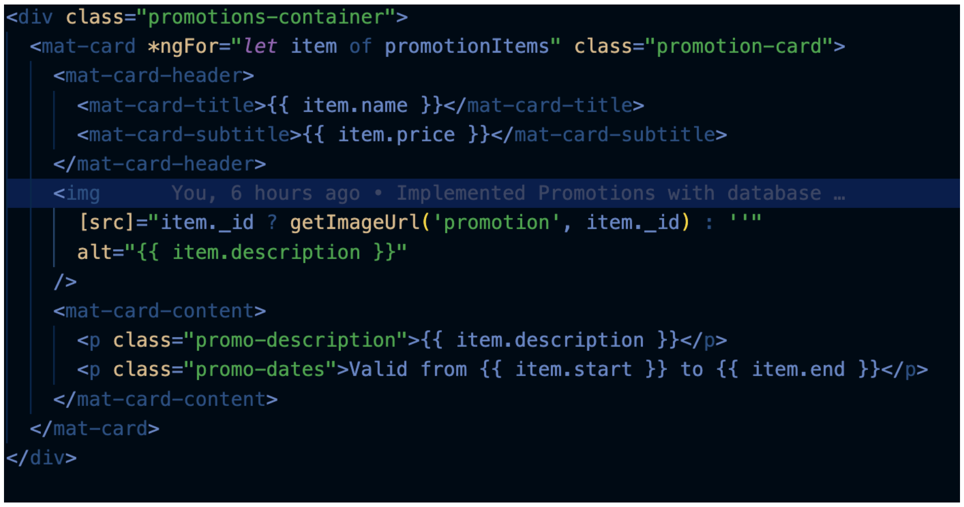
Promotions.component (body.component 2):

The promotions component is the second body component in our project. This component displays the current promotions available at our restaurant. On initialization it pulls the data from the observables that it is subscribed to and displays the promotions on the promotions page.



```
ngOnInit() {
  this.dataService.getPromotions().~~subscribe~~(
    (promotions) => {
      this.promotionItems = promotions;
    },
    (error) => {
      console.error('Error fetching promotion items:', error);
    }
  );
}
```

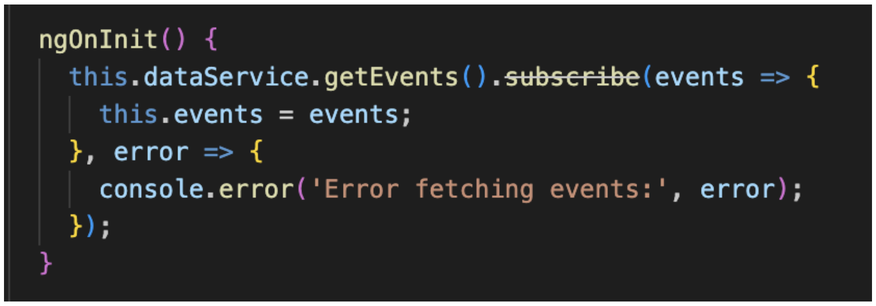
The data is displayed on the page using the matcard utility. Making it easy to format and add animations to.



```
<div class="promotions-container">
  <mat-card *ngFor="let item of promotionItems" class="promotion-card">
    <mat-card-header>
      <mat-card-title>{{ item.name }}</mat-card-title>
      <mat-card-subtitle>{{ item.price }}</mat-card-subtitle>
    </mat-card-header>
    <img      You, 6 hours ago • Implemented Promotions with database …
      [src]="item._id ? getImageUrl('promotion', item._id) : ''"
      alt="{{ item.description }}"
    />
    <mat-card-content>
      <p class="promo-description">{{ item.description }}</p>
      <p class="promo-dates">Valid from {{ item.start }} to {{ item.end }}</p>
    </mat-card-content>
  </mat-card>
</div>
```

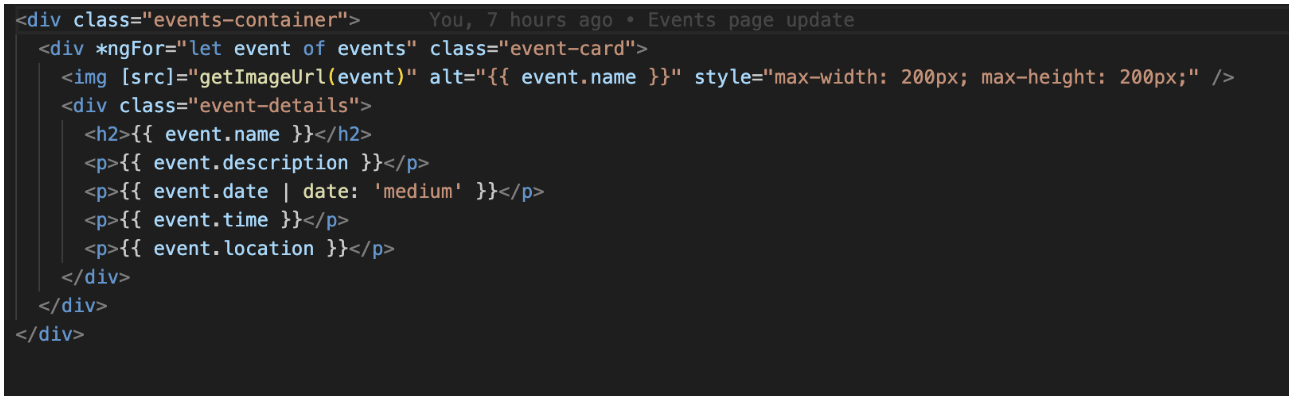
Events.component (body.component 3):

The events component is the third body component in our project. This component displays the upcoming events at the restaurant to attract new customers. It operates similarly to how the promotions.component does. The events are pulled out of the data that is acquired by subscribing to dataService.

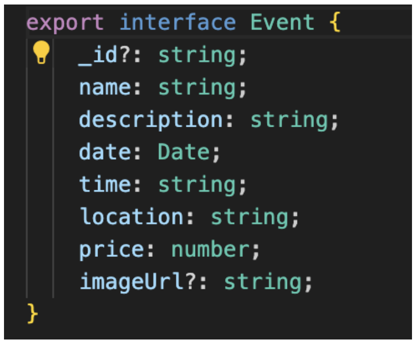


```
ngOnInit() {
  this.dataService.getEvents().~~subscribe~~(events => {
    this.events = events;
  }, error => {
    console.error('Error fetching events:', error);
  });
}
```

The data is brought to the front end using an *ngFor loop, allowing every item in the events array to be displayed in the same way. Attributes like “name” and “price” are easy to call upon because of the way we built our models.



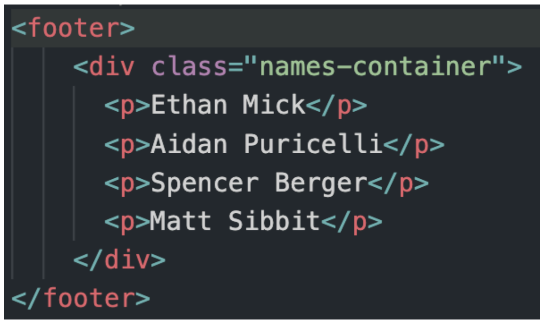
```
<div class="events-container">        You, 7 hours ago • Events page update
  <div *ngFor="let event of events" class="event-card">
    <img [src]="getImageUrl(event)" alt="{{ event.name }}" style="max-width: 200px; max-height: 200px;" />
    <div class="event-details">
      <h2>{{ event.name }}</h2>
      <p>{{ event.description }}</p>
      <p>{{ event.date | date: 'medium' }}</p>
      <p>{{ event.time }}</p>
      <p>{{ event.location }}</p>
    </div>
  </div>
</div>
```



```
export interface Event {
    _id?: string;
    name: string;
    description: string;
    date: Date;
    time: string;
    location: string;
    price: number;
    imageUrl?: string;
}
```

footer.component:

The footer component in our application consists of simple HTML code to display the group members names as defined by the project requirements. It is styled for mobile and desktop screens.



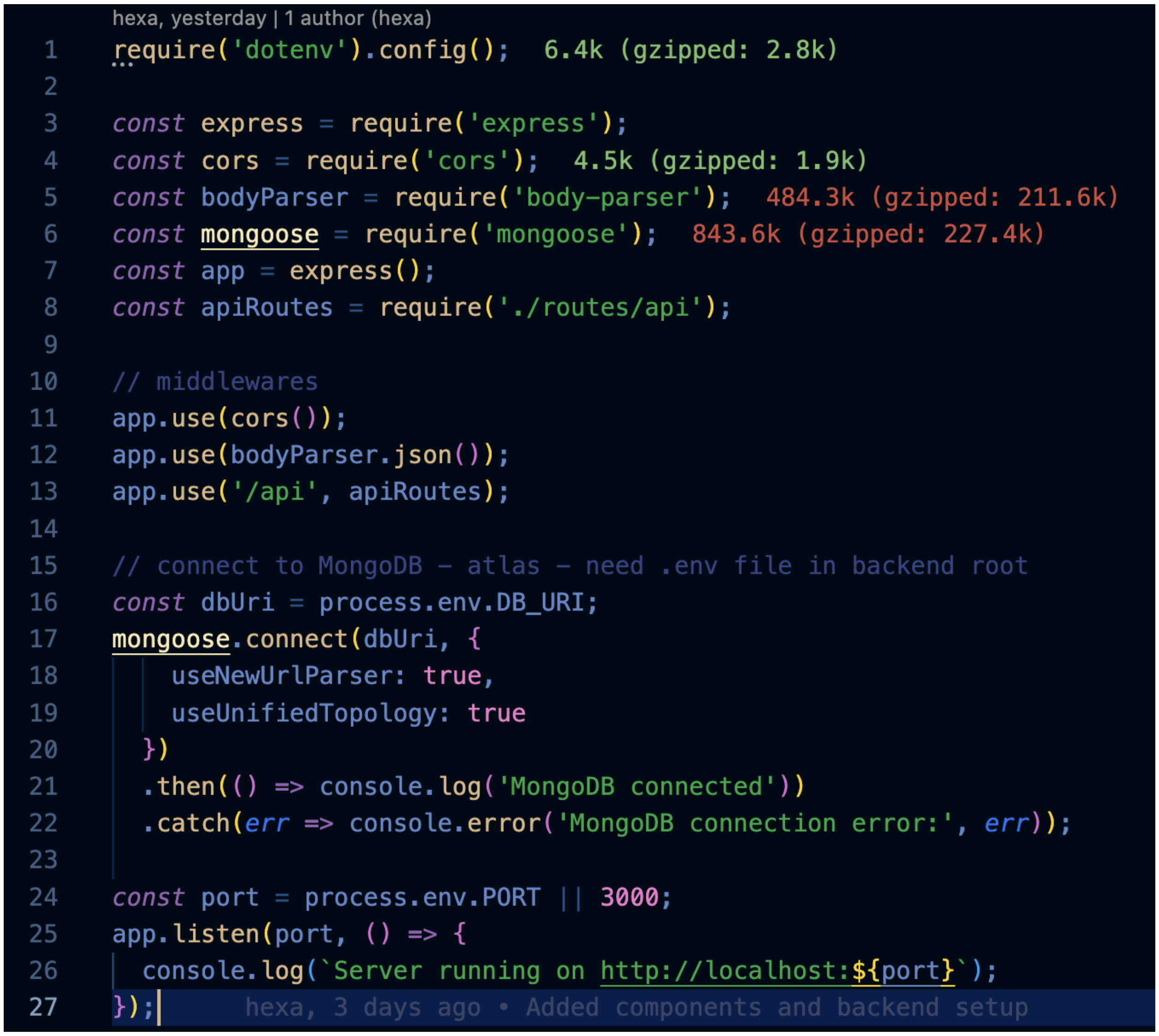
```
<footer>
    <div class="names-container">
      <p>Ethan Mick</p>
      <p>Aidan Puricelli</p>
      <p>Spencer Berger</p>
      <p>Matt Sibbit</p>
    </div>
</footer>
```

2.2 NodeJS

2. NodeJS (Points 25) You will use NodeJS to build your backend. Your node must use HTTP for the connection for the data between your database and your frontend.

Our Project:

1. **NodeJS Backend:** The code utilizes NodeJS to create the backend server. It imports necessary modules such as express, cors, body-parser, and mongoose which are commonly used in NodeJS applications.
2. **HTTP Connection:** The backend communicates with the frontend using HTTP. This is achieved through the use of express, which is a web application framework for NodeJS that provides features for building web servers. By defining routes and handling HTTP requests, the backend establishes communication with the frontend.
3. **MongoDB Connection:** The backend connects to a MongoDB database using Mongoose, a MongoDB object modeling tool designed to work in an asynchronous environment. The MongoDB connection string is retrieved from the environment variables (process.env.DB_URI) using dotenv to load variables from a .env file. This ensures that sensitive information such as database credentials are not hard-coded into the source code.



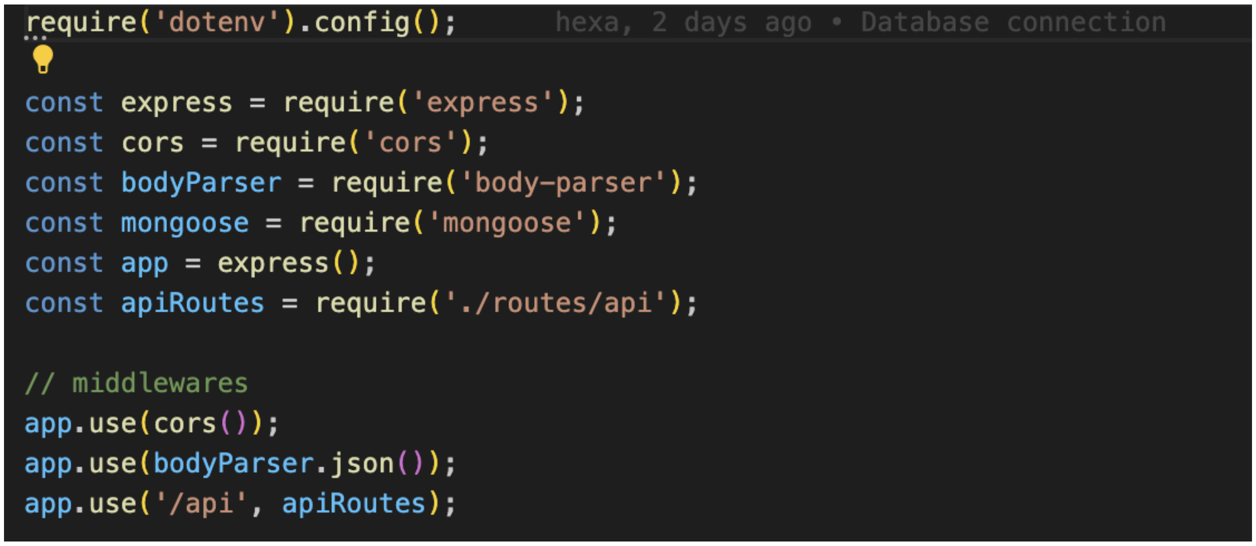
```
hexa, yesterday | 1 author (hexa)
require('dotenv').config();  6.4k (gzipped: 2.8k)

const express = require('express');
const cors = require('cors');  4.5k (gzipped: 1.9k)
const bodyParser = require('body-parser');  484.3k (gzipped: 211.6k)
const mongoose = require('mongoose');  843.6k (gzipped: 227.4k)
const app = express();
const apiRoutes = require('./routes/api');

// middlewares
app.use(cors());
app.use(bodyParser.json());
app.use('/api', apiRoutes);

// connect to MongoDB - atlas - need .env file in backend root
const dbUri = process.env.DB_URI;
mongoose.connect(dbUri, {
    useNewUrlParser: true,
    useUnifiedTopology: true
  })
  .then(() => console.log('MongoDB connected'))
  .catch(err => console.error('MongoDB connection error:', err));

const port = process.env.PORT || 3000;
app.listen(port, () => {
  console.log(`Server running on http://localhost:${port}`);
});      hexa, 3 days ago • Added components and backend setup
```

2.3 Express

3. Express.js (Points 25) You must use express to create restful API between your backend and your service.ts file to be used in your applications. There will be no direct connection between your backend node and your frontend. All communication and data transfer will be done on your http url from your node. The angular will not be able to connect and access the database.

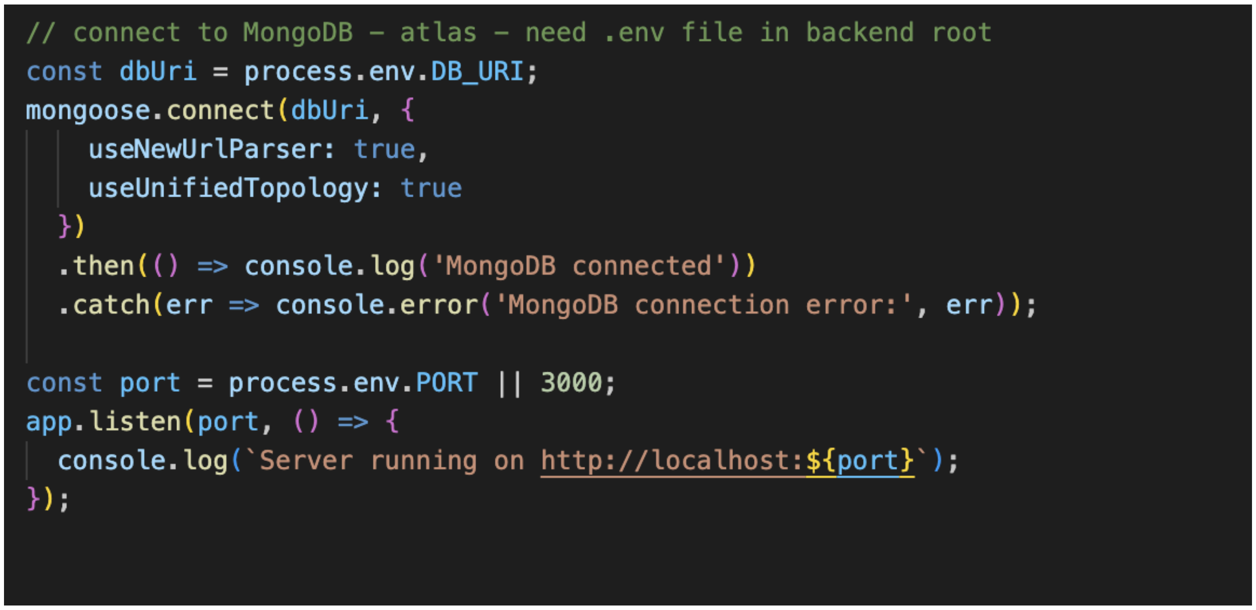
Our Project:

The server.js file serves as the entry point for our backend application. Here, we initialize an Express app, configure middleware such as cors and body-parser for handling requests, and define routes using the apiRoutes module.



```
require('dotenv').config();        hexa, 2 days ago • Database connection

const express = require('express');
const cors = require('cors');
const bodyParser = require('body-parser');
const mongoose = require('mongoose');
const app = express();
const apiRoutes = require('./routes/api');

// middlewares
app.use(cors());
app.use(bodyParser.json());
app.use('/api', apiRoutes);
```

Additionally, the app listens for incoming requests on a specified port, establishing the HTTP connection through which data is transferred between the backend and frontend.



```
// connect to MongoDB - atlas - need .env file in backend root
const dbUri = process.env.DB_URI;
mongoose.connect(dbUri, {
    useNewUrlParser: true,
    useUnifiedTopology: true
  })
  .then(() => console.log('MongoDB connected'))
  .catch(err => console.error('MongoDB connection error:', err));

const port = process.env.PORT || 3000;
app.listen(port, () => {
  console.log(`Server running on http://localhost:${port}`);
});
```

.env

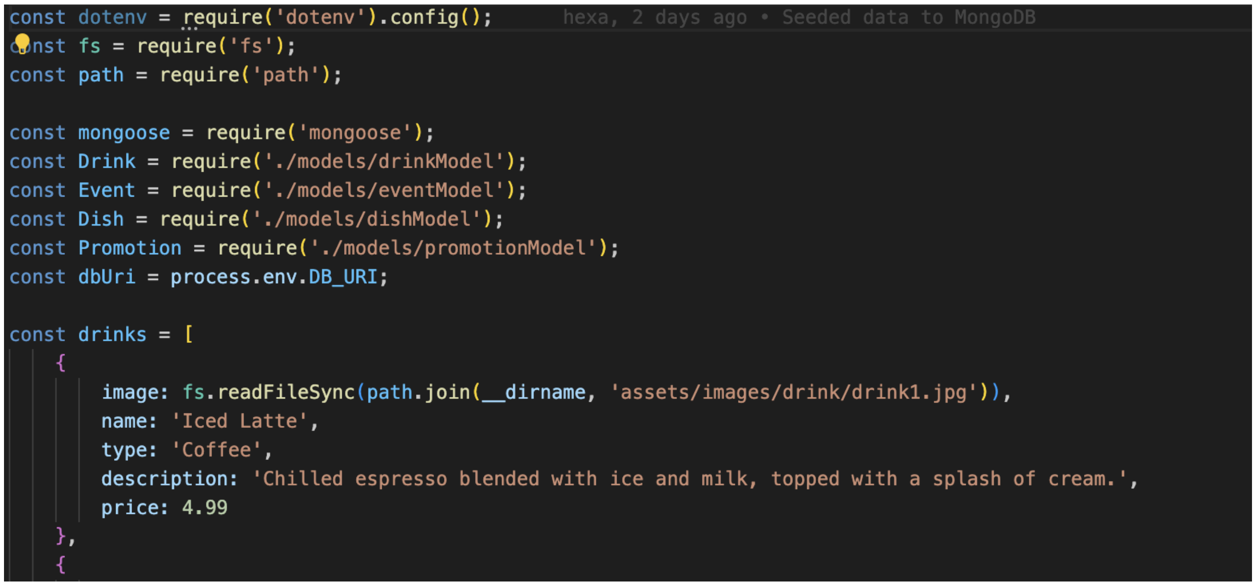
The backend connects to a MongoDB database hosted on the cloud using the Mongoose library. The database connection string is securely stored in the .env file and accessed via the process.env object.

This is the only line in the .env file:

DB_URI='mongodb+srv://ethanmick741:2DCuVGmZ0xyL7qUy@web-dev-2.07r753s.mongodb.net/restaurant_menu?retryWrites=true&w=majority&appName=web-dev-2'

Seed.js

To populate the database with initial data, we utilize the seed.js file. This script inserts predefined drink, dish, event, and promotion objects into their respective collections within the MongoDB database.



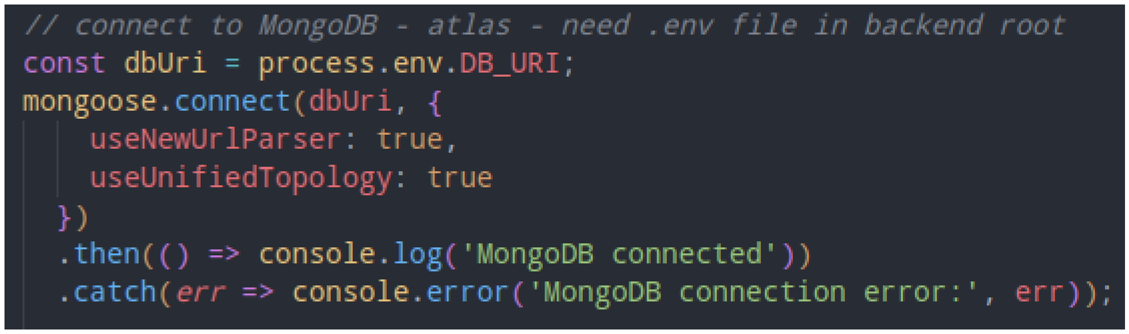
```
const dotenv = require('dotenv').config();      hexa, 2 days ago • Seeded data to MongoDB
const fs = require('fs');
const path = require('path');

const mongoose = require('mongoose');
const Drink = require('./models/drinkModel');
const Event = require('./models/eventModel');
const Dish = require('./models/dishModel');
const Promotion = require('./models/promotionModel');
const dbUri = process.env.DB_URI;

const drinks = [
    {
        image: fs.readFileSync(path.join(__dirname, 'assets/images/drink/drink1.jpg')),
        name: 'Iced Latte',
        type: 'Coffee',
        description: 'Chilled espresso blended with ice and milk, topped with a splash of cream.',
        price: 4.99
    },
    {
```

Each object contains essential information such as name, description, price, date, time, and image, ensuring that our application starts with a diverse range of content for users to explore.

2.4 MongoDB

4. MongoDB (Points 25) You must have a database running on the cloud that your application will connect to and add/remove information from. The data from the database must be seen in the website

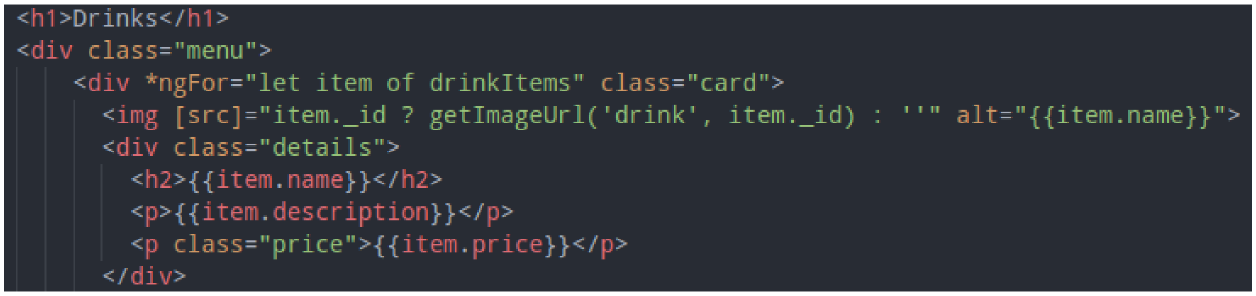
Our Project:



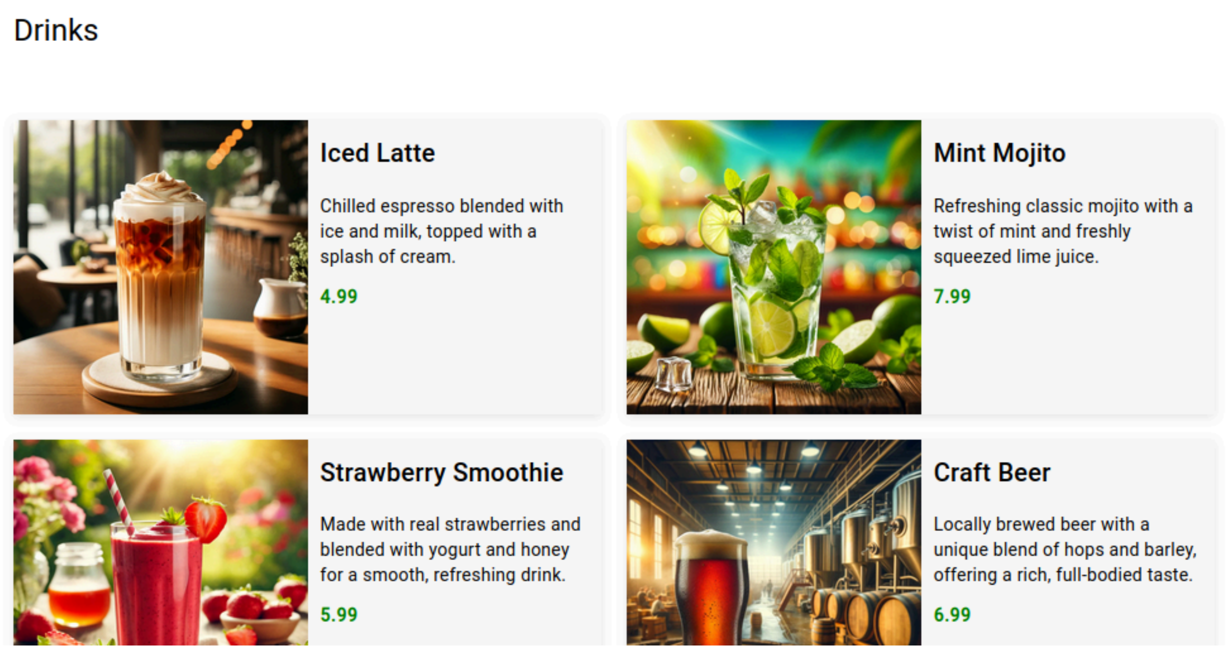
```
// connect to MongoDB - atlas - need .env file in backend root
const dbUri = process.env.DB_URI;
mongoose.connect(dbUri, {
    useNewUrlParser: true,
    useUnifiedTopology: true
  })
  .then(() => console.log('MongoDB connected'))
  .catch(err => console.error('MongoDB connection error:', err));
```

Above: Server.js: This shows the connection that is established upon launch of the backend. The MongoDB database is hosted in the cloud via Atlas and the environment variable establishes connection.

Our project utilizes MongoDB Atlas to host the database in the cloud and pull the necessary information used throughout the app. Information is added manually via a "seed.js' file which directly interacts with the database for inserting into collections. Data is fetched from the database for all components and their relevant information, including images which are stored as binary buffers and displayed dynamically.



```
<h1>Drinks</h1>
<div class="menu">
    <div *ngFor="let item of drinkItems" class="card">
      <img [src]="item._id ? getImageUrl('drink', item._id) : ''" alt="{{item.name}}">
      <div class="details">
        <h2>{{item.name}}</h2>
        <p>{{item.description}}</p>
        <p class="price">{{item.price}}</p>
      </div>
```


Drinks

Iced Latte

Chilled espresso blended with ice and milk, topped with a splash of cream.

4.99

Mint Mojito

Refreshing classic mojito with a twist of mint and freshly squeezed lime juice.

7.99

Strawberry Smoothie

Made with real strawberries and blended with yogurt and honey for a smooth, refreshing drink.

5.99

Craft Beer

Locally brewed beer with a unique blend of hops and barley, offering a rich, full-bodied taste.

6.99

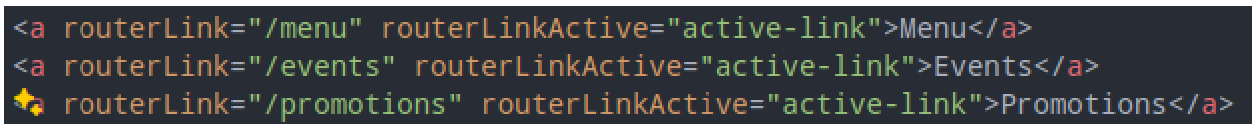
Above: HTML for the displaying of drinks and how it looks in-app. The service section will describe how the app actually handles the process of getting here, but this screenshot shows how the app is able to effectively display information from the database.

2.5 Routing

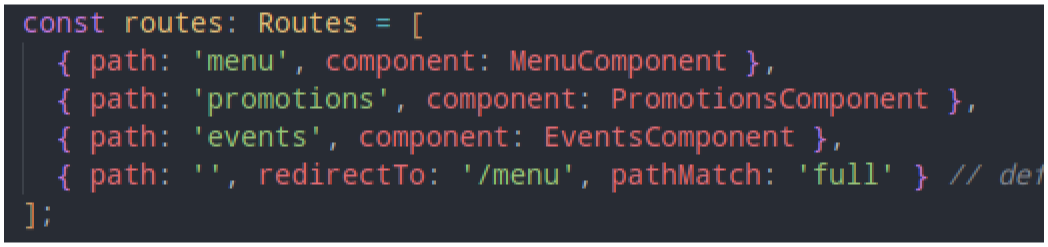
5. Routing.ts (Points 25) You need a routing file in your website that handles what component is to be displayed for each route. The routing information will be used with the header.component

Our Project:

The routing is simple in our application; as is required we include a header component that acts as the user's interface for navigation throughout the app. Users can navigate back and forth from the Menu, Promotions, and Events components freely by using the content displayed in header.component. The below images display both what this looks like in-app and the corresponding code snippets that make it possible.



```
<a routerLink="/menu" routerLinkActive="active-link">Menu</a>
<a routerLink="/events" routerLinkActive="active-link">Events</a>
  routerLink="/promotions" routerLinkActive="active-link">Promotions</a>
```



```
const routes: Routes = [
  { path: 'menu', component: MenuComponent },
  { path: 'promotions', component: PromotionsComponent },
  { path: 'events', component: EventsComponent },
  { path: '', redirectTo: '/menu', pathMatch: 'full' } // def
];
```

Above: (TOP) The HTML (header.component.html). The routerLink directive is used to define the target route for a link, while the 'routerLinkActive' directive is used to apply a class when the link is active. (Bottom) app-routing.module.ts: This code comes directly from the routing module, which actually interacts with the HTML to understand what is active and when and route appropriately via the header component.

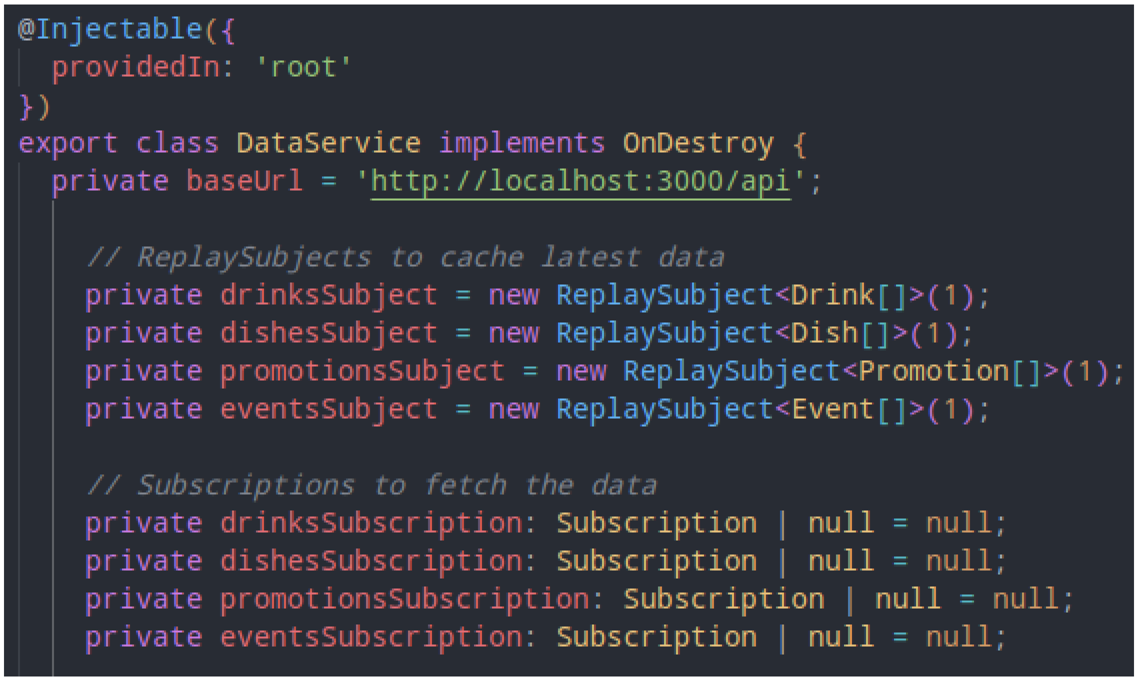
2.6 Data Service

6. Service.js (Points 25) The service file will contain the restful api and data transfer between components. You are allowed to use binding and input/output on less sensitive data but there does need to be a service.ts to handle frontend backend connections

Our Project:

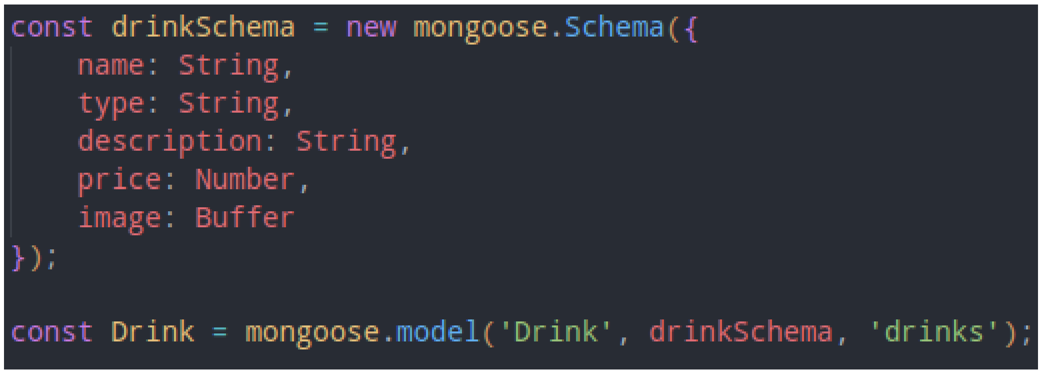
The file 'data.service.ts" handles everything involving data transfer, chiefly the connection that ensures that the frontend interacts with the backend which actually communicates with the database to fetch data.

The use of RESTful APIs facilitate data transfer between components. It uses HttpClient to fetch data for various models like Drink, Dish, Promotion, and Event, and leverages ReplaySubject to cache the latest data for each, ensuring efficient data binding. The service, marked with @Injectable for dependency injection, also provides public getters to access the cached data as observables, aligning with the requirement for managing frontend-backend connections. Additionally, the getImageUrl method allows retrieval of images for the specified types and items, and the ngOnDestroy method ensures proper subscription cleanup. The below image shows the initial setup of the service, where setup of Subscriptions can be seen which will be discussed later.

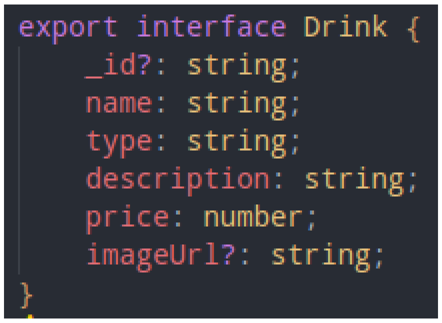


```
@Injectable({
  providedIn: 'root'
})
export class DataService implements OnDestroy {
  private baseUrl = 'http://localhost:3000/api';

    // ReplaySubjects to cache latest data
    private drinksSubject = new ReplaySubject<Drink[]>(1);
    private dishesSubject = new ReplaySubject<Dish[]>(1);
    private promotionsSubject = new ReplaySubject<Promotion[]>(1);
    private eventsSubject = new ReplaySubject<Event[]>(1);

    // Subscriptions to fetch the data
    private drinksSubscription: Subscription | null = null;
    private dishesSubscription: Subscription | null = null;
    private promotionsSubscription: Subscription | null = null;
    private eventsSubscription: Subscription | null = null;
```

Also of note is the use of models for the data that are similar but distinctly separate files used within Angular for the purposes of displaying information correctly, and in the backend for setting up the schema necessary for adding info to the database. Below the model for drinks can be seen. The first image is the backend schema, and the second is the angular model:



```
const drinkSchema = new mongoose.Schema({
    name: String,
    type: String,
    description: String,
    price: Number,
    image: Buffer
});

const Drink = mongoose.model('Drink', drinkSchema, 'drinks');
```



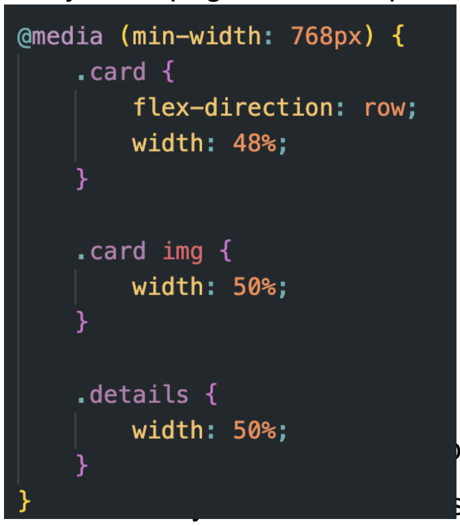
```
export interface Drink {
    _id?: string;
    name: string;
    type: string;
    description: string;
    price: number;
    imageUrl?: string;
}
```

2.7 CSS (Layout and Component Wise)

7. Component.css (Points 25) Your application must contain a proper layout using css padding and margins. The web application must be visually symmetric using css with the components.

Our Project:

Our project uses global CSS styles along with component specific styles to create a display that is appealing to the user and easy to navigate. We also used media queries to style our pages and components for mobile and other screen sizes.



```
@media (min-width: 768px) {
    .card {
        flex-direction: row;
        width: 48%;
    }

    .card img {
        width: 50%;
    }

    .details {
        width: 50%;
    }
}
```

when the screen width exceeds 768px, the cards in our e two to a row rather than stacked on top of each other smaller screens.

2.8 Subjects

8. Subjects(points 10) You will need to use subjects and subscriptions to handle the

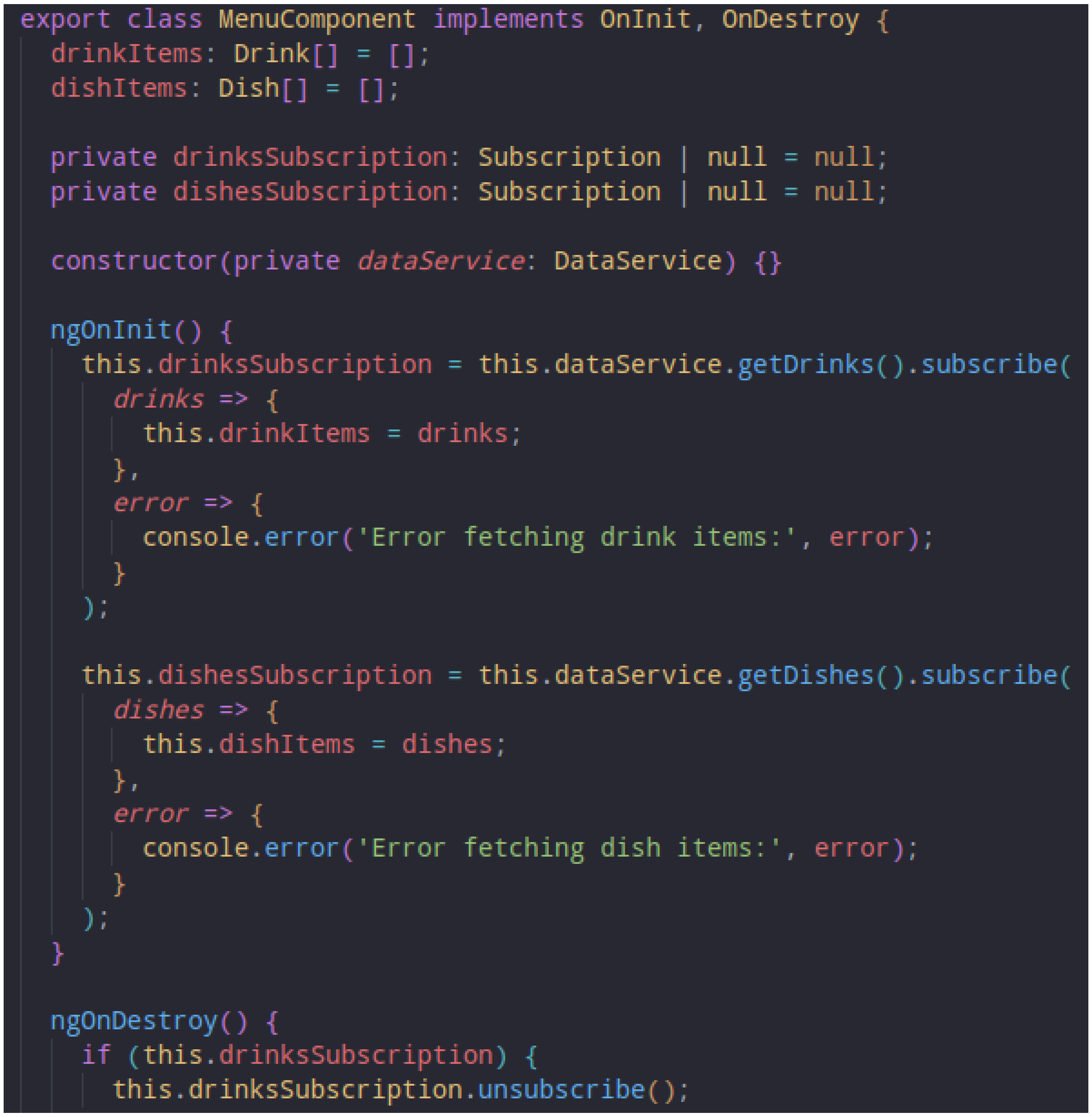
data that will be displayed from your database.

Our Project:

Section 2.6 shows a snippet which displays how Subscriptions are set up in the data service, but for additional clarity in this section, it's important to highlight how the data is displayed, showing specifically what goes on in the frontend in a given component utilizing subjects and subscriptions.

We'll use the menu component as an example, The `MenuComponent` in `menu.component.ts` effectively utilizes RxJS subjects and subscriptions to manage data displayed from the database, meeting the stated requirements. By subscribing to the observables provided by `DataService` for `Drink` and `Dish` items, it ensures the component stays updated with the latest data. The component initializes these subscriptions in `ngOnInit`, listening for updates from the backend, and appropriately unsubscribes in `ngOnDestroy` to prevent memory leaks.

The usage of `Subject`s in the service allows for efficient and reactive data handling, while the component's lifecycle methods ensure proper management of these subscriptions. Additionally, the `getImageUrl` method facilitates retrieving images for the items, further showcasing how subjects and subscriptions are effectively employed to display dynamic data from the database.



```
export class MenuComponent implements OnInit, OnDestroy {
  drinkItems: Drink[] = [];
  dishItems: Dish[] = [];

  private drinksSubscription: Subscription | null = null;
  private dishesSubscription: Subscription | null = null;

  constructor(private dataService: DataService) {}

  ngOnInit() {
    this.drinksSubscription = this.dataService.getDrinks().subscribe(
      drinks => {
        this.drinkItems = drinks;
      },
      error => {
        console.error('Error fetching drink items:', error);
      }
    );

    this.dishesSubscription = this.dataService.getDishes().subscribe(
      dishes => {
        this.dishItems = dishes;
      },
      error => {
        console.error('Error fetching dish items:', error);
      }
    );
  }

  ngOnDestroy() {
    if (this.drinksSubscription) {
      this.drinksSubscription.unsubscribe();
```