TEXTUAL DESCRIPTION FOR E-COMMERCE WEB APPLICATION

PREPARED BY

Michelle Nguyen

05/02/2024

TABLE OF CONTENT

1. Overview

1.1 PROJECT OBJECTIVE

The objective of this project is to develop an ecommerce web application using MERN stack (MongoDB, Express.js, React.js, Node.js).

The application will consist of two different interfaces for different user roles: the client, who can browse and purchase items, and the store owner, who can manage the store seamlessly without any prior coding knowledge. With this project, we aim to provide a reliable and intuitive platform for hassle-free transactions for both clients and store owners.

1.2 PROJECT OVERVIEW

The clients can access a user-friendly and visually-appealing website application to browse for college-level textbooks. The clients can filter products by category, brand, or price and view detailed listings before adding items to their cart. At the checkout, they can enter shipping and payment details, receiving confirmation via email upon successful purchase.

The store owner can manage the business via an admin dashboard, allowing them to update inventory and monitor orders. They can easily modify product listings, track inventory, and generate sales reports.

1.3 FUNCTIONAL EXPECTATION

A. Client Side:

- **Client Dashboard**: Client can access the client dashboard to manage inventory, orders, and customer data.
- **User Authentication**: Clients can register and sign in to track the status of their order.
- **Browsing Inventory**: Clients can search, filter, and browse through items with detailed information such as brand, ID, price, etc.
- **Shopping Cart**: Clients can add items to their cart, view cart, and proceed to checkout.
- **Secure Payment**: Clients can secure transactions with payment gateway that upholds functionality and security standards.

B. Store Owner Side:

- **Admin Dashboard**: Store owners have access to a centralized dashboard to manage inventory, orders, and customer data.
- **Inventory Management**: Store owners can add new items, update existing ones, and remove items from the inventory.
- **Order Management**: Store owners can view and manage incoming orders, mark orders as fulfilled.
- **Sales Report**: Store owners can log weekly sales data and generate reports for sales data, including trends, totals, and summaries.

2. System Structure

2.1 TECHNOLOGY STACK:

Frontend	React.js for building responsive user interfaces.
Backend	Node.js and Express.js for creating a server-side application.
Database	MongoDB's Data Service for efficient data storage and retrieval with NoSQL data.
User Authentication	MongoDB's App Service for authentication and authorization through custom JWT.
Payment Gateway Integration	Stripe API to create mock transactions using a variety of payment methods.

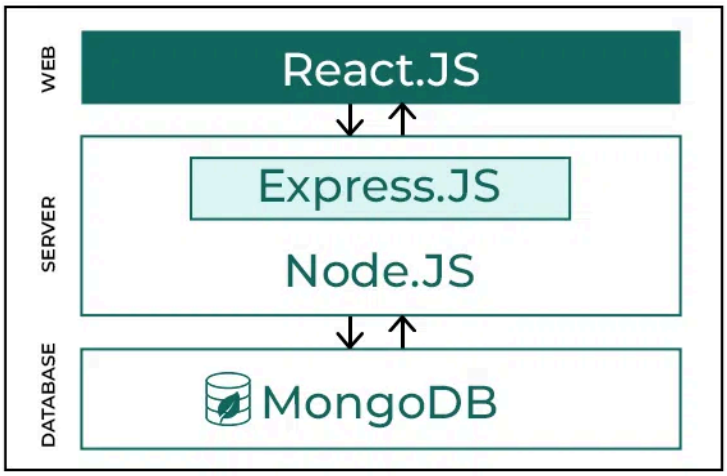
2.2 TECHNOLOGY DETAILS:

- **React** - version 18.2.0
- **Node** - version 20.7.0
- **Express** - version 4.18.2
- **MongoDB** - version 7.0
- **Deployment** - Vercel
- **User Authentication -** Realm Java SDK
- **Language -** JavaScript, Typescript, HTML, CSS, Python
- **Version Control** - Bitbucket, Trello

2.3 SYSTEM STRUCTURE:

The MERN stack consists of four main components:

1. **MongoDB**: A NoSQL database used to store product information, user data, orders, and other relevant data.
2. **Express.js**: A backend framework for Node.js that handles HTTP requests, routing, middleware, and interacts with the MongoDB database.
3. **React**: A frontend library for building user interfaces that handles the presentation layer of the ecommerce website.
4. **Node.js**: A JavaScript runtime environment that executes server-side code, handles requests from the frontend, and interacts with the MongoDB database through Express.js.



3. IMPLEMENTED ALGORITHMS

3.1 USER AUTHENTICATION ALGORITHM:

The UserContext is a React context that provides a set of functions for user authentication using MongoDB Realm's authentication system. It includes functions for logging in, signing up, resetting passwords, fetching user data, and logging out.

By wrapping the application's root component with the UserProvider component that utilizes this context, these authentication functions become available to all components within the app. Components can use the useUser hook provided by the UserContext to access the current user object and these authentication functions, enabling seamless integration of user authentication across the web app.

```
import { createContext, useState } from "react";
import { App, Credentials } from "realm-web";
import { APP_ID } from "./constants";

const app = new App(APP_ID);

export const UserContext = createContext();

export const UserProvider = ({ children }) => {
  const [user, setUser] = useState(null);

  const emailPasswordLogin = async (email, password) => {
    const credentials = Credentials.emailPassword(email, password);
    const authenticatedUser = await app.logIn(credentials);
    setUser(authenticatedUser);
    return authenticatedUser;
  };

  const emailPasswordSignup = async (email, password) => {
    try {
      await app.emailPasswordAuth.registerUser(email, password);
      // return emailPasswordLogin(email, password);
      const authenticatedUser = await emailPasswordLogin(email, password);
      return authenticatedUser;
    } catch (error) {
      throw error;
    }
  };
```

```
  const passwordReset = async (token, tokenId, password) => {
    try {
      await app.emailPasswordAuth.resetPassword(token, tokenId, password);
    } catch (error) {
      throw error;
    }
  };

  const fetchUser = async () => {
    if (!app.currentUser) return false;
    try {
      await app.currentUser.refreshCustomData();
      setUser(app.currentUser);
      return app.currentUser;
    } catch (error) {
      throw error;
    }
  }

  const logOutUser = async () => {
    if (!app.currentUser) return false;
    try {
      await app.currentUser.logOut();
      setUser(null);
      return true;
    } catch (error) {
      throw error
    }
  }

  return <UserContext.Provider
        value={{ user, setUser, fetchUser, emailPasswordLogin,
                 emailPasswordSignup, logOutUser,
                 emailPasswordReset, passwordReset }}>
    {children}
  </UserContext.Provider>;
}
```

The UserProvider is wrapping several components, providing authentication-related functions and user state to all components rendered within.

```
function App() {
  return (
    <UserProvider>
      <ShoppingCartProvider>
        <Routes/>
      </ShoppingCartProvider>
    </UserProvider>
  );
}
```

3.2 SEARCHING ALGORITHM:

The searching algorithm is implemented using Fuse.js, a JavaScript library that performs simple yet efficient fuzzy search on arrays of strings or objects.

When the searchQuery changes, a new Fuse instance is created with the inventory array as the dataset. The keys option specifies the object properties to search for matches. By setting includeScore to true, a relevance score is included for each search result, and threshold determines the required match closeness.

The search method is then used to find matching results, which are mapped to extract the original items. These results update the searchResults state, enabling real-time product searches based on title, ISBN, author, or subject.

```
useEffect(() => {
  if (searchQuery.trim() === '') {
    setSearchResults([]);
    return;
  }

  const fuse = new Fuse(inventory, {
    keys: ['title', 'isbn', 'author', 'subject'],
    includeScore: true,
    threshold: 0.3,
  });
  const searchResult = fuse.search(searchQuery);
  setSearchResults(searchResult.map(result => result.item));
}, [inventory, searchQuery]);
```

3.3 FILTERING & SORTING ALGORITHM:

The filtering and sorting algorithm filters an inventory based on selected class subject (selectedTags) and price ranges (selectedPriceRange), ensuring each item includes all selected tags in its subject property and falls within the selected price ranges.

It then sorts the filtered inventory based on a selected sortBy parameter, such as lowPrice, highPrice, or alphabetic order by title, assigning the sorted inventory to the sortedInventory variable. If no specific sorting criteria is selected, the inventory remains unsorted.

```
const filteredInventory = inventory.filter((inventory) =>
  selectedTags.every((tag) => inventory.subject.includes(tag)) &&
  (selectedPriceRange.length === 0 ||
    selectedPriceRange.some((range) => {
      if (range === "$200 and above") {
        return inventory.price >= 200;
      } else {
        const [min, max] = range.split(" - ").map((str) => parseInt(str.replace(/\$/g, ""
        return inventory.price >= min && inventory.price <= max;
      }
    }))
);

let sortedInventory;
if (sortBy === "lowPrice") {
  sortedInventory = filteredInventory.sort((a, b) => a.price - b.price);
} else if (sortBy === "highPrice") {
  sortedInventory = filteredInventory.sort((a, b) => b.price - a.price);
} else if (sortBy === "alphabetic") {
  sortedInventory = filteredInventory.sort((a, b) => a.title.localeCompare(b.title));
} else {
  sortedInventory = filteredInventory;
}
```

3.4 PAYMENT PROCESSING ALGORITHM:

The payment processing algorithm operates as follows:

On the frontend, when a user initiates a payment by submitting their card information, the handlePayment function is triggered. This function uses the Stripe API to create a payment method and sends the payment details to the backend server using an HTTP POST request.

```
const handlePayment = async (e) => {
  e.preventDefault();
  const { error, paymentMethod } = await stripe.createPaymentMethod({
    type: 'card',
    card: elements.getElement(CardElement),
    billing_details: {      },
  });

  if (!error) {
    try {
      const { id } = paymentMethod;
      const response = await axios.post('http://localhost:5050/payment', {
        amount: Math.round(total * 100),
        id,
      });

      if (response.data.success) {
        console.log('Successful payment');
        setSuccess(true);
        localStorage.removeItem('shopping-cart');

        const paymentId = id;
        console.log('Payment ID:', paymentId);

      }
    } catch (error) { console.log('Error', error); }
  } else {
    console.log(error.message);
  }
};
```

The backend server receives the payment details, including the amount and payment method ID, and uses the Stripe API to create a payment intent. If the payment is successful, the server responds with a success message, and the frontend sets a success state, removes the shopping cart items from local storage, and logs the payment ID.

```
const stripeConnection = process.env.STRIPE_SECRET_TEST || "";
const stripe = new stripePackage(stripeConnection);

router.post("/", async (req, res) => {
    let { amount, id } = req.body;
    try {
        const payment = await stripe.paymentIntents.create({
            amount,
            currency: "USD",
            description: "MERNTextBook",
            payment_method: id,
            confirm: true,
            return_url: 'http://localhost:3000/shop/checkout'
        });

        res.json({
            message: "Payment successful",
            success: true
        });
    } catch (error) {
        console.log("Error", error);
        res.json({
            message: "Payment failed",
            success: false
        });
    }
});

export default router;
```

If an error occurs during the payment process, the server responds with an error message, which is logged on the frontend. This payment processing algorithm ensures secure and seamless payment transactions for users.

3.4 UPDATE INVENTORY/ORDER ALGORITHM:

The update inventory algorithm works as follows:

On the frontend, the Edit component fetches the current inventory item with the specified id from the backend using an HTTP GET request. If the item is found, its details are set in the component's local state. The user can then update the item's details in the form, triggering the updateForm function to update the local state. When the user submits the form, an HTTP PATCH request is sent to the backend with the updated item details.

On the backend, the router receives the PATCH request and updates the corresponding item in the database using the MongoDB updateOne method. If the update is successful, the router responds with a status of 200 and the updated item.

```
router.patch("/:id", async (req, res) => {
  const query = { _id: new ObjectId(req.params.id) };
  const updates =  {
    $set: {
      isbn: req.body.isbn,
      title: req.body.title,
      description: req.body.description,
      edition: req.body.edition,
      author: req.body.author,
      publisher: req.body.publisher,
      publicationYear: req.body.publicationYear,
      quantity: req.body.quantity,
      price: req.body.price,
      subject: req.body.subject,
      rating: req.body.rating,
      ratingTotal: req.body.ratingTotal,
      imageurl: req.body.imageurl,
      showListing: req.body.showListing,
    }
  };

  let collection = await db.collection("inventory");
  let result = await collection.updateOne(query, updates);
  res.send(result).status(200);
});
```

```
const Edit = ({ id }) => {
  const [form, setForm] = useState({  });
  const navigate = useNavigate();

  useEffect(() => {
    async function fetchData() {
      const response = await fetch(`http://localhost:5050/inventory/${id}`);

      if (!response.ok) {
        window.alert(`An error has occurred: ${response.statusText}`);
        return;
      }

      const inventory = await response.json();
      if (!inventory) {
        window.alert(`Inventory item with id ${id} not found`);
        return;
      }

      setForm(inventory);
    }

    fetchData();

    return;
  }, [id, navigate, handleSwitchOff]);

  function updateForm(value) {
    return setForm((prev) => {
      return { ...prev, ...value };
    });
  }
  async function onSubmit(e) {
    e.preventDefault();
    const editedItem = { };

    await fetch(`http://localhost:5050/inventory/${id}`, {
      method: "PATCH",
      body: JSON.stringify(editedItem),
      headers: {
        'Content-Type': 'application/json'
      },
    });

    handleSwitchOff();
  }
```