GROCERY ECOMMERCE SYSTEM

1. Introduction

The **Grocery Ecommerce System** is designed to revolutionize the way grocery shopping is done online by making it efficient, user-friendly, and accessible. The goal of this system is to allow users to browse a variety of grocery products, place orders, and manage their purchases from the comfort of their homes.

- **Target Audience**: Consumers who want to shop for groceries online.
- **Business Value**: It facilitates quick, easy access to groceries with an organized platform for store management and order fulfillment.

The system aims to achieve:

- Real-time product browsing and order tracking.
- Seamless customer experience, from placing an order to receiving notifications about status updates.
- Robust administrative capabilities for product, user, and order management.

The platform integrates a user-friendly frontend with a powerful backend that supports features like product management, customer authentication, and order processing.

2. Technologies Used

2.1 Backend

Python and **Django** are at the core of the system. Here's why:

- **Python** is easy to learn and has a rich ecosystem of libraries, making it an excellent choice for both development and rapid iteration.

- **Django** provides built-in tools that enable rapid web application development. It includes an ORM for database management, built-in user authentication, and security features like protection against cross-site scripting and SQL injection.

The **Django REST Framework (DRF)** takes Django's capabilities to the next level by allowing the backend to expose APIs. These APIs serve as the intermediary between the frontend and backend, allowing users to interact with the system seamlessly.

2.2 Database Management

PostgreSQL is chosen for its ability to handle relational data with ease, offering:

- **ACID compliance**: Ensures that transactions are processed reliably.
- **Scalability**: PostgreSQL can handle large volumes of data as the system grows.

We have designed a relational database schema to store and organize data efficiently. Key data entities like products, customers, and orders are stored in separate tables, and relationships between them are managed using foreign keys.

2.3 Asynchronous Task Management

Celery is used to handle background tasks such as:

- Processing orders
- Sending SMS and email notifications These tasks are offloaded to workers, which ensures that the main application remains responsive to user interactions without getting bogged down by time-consuming processes.

2.4 Notifications

- **SMS Notifications**: Powered by **Africa's Talking API**, the system sends SMS updates to customers regarding the status of their orders.
- **Email Notifications**: SMTP or an email API is used to send updates to both customers and admins. This keeps everyone involved in the order process informed in real-time.

3. System Architecture

3.1 High-Level Architecture

The system is built with a **microservices-based architecture** to promote scalability and modularity. Key components of the system include:

- **Frontend Layer**: This is where customers interact with the system, browsing products, adding them to the cart, and checking out.
- **API Layer**: Responsible for serving as the communication bridge between the frontend and the backend, the API layer (built using Django REST) processes requests, performs business logic, and returns data.
- **Authentication Layer**: Ensures secure access to the system using **JWT tokens** for user authentication.
- **Database Layer**: The PostgreSQL database stores data for products, orders, and users, ensuring that data is consistently available and reliable.
- **Notification Layer**: Handles the sending of updates via **SMS and email**.
- **Background Worker (Celery)**: Processes background tasks, ensuring that actions like order fulfillment and notifications don't interrupt the user experience.

3.2 Flow of Data

1. **User Request**: The user sends a request (e.g., product search, place an order) from the frontend.
2. **API Handling**: The API layer receives the request and processes it (e.g., fetching data from the database, validating user input).
3. **Business Logic Execution**: Business logic (like applying discounts, calculating prices) is applied.
4. **Response**: The API sends a response back to the frontend with the necessary data (e.g., order confirmation, product list).
5. **Notification**: Asynchronous notifications (SMS or email) are triggered as part of the order flow.

4. Database Schema Design

4.1 Tables & Relationships

- **Customers Table**: Stores personal information such as name, email, and phone number.
- **Products Table**: Contains product details like name, description, price, and category.
- **Categories Table**: Products are grouped into categories (e.g., Dairy, Snacks).
- **Orders Table**: This table stores order information, including which customer placed the order and the products ordered.

4.2 Data Integrity

Foreign keys ensure referential integrity between related tables:

- Orders reference customers via a foreign key.
- Products reference categories via a foreign key.

We also have cascading delete rules in place, so when a customer or product is deleted, the related orders or entries are automatically cleaned up.

5. API Documentation

API endpoints are designed to be flexible and efficient. Below are the key functionalities offered by the system.

5.1 Authentication Endpoints

- **POST /auth/register**: Registers a new user by collecting personal details such as name, email, and password.
- **POST /auth/login**: Authenticates a user based on credentials and returns a JWT token to authenticate further requests.
- **POST /auth/logout**: Logs the user out and invalidates their token, ensuring secure access control.

5.2 Product Management Endpoints

- **GET /products/**: Retrieves a list of all products available in the system.
- **POST /products/**: Adds a new product (admin-only).
- **PUT /products/{id}**: Updates a product's details (admin-only).
- **DELETE /products/{id}**: Deletes a product from the system (admin-only).

5.3 Order Management Endpoints

- **POST /orders/**: Allows customers to place new orders by adding products to their cart.
- **GET /orders/{id}**: Retrieves order details for a particular order (e.g., items, status).
- **PATCH /orders/{id}/status**: Allows the admin to update the status of an order (e.g., Shipped, Delivered).

6. Authentication & Authorization

6.1 JWT Authentication Flow

JWT authentication ensures that users only access resources they are authorized to use. Here's how it works:

1. **Login**: The user enters credentials (email and password).
2. **Token Issuance**: The system validates credentials and generates a JWT token.
3. **Token Use**: The token is passed in the Authorization header for every subsequent request.
4. **Validation**: The backend verifies the JWT on each request to ensure it's valid and hasn't expired.

7. Notifications

7.1 SMS Notifications via Africa's Talking API

When a customer places an order, the system automatically sends a **confirmation SMS** to the customer's phone number. Notifications also include updates on order status (e.g., "Your order has been shipped").

7.2 Email Notifications

Admin and customer notifications are sent via email using SMTP or a third-party email API. Admins are notified whenever an order is placed, updated, or canceled.

8. Deployment Instructions

8.1 Local Development Setup

For local development, the application can be set up on any machine by following these steps:

1. **Clone the repository**: git clone https://github.com/eKidenge/grocery-ecommerce.git
2. **Set up a virtual environment**: python -m venv env
   source env/bin/activate # On Windows: env\Scripts\activate
3. **Install dependencies**: pip install -r requirements.txt
4. **Run database migrations**: python manage.py migrate
5. **Start the server**: python manage.py runserver 127.0.0.1:8000

8.2 Production Setup

In production, we recommend deploying the app in **Docker containers** and using **Kubernetes** for orchestration. This setup ensures that the app is scalable and can handle traffic efficiently.

1. **Build Docker Image**: docker build -t grocery-ecommerce .
2. **Push to DockerHub**: docker push username/grocery-ecommerce
3. **Deploy with Kubernetes**: kubectl apply -f k8s/deployment.yaml

9. Testing Strategy

Testing ensures that every part of the system works as expected:

9.1 Unit Testing

Unit tests verify individual components. For example, testing the order model to ensure that it correctly calculates the total price.

9.2 Integration Testing

Integration tests ensure that the system functions as a whole, validating interactions between the database, backend, and frontend.

9.3 End-to-End Testing

Simulates the complete user flow (e.g., browsing products, placing orders) to ensure that the end-user experience is seamless.

10. Troubleshooting & Debugging

In case of issues, here are some common troubleshooting steps:

- **Database Issues**: Ensure that PostgreSQL is running and accessible.
- **API Errors**: Check if the API endpoints are responding correctly. Use tools like Postman for API testing.
- **Notification Failures**: Verify your configuration for SMS and email services.