# Backend Web Services for eCommerce Platform

Student Name(s): Karim Abboud, Rayan Fakhreddine

Date of Submission: 1/12/2024

# Table of Contents

1. Title Page  
2. Table of Contents  
3. Introduction  
4. System Architecture  
5. Implementation Details  
6. Database Design  
7. Error Handling and Validation  
8. Testing  
9. Deployment and Integration  
10. Documentation and Profiling  
11. GitHub and Version Control  
12. Docker and Images  
13. Validation and Sanitization  
14. User Authentication  
15. Moderation  
16. Additional Professional Tasks  
17. References  
18. Appendices

# Introduction

## Project Overview

## The eCommerce backend project is a comprehensive solution designed to handle critical functionalities of an online shopping platform, including customer management, inventory control, sales transactions, and product reviews. The backend comprises four containerized services, each dedicated to a specific aspect of the platform, ensuring modularity and efficient communication via APIs. Utilizing Supabase for the database, the project implements email verification and advanced data security through third-party services. By leveraging Docker, Python, and Thunder Client (same as Postman), the system is built for scalability, security, and seamless integration. The project’s focus on containerization and API-driven design ensures a robust and adaptable solution suitable for modern eCommerce applications.

## Objectives

1. **Develop Modular Backend Services**:
   * Create four distinct, containerized services (Customers, Inventory, Sales, and Reviews) to handle separate functionalities of an eCommerce platform efficiently.
2. **Enable API-Driven Communication**:
   * Implement robust RESTful APIs to ensure seamless communication between the services, facilitating scalable and flexible integration.
3. **Facilitate Customer Management**:
   * Provide APIs to manage customer data, including registration, updates, account charging, and wallet deductions, while ensuring unique usernames and secure data handling.
4. **Streamline Inventory Management**:
   * Develop functionalities to add, update, and deduct items in stock, ensuring accurate tracking of product availability and details.
5. **Manage Sales Transactions**:
   * Enable efficient sales processing, including checking customer wallet balance and item stock availability, deducting payments, and maintaining a history of customer purchases.
6. **Incorporate Review and Rating System**:
   * Allow customers to submit, update, and delete product reviews, retrieve reviews for products or customers, and provide moderation capabilities to maintain review integrity.
7. **Leverage Modern Technologies**:
   * Utilize Docker for containerization, Supabase for database management, and third-party services for email verification, ensuring security, scalability, and reliability.
8. **Test APIs with Thunder Client (Postman)**:
   * Thoroughly test all API functionalities using Postman to ensure correctness, reliability, and compliance with project requirements.

These objectives collectively aim to deliver a robust, efficient, and secure backend solution for an eCommerce platform.

# System Architecture

## Service Description

**MVC Architecture Overview**

The eCommerce backend follows the **Model-View-Controller (MVC)** architectural pattern, which organizes the application into three interconnected components:

1. **Model**:
   * Represents the **data layer** of the application.
   * Contains the logic for interacting with the database hosted on **Supabase**, including customer details, inventory records, sales history, and reviews.
2. **View**:
   * Acts as the **semi-frontend** layer implemented using **FastAPI**.
   * Defines the endpoints and API responses that the user interacts with via tools like Postman.
   * Ensures that the data presented to users is structured and easy to understand.
3. **Controller**:
   * Serves as the **business logic layer**, handling requests from the View and performing necessary operations using the Model.
   * Processes API requests, validates input, applies logic, and manages communication between the View and the Model.

This structure ensures a clear separation of concerns, improving maintainability and scalability while streamlining the development and debugging process.

**1. Customers Service (Rayan)**

This service is responsible for managing customer data and wallet operations.  
**Key Responsibilities**:

* **Customer Registration**:
  + Registers new customers by storing essential details such as full name, username, password, age, address, gender, and marital status.
  + Validates the uniqueness of the username, returning an error if it is already taken.
* **Customer Management**:
  + Provides functionality to update or delete customer information.
  + Supports updates to multiple fields in a single request.
* **Wallet Management**:
  + Allows customers to charge their wallet with a specified dollar amount.
  + Deducts money from the wallet during purchases.
* **Data Retrieval**:
  + Retrieves a list of all registered customers.
  + Fetches customer details based on their unique username.

**2. Inventory Service (Karim)**

This service manages the inventory of goods available for sale.  
**Key Responsibilities**:

* **Adding Goods**:
  + Adds new items to the inventory, storing details such as name, category (e.g., food, clothes, electronics), price per item, description, and stock count.
* **Updating Goods**:
  + Allows updates to fields related to a specific item, including price, description, and stock count.
* **Stock Management**:
  + Deducts items from stock when a sale is made, ensuring inventory accuracy.
* **Data Consistency**:
  + Maintains data integrity for stock levels and prevents the sale of out-of-stock items.

**3. Sales Service (karim)**

This service handles customer purchases and manages sales data.  
**Key Responsibilities**:

* **Displaying Available Goods**:
  + Provides a list of items in the inventory, including their name and price.
* **Fetching Product Details**:
  + Retrieves complete information about a specific product, including description, category, and stock availability.
* **Processing Sales**:
  + Ensures the following conditions before processing a sale:
    - The customer has sufficient funds in their wallet.
    - The item is in stock.
  + Deducts the item from inventory and the corresponding amount from the customer's wallet.
* **Purchase History**:
  + Tracks and stores the history of all purchases made by customers for future reference.

**4. Reviews Service (Rayan)**

This service manages product reviews and ratings, providing a platform for customer feedback.  
**Key Responsibilities**:

* **Submitting Reviews**:
  + Allows customers to post reviews for products, including a rating (e.g., 1-5 stars) and a comment.
* **Updating Reviews**:
  + Enables customers to modify their reviews, ensuring up-to-date feedback.
* **Deleting Reviews**:
  + Allows customers and administrators to delete inappropriate or outdated reviews.
* **Fetching Reviews**:
  + Retrieves all reviews for a specific product, providing insights into customer experiences.
  + Lists all reviews submitted by a specific customer for historical tracking.
* **Moderating Reviews**:
  + Administrators can moderate reviews by flagging or approving them to maintain content integrity.
* **Review Details**:
  + Retrieves detailed information about a specific review, including the username, product name, rating, and comments.

Each service is developed as an independent module with specific responsibilities, making the system modular, scalable, and maintainable. These services interact seamlessly through their RESTful APIs to deliver the functionalities of a fully operational eCommerce platform.

# Implementation Details

## Service-Specific Implementation

**Service 1 – Customers (Rayan)**

**APIs and Challenges:**

* **APIs**: Provides CRUD operations for customers and wallet management features.
  + Add funds to the wallet: PUT /customers/add\_money/{id}
  + Deduct funds from the wallet: PUT /customers/deduct\_money/{id}
  + Query all or individual customers: GET /customers/, GET /customers/{id}
  + Create or update multiple customers: POST /customers/, POST /customers/many
  + Delete customer records: DELETE /customers/{id}
* **Challenges**:
  + Ensuring accurate wallet transactions with sufficient balance checks.
  + Error handling for scenarios like invalid IDs or concurrent updates to wallet.

**Service 2 – Inventory (Karim)**

**Details about Managing Goods:**

* **APIs**: CRUD operations for managing inventory stock.
  + Retrieve inventory items: GET /inventory/, GET /inventory/{id}
  + Create/update inventory items: POST /inventory/, PUT /inventory/{id}
  + Bulk creation: POST /inventory/many
  + Delete inventory items: DELETE /inventory/{id}
* **Challenges**:
  + Maintaining data integrity during stock updates.
  + Preventing inconsistencies during concurrent inventory modifications.

**Service 3 – Sales (Karim)**

**Details about Processing Sales:**

* **Features**:
  + Process sales transactions.
  + Store historical sales data for analysis.
  + Automatically update inventory levels upon sale completion.
* **Challenges**:
  + Implementing atomic operations to handle sales and inventory updates.
  + Storing and querying large volumes of historical sales data efficiently.

**Service 4 – Reviews (Rayan)**

**Details about Review Submission:**

* **APIs**:
  + Submit product reviews: POST /reviews/
  + Update existing reviews: PUT /reviews/{id}
  + Moderate reviews to manage inappropriate content.
* **Challenges**:
  + Automating moderation while ensuring fairness.
  + Handling spam submissions and maintaining database performance.

**Base Router Overview**

The BaseRouter provides a generic implementation for CRUD operations, minimizing code duplication. Key features include:

* Dynamic model-based query parameter parsing.
* Error handling with detailed responses.
* Modular router creation for reusable functionality.

**Example: Customer Wallet Operations**

* **Add Funds**: Uses add\_money\_to\_wallet to increase the wallet balance, ensuring no negative values.
* **Deduct Funds**: Validates sufficient funds before deduction using deduct\_money.

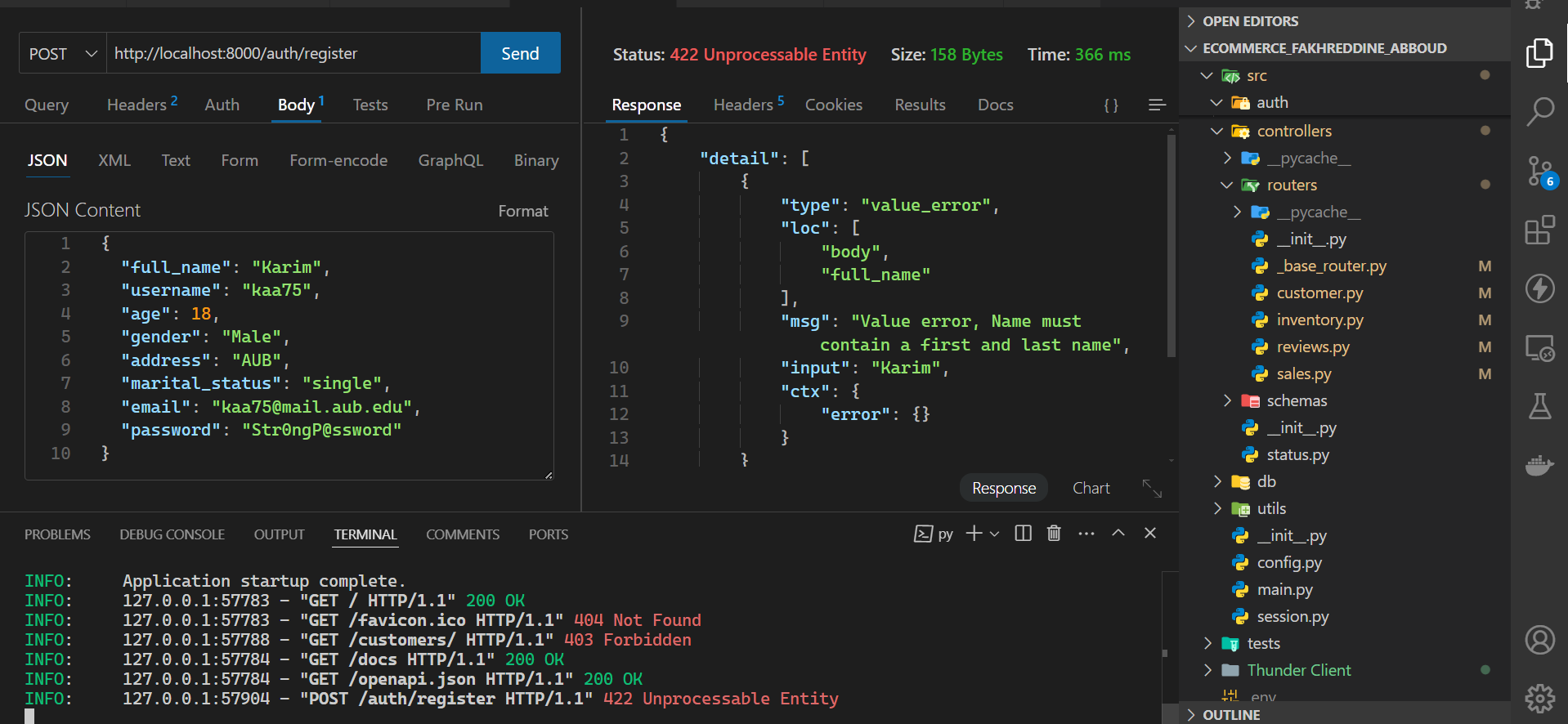
**Key Challenges in Project:**

* Ensuring consistency in distributed transactions (e.g., wallet deduction and inventory updates).
* Designing reusable components like BaseRouter to minimize boilerplate.
* Balancing performance with flexibility in query and transaction handling.

## API Documentation

A screenshot of a computer

Description automatically generated

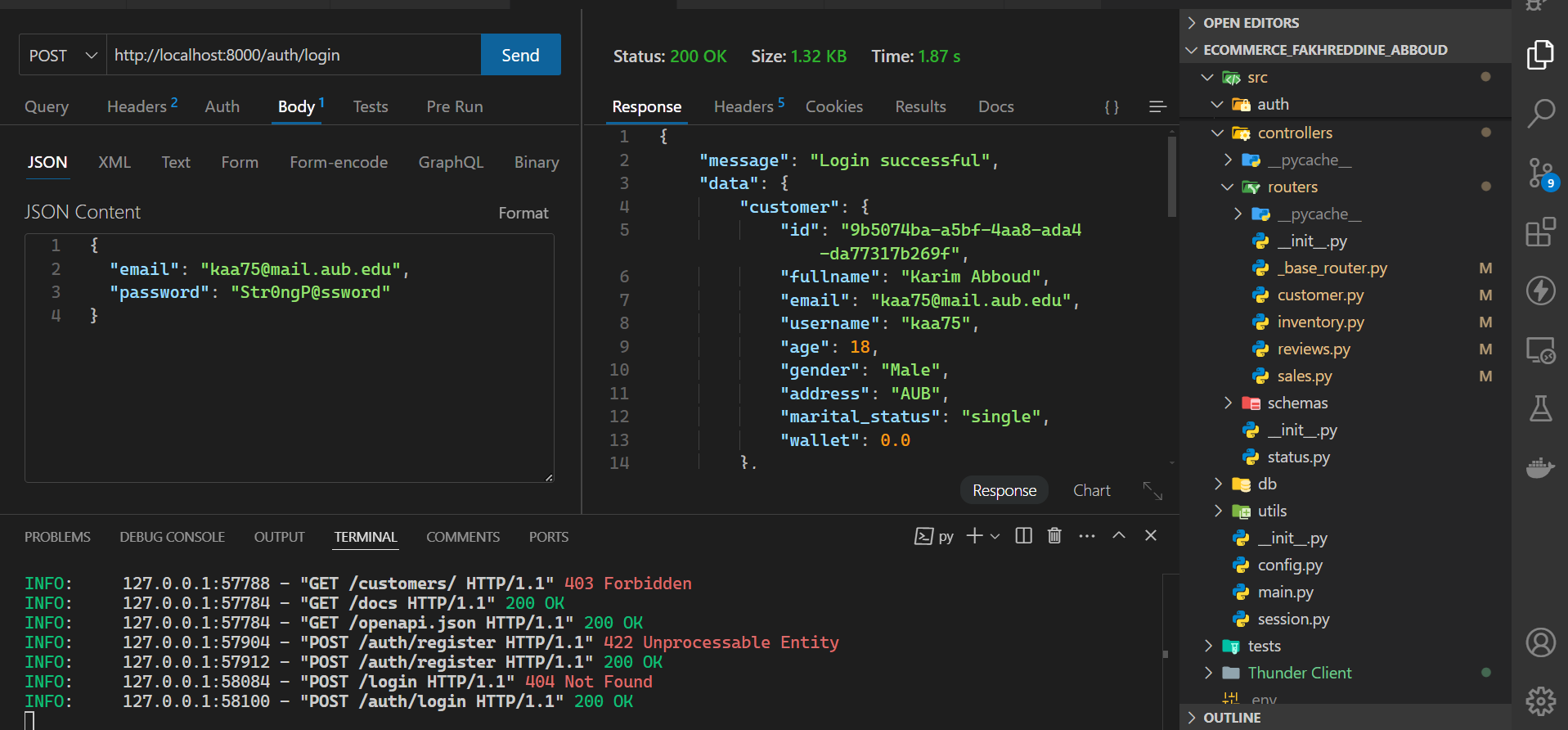


A screenshot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generated



Some Features are not Available on Thunder Client, so we Switched back to Postman.

# Database Design

## Schema Diagram

Show the database schema, including tables and fields. (Rayan)

Provided by SupaBase:

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

# Error Handling and Validation

## Error Management

In the system, error management is handled systematically to ensure that errors are caught early and managed gracefully. The core principles include:

1. **Centralized Error Handling:**
   * All errors are captured in a centralized location, often using try-catch blocks or error middleware (in web frameworks like Flask or Express).
   * This ensures that errors do not propagate out of control, which might crash the application or expose sensitive information to the user.
2. **Error Logging:**
   * Errors are logged with sufficient detail, including the error type, message, timestamp, and the context in which the error occurred. This helps in troubleshooting and improving system stability.
   * Logging mechanisms often store errors in local log files or send them to external monitoring systems (e.g., Sentry, Loggly).
3. **Graceful Degradation:**
   * When an error occurs, the system does not abruptly stop. Instead, it handles the error and ensures the system continues functioning in a degraded mode if necessary. For instance, displaying a user-friendly error message while logging the technical details.
4. **Error Response to Users:**
   * The user interface provides feedback to the user about errors in a user-friendly manner (e.g., “Something went wrong, please try again later”). This avoids showing raw technical details that might confuse the user or pose a security risk.
5. **Error Notification:**
   * If critical errors occur, automated notifications (such as emails or alerts) are sent to the development or support team for immediate action.

## Validation

1. **Input Validation:**
   * Input from users or external systems is validated at various stages of data processing. For example, form inputs are checked for the correct type (e.g., numeric, string), length, and format (e.g., email address format, date format).
   * In web applications, client-side validation using JavaScript ensures immediate feedback, while server-side validation ensures security and correctness after the data reaches the server.
2. **Business Rule Validation:**
   * Data is validated against predefined business rules to ensure it adheres to the required logic. For example, if a user is registering an account, the system checks whether the username is unique.
3. **Regular Expressions:**
   * Regular expressions are used for pattern-based validation, such as ensuring that emails, phone numbers, and postal codes are in the correct format.
4. **Boundary Checks:**
   * For numeric inputs, boundary checks are performed to ensure the input value falls within acceptable ranges (e.g., age between 18 and 100, quantity between 1 and 1000).
5. **Error Messages for Invalid Inputs:**
   * Clear error messages are provided when validation fails, describing why the input is invalid (e.g., "Username must be at least 6 characters long" or "Invalid email format").
   * These messages are localized or internationalized to ensure accessibility for users in different regions.
6. **Cross-field Validation:**
   * Sometimes, validation must occur across multiple fields. For instance, confirming that the “password” and “confirm password” fields match or that the "start date" is earlier than the "end date."
7. **Server-Side Validation:**
   * While client-side validation is useful for convenience, server-side validation is essential for security. The system ensures that all incoming data is properly sanitized and validated to prevent malicious input like SQL injection or XSS attacks.

By combining effective error handling and robust validation strategies, the system ensures stability, security, and a smooth user experience.

# Testing (Rayan)

## Testing Strategy

1. **Unit Testing:**

* **Purpose:** Test individual components or functions in isolation.
* **Approach:** Write tests for each function, focusing on edge cases and expected behavior.
* **Tools:** **Pytest** (for Python), **unittest**.

2. **Integration Testing:**

* **Purpose:** Test interactions between modules (e.g., frontend-backend, database-API).
* **Approach:** Ensure data flows correctly between components.
* **Tools:** **ThunderClient** (for API testing), **Postman**.

3. **System Testing:**

* **Purpose:** Validate the complete system as a whole.
* **Approach:** Perform end-to-end tests, ensuring all components work together.
* **Tools:** Pytest, Unittest and Manual testing.

4. **User Acceptance Testing (UAT):**

* **Purpose:** Ensure the system meets business requirements and user needs.
* **Approach:** Gather feedback from users to verify system usability.
* **Tools:** Manual testing, user surveys.

5. **Security Testing:**

* **Purpose:** Ensure the system is secure from vulnerabilities.
* **Approach:** Test for common security threats (e.g., SQL injection, XSS).
* **Tools:** Manual Testing.

6. **Performance Testing:**

* **Purpose:** Verify system performance under load.
* **Approach:** Simulate high user load and measure response times.
* **Tools:** **coverage.py** (for performance analysis and identifying bottlenecks).

## Test Cases

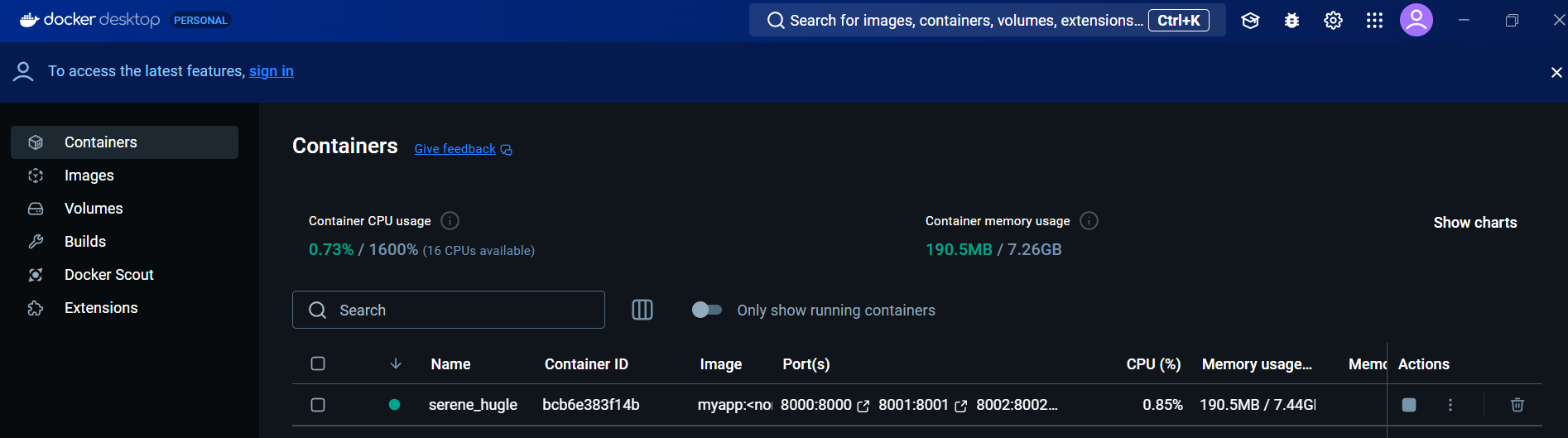
Provide examples of test cases and results.

[INSERT IMAGE HERE for Pytest Results]

# Deployment and Integration (Karim)

## Docker Setup

Explain the containerization process with Docker. Include Dockerfile and docker-compose.yml details.



A screenshot of a computer

Description automatically generated

Ran Dockerfile using the commands:

1. docker build -t ecommerce\_fakhreddine\_abboud

2.

Ran it using:

1. docker run -d -p 8000:800 8001:8001 8002:8002 8003:8003 –name ecommerce\_fakhreddine\_abboud ecommerce\_fakhreddine\_abboud

2.

Docker file:

1. FROM python:3.12-alpine

2.

3. # Set work directory

4. WORKDIR /app

5.

6. # Install dependencies

7. COPY requirements.txt .

8. RUN pip install --no-cache-dir -r requirements.txt

9.

10. # Copy project

11. COPY . .

12.

13. # Expose ports for each service

14. EXPOSE 8000 8001 8002 8003

15.

16. # Start multiple services

17. CMD ["sh", "-c", "uvicorn src.main:app --host 0.0.0.0 --port 8000 & uvicorn src.main:app --host 0.0.0.0 --port 8001 & uvicorn src.main:app --host 0.0.0.0 --port 8002 & uvicorn src.main:app --host 0.0.0.0 --port 8003"]

18.

Docker-compose.yml

1. services:

2.   auth\_service:

3.     build:

4.       context: ./auth\_service

5.     ports:

6.       - "8000:8000"

7.   customers\_service:

8.     build:

9.       context: ./customers\_service

10.     ports:

11.       - "8001:8000"

12.   inventory\_service:

13.     build:

14.       context: ./inventory\_service

15.     ports:

16.       - "8002:8000"

17.   review\_service:

18.     build:

19.       context: ./review\_service

20.     ports:

21.       - "8003:8000"

22.

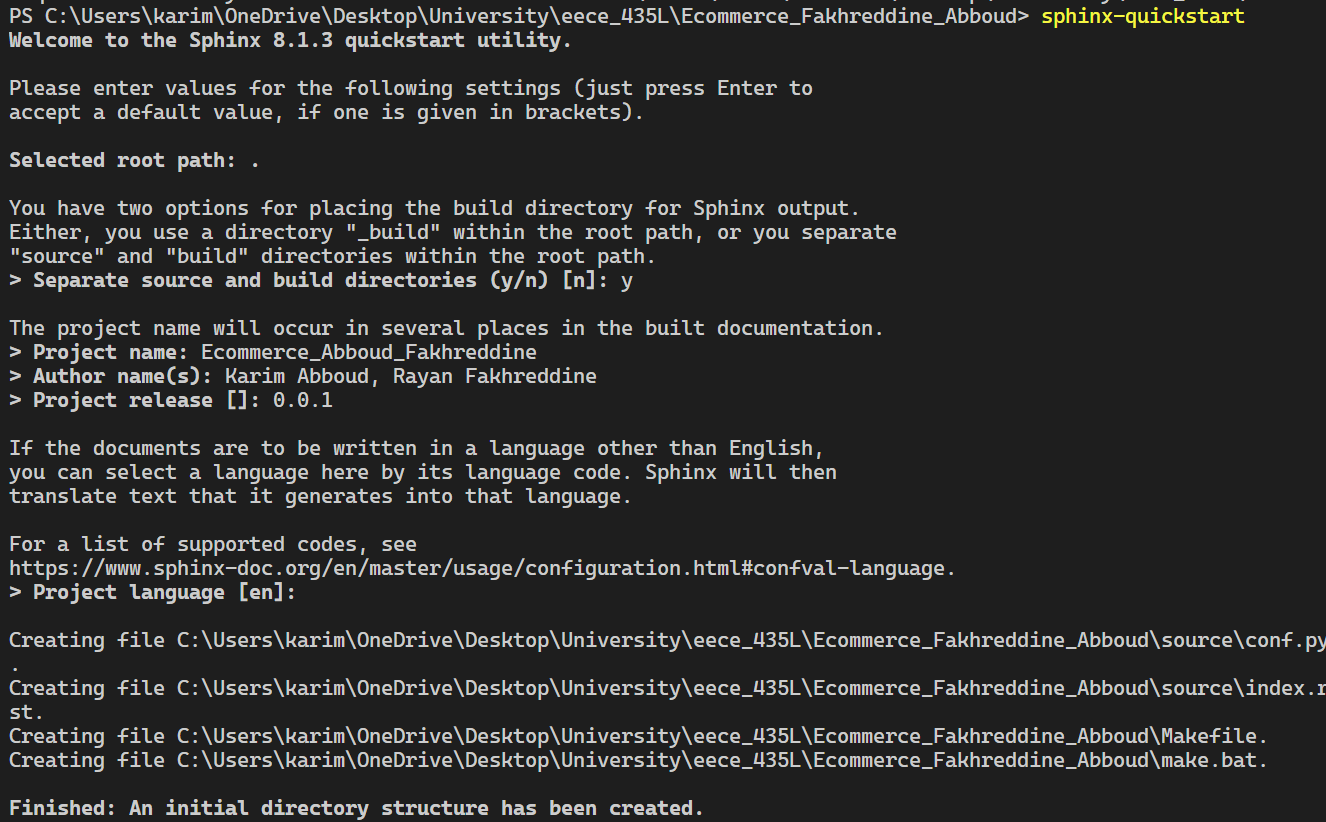
# Documentation and Profiling

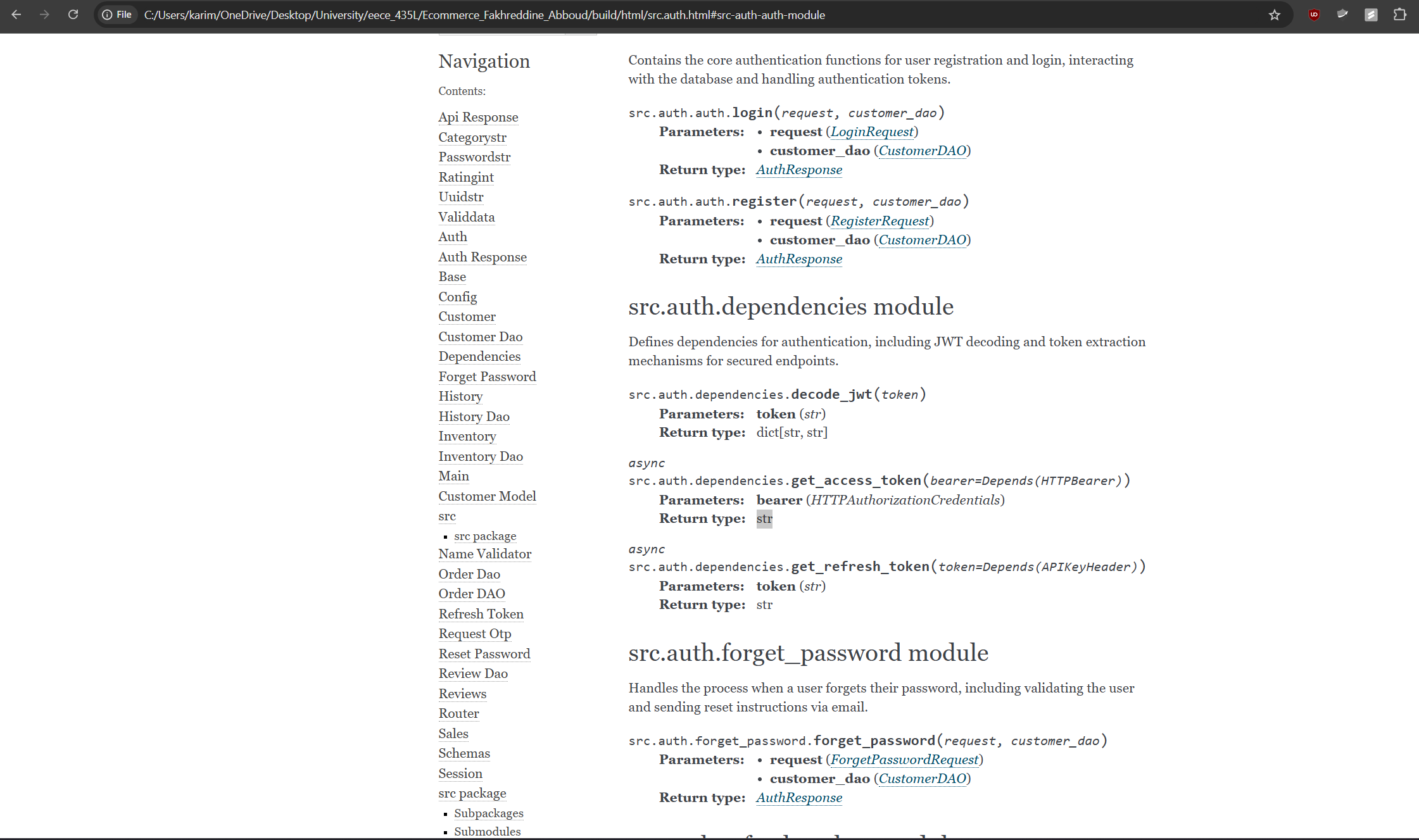
## Documentation (Karim)

all Documentation are included in the build/index.html (in the GitHub repo)

and built from source/index.rst (in the GitHub repo)

A screenshot of a computer

Description automatically generated  A computer screen shot of a program

Description automatically generated 

## Performance Profiling (Rayan)

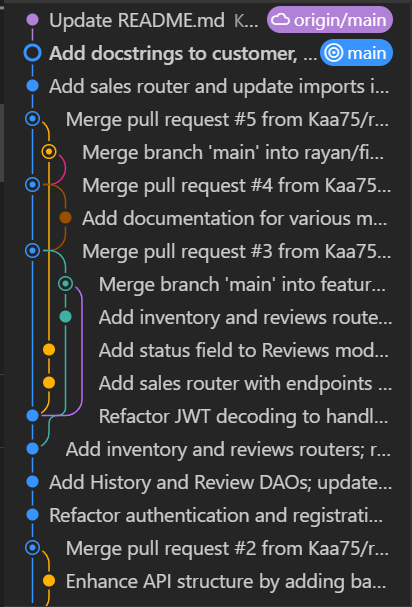
Provide snapshots of profiling results for performance and memory.

[INSERT IMAGE HERE for Profiling Results]

# GitHub and Version Control (Karim)

## Repository Links

<https://github.com/Kaa75/Ecommerce_Fakhreddine_Abboud>

 A diagram of a diagram

Description automatically generated with medium confidence

A screenshot of a black screen

Description automatically generated

A black background with white text

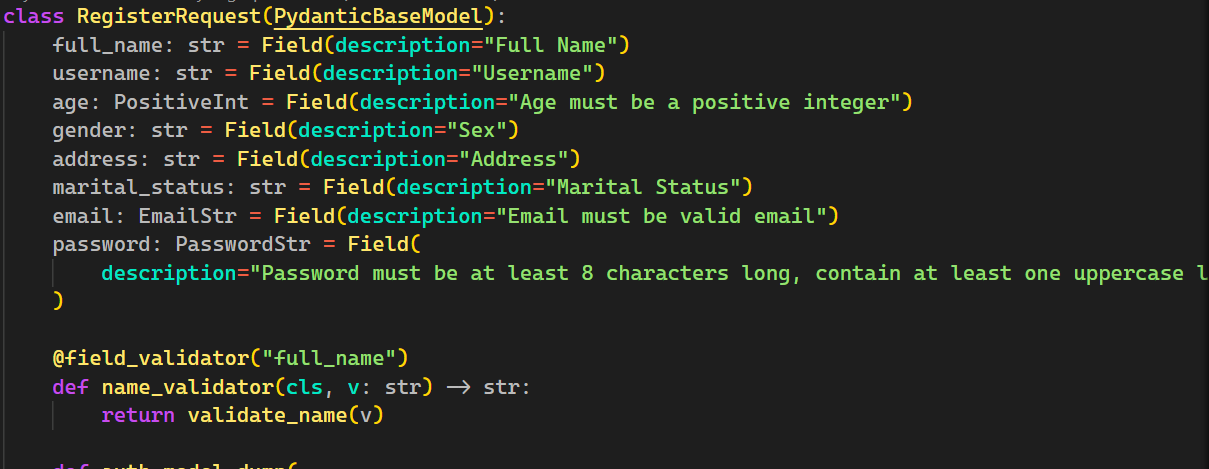
Description automatically generated

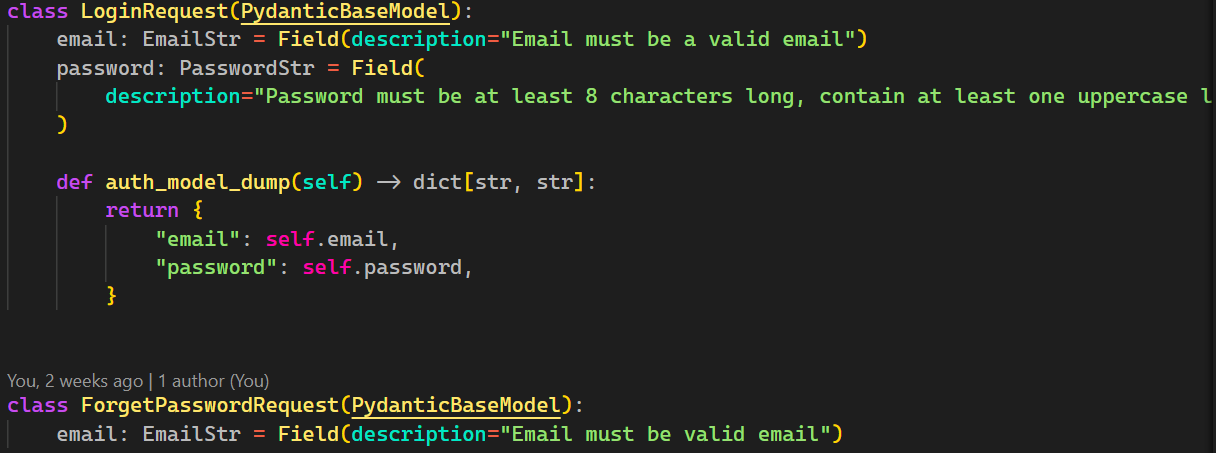
# Validation and Sanitization

In our project, most of the validation is handled server-side using Supabase, which ensures that the data conforms to the expected formats and business rules. As a result, we did not need to implement custom sanitization logic for user inputs on the client-side.

For the validation process, we leverage FastAPI's built-in validation mechanisms, which utilize Pydantic models and request bodies to ensure that incoming data is correctly structured. Additionally, validation is implemented using FastAPI's pipes, which automatically parse and validate the data directly from the JSON payloads. These pipes ensure that only properly formatted data is accepted, reducing the chances of errors or malicious input.

By relying on FastAPI and Supabase for validation and sanitization, we ensure that the application is both secure and efficient, with minimal manual input handling.





A screen shot of a computer program

Description automatically generated

# User Authentication

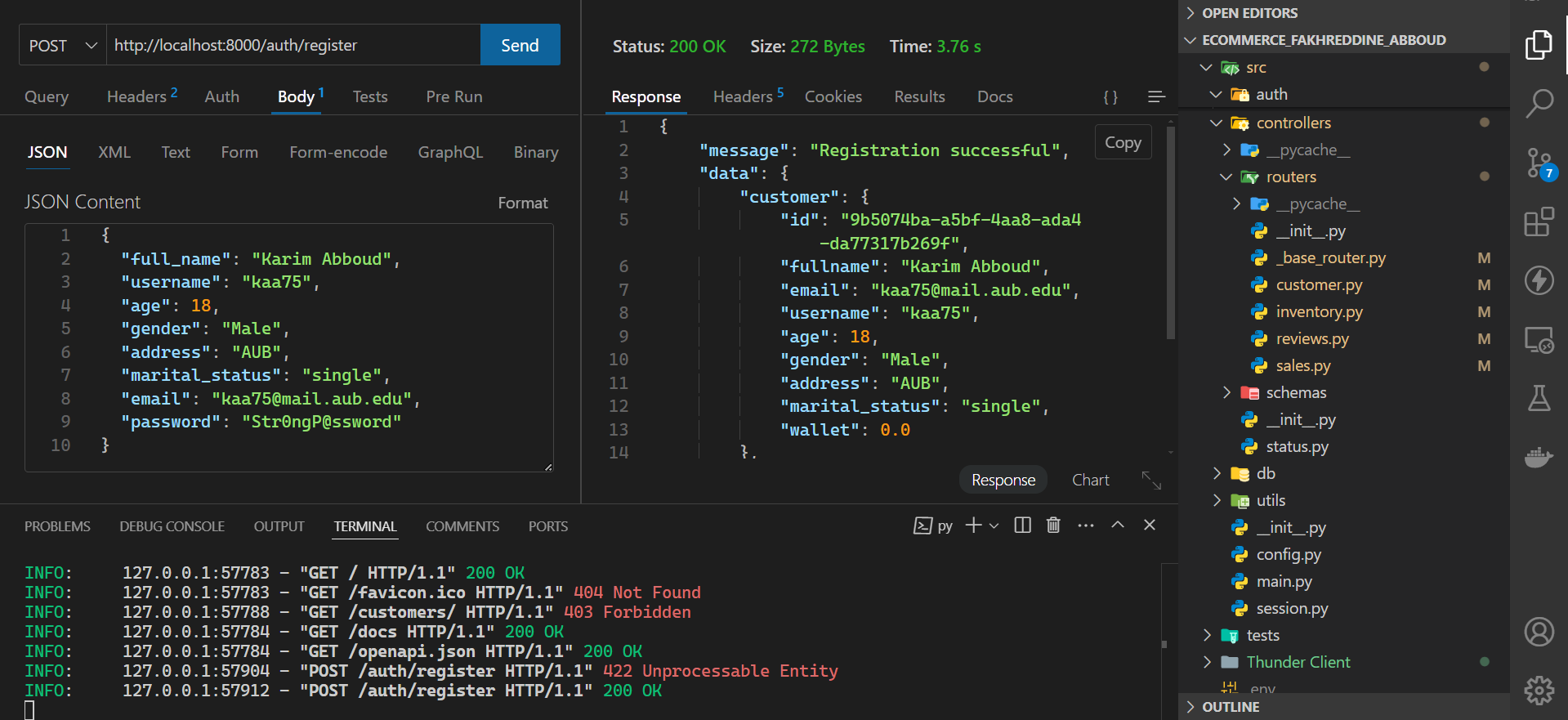
**User Authentication and Authorization**  
In our application, user authentication and authorization are handled securely using Supabase, which provides built-in features for both. Supabase uses JSON Web Tokens (JWT) for authenticating users and ensuring secure access to the application.

1. **Authentication**:
   * When users log in, their credentials (username/email and password) are verified through Supabase's authentication API.
   * Upon successful login, Supabase issues a JWT token that is stored on the client side, typically in local storage or cookies, to maintain the user's session.
   * This token is used in subsequent requests to authenticate the user, allowing them to access protected resources and endpoints.
2. **Authorization**:
   * Authorization checks are performed on the server side to ensure that users have the appropriate permissions to access specific resources or perform certain actions.
   * Supabase offers role-based access control (RBAC), allowing us to define different roles (e.g., admin, user, guest) and set granular permissions on database tables and API endpoints.
   * Each request to the server includes the JWT token, which is validated to ensure the user is authorized to perform the requested action.
   * Authorization logic is implemented in FastAPI, where specific routes are protected using dependencies that check the user’s role or permissions.
3. **Session Management**:
   * The JWT token expires after a predefined period, prompting users to reauthenticate after the token's expiration.
   * We also implement token refresh mechanisms to extend the session when needed, ensuring a seamless user experience.

By combining Supabase's built-in authentication system with role-based authorization in FastAPI, we maintain secure user access control and data protection throughout the application.

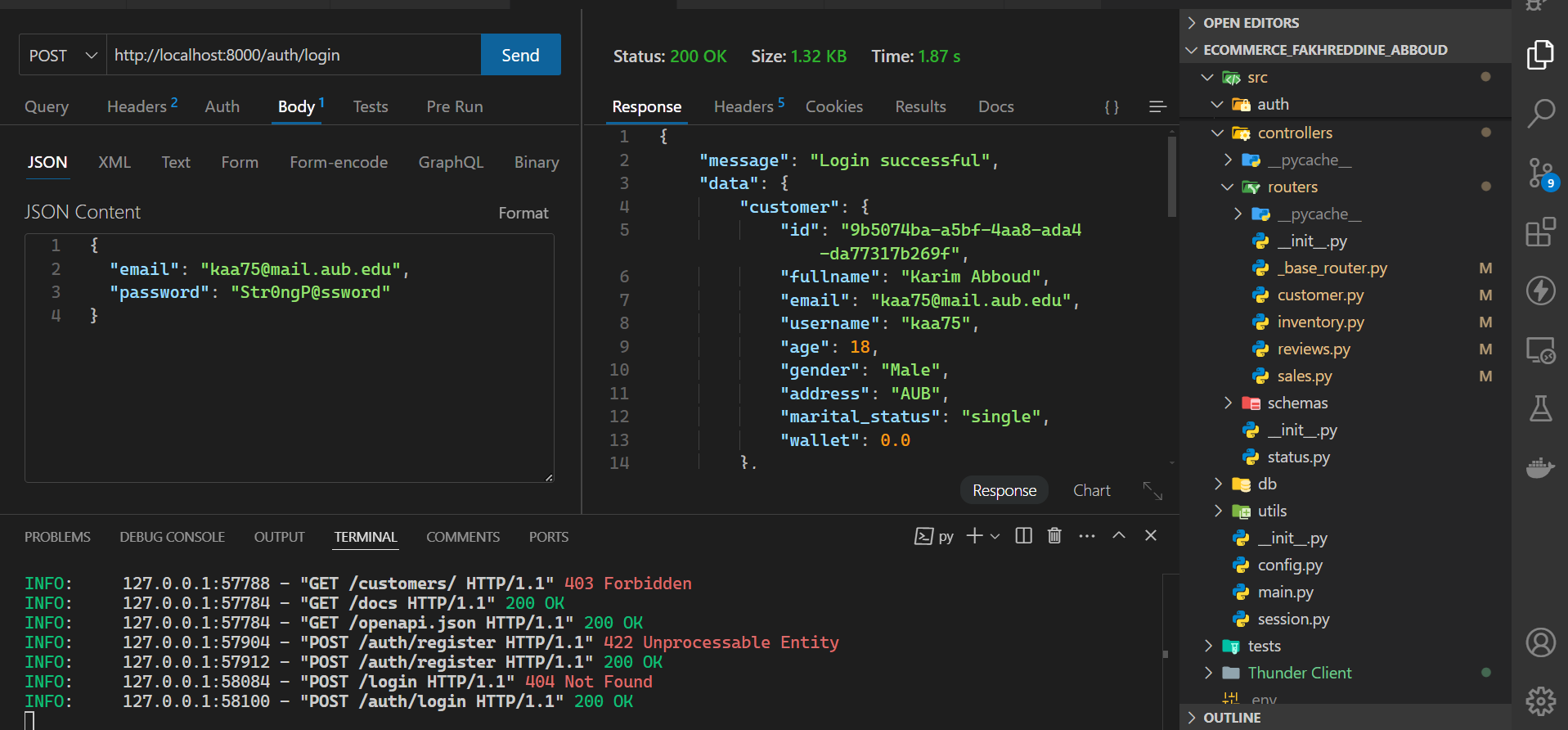
A screenshot of a computer

Description automatically generated



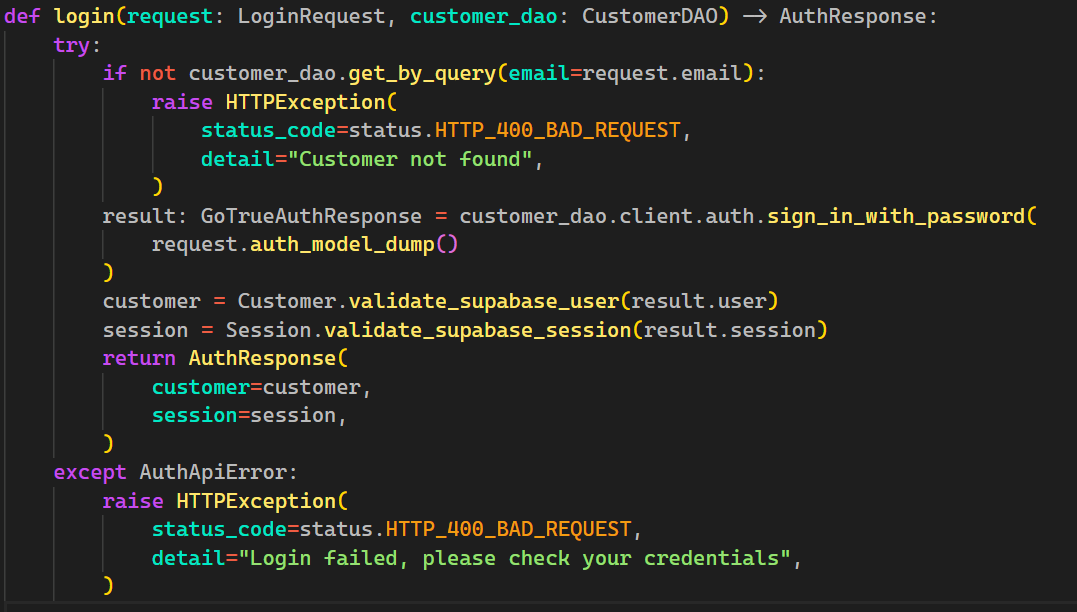
A screenshot of a computer

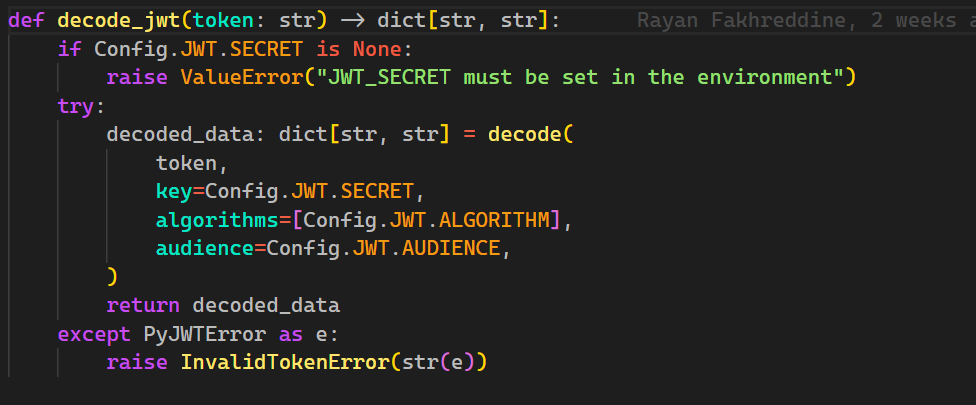
Description automatically generated

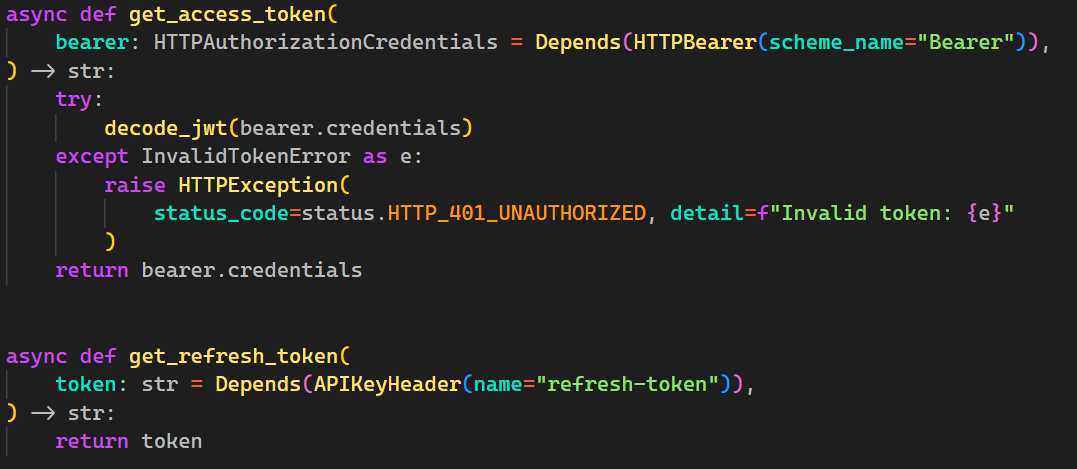


A screen shot of a computer program

Description automatically generated



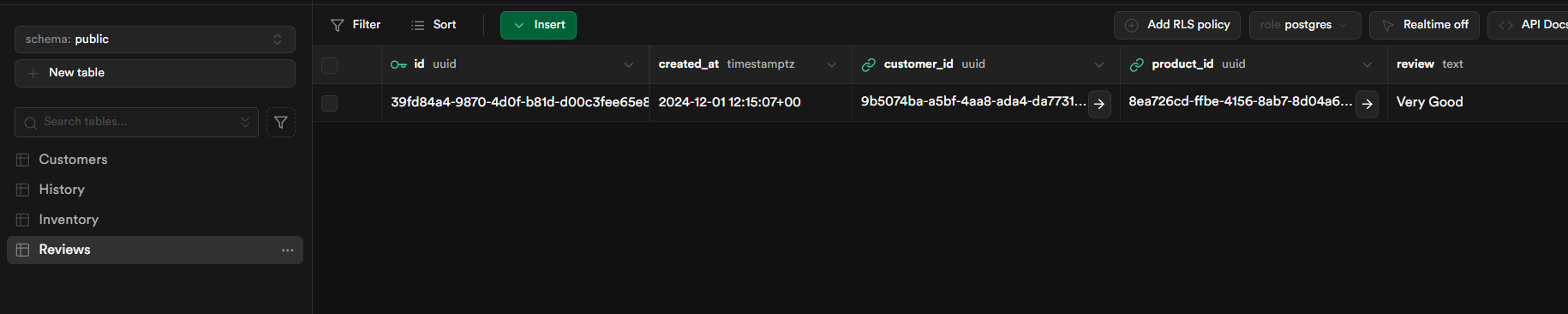




# Moderation (Rayan)

To ensure the platform maintains high-quality content, we implemented a backend moderation system for handling inappropriate reviews. The system allows users to flag reviews, and administrators can review flagged content, act, and update the review status accordingly. The moderation process is integrated with the frontend to display the review status appropriately without revealing any rejected statuses to the end user.

1. **Review Submission and Flagging**:
   * When users submit a review, it is stored in the database with an initial **"pending"** status.
   * Users have the option to **flag** a review if they find it inappropriate. When flagged, the review’s status is updated to **"pending review"**, indicating it is under moderation.
   * The flagged review is sent to a **moderation queue** in the backend, where it is reviewed by an administrator or subject to automated content filters.
2. **Automated Moderation**:
   * The system runs automated checks on the review's content to detect potential issues such as offensive language, spam, or personal attacks using content moderation APIs (like the Perspective API).
   * If the automated system detects problematic content, the review is flagged for administrative review, and the status is marked as **"pending review"**.
3. **Administrator Moderation**:
   * Administrators have access to a moderation dashboard to view flagged reviews. The administrator can either:
     + **Accept** the review, changing the status to **"accepted"**.
     + **Reject** or **delete** the review if it violates the platform’s community guidelines, updating the review status accordingly (though this status is not shown to users on the frontend).
   * Rejected reviews are not shown with a **"rejected"** status to users, as the frontend only displays reviews marked as **"accepted"** or **"pending review"**.
   * Administrators can also choose to **suspend or ban users** who repeatedly submit inappropriate content.
4. **Frontend Display(if Implemented)**:
   * The **frontend** only displays reviews with two possible statuses:
     + **Accepted**: Reviews that have been reviewed and approved by the administrator or passed the automated moderation checks.
     + **Pending Review**: Reviews that are flagged and are awaiting approval, indicating they are under review but not yet processed.
   * The **rejected** reviews are **not shown** on the frontend to avoid exposing any negative or rejected content to users. Rejected content is either deleted or hidden entirely.
5. **Review Status Update**:
   * Once an administrator reviews a flagged review, the status is updated in the database to either **"accepted"** or **"pending review"** (if it’s still under review). If a review is rejected, it’s removed from the database, ensuring no rejected content is exposed.
   * The **frontend** periodically checks for status updates (through API calls) to update the review's display status accordingly. If a review’s status is changed from "pending review" to "accepted," it becomes visible to users. If it's still pending, it remains in the queue for review.
6. **User Feedback(if frontend is implemented)**:
   * While users can flag reviews they believe are inappropriate, they will not see a specific **rejected status** or reason. This ensures that rejected reviews are kept private and not displayed to other users.
   * After a review is moderated, users who flagged the content may receive general notifications about actions taken (e.g., "Your flagged review has been reviewed"), but no details about the rejection process are exposed to maintain privacy and prevent confusion.
7. **Audit Logs**:
   * The backend maintains **audit logs** that record actions taken on flagged reviews, including the status changes (accepted, pending review, rejected), user flags, and administrator decisions. This log is accessible only to administrators for review and accountability.

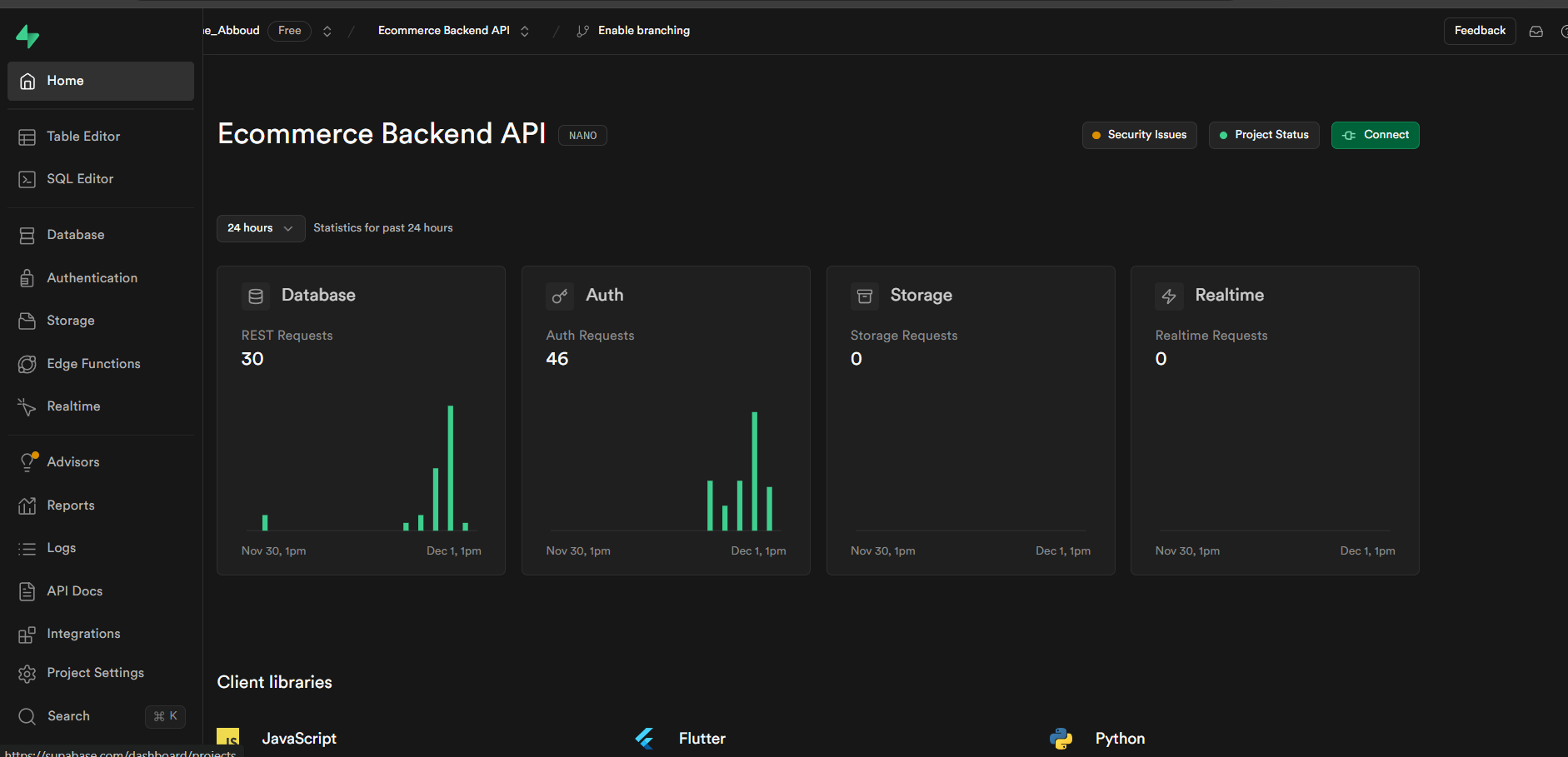


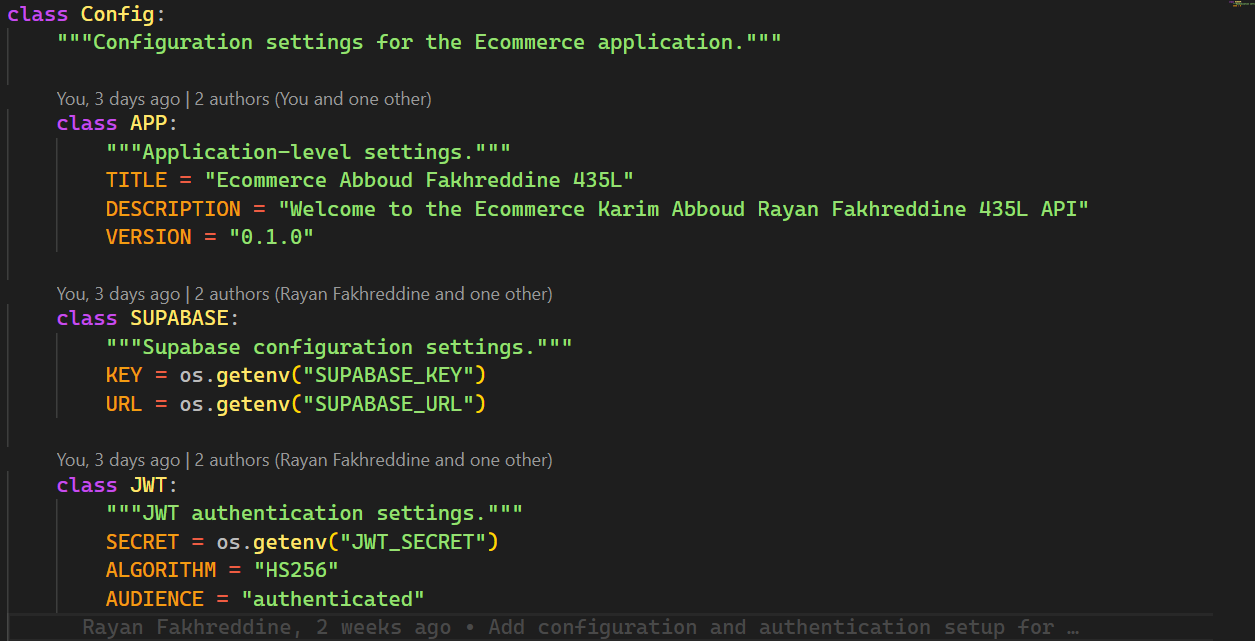
# Additional Professional Tasks

Several additional features were integrated into the application to enhance its functionality, security, and user experience. These include third-party services, Supabase integration, email verification, authentication, validation, and overall security measures.

1. **Third-Party Services**:
   * We integrated **third-party services** for various purposes, such as communication, storage, and analytics. These services were selected based on their reliability, scalability, and ability to enhance user experience.
   * For instance, we used services like **email providers** for sending verification emails and notifications. These providers were integrated through APIs, enabling seamless communication with users.
2. **Supabase Integration**:
   * **Supabase** was chosen as our backend-as-a-service solution. It provides authentication, real-time database, and storage functionalities, making it an efficient and secure choice for managing user data.
   * Supabase’s **authentication API** is used to handle user login, registration, and token management. This service simplifies authentication by providing built-in support for JWT tokens and session management, which are crucial for maintaining secure user sessions.
   * Supabase also offers role-based access control (RBAC) and fine-grained permissions, allowing us to secure specific resources based on the user’s role.
3. **Email Verification**:
   * To ensure the legitimacy of users, **email verification** was implemented during the registration process. After signing up, users receive a verification email with a unique link. Once the user clicks the link, their email is confirmed, enabling them to fully access the application.
   * This feature was integrated using **Supabase's built-in email verification system**, which automatically handles the email sending process and ensures that users cannot log in until their email is verified.
4. **Authentication**:
   * **User authentication** is managed via Supabase's authentication API, where users authenticate using their email and password. Upon successful login, a JWT token is issued and stored securely on the client side (usually in local storage or cookies). This token is used in subsequent requests to verify the identity of the user.
   * Token expiration and refresh mechanisms were implemented to ensure session integrity and security.
5. **Validation and Sanitization**:
   * Data **validation** is done on the server-side using FastAPI’s powerful validation features, which utilize Pydantic models to ensure that incoming data conforms to the required format.
   * **Sanitization** is handled indirectly through the use of Supabase and FastAPI, which automatically validate and sanitize inputs such as email addresses, usernames, and other user-generated data to protect against SQL injection and other common vulnerabilities.
   * FastAPI's pipes are used to parse data directly from the JSON payload, ensuring that only clean, properly structured data is accepted by the application.
6. **Security Measures**:
   * To protect sensitive user data, we implemented **SSL/TLS encryption** for all communication between the client and server, ensuring that data is transmitted securely over the network.
   * Passwords are hashed using modern cryptographic algorithms before being stored in the Supabase database, ensuring that user credentials remain secure even in case of a data breach.
   * Additionally, the application implements proper **rate-limiting** and **brute-force protection** mechanisms to prevent abuse, particularly on sensitive endpoints like login and registration.

By integrating these additional features, we ensure that the application remains secure, user-friendly, and scalable, with robust authentication, validation, and security measures in place.





A screenshot of a computer

Description automatically generated

# References

1. Frameworks **and Libraries:**

* **FastAPI**: For building the backend API and handling server-side validation, authentication, and routing.
* **Pydantic**: For data validation and parsing in FastAPI.
* **Supabase**: Used for backend services like user authentication, database storage, real-time functionality, and role-based access control (RBAC).
* **JWT (JSON Web Tokens)**: For handling user authentication and session management securely.

2. **Authentication and Authorization Services:**

* **Supabase Authentication**: To manage user sign-up, login, email verification, and session handling.
* **JWT Authentication**: For securing API endpoints with token-based user validation.

3. **Third-Party Services:**

* **Email Providers (from SupaBase)**: For sending verification emails and notifications to users.

4. **Development Tools and Environment:**

* **VSCode**: For development, with extensions for Python, React, and FastAPI.
* **Docker (if used)**: For containerizing the application and setting up a consistent development and production environment.
* **Postman/Thunder Client**: For manually testing API endpoints during development.
* **Git**: For version control and collaboration.
* **GitHub/GitLab**: For hosting the repository and collaboration.

5. **Databases and Storage:**

* **Supabase Database**: A PostgreSQL database provided by Supabase for storing user data, authentication credentials, and other application data.

6. **Security Tools and Libraries:**

* **FastAPI Security**: For adding authorization checks and protecting routes with permissions.
* **SSL/TLS Encryption**: For securing the communication between the client and server over HTTPS.

7. **Testing Tools:**

* **Pytest, Unittest**: For testing the backend logic, including validation, API endpoints, and database interactions.

8. **Documentation and Reference Materials:**

* **FastAPI Documentation**: Official documentation for learning how to use FastAPI effectively.
* **Supabase Documentation**: Guides for setting up authentication, databases, and storage with Supabase.
* **JWT.io**: For learning about JSON Web Tokens and how to use them for secure authentication.

9. **Deployment Tools:**

* **Docker**: For creating containerized environments for development, testing, and production.

10. **Collaboration and Project Management Tools:**

* **Trello:** For task tracking and organizing the project workflow.
* **WhatsApp/ GitHub/ Discord**: For team communication and discussions.