GROCERY WEB APP USING MERN STACK

S.NO	NAME	ROLE
1	PRAVEEN I	BACKEND
2	DHANUSH R	FRONTEND
3	STEPHY A	FRONTEND
4	RAGHINI R	DATABASE

TABLE OF CONTENTS:

ABSTRACT:

SmartGrocery is a web-based application designed to facilitate online grocery shopping, leveraging the MERN (MongoDB, Express.js, React.js, and Node.js) stack. This platform aims to provide a seamless and efficient shopping experience for users, while streamlining inventory management and order fulfillment for grocery stores. The grocery shopping experience has undergone significant transformations with the advent of e-commerce and digital technologies. Online grocery shopping has become increasingly popular, offering convenience, flexibility, and time-saving benefits to consumers. However, existing solutions often face challenges related to inventory management, order fulfillment, and user experience. To address these issues, this project proposes the development of SmartGrocery, a web-based application leveraging the MERN (MongoDB, Express.js, React.js, and Node.js) stack. SmartGrocery aims to provide an intuitive, efficient, and scalable platform for online grocery shopping, streamlining the shopping experience for users and inventory management for store owners.

PROJECT OVERVIEW:

SmartGrocery is a web-based application designed to facilitate online grocery shopping. The platform allows users to browse and purchase groceries from local stores, while providing store owners with an efficient inventory management system.

PURPOSE:

Dvelop a user-friendly web application for online grocery shopping.Implement efficient inventory management and order fulfillment.Integrate secure payment gateways. Provide real-time

order tracking and status updates. Secure payment processing.Real-time order tracking and status updates.

FEATURES:

Customer-Facing Features:

1. User Registration and Login
2. Product Search and Filtering
3. Product Details and Reviews
4. Shopping Cart and Checkout
5. Secure Payment Processing
6. Order Tracking and Status Updates
7. Personalized Recommendations
8. Shopping Lists and Favorites
9. Ratings and Reviews
10. Customer Support

Admin Features:

1. User Management
2. Store Management
3. Order Management
4. Product Management
5. Reporting and Analytics
6. System Configuration
7. Security and Access Control
8. Integration with third-party services
9. Content Management

10. Customer Support

Security Features:

1. Two-Factor Authentication
2. Access Control
3. Regular Security Audits

ARCHITECTURE:

SETUP INSTRUCTION:

Hardware Requirements:

1. Server: Dedicated or cloud-based (e.g., AWS, Google Cloud)
2. Database Server: MySQL, MongoDB, or other NoSQL databases
3. Load Balancer: For distributing traffic across multiple servers
4. Storage: Sufficient storage for product images, videos, and documents
5. Network: Reliable internet connectivity

Software Requirements:

1. Operating System: Linux, Windows, or macOS
2. Web Server: Apache, Nginx, or IIS
3. Programming Languages:

- Frontend: JavaScript (React, Angular, Vue), HTML, CSS
- Backend: Node.js, Python, Ruby, PHP

4. Frameworks:
   - Frontend: React, Angular, Vue
   - Backend: Express.js, Django, Ruby on Rails
5. Database Management System: MySQL, MongoDB, PostgreSQL
6. Payment Gateway Integration: Stripe, PayPal, (link unavailable)
7. Security Software: SSL certificates, firewall, antivirus

Technical Skills:

1. Frontend development: HTML, CSS, JavaScript, React, Angular, Vue
2. Backend development: Node.js, Python, Ruby, PHP, Express.js, Django
3. Database administration: MySQL, MongoDB, PostgreSQL
4. API integration: RESTful APIs, GraphQL
5. Security: SSL, firewall, antivirus
6. Testing: Unit testing, integration testing, UI testing
7. Deployment: Cloud deployment (AWS, Google Cloud), containerization (Docker)

INSTALLATION:

Step 1: Install Node.js and npm

1. Download and install Node.js from the official website: (link unavailable)
2. Verify Node.js installation: node -v
3. Verify npm installation: npm -v

Step 2: Install MongoDB

1. Download and install MongoDB Community Server from the official website: (link unavailable)
2. Follow the installation instructions for your operating system.
3. Verify MongoDB installation: mongod --version

Step 3: Install Express.js

1. Create a new project directory: mkdir grocery-web-app
2. Navigate to the project directory: cd grocery-web-app
3. Initialize a new npm project: npm init
4. Install Express.js: npm install express

Step 4: Install React.js

1. Create a new React app: npx create-react-app client
2. Navigate to the client directory: cd client
3. Install required dependencies: npm install

Step 5: Install Required Dependencies

Backend (Node.js/Express.js)

1. Install mongoose (MongoDB ORM): npm install mongoose
2. Install bcrypt (password hashing): npm install bcrypt
3. Install jsonwebtoken (JWT authentication): npm install jsonwebtoken
4. Install cors (CORS middleware): npm install cors

Frontend (React.js)

1. Install axios (HTTP client): npm install axios
2. Install react-router-dom (client-side routing): npm install react-router-dom

Step 6: Set up MongoDB Database

1. Create a new MongoDB database: use grocery-db
2. Create collections for users, products, orders, etc.

Step 7: Configure Backend (Node.js/Express.js)

1. Create a new file server.js

2. Import required dependencies: express, mongoose, bcrypt, jsonwebtoken, cors

3. Set up Express.js app: const app = express()

4. Configure MongoDB connection: mongoose.connect('mongodb://localhost/grocery-db')

5. Define API routes: /api/users, /api/products, /api/orders

Step 8: Configure Frontend (React.js)

1. Create a new file index.js

2. Import required dependencies: react, react-dom, axios, react-router-dom

3. Set up React app: ReactDOM.render(<App />, document.getElementById('root'))

4. Define components: App.js, ProductList.js, OrderForm.js

Step 9: Implement Authentication and Authorization

Backend (Node.js/Express.js)

1. Implement user registration: /api/users/register

2. Implement user login: /api/users/login

3. Implement JWT authentication: jsonwebtoken

Frontend (React.js)

1. Implement login and registration forms
2. Implement JWT token storage: localStorage

Step 10: Implement Payment Gateway

1. Choose a payment gateway (e.g., Stripe, PayPal)
2. Implement payment gateway integration

Step 11: Test and Deploy

1. Test application functionality
2. Deploy application to production environment (e.g., Heroku, AWS)

FOLDER STTUCTURE :

The root directory, grocery-web-app, contains two main folders: client and server. The client folder holds the frontend code, with subfolders including public for static assets, src for source code, components for reusable UI components, containers for container components, actions and reducers for Redux management, store for Redux store configuration, utils for utility functions, images for image assets, and styles for CSS styles. The src folder also contains the index.js entry point.The server folder holds the backend code, with subfolders including config for configuration files, controllers for API controllers, models for database models, routes for API routes, services for business logic services, and utils for utility functions. The server folder also contains the app.js entry point.The root directory also contains a package.json file for project dependencies and scripts

Client (Frontend) Folder Structure:

client/

public/

index.html

favicon.ico

src/

components/

App.js

Header.js

Footer.js

ProductList.js

OrderForm.js

...

containers/

ProductDetails.js

OrderSummary.js

...

actions/

productActions.js

orderActions.js

...

reducers/

productReducers.js

orderReducers.js

...

store/

store.js

...

utils/

api.js

constants.js

...

images/

product-images/

...

styles/

global.css

...

index.js

package.json

(link unavailable)

Server (Backend) Folder Structure:

server/

config/

database.js

auth.js

...
controllers/
productController.js
orderController.js
...
models/
productModel.js
orderModel.js
...
routes/
productRoutes.js
orderRoutes.js
...
services/
productService.js
orderService.js
...
utils/
auth.js
errors.js
...
app.js
package.json

RUNNING THE APPLICATION:

Frontend (Client) Server:

1. Navigate to the client directory: cd client

2. Install dependencies (if not already installed): npm install

3. Start the frontend server: npm start

Default port: 3000

Access the frontend at: http://localhost:3000

Backend (Server) Server:

1. Navigate to the server directory: cd server

2. Install dependencies (if not already installed): npm install

3. Start the backend server: npm start

Default port: 8080 (or configured port in server/app.js)

Access the backend API at: http://localhost:8080/api

- To start both frontend and backend servers concurrently

1. Install concurrently package: npm install concurrently

2. Add scripts to package.json in the root directory:

```
"scripts": {
  "start:client": "npm start --prefix client",
  "start:server": "npm start --prefix server",
  "start": "concurrently \"npm run start:client\" \"npm run start:server\""
}
```

3. Start both servers: npm start

API DOCUMENTATION:

Authentication Endpoints

1. Register User:

- Method: POST
- Endpoint: /api/auth/register
- Parameters:
  - name: String (required)
  - email: String (required, unique)
  - password: String (required)
- Request Body:

```
{
  "name": "John Doe",
  "email": "johndoe@example.com",
  "password": "password123"
}
```

- Response:

```
{
  "message": "User created successfully",
  "token": "jwt-token"
}
```

2. Login User:

- Method: POST
- Endpoint: /api/auth/login

- Parameters:
    - email: String (required)
    - password: String (required)
- Request Body:

```
{
  "email": "johndoe@example.com",
  "password": "password123"
}
```

- Response:

```
{
  "message": "Login successful",
  "token": "jwt-token"
}
```

Product Endpoints

1. Get All Products

- Method: GET
- Endpoint: /api/products
- Parameters: None
- Response:

```
[
  {
    "_id": "product-id",
    "name": "Product Name",
```

```
    "price": 10.99,
    "description": "Product description"
  },
  ...
]
```

2. Get Product by ID

- Method: GET
- Endpoint: /api/products/:id
- Parameters:
    - id: String (required)
- Response:

```
{
  "_id": "product-id",
  "name": "Product Name",
  "price": 10.99,
  "description": "Product description"
}
```

3. Create Product

- Method: POST
- Endpoint: /api/products
- Parameters:
    - name: String (required)
    - price: Number (required)

    - description: String (required)
- Request Body:

```
{
  "name": "New Product",
  "price": 9.99,
  "description": "New product description"
}
```

- Response:

```
{
  "message": "Product created successfully",
  "_id": "new-product-id"
}
```

4. Update Product

- Method: PUT
- Endpoint: /api/products/:id
- Parameters:
    - id: String (required)
    - name: String (optional)
    - price: Number (optional)
    - description: String (optional)
- Request Body:

```
{
  "name": "Updated Product Name"
}
```

- Response:

```
{
  "message": "Product updated successfully"
}
```

5. Delete Product

- Method: DELETE
- Endpoint: /api/products/:id
- Parameters:
    - id: String (required)
- Response:

```
{
  "message": "Product deleted successfully"
}
```

Order Endpoints

1. Create Order

- Method: POST
- Endpoint: /api/orders
- Parameters:
    - products: Array of product IDs (required)
    - total: Number (required)
- Request Body:

{
  "products": ["product-id-1", "product-id-2"],
  "total": 20.98
}
- Response:
{
  "message": "Order created successfully",
  "_id": "order-id"
}
2. Get Order by ID

- Method: GET
- Endpoint: /api/orders/:id
- Parameters:
    - id: String (required)
- Response:
{
  "_id": "order-id",
  "products": ["product-id-1", "product-id-2"],
  "total": 20.98,
  "status": "pending"
}
3. Update Order Status

- Method: PUT

- Endpoint: /api/orders/:id
- Parameters:
  - id: String (required)
  - status: String (required, e.g., "shipped", "delivered")
- Request Body:

```
{
  "status": "shipped"
}
```

- Response:

```
{
  "message": "Order status updated successfully"
}
```

AUTHENTICATION:

Authentication:

1. User Registration: When a user registers, their password is hashed using bcrypt and stored in the MongoDB database.

2. User Login: When a user logs in, the provided password is hashed and compared to the stored hash.

3. JWT (JSON Web Tokens)Generation: Upon successful login, a JWT is generated using the user's ID, email, and other relevant information.

4. Token Signing: The JWT is signed with a secret key using the HS256 algorithm.

Authorization:

1. Token Verification: On subsequent requests, the client sends the JWT in the Authorization header.

2. Token Verification: The server verifies the token using the secret key and checks its expiration.

3. User Retrieval: If valid, the server retrieves the user's information from the database using the ID in the token.

4. Permission Checking: The server checks the user's permissions and roles to determine access to protected routes.

Token-Based Authentication:

1. Token Structure: The JWT contains the following information:

- User ID (sub)
- Email (email)
- Role (role)
- Expiration Time (exp)

2. Token Storage: The client stores the JWT in local storage or cookies.

3. Token Renewal: The token is renewed upon expiration or when the user logs out.