MAD-2 Project: Grocery Store Application - GrocerEase

Author Name: Archit Handa | **Roll Number:** 22f2000744 | **Student Email:** 22f2000744@ds.study.iitm.ac.in
A budding computer scientist currently pursuing BS in Data Science and Applications from IIT Madras.

Description

The web-based Grocery Store application aims to offer a seamless online shopping experience. Key features include product browsing, cart management, order placement for users; product CRUD, raise request for category CRUD, statistical summaries for store managers; category CRUD, store manager sign up and request approvals for admin.

Technologies Used

- **Frontend**
  - **Vue.js 2** - A JavaScript framework for building user interfaces
  - **VueRouter** - A router for Vue.js allowing for fine-grained navigation control
  - **Vuex** - A state management library for Vue.js
- **Backend**
  - **Flask** - A web application backend framework that allows creation of controllers to tie-up the entire application
  - **Flask-SQLAlchemy** - A Flask extension that supports SQLAlchemy (an ORM) to design database models
  - **Flask-RESTful** - A Flask extension that allows effective creation of REST APIs
  - **Flask-Caching** - A Flask extension for caching support for backend
  - **Flask-Security** - A Flask extension that adds security mechanisms and authentication features
  - **Celery** - An asynchronous task queue/job queue implementation for Python
  - **smtplib** - A Python library for sending emails
- **Database**
  - **Redis** - In-memory data structure store used as message broker for Celery and store for caching
- **Development Environment** - VS Code

Architecture and Features

The grocery store application, which I have named ***GrocerEase***, is a **Web-based** application built upon the **Client-Server** architecture. It primarily follows the Model-View-Controller (**MVC**) design pattern approach with a particular focus on 'Separation of Concerns', allowing for modularity and ensuring maintainability.

The **Model** component is defined within the *models.py* file, where classes have been created for individual relations in the database. Leveraging the ORM, these classes encapsulate attributes and relationships specific to each entity.

The **Controllers** have been implemented across the *app.py* and *resources.py* files. Here, employing the Flask frameworks, separate view functions and **API** endpoints are defined respectively, which are triggered as and when a particular URL endpoint is requested.

The **View** of the application is controlled using the Vue.js library that renders the appropriate Vue components as per the need. The application is designed as a Single Page Application (**SPA**), meaning there is just one *index.html*, situated in the *templates* folder, whose **DOM** is manipulated by *app.js* and its several components, residing in the *static* folder.

The **Database** employs SQLite database engine and is set up as *development.sqlite3* under the *instance* folder.

To configure the application for the intended purpose, development in this case, a separate *config.py* and *celeryconfig.py* have been created.

The *backend* folder houses various *.py* files for specialized purposes. The feature to send email is set up in *mail_service.py*, while the backend tasks/jobs are defined in the *tasks.py*. The celery initialization is done in the *worker.py* file, caching instance is created in the *instances.py*, and finally, *security.py* creates a Flask-Security datastore employing Users and Roles tables of the database.

The application allows an admin to login and approve the sign up of a new store manager. The admin can also perform CRUD operations on the categories. The app even allows store managers to modify the categories by raising specific requests that are later approved or declined by the admin before being published. The store manager, on the other hand, has been given entire control over CRUD of products. As an additional feature, the store manager can also get an overview of how sales and revenue figures are behaving with respect to each category and product by checking the data and stats page that provides diagrams for the same. Moreover, the store manager can export the inventory details as a CSV or XLSX file.

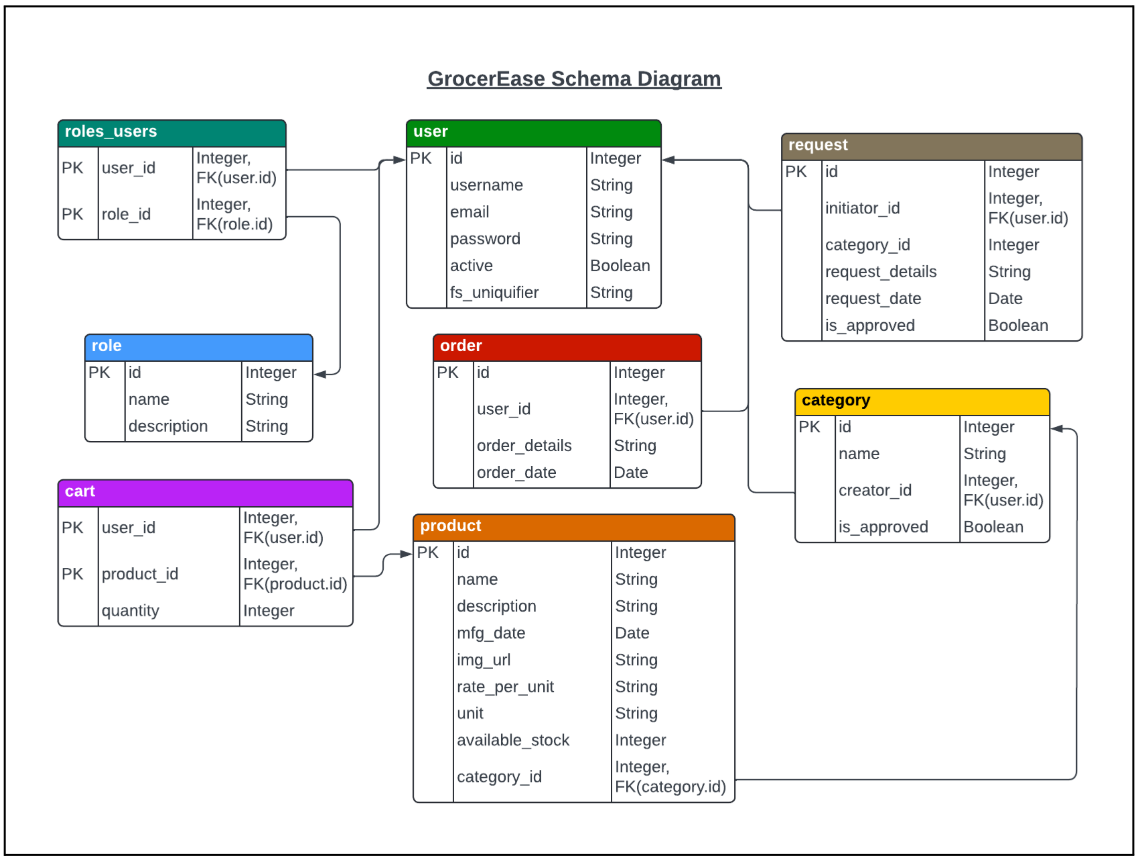
Users can log in, apply filters for a customized view, and fill their carts with products across categories. The cart offers a summary of selected items and the total checkout price. The orders page allows users to view their past purchases.

The app ensures security through authentication tokens. It sends daily reminders to users to visit the app and monthly order summaries that can be saved as PDFs. It even sends notifications to the store managers about low or out-of-stock products.

In order to improve the performance of the app, certain endpoints have been cached at the backend, which are cleared as and when a non-GET request is accepted.

Finally, a *manifest.json* is also loaded by the app that allows the app to be added to the desktop, and works like a native app. This is step towards making the app a Progressive Web App (**PWA**)

<u>Database Schema Design</u>



roles_users

PK	user_id	Integer, FK(user.id)
PK	role_id	Integer, FK(role.id)

user

PK	id	Integer
	username	String
	email	String
	password	String
	active	Boolean
	fs_uniquifier	String

request

PK	id	Integer
	initiator_id	Integer, FK(user.id)
	category_id	Integer
	request_details	String
	request_date	Date
	is_approved	Boolean

role

PK	id	Integer
	name	String
	description	String

order

PK	id	Integer
	user_id	Integer, FK(user.id)
	order_details	String
	order_date	Date

category

PK	id	Integer
	name	String
	creator_id	Integer, FK(user.id)
	is_approved	Boolean

cart

PK	user_id	Integer, FK(user.id)
PK	product_id	Integer, FK(product.id)
	quantity	Integer

product

PK	id	Integer
	name	String
	description	String
	mfg_date	Date
	img_url	String
	rate_per_unit	String
	unit	String
	available_stock	Integer
	category_id	Integer, FK(category.id)

User, *Role*, *Category*, *Product*, *Order*, *Request*, and *Cart* are the entity sets in the database. *RolesUsers* act as a relationship set between *User-Role* so as to work with Flask-Security datastore requirements.

<u>API Design</u>

API endpoints have been designed to provide essential functionality for both customers and staff, such as fetching carts, orders, and even statistical summaries by sales and revenue. These are a part of the *resources.py* file.

<u>Future Scope</u>

While the app has achieved its primary objectives, there are opportunities for future enhancements:

1. Implement more responsive features to create a lively and engaging UI/UX by exploring more JS+CSS concepts.
2. Enhance performance by employing a declarative style of programming for certain imperative pieces of code.
3. Explore Vuex capabilities for enhanced state management, ensuring data consistency and simplifying complex state transitions.
4. Integrate web workers to add PWA functionality, enabling background processes and improving offline capabilities. This enhancement can contribute to a more resilient and user-friendly application.

While developing the app, I have grown, navigating a learning curve and embracing a continuous improvement mindset. Excited, I look forward to an ongoing journey of growth and refinement in my role as a developer.

Video

[Video Link - https://drive.google.com/file/d/197O9ahPfcgen4HxURwb2WFJvEkXB7uFr/view?usp=sharing](https://drive.google.com/file/d/197O9ahPfcgen4HxURwb2WFJvEkXB7uFr/view?usp=sharing)