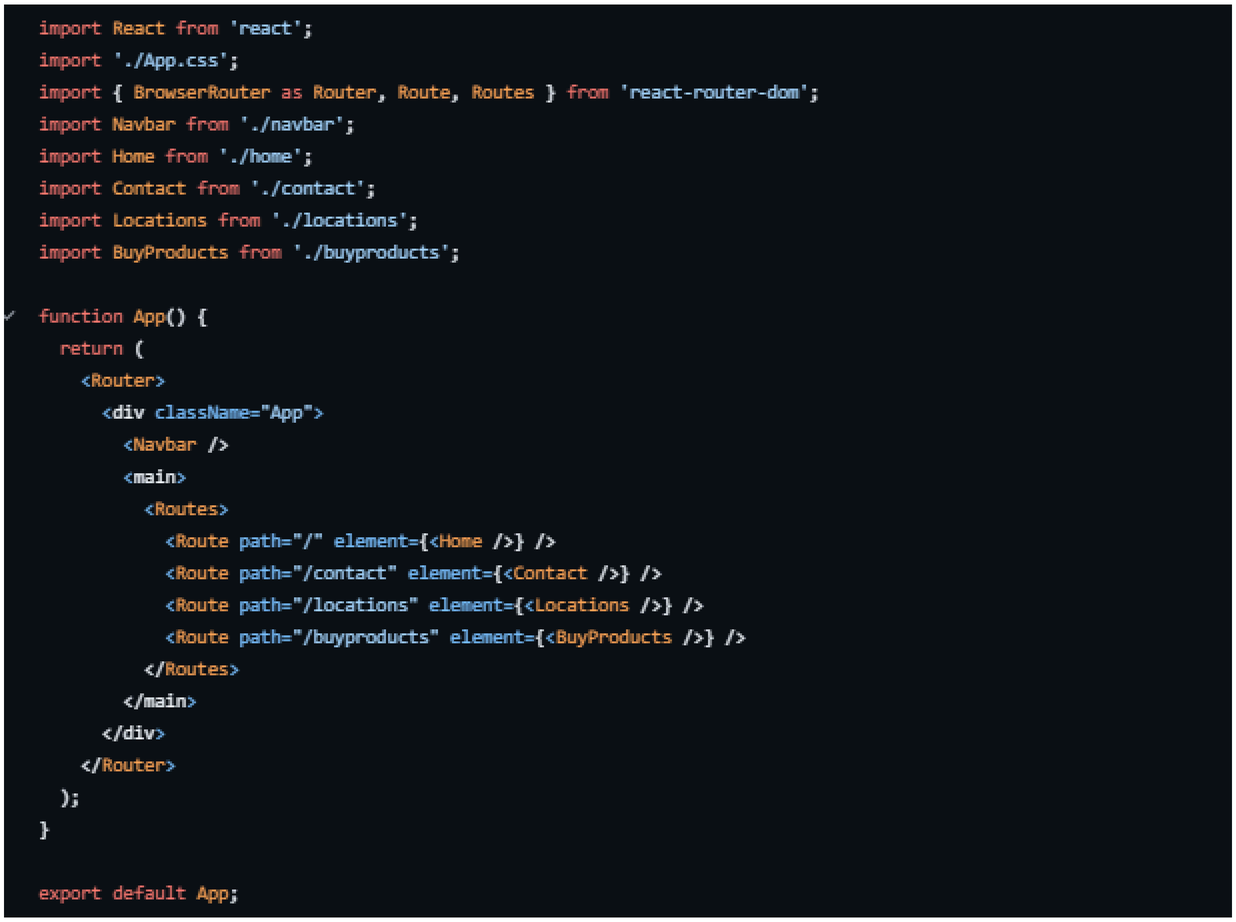
MUSIC FESTIVAL WEBSITE(355 Project 2):
Github link:
Group: Steven Campeche, Fahim Tanvir, Jamal Siddiqui

GRAY-PAPER

App.js:

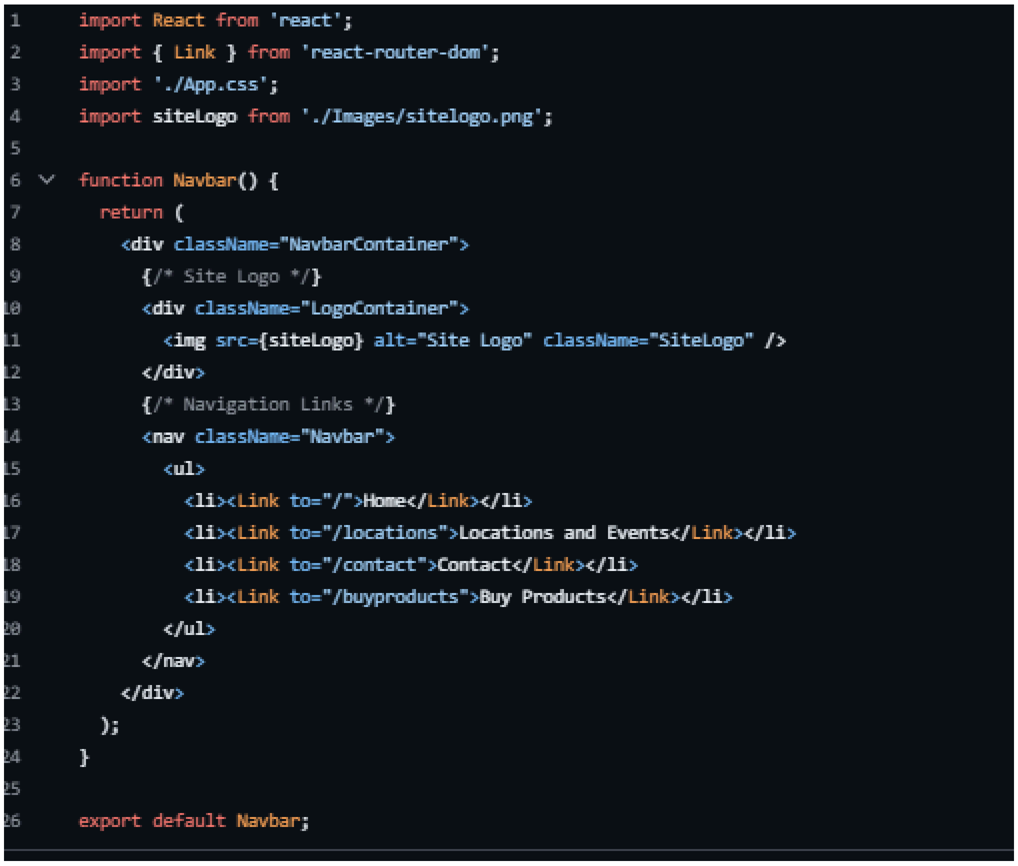
We took advantage of reacts reusability with code. Integrating the Navbar, and every page was relatively simple. Just import and declare it within the JavaScript.



```
import React from 'react';
import './App.css';
import { BrowserRouter as Router, Route, Routes } from 'react-router-dom';
import Navbar from './navbar';
import Home from './home';
import Contact from './contact';
import Locations from './locations';
import BuyProducts from './buyproducts';

function App() {
  return (
    <Router>
      <div className="App">
        <Navbar />
        <main>
          <Routes>
            <Route path="/" element={<Home />} />
            <Route path="/contact" element={<Contact />} />
            <Route path="/locations" element={<Locations />} />
            <Route path="/buyproducts" element={<BuyProducts />} />
          </Routes>
        </main>
      </div>
    </Router>
  );
}

export default App;
```

NavBar:

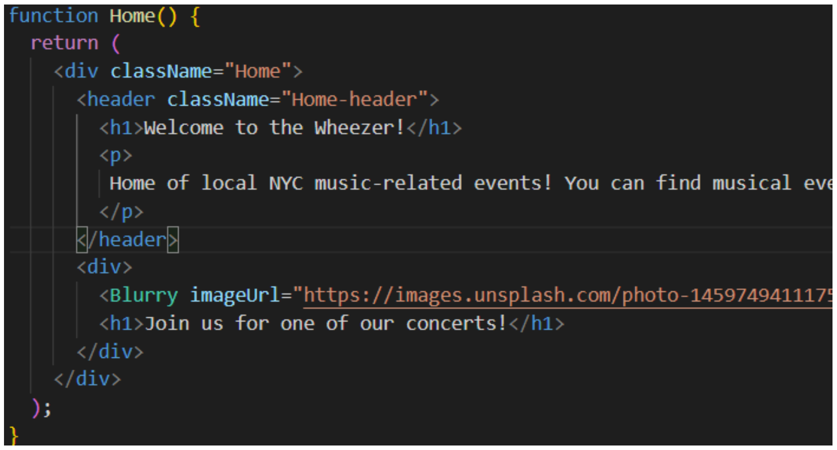
Just like App.js, we just imported the other pages linked to the navigation bar and put it as links.



```
import React from 'react';
import { Link } from 'react-router-dom';
import './App.css';
import siteLogo from './Images/sitelogo.png';

function Navbar() {
  return (
    <div className="NavbarContainer">
      {/* Site Logo */}
      <div className="LogoContainer">
        <img src={siteLogo} alt="Site Logo" className="SiteLogo" />
      </div>
      {/* Navigation Links */}
      <nav className="Navbar">
        <ul>
          <li><Link to="/">Home</Link></li>
          <li><Link to="/locations">Locations and Events</Link></li>
          <li><Link to="/contact">Contact</Link></li>
          <li><Link to="/buyproducts">Buy Products</Link></li>
        </ul>
      </nav>
    </div>
  );
}

export default Navbar;
```

Homejs:

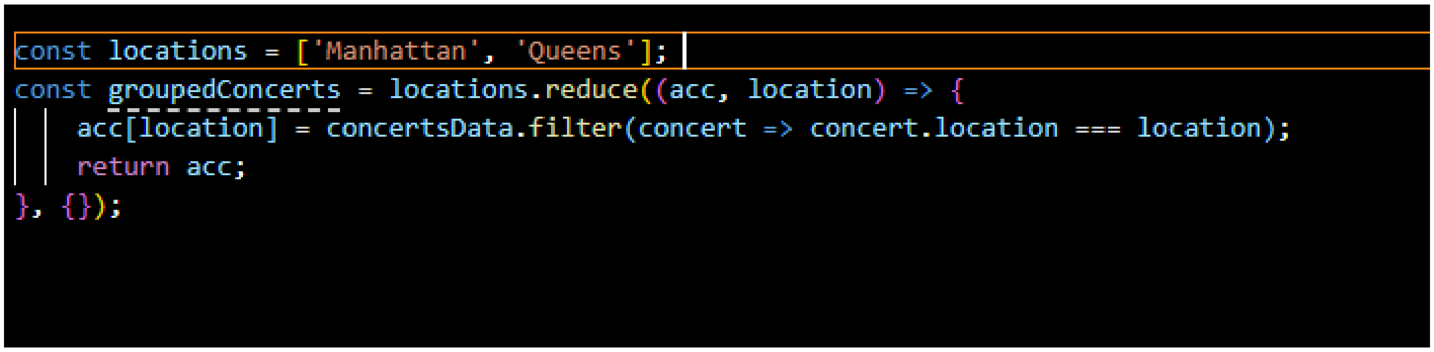
The homepage for our website, which does not contain much but uses the Blurry component that we developed based on the blurry loading image from the 5th project from the Udemy course.



```
function Home() {
  return (
    <div className="Home">
      <header className="Home-header">
        <h1>Welcome to the Wheezer!</h1>
        <p>
          Home of local NYC music-related events! You can find musical eve
        </p>
      </header>
      <div>
        <Blurry imageUrl="https://images.unsplash.com/photo-145974941117
        <h1>Join us for one of our concerts!</h1>
      </div>
    </div>
  );
}
```

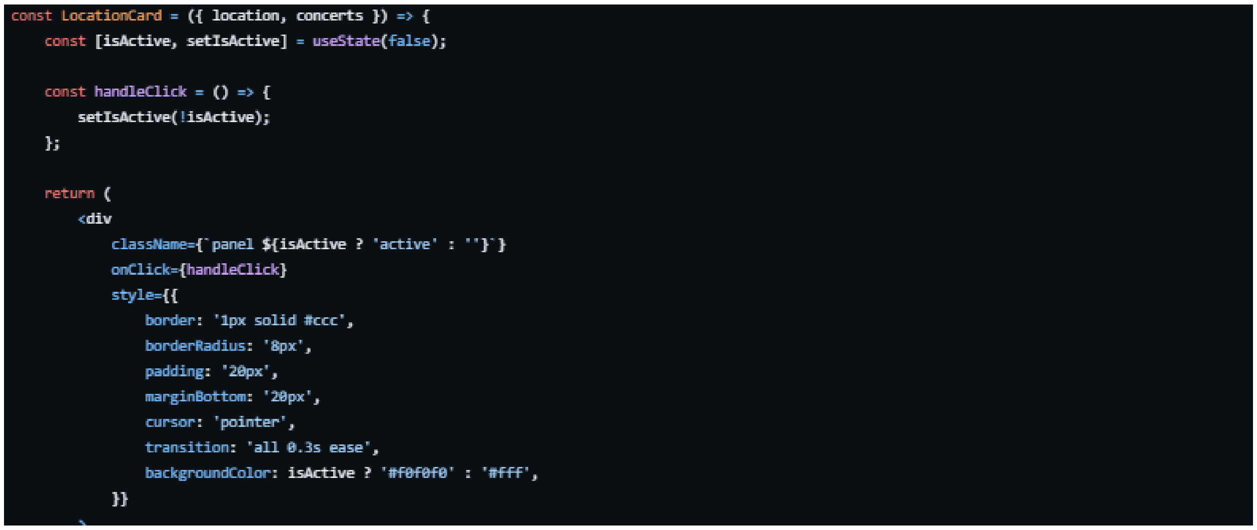
Locations.js

The Locations groups concert data by location. Each concert in the concertsData array is assigned a location (e.g., "Manhattan", "Queens"). This allows the component to filter and display only the concerts that belong to a specific location, creating an organized structure where users can explore events by borough or area.



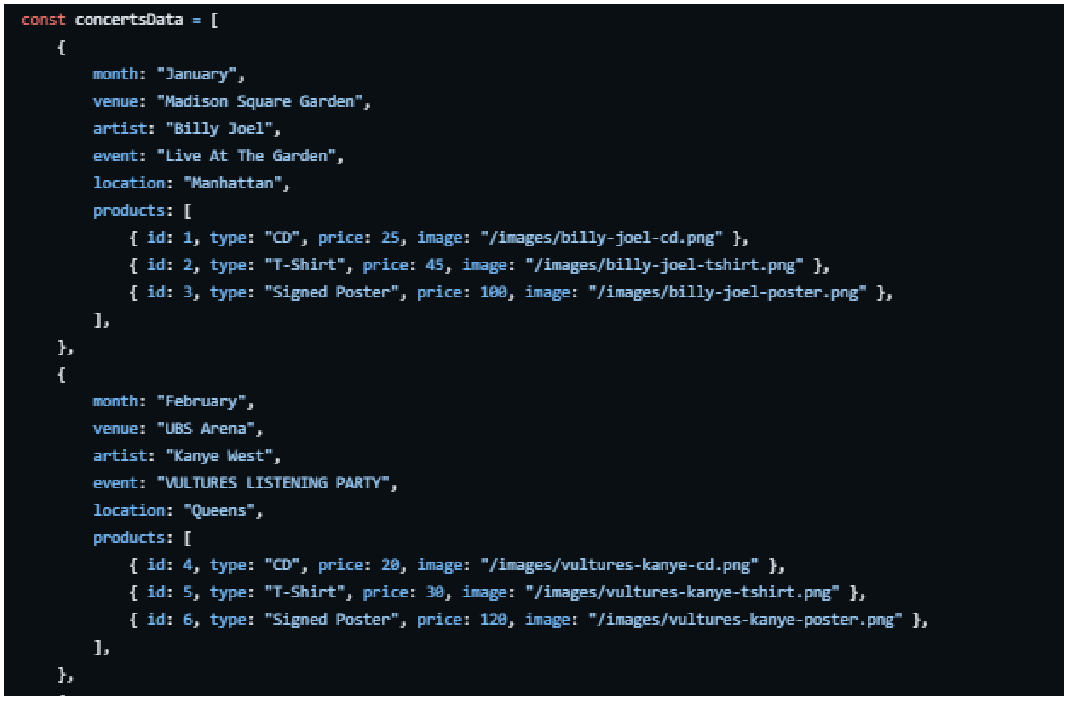
```
const locations = ['Manhattan', 'Queens'];
const groupedConcerts = locations.reduce((acc, location) => {
    acc[location] = concertsData.filter(concert => concert.location === location);
    return acc;
}, {});
```

The component uses interactive expanding cards, which are clickable. Each card represents a location and toggles between showing brief information or detailed event listings. When a user clicks a card, the LocationCard expands, revealing the concerts and associated products. The component uses React’s useState hook to manage the active state of each LocationCard. When a user clicks on a location, the state toggles, causing the card to expand and show the details for that particular location. This dynamic interaction makes the user interface more engaging and informative.



```
const LocationCard = ({ location, concerts }) => {
    const [isActive, setIsActive] = useState(false);

    const handleClick = () => {
        setIsActive(!isActive);
    };

    return (
        <div
            className={`panel ${isActive ? 'active' : ''}`}
            onClick={handleClick}
            style={{
                border: '1px solid #ccc',
                borderRadius: '8px',
                padding: '20px',
                marginBottom: '20px',
                cursor: 'pointer',
                transition: 'all 0.3s ease',
                backgroundColor: isActive ? '#f0f0f0' : '#fff',
            }}
```

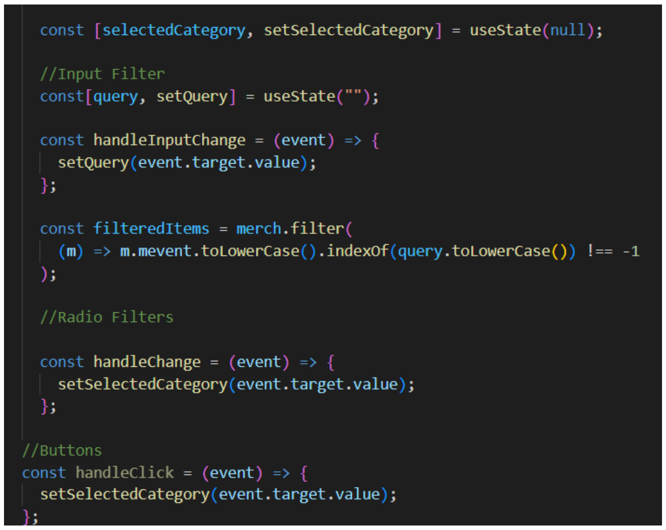
Each LocationCard displays detailed concert information when clicked. This includes the artist performing, the venue, the month of the concert, and associated products for sale (such as CDs, T-shirts, and signed posters). The products are displayed with images, prices, and types, allowing users to see the available merchandise related to each concert.



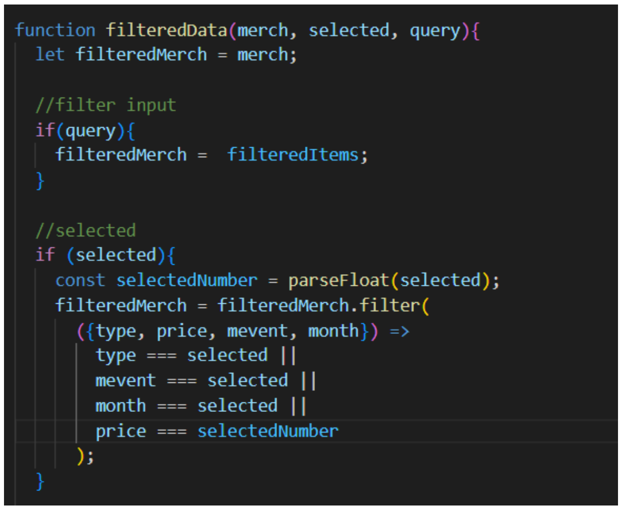
```
const concertsData = [
    {
        month: "January",
        venue: "Madison Square Garden",
        artist: "Billy Joel",
        event: "Live At The Garden",
        location: "Manhattan",
        products: [
            { id: 1, type: "CD", price: 25, image: "/images/billy-joel-cd.png" },
            { id: 2, type: "T-Shirt", price: 45, image: "/images/billy-joel-tshirt.png" },
            { id: 3, type: "Signed Poster", price: 100, image: "/images/billy-joel-poster.png" },
        ],
    },
    {
        month: "February",
        venue: "UBS Arena",
        artist: "Kanye West",
        event: "VULTURES LISTENING PARTY",
        location: "Queens",
        products: [
            { id: 4, type: "CD", price: 20, image: "/images/vultures-kanye-cd.png" },
            { id: 5, type: "T-Shirt", price: 30, image: "/images/vultures-kanye-tshirt.png" },
            { id: 6, type: "Signed Poster", price: 120, image: "/images/vultures-kanye-poster.png" },
        ],
    },
```

buyproducts.js

This page consists of multiple objects and methods in order to include a filter and search system into our product page. Each section is commented on for what they accomplish, and further down the actual filters that were originally intended for use are part of the function. The input is filtered using queries sent into the search bar's input in nav.js. For the radio buttons, they are filtered and specified in the function using selected, and using selectedNumber as a way to compare the price. We were unable to get this function to fully work properly, as instead of filtering in the range for the price, it only shows those items that are the exact amount of the upper limit.

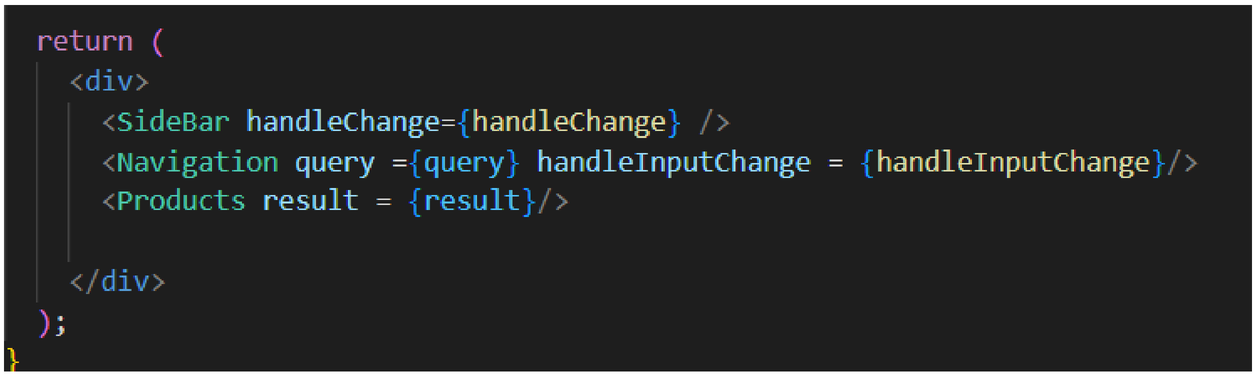


```
  const [selectedCategory, setSelectedCategory] = useState(null);

  //Input Filter
  const[query, setQuery] = useState("");

  const handleInputChange = (event) => {
    setQuery(event.target.value);
  };

  const filteredItems = merch.filter(
    (m) => m.mevent.toLowerCase().indexOf(query.toLowerCase()) !== -1
  );

  //Radio Filters

  const handleChange = (event) => {
    setSelectedCategory(event.target.value);
  };

//Buttons
const handleClick = (event) => {
  setSelectedCategory(event.target.value);
};
```



```
function filteredData(merch, selected, query){
  let filteredMerch = merch;

  //filter input
  if(query){
    filteredMerch =  filteredItems;
  }

  //selected
  if (selected){
    const selectedNumber = parseFloat(selected);
    filteredMerch = filteredMerch.filter(
      ({type, price, mevent, month}) =>
        type === selected ||
        mevent === selected ||
        month === selected ||
        price === selectedNumber
      );
  }
```

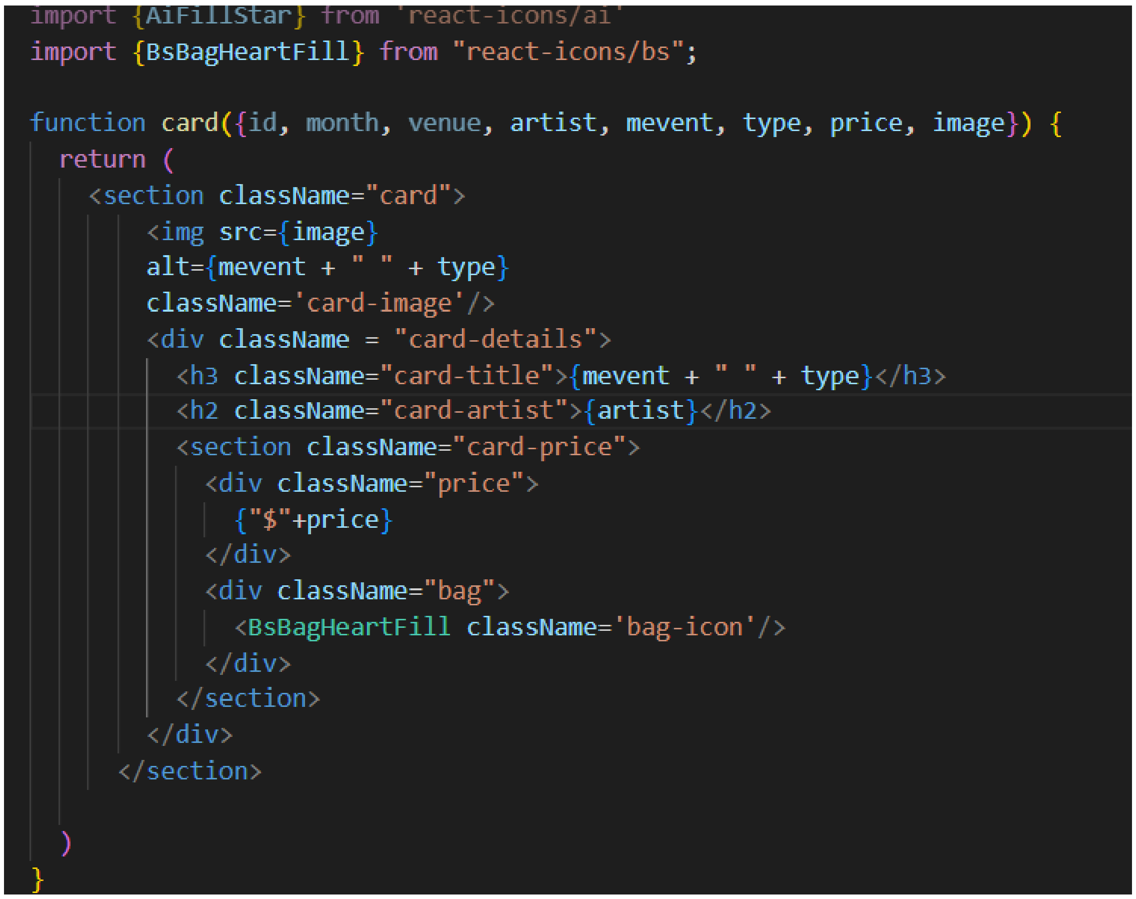
For the main components being loaded, they must be able to filter using the established items above, and the navigation specifically must be able to handle query changes as well instead of just passing specific values from out products.



```
  return (
    <div>
      <SideBar handleChange={handleChange} />
      <Navigation query ={query} handleInputChange = {handleInputChange}/>
      <Products result = {result}/>

    </div>
  );
}
```

card.js

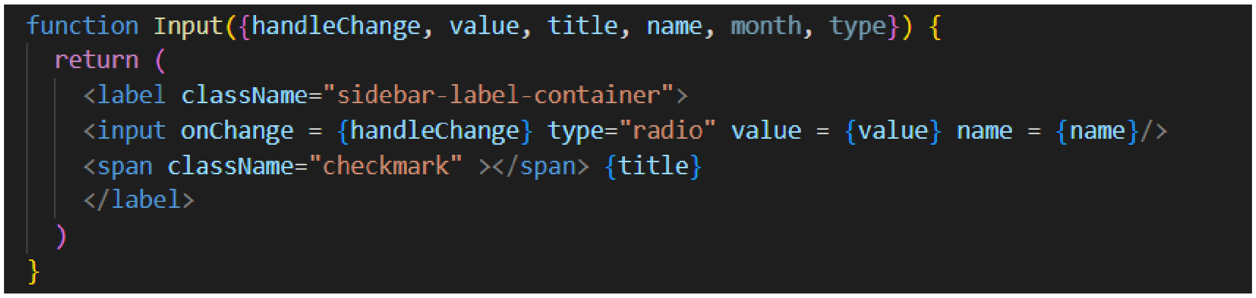
For this component, our products all use this template. Although all attributes are passed through, not all of them are used or displayed. Originally it was intended to be able to filter by each month, it proved impractical in terms of the amount of buttons that would be displayed and needed.



```
import {AiFillStar} from 'react-icons/ai'
import {BsBagHeartFill} from "react-icons/bs";

function card({id, month, venue, artist, mevent, type, price, image}) {
  return (
    <section className="card">
        <img src={image}
        alt={mevent + " " + type}
        className='card-image'/>
        <div className = "card-details">
          <h3 className="card-title">{mevent + " " + type}</h3>
          <h2 className="card-artist">{artist}</h2>
          <section className="card-price">
            <div className="price">
              {"$"+price}
            </div>
            <div className="bag">
              <BsBagHeartFill className='bag-icon'/>
            </div>
          </section>
        </div>
      </section>

  )
}
```

input.js

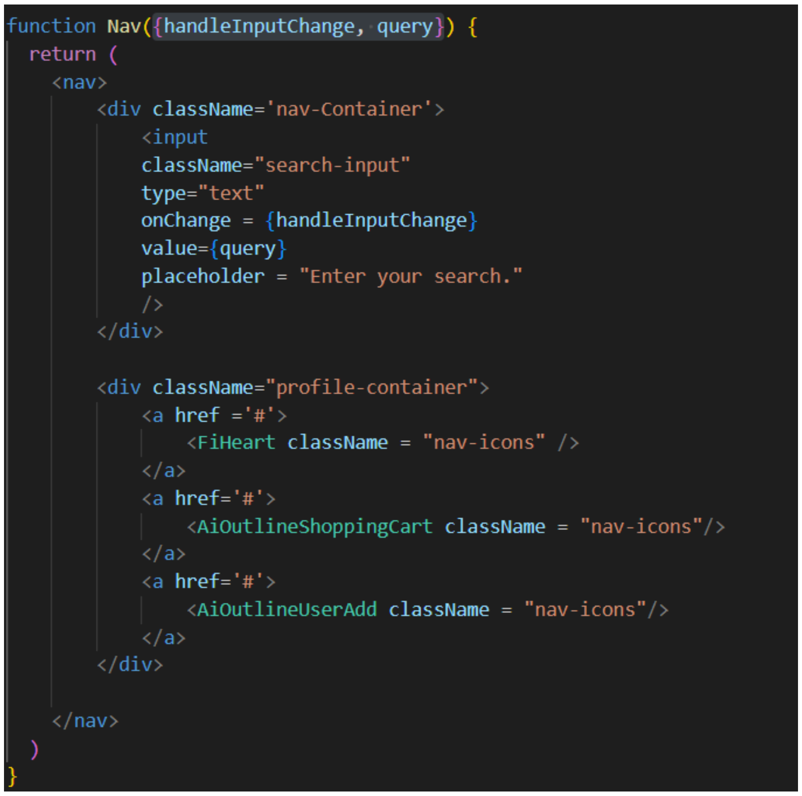
The input component was used for all of the radio buttons used for filtering. By doing this, based on passed in names and values, it was easy to specify which items were to be displayed or filtered out. This also utilizes a passed in handler for changes of selected filters, referring back to the object passed from the buyproducts.js page.



```
function Input({handleChange, value, title, name, month, type}) {
  return (
    <label className="sidebar-label-container">
    <input onChange = {handleChange} type="radio" value = {value} name = {name}/>
    <span className="checkmark" ></span> {title}
    </label>
  )
}
```

Nav.js

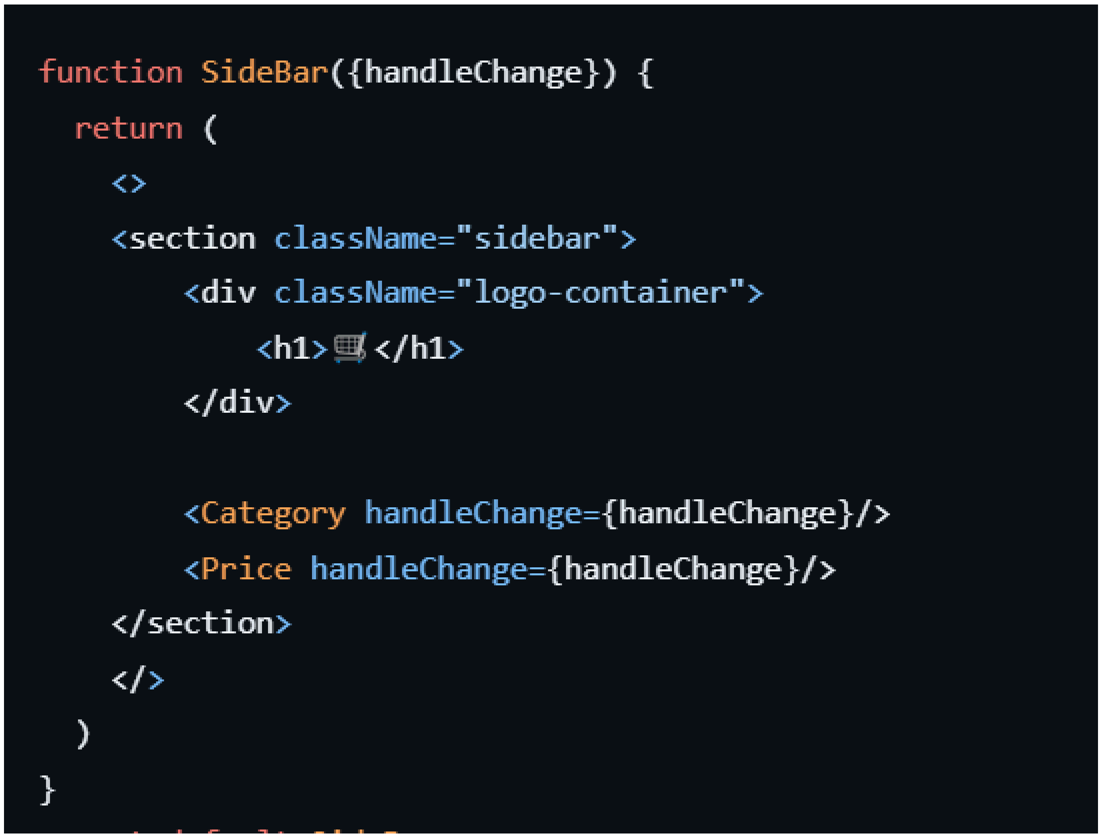
This nav is not the navbar, but refers to the navigation for the products page, which includes icons as well as the search bar for filtering product cards. It works by using the handle input change object, as well as setting value to the query being asked/input. By doing this, it can actively change the cards included within its filter as the user types into the search bar.



```
function Nav({handleInputChange, query}) {
  return (
    <nav>
        <div className='nav-Container'>
            <input
            className="search-input"
            type="text"
            onChange = {handleInputChange}
            value={query}
            placeholder = "Enter your search."
            />
        </div>

        <div className="profile-container">
            <a href ='#'>
                <FiHeart className = "nav-icons" />
            </a>
            <a href='#'>
                <AiOutlineShoppingCart className = "nav-icons"/>
            </a>
            <a href='#'>
                <AiOutlineUserAdd className = "nav-icons"/>
            </a>
        </div>

    </nav>
  )
}
```

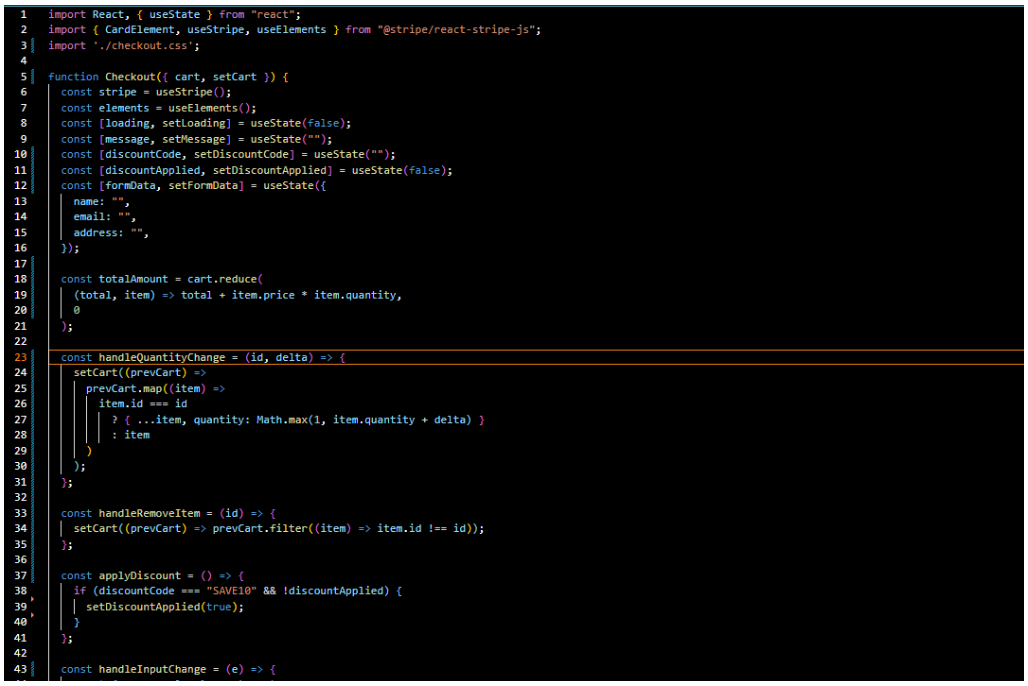
SideBar.js

The sidebar.js is another function used for the product page. It contains mostly the radio buttons and their containers, as well as manages them by passing the required handler for any changes to the buttons being selected.

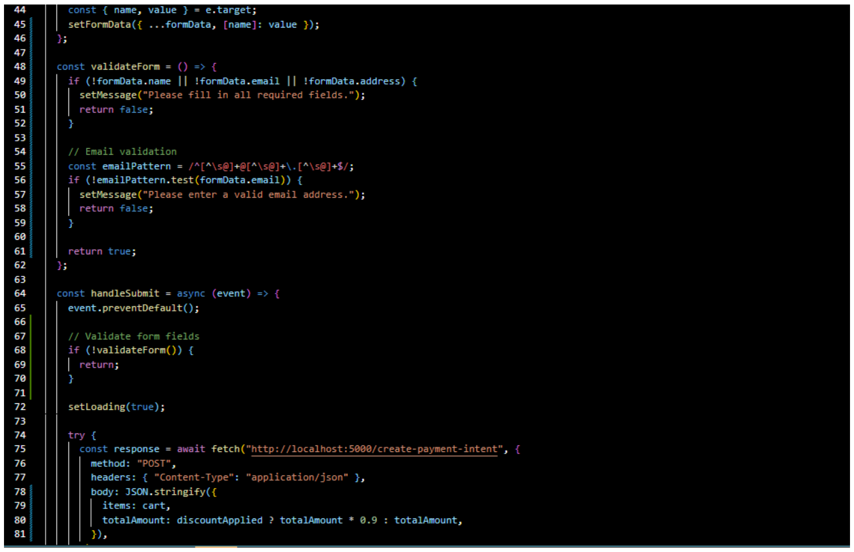


```
function SideBar({handleChange}) {
  return (
    <>
    <section className="sidebar">
        <div className="logo-container">
            <h1>🛒</h1>
        </div>

        <Category handleChange={handleChange}/>
        <Price handleChange={handleChange}/>
    </section>
    </>
  )
}
```

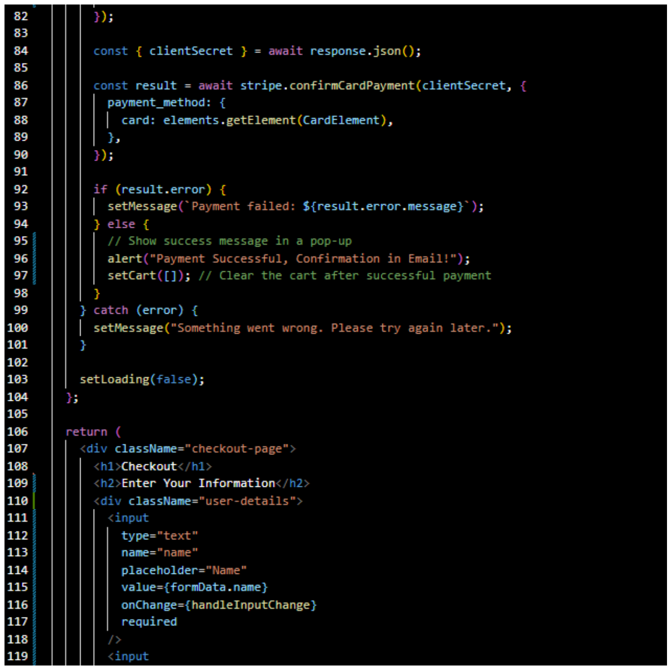
Checkout.js



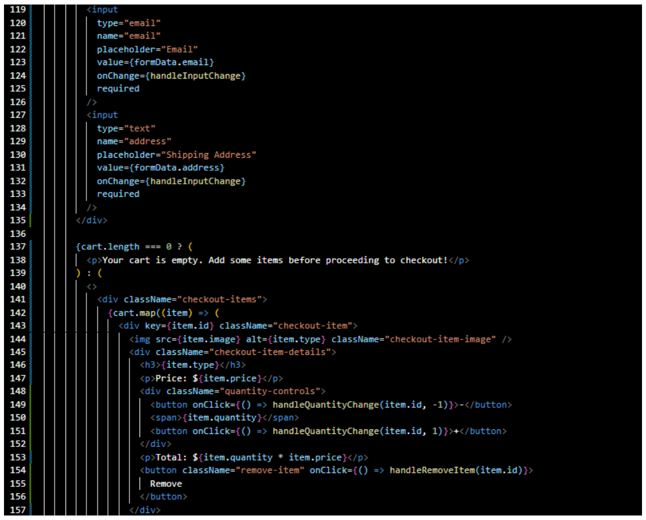
```
import React, { useState } from "react";
import { CardElement, useStripe, useElements } from "@stripe/react-stripe-js";
import './checkout.css';

function Checkout({ cart, setCart }) {
  const stripe = useStripe();
  const elements = useElements();
  const [loading, setLoading] = useState(false);
  const [message, setMessage] = useState("");
  const [discountCode, setDiscountCode] = useState("");
  const [discountApplied, setDiscountApplied] = useState(false);
  const [formData, setFormData] = useState({
    name: "",
    email: "",
    address: "",
  });

  const totalAmount = cart.reduce(
    (total, item) => total + item.price * item.quantity,
    0
  );

  const handleQuantityChange = (id, delta) => {
    setCart((prevCart) =>
      prevCart.map((item) =>
        item.id === id
          ? { ...item, quantity: Math.max(1, item.quantity + delta) }
          : item
      )
    );
  };

  const handleRemoveItem = (id) => {
    setCart((prevCart) => prevCart.filter((item) => item.id !== id));
  };

  const applyDiscount = () => {
    if (discountCode === "SAVE10" && !discountApplied) {
      setDiscountApplied(true);
    }
  };

  const handleInputChange = (e) => {
```



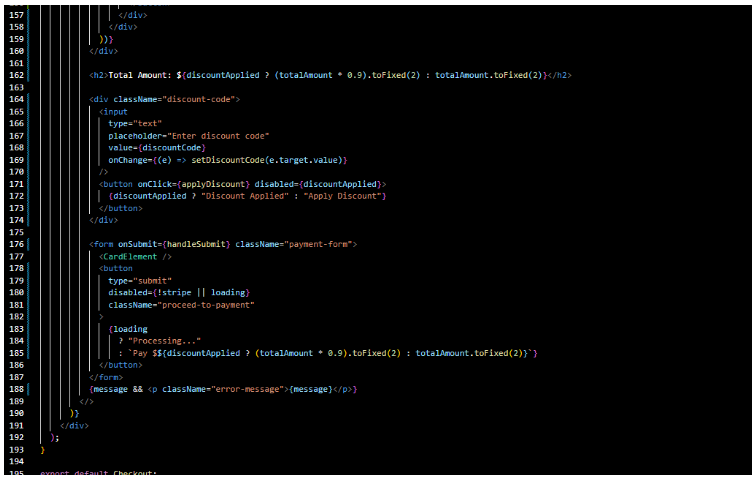
```javascript
    const { name, value } = e.target;
    setFormData({ ...formData, [name]: value });
  };

  const validateForm = () => {
    if (!formData.name || !formData.email || !formData.address) {
      setMessage("Please fill in all required fields.");
      return false;
    }

    // Email validation
    const emailPattern = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
    if (!emailPattern.test(formData.email)) {
      setMessage("Please enter a valid email address.");
      return false;
    }

    return true;
  };

  const handleSubmit = async (event) => {
    event.preventDefault();

    // Validate form fields
    if (!validateForm()) {
      return;
    }

    setLoading(true);

    try {
      const response = await fetch("http://localhost:5000/create-payment-intent", {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify({
          items: cart,
          totalAmount: discountApplied ? totalAmount * 0.9 : totalAmount,
        }),
```



```javascript
      });

      const { clientSecret } = await response.json();

      const result = await stripe.confirmCardPayment(clientSecret, {
        payment_method: {
          card: elements.getElement(CardElement),
        },
      });

      if (result.error) {
        setMessage(`Payment failed: ${result.error.message}`);
      } else {
        // Show success message in a pop-up
        alert("Payment Successful, Confirmation in Email!");
        setCart([]); // Clear the cart after successful payment
      }
    } catch (error) {
      setMessage("Something went wrong. Please try again later.");
    }

    setLoading(false);
  };

  return (
    <div className="checkout-page">
      <h1>Checkout</h1>
      <h2>Enter Your Information</h2>
      <div className="user-details">
        <input
          type="text"
          name="name"
          placeholder="Name"
          value={formData.name}
          onChange={handleInputChange}
          required
        />
        <input
```



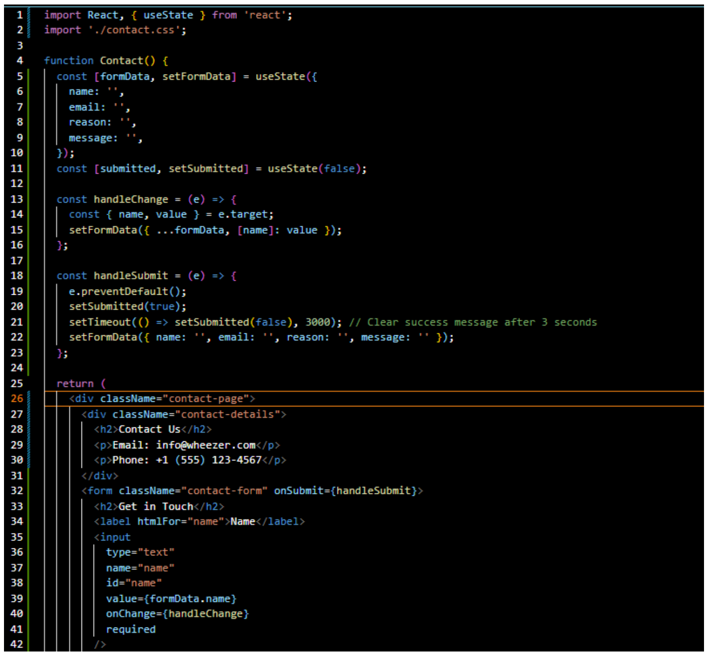
```javascript
        <input
          type="email"
          name="email"
          placeholder="Email"
          value={formData.email}
          onChange={handleInputChange}
          required
        />
        <input
          type="text"
          name="address"
          placeholder="Shipping Address"
          value={formData.address}
          onChange={handleInputChange}
          required
        />
      </div>

      {cart.length === 0 ? (
        <p>Your cart is empty. Add some items before proceeding to checkout!</p>
      ) : (
        <>
          <div className="checkout-items">
            {cart.map((item) => (
              <div key={item.id} className="checkout-item">
                <img src={item.image} alt={item.type} className="checkout-item-image" />
                <div className="checkout-item-details">
                  <h3>{item.type}</h3>
                  <p>Price: ${item.price}</p>
                  <div className="quantity-controls">
                    <button onClick={() => handleQuantityChange(item.id, -1)}>-</button>
                    <span>{item.quantity}</span>
                    <button onClick={() => handleQuantityChange(item.id, 1)}>+</button>
                  </div>
                  <p>Total: ${item.quantity * item.price}</p>
                  <button className="remove-item" onClick={() => handleRemoveItem(item.id)}>
                    Remove
                  </button>
                </div>
```



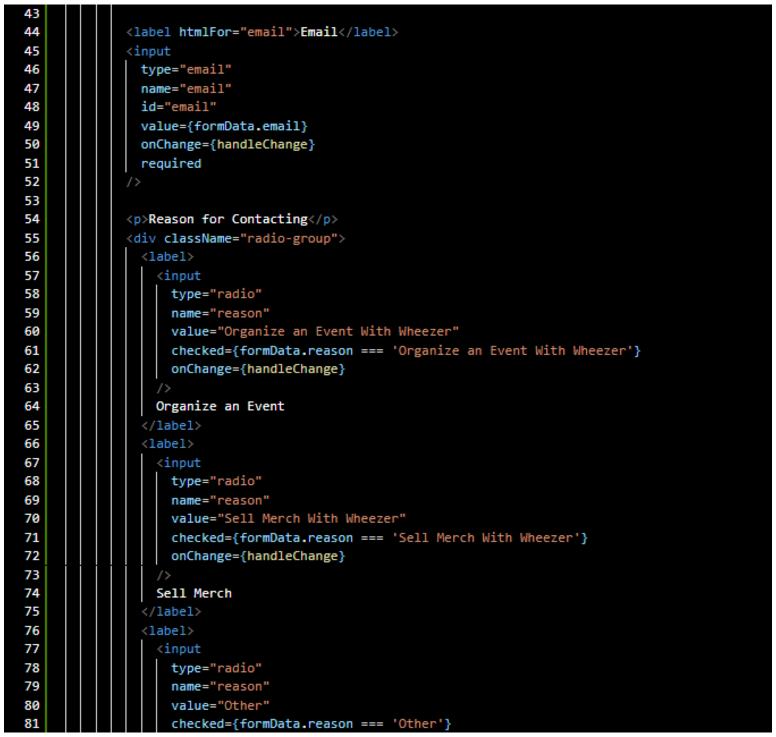
```javascript
                </div>
              </div>
            ))}
          </div>

          <h2>Total Amount: ${discountApplied ? (totalAmount * 0.9).toFixed(2) : totalAmount.toFixed(2)}</h2>

          <div className="discount-code">
            <input
              type="text"
              placeholder="Enter discount code"
              value={discountCode}
              onChange={(e) => setDiscountCode(e.target.value)}
            />
            <button onClick={applyDiscount} disabled={discountApplied}>
              {discountApplied ? "Discount Applied" : "Apply Discount"}
            </button>
          </div>

          <form onSubmit={handleSubmit} className="payment-form">
            <CardElement />
            <button
              type="submit"
              disabled={!stripe || loading}
              className="proceed-to-payment"
            >
              {loading
                ? "Processing..."
                : `Pay $${discountApplied ? (totalAmount * 0.9).toFixed(2) : totalAmount.toFixed(2)}`}
            </button>
          </form>
          {message && <p className="error-message">{message}</p>}
        </>
      )}
    </div>
  );
}

export default Checkout;
```

The Checkout.js component manages the checkout
process by collecting and validating user input (name, email, and address) and ensuring proper formatting (e.g., email must include @). It calculates the total cart amount, applies a discount if a valid code is entered, and processes payments securely using the Stripe API. Users can adjust item quantities, remove items, and proceed to payment only if all fields are correctly filled. Upon successful payment, the cart is cleared, and a success pop-up confirms the transaction with a message stating, "Payment Successful, Confirmation in Email!" Error handling and user feedback guide the process seamlessly.

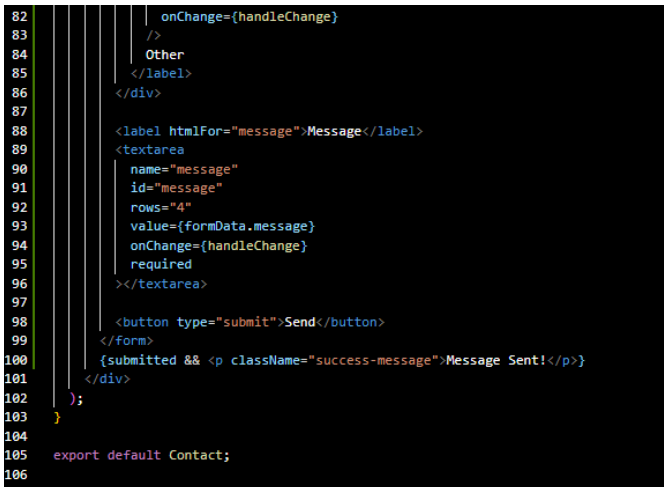
Contact.js



```
import React, { useState } from 'react';
import './contact.css';

function Contact() {
  const [formData, setFormData] = useState({
    name: '',
    email: '',
    reason: '',
    message: '',
  });
  const [submitted, setSubmitted] = useState(false);

  const handleChange = (e) => {
    const { name, value } = e.target;
    setFormData({ ...formData, [name]: value });
  };

  const handleSubmit = (e) => {
    e.preventDefault();
    setSubmitted(true);
    setTimeout(() => setSubmitted(false), 3000); // Clear success message after 3 seconds
    setFormData({ name: '', email: '', reason: '', message: '' });
  };

  return (
    <div className="contact-page">
      <div className="contact-details">
        <h2>Contact Us</h2>
        <p>Email: info@wheezer.com</p>
        <p>Phone: +1 (555) 123-4567</p>
      </div>
      <form className="contact-form" onSubmit={handleSubmit}>
        <h2>Get in Touch</h2>
        <label htmlFor="name">Name</label>
        <input
          type="text"
          name="name"
          id="name"
          value={formData.name}
          onChange={handleChange}
          required
        />
```



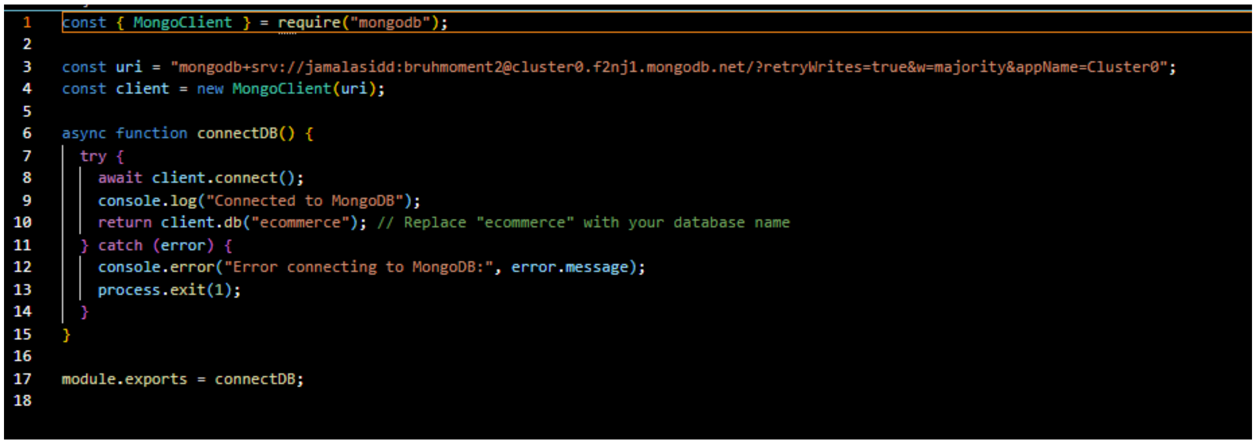
```

        <label htmlFor="email">Email</label>
        <input
          type="email"
          name="email"
          id="email"
          value={formData.email}
          onChange={handleChange}
          required
        />

        <p>Reason for Contacting</p>
        <div className="radio-group">
          <label>
            <input
              type="radio"
              name="reason"
              value="Organize an Event With Wheezer"
              checked={formData.reason === 'Organize an Event With Wheezer'}
              onChange={handleChange}
            />
            Organize an Event
          </label>
          <label>
            <input
              type="radio"
              name="reason"
              value="Sell Merch With Wheezer"
              checked={formData.reason === 'Sell Merch With Wheezer'}
              onChange={handleChange}
            />
            Sell Merch
          </label>
          <label>
            <input
              type="radio"
              name="reason"
              value="Other"
              checked={formData.reason === 'Other'}
```



```
              onChange={handleChange}
            />
            Other
          </label>
        </div>

        <label htmlFor="message">Message</label>
        <textarea
          name="message"
          id="message"
          rows="4"
          value={formData.message}
          onChange={handleChange}
          required
        ></textarea>

        <button type="submit">Send</button>
      </form>
      {submitted && <p className="success-message">Message Sent!</p>}
    </div>
  );
}

export default Contact;

```

The Contact.js component is a React functional component that implements a dynamic contact form. It uses useState to manage form data (name, email, reason, and message) and submission status. When a user types in the form, the handleChange function updates the corresponding state. Upon submission, the handleSubmit function prevents default browser behavior, displays a temporary success message ("Message Sent!"), and clears the form fields. Required fields ensure the user provides valid input before submission. The component also displays contact details (email and phone) and provides a concise, user-friendly interface for getting in touch.

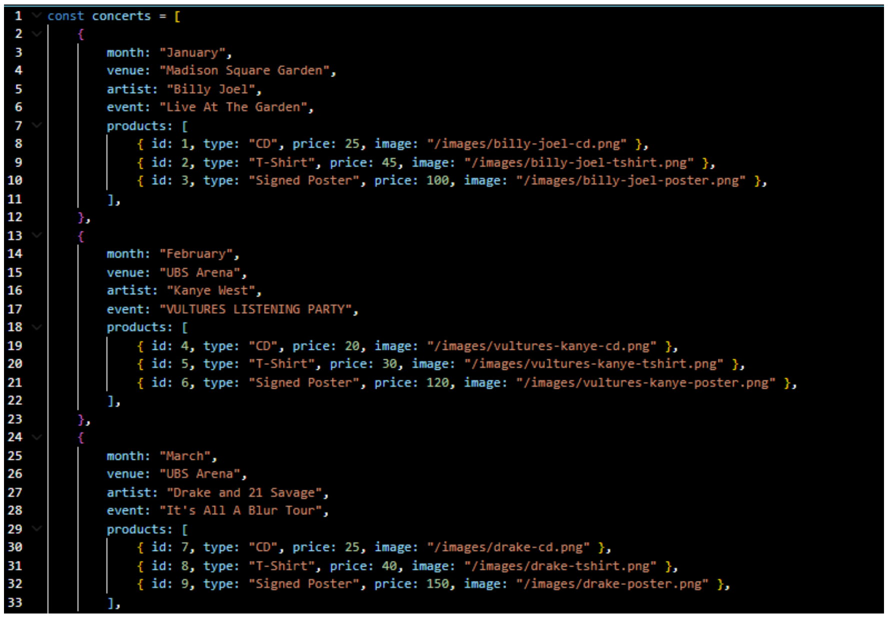
db.js



```javascript
const { MongoClient } = require("mongodb");

const uri = "mongodb+srv://jamalasidd:bruhmoment2@cluster0.f2nj1.mongodb.net/?retryWrites=true&w=majority&appName=Cluster0";
const client = new MongoClient(uri);

async function connectDB() {
  try {
    await client.connect();
    console.log("Connected to MongoDB");
    return client.db("ecommerce"); // Replace "ecommerce" with your database name
  } catch (error) {
    console.error("Error connecting to MongoDB:", error.message);
    process.exit(1);
  }
}

module.exports = connectDB;
```

This code connects a Node.js application to a MongoDB database using the mongodb library. It imports the MongoClient class from the library, which is used to interact with the database. The uri variable holds the connection string for the MongoDB instance, which includes the username, password, and cluster information. A new MongoClient instance is created using this URI.
The connectDB function is defined as an asynchronous function to establish a connection to the database. Within it, client.connect() is called to initiate the connection, and a success message is logged if the connection is successful. The function returns a reference to a specific database (ecommerce in this case). If the connection fails, the error is caught, and an error message is logged. The process is terminated with an exit code of 1 to indicate failure. Finally, the connectDB function is exported as a module so that it can be used elsewhere in the application to access the database.

concerts.js



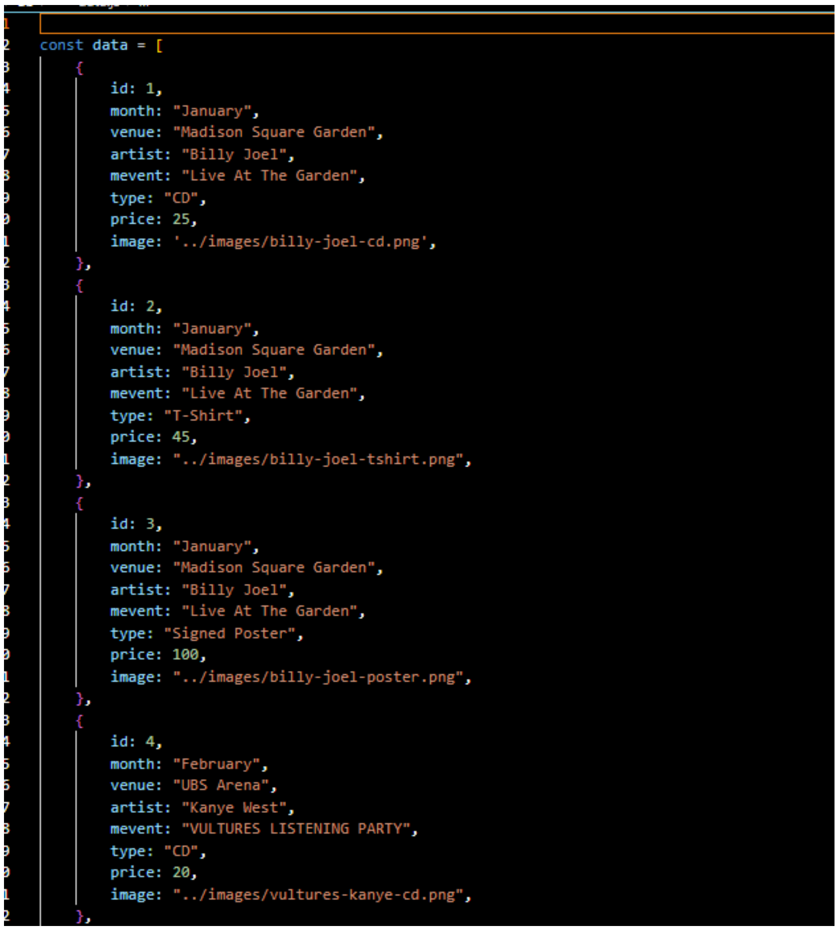
```javascript
const concerts = [
    {
        month: "January",
        venue: "Madison Square Garden",
        artist: "Billy Joel",
        event: "Live At The Garden",
        products: [
            { id: 1, type: "CD", price: 25, image: "/images/billy-joel-cd.png" },
            { id: 2, type: "T-Shirt", price: 45, image: "/images/billy-joel-tshirt.png" },
            { id: 3, type: "Signed Poster", price: 100, image: "/images/billy-joel-poster.png" },
        ],
    },
    {
        month: "February",
        venue: "UBS Arena",
        artist: "Kanye West",
        event: "VULTURES LISTENING PARTY",
        products: [
            { id: 4, type: "CD", price: 20, image: "/images/vultures-kanye-cd.png" },
            { id: 5, type: "T-Shirt", price: 30, image: "/images/vultures-kanye-tshirt.png" },
            { id: 6, type: "Signed Poster", price: 120, image: "/images/vultures-kanye-poster.png" },
        ],
    },
    {
        month: "March",
        venue: "UBS Arena",
        artist: "Drake and 21 Savage",
        event: "It's All A Blur Tour",
        products: [
            { id: 7, type: "CD", price: 25, image: "/images/drake-cd.png" },
            { id: 8, type: "T-Shirt", price: 40, image: "/images/drake-tshirt.png" },
            { id: 9, type: "Signed Poster", price: 150, image: "/images/drake-poster.png" },
        ],
```

This concerts array is a structured dataset that provides detailed information about various events, organized by month. Each object within the array represents a specific event and contains multiple attributes. The month attribute specifies the timing of the event (e.g., "January," "February"). The venue attribute highlights the location of the event, such as "Madison Square Garden" or "UBS Arena." The artist attribute indicates the performer or group headlining the event (e.g., "Billy Joel," "Kanye West"). The event attribute provides a descriptive title for the event (e.g., "Live At The Garden").
Each event also includes a products array, which details merchandise associated with the event. Each product in this array has:

- A unique ID to identify the product.
- A type that specifies the kind of product (e.g., "CD," "T-Shirt," "Signed Poster").
- A price that reflects the cost of the product.
- An image that provides the file path for the product's image to be displayed on the frontend.

This dataset can be used dynamically in an application to display event details, allow users to browse merchandise, or filter events and products by criteria such as month, artist, or type. The structure ensures scalability and flexibility, making it suitable for applications like event management systems, e-commerce platforms for event merchandise, or promotional websites.

data.js



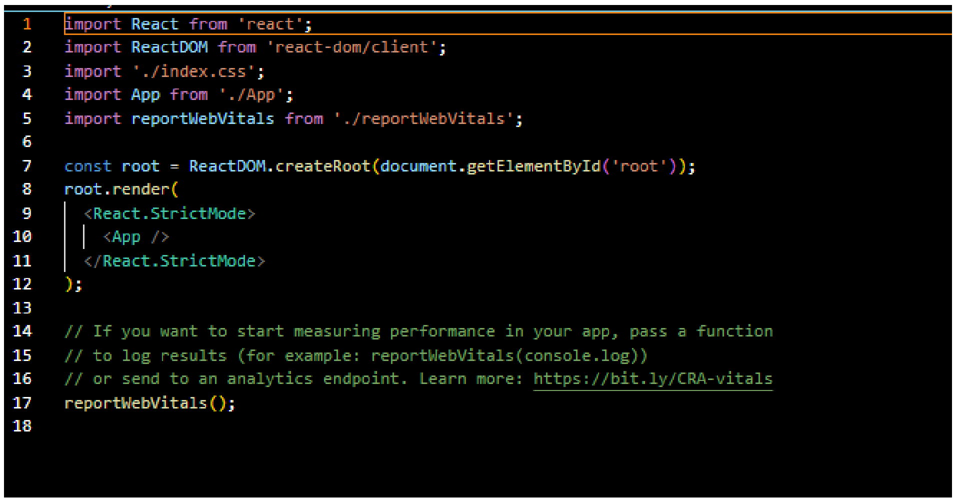
```javascript

const data = [
    {
        id: 1,
        month: "January",
        venue: "Madison Square Garden",
        artist: "Billy Joel",
        mevent: "Live At The Garden",
        type: "CD",
        price: 25,
        image: '../images/billy-joel-cd.png',
    },
    {
        id: 2,
        month: "January",
        venue: "Madison Square Garden",
        artist: "Billy Joel",
        mevent: "Live At The Garden",
        type: "T-Shirt",
        price: 45,
        image: "../images/billy-joel-tshirt.png",
    },
    {
        id: 3,
        month: "January",
        venue: "Madison Square Garden",
        artist: "Billy Joel",
        mevent: "Live At The Garden",
        type: "Signed Poster",
        price: 100,
        image: "../images/billy-joel-poster.png",
    },
    {
        id: 4,
        month: "February",
        venue: "UBS Arena",
        artist: "Kanye West",
        mevent: "VULTURES LISTENING PARTY",
        type: "CD",
        price: 20,
        image: "../images/vultures-kanye-cd.png",
    },
```

The data.js file is an array of objects, where each object represents a specific merchandise item associated with various concerts or events. Here's how the structure works:

- id: A unique identifier for each item, ensuring it can be referenced individually within the application.
- month: Indicates the month when the associated concert or event is happening, such as "January" or "February."
- venue: Specifies the location of the event, e.g., "Madison Square Garden" or "UBS Arena."
- artist: Names the performer or group for the event, like "Billy Joel" or "Kanye West."
- mevent: Describes the event associated with the item, such as "Live At The Garden" or "VULTURES LISTENING PARTY."
- type: Specifies the type of merchandise, such as "CD," "T-Shirt," or "Signed Poster."
- price: Indicates the cost of the merchandise in dollars.
- image: Provides the file path to the image of the merchandise, which can be displayed on a website or application.

This structure is highly useful for e-commerce or event management applications. It allows the app to dynamically display merchandise details, filter items by event, type, or artist, and manage interactions such as adding items to a cart or showing detailed information about the product. The detailed attributes make it easy to provide users with a rich and interactive experience when browsing concert-related merchandise.

index.js



```javascript
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);

// If you want to start measuring performance in your app, pass a function
// to log results (for example: reportWebVitals(console.log))
// or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals
reportWebVitals();
```

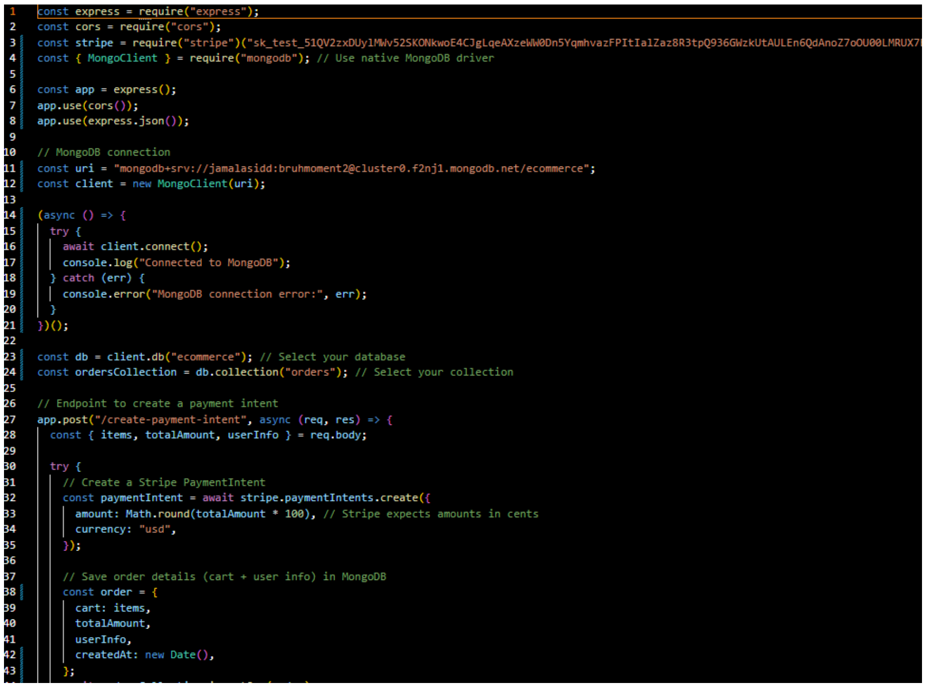
The provided code is the entry point for a React application, typically found in the index.js file. Here's how it functions:

1. Imports:
    - React and ReactDOM: These are core libraries for building and rendering React applications.
    - ./index.css: The main CSS file for global styles in the application.
    - App: The main component that serves as the root of the application's component tree.

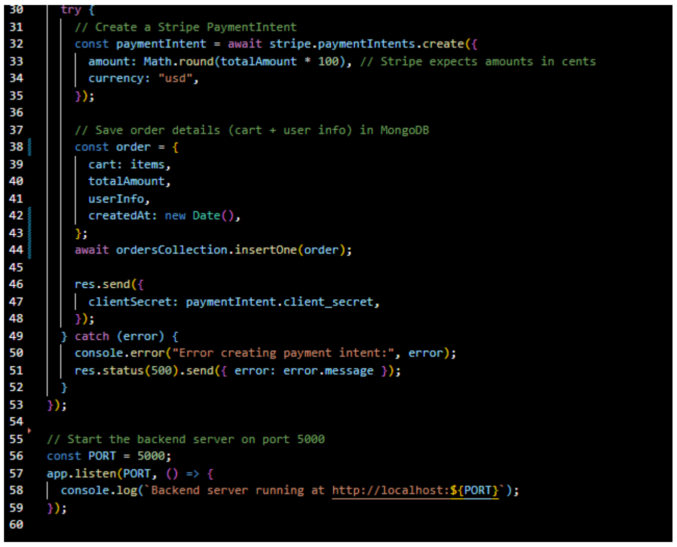
    - reportWebVitals: A utility for measuring and reporting app performance metrics.
2. Creating the Root:
    - ReactDOM.createRoot(document.getElementById('root')): This sets up the root element in the DOM where the React application will be rendered. The root ID corresponds to an element in the index.html file.
3. Rendering:
    - The root.render() method renders the React application into the DOM. Inside the render() method:
        - React.StrictMode wraps the App component. It is a development tool that helps identify potential problems in the React app by highlighting issues such as deprecated methods or side effects.
4. Performance Monitoring:
    - The reportWebVitals function can log app performance metrics to the console or send them to an analytics endpoint. This is optional and useful for optimizing the app.

In summary, this code initializes the React application, renders the App component into the DOM, and provides optional performance monitoring functionality. It acts as the entry point where the application lifecycle begins.

server.js



```javascript
const express = require("express");
const cors = require("cors");
const stripe = require("stripe")("sk_test_51QV2zxDUy1MWv52SKONkwoE4CJgLqeAXzeWW0Dn5YqmhvazFPItIalZaz8R3tpQ936GWzkUtAULEn6QdAnoZ7oOU00LMRUX7
const { MongoClient } = require("mongodb"); // Use native MongoDB driver

const app = express();
app.use(cors());
app.use(express.json());

// MongoDB connection
const uri = "mongodb+srv://jamalasidd:bruhmoment2@cluster0.f2nj1.mongodb.net/ecommerce";
const client = new MongoClient(uri);

(async () => {
  try {
    await client.connect();
    console.log("Connected to MongoDB");
  } catch (err) {
    console.error("MongoDB connection error:", err);
  }
})();

const db = client.db("ecommerce"); // Select your database
const ordersCollection = db.collection("orders"); // Select your collection

// Endpoint to create a payment intent
app.post("/create-payment-intent", async (req, res) => {
  const { items, totalAmount, userInfo } = req.body;

  try {
    // Create a Stripe PaymentIntent
    const paymentIntent = await stripe.paymentIntents.create({
      amount: Math.round(totalAmount * 100), // Stripe expects amounts in cents
      currency: "usd",
    });

    // Save order details (cart + user info) in MongoDB
    const order = {
      cart: items,
      totalAmount,
      userInfo,
      createdAt: new Date(),
    };
```



```javascript
  try {
    // Create a Stripe PaymentIntent
    const paymentIntent = await stripe.paymentIntents.create({
      amount: Math.round(totalAmount * 100), // Stripe expects amounts in cents
      currency: "usd",
    });

    // Save order details (cart + user info) in MongoDB
    const order = {
      cart: items,
      totalAmount,
      userInfo,
      createdAt: new Date(),
    };
    await ordersCollection.insertOne(order);

    res.send({
      clientSecret: paymentIntent.client_secret,
    });
  } catch (error) {
    console.error("Error creating payment intent:", error);
    res.status(500).send({ error: error.message });
  }
});

// Start the backend server on port 5000
const PORT = 5000;
app.listen(PORT, () => {
  console.log(`Backend server running at http://localhost:${PORT}`);
});
```

1. Imports and Setup:
    - The file imports necessary modules such as express for creating a server, cors for handling cross-origin requests, stripe for integrating Stripe payment services, and mongodb for connecting to a MongoDB database.
    - A new MongoClient is initialized to connect to the MongoDB instance with the given URI.
2. Middleware:
    - The express.json() middleware is used to parse incoming JSON requests.
    - cors middleware allows the server to handle requests from different origins, enabling frontend-backend communication.
3. MongoDB Connection:
    - The server connects to a MongoDB database using the provided connection string.
    - Upon a successful connection, it logs a confirmation message. If the connection fails, it logs an error and exits the process.
4. Database and Collection:
    - The code initializes access to the ecommerce database and the orders collection to store order information.
5. Payment Intent Endpoint:
    - The /create-payment-intent endpoint listens for POST requests.
    - It extracts items, the total amount, and user information from the request body.
    - A Stripe PaymentIntent is created to manage the payment process, with the total amount converted to cents (as Stripe expects).
    - The cart details, total amount, user information, and a timestamp are saved in the MongoDB database as an order document.
6. Error Handling:
    - If the payment intent creation or database insertion fails, the server responds with a 500 status and logs the error.
7. Server Initialization:
    - The server starts on port 5000 and logs a confirmation message with the server's URL (http://localhost:5000).

This server.js file acts as a bridge between the frontend and the backend. It manages payment transactions through Stripe and ensures that order data is securely saved in MongoDB. It is essential for processing payments, validating user inputs, and maintaining a record of purchases.