MINISTRY OF EDUCATION AND SCIENCE OF UKRAINE

SUMY STATE UNIVERSITY

FACULTY OF ELECTRONICS AND INFORMATION TECHNOLOGY

COMPUTER SCIENCE DEPARTMENT

"Approved for defense."

Acting Head of the Department

Igor SHELEHOV

____

(signature)

09.06.2023

____

GRADUATION THESIS

for obtaining the educational degree of Bachelor

in the specialty 122 - Computer Science,
educational-professional program "Informatics"
on the topic: " REST API for a store management system using Flask"
by the student of group IN-95AH, Ukpongson Miracle Praise.

The Bachelor Graduation Thesis contains the results of original research. The use of ideas, results, and texts of other authors is properly referenced to the respective sources.

[illegible]

________________ **Ukpongson Miracle P.**

(signature)

Supervisor
Candidate of Physical and Mathematical Sciences, Senior lecturer. **Oleksiienko G.A.** __________

(signature)

Sumy – 2023

SUMY STATE UNIVERSITY

FACULTY OF ELECTRONICS AND INFORMATION TECHNOLOGY

COMPUTER SCIENCE DEPARTMENT

«Approved»

Acting Head of the Department

Igor SHELEHOV

____
(signature)

TASK FOR THE GRADUATION THESIS

to obtain the educational degree of Bachelor

in the specialty 122 - Computer Science,

educational-professional program "Informatics"

by the student of group IN-95AH, Ukpongson Miracle Praise.

1. Topic of the Bachelor Graduation Thesis: "REST API FOR A STORE MANAGEMENT SYSTEM USING FLASK"
approved by the order of SumDU on *June 1, 2023.№ 0475-VI*

2. The deadline for the submission of the Bachelor Graduation Thesis *until June 9, 2023.*

3. Input data for the qualification work

4. Table of Contents for the Explanatory Memorandum (List of questions to be addressed) *1) Analysis of the subject area, defining the purpose, and forming the tasks of the Bachelor Graduation Thesis. 2) Review of the theoretical material. 3) Development of the REST API for a store management system using Flask4) Analysis of the obtained results.*

5. List of graphic materials (with specific mention of mandatory drawings)

6. Project consultants (with the corresponding sections of the project they are associated with).

Section	Consultant	Signature, date	
		The assignment has been issued	The assignment has been accepted

7. Date of assignment issuance «____» ________________ 20 ___

The assignment has been accepted for execution

Supervisor

[illegible]
_________________ _______________

(signature) (signature)

Calendar Plan

№	Titles of the stages of the Bachelor Graduation Thesis	Deadline	Note
1	*Analysis of the subject area, defining the purpose, and forming the tasks of the Bachelor Graduation Thesis*		
2	*Review of the theoretical material*		
3	*Development of the automation of an inventory system*		
4	*Analysis of the obtained results*		
5	*Bachelor Graduation Thesis Formatting*		

Higher education student [illegible] Supervisor

(signature) (signature)

ABSTRACT

Note: 41 pages, 15 figures, 14 references, 1 app.

Research object: Rest Api For a Store Management System Using Flask

Research objective: is to completely design, implement and deploy a REST API service for Store Management.

Research methods: Comparative analysis, experimental research, literature reviews, deductive analysis.

Results: a fully functional and deployed REST API, with implemented error handling, which allows it stay operational with limited developer oversight.

REST API’S, FLASK, PYTHON, POSTGRES, STORE MANAGEMENT, DATABASE MANAGEMENT, POSTMAN

CONTENTS

INTRODUCTION

In recent years, the growth of online stores and e-commerce has revolutionized the retail industry. With the increasing adoption of digital platforms, businesses are presented with new opportunities for expansion and reaching a wider customer base. However, managing store inventory and items effectively in the online realm poses unique challenges that require innovative solutions.4

To address these challenges, the design and implementation of a robust and user-friendly application programming interface (API) can greatly enhance store and item management systems. The use of a RESTful API, in particular, provides a standardized and flexible approach to building web services and facilitates integration with various platforms and technologies.

The primary objective of this thesis is to design and implement a REST API using the Flask framework for a comprehensive store and item management system. This API aims to streamline the process of managing store inventory, enabling businesses to efficiently handle items, track store contents, and retrieve item information. By utilizing Flask, a lightweight and extensible web framework written in Python, we can leverage its simplicity and flexibility to develop a scalable and efficient API.

The scope of this thesis encompasses the development of an API that supports essential functionalities for store and item management. This includes creating stores, managing item details (such as name and price), retrieving item information, and searching for specific items based on various tags. By focusing on these core functionalities, the API will provide users with a solid foundation for effectively managing their online stores.

Through the implementation of this REST API, we aim to improve the efficiency and accuracy of store and item management, empowering users to effectively handle inventory, and optimize their overall operations. The thesis will delve into the design, implementation, and evaluation of the API, providing insights into the technical aspects of its development and its potential impact on store management systems.

In the following sections, we will explore existing literature on REST APIs, delve into the Flask framework, review related store management systems, outline the system requirements, present the design and architecture of the API, discuss the implementation details, and evaluate the performance and usability through testing. Through this thesis, we aim to contribute to the field of store management systems by providing a practical and efficient solution for store and item management through the development of a REST API using Flask.

1 LITERATURE REVIEW

1.1 Analysis of the current state of the subject area

In this literature review, we will explore relevant studies, research papers, and existing frameworks that are pertinent to the design and implementation of a REST API using Flask for store and item management systems.

REST APIs [4][5]: Representational State Transfer (REST) is an architectural style for designing networked applications, widely used for building web services. REST APIs provide a standardized approach for creating, updating, and retrieving resources over the web through the use of HTTP methods such as GET, POST, PUT, and DELETE. The simplicity, scalability, and statelessness of REST make it an ideal choice for developing APIs in various domains. The principles of REST, including the use of resource-oriented URLs, stateless communication, and hypermedia as the engine of application state (HATEOAS)[10], ensure interoperability and ease of integration.

Store Management Systems: Several store management systems are available in the market, each offering a range of features for efficiently managing store inventory and items. For instance, Shopify is a widely-used e-commerce platform that provides comprehensive store management capabilities, including inventory tracking, order processing, and reporting. WooCommerce, another popular e-commerce platform that integrates with WordPress and provides extensive features for managing products, inventory, and customer relationships.. Understanding these existing systems helps in identifying best practices and functionalities that can be incorporated into the design and implementation of the REST API for store and item management.

By reviewing the existing literature on REST APIs, Flask framework, and store management systems, we can identify proven methodologies, design patterns, and best practices that can inform the development of our REST API for store and item management. This knowledge base will enable us to make informed decisions in the design and implementation stages, ensuring the API's efficiency, scalability, and usability.

In the following sections, we will build upon this foundation and leverage the knowledge gained from the literature review to design and implement a robust and user-friendly REST API using Flask for store and item management.

1.2 Review of known solutions

In this section, we will review and analyse three well-known solutions that utilize REST APIs for store-related applications. These solutions have been widely adopted by businesses and developers to enhance store management, integrate with external systems, and improve the overall customer experience. The following are the reviewed solutions:

1. Shopify API [1]:
    - Overview: Shopify, a popular e-commerce platform, provides a robust RESTful API that enables developers to build applications and integrations with Shopify stores.
    - Functionality: The Shopify API allows for managing various aspects of the store, including products, orders, customers, and inventory. Developers can create custom storefronts, automate tasks, and integrate third-party systems using the API.
    - Benefits: The Shopify API provides a comprehensive set of endpoints and resources, allowing businesses to extend the capabilities of their Shopify stores and streamline their e-commerce operations. The API's scalability and developer-friendly features make it a preferred choice for many businesses.
2. WooCommerce API [9]:
    - Overview: WooCommerce, a popular WordPress plugin for online stores, offers a REST API that allows developers to interact with WooCommerce stores programmatically.
    - Functionality: The WooCommerce API provides endpoints for managing products, orders, customers, and other store-related data. Developers can leverage the API to create custom storefronts, automate processes, and build personalized shopping experiences.
    - Benefits: The WooCommerce API offers seamless integration with WordPress-powered online stores, providing developers with flexibility and customization options. The API's extensive documentation and active developer community make it a reliable solution for businesses looking to enhance their WooCommerce stores.
3. Square API [9]:

- Overview: Square, a leading payment processing and point-of-sale provider, offers a comprehensive REST API for developers to build applications that interact with Square's services.
- Functionality: The Square API enables developers to manage inventory, process payments, track sales, and access other store-related data. It allows businesses to accept payments, manage transactions, and synchronize data across various systems.
- Benefits: The Square API provides a unified solution for payment processing and store management. It offers robust features and developer-friendly documentation, making it a popular choice for businesses across different industries.

These reviewed solutions demonstrate the effectiveness and versatility of REST APIs in-store management. They showcase how businesses can leverage these APIs to extend the functionality of their stores, integrate with external systems, and provide seamless customer experiences. By studying these known solutions, we gain valuable insights into the best practices, design patterns, and implementation strategies employed in building REST APIs for store-related applications.

1.3 System requirements

The system requirements subsection outlines the functional and non-functional requirements of the REST API for store and item management. These requirements define the essential features and performance expectations of the system.

Functional Requirements:

1. Store Management:
   - Create a new store with details such as name, address, and contact information.
   - Update store information, including name, address, and contact details.
   - Retrieve all items in a store.
   - Delete a store from the system.
2. Item Management:
   - Add a new item to a store with details such as name and price.
   - Update item details.
   - Delete an item from a store.

- Retrieve item details.
- Search for items based on specific tags.

3. User Authentication:
    - Implement user authentication mechanisms to secure access to the API endpoints.
    - Allow users to register and create an account.
    - Authenticate users using credentials such as username and password.
    - Provide token-based authentication using JSON Web Tokens (JWT) to authenticate subsequent API requests.

Non-functional Requirements:

1. Performance:
    - The API should support concurrent user requests without significant degradation in performance.
    - The system should be capable of handling a high volume of data without compromising performance.
2. Security:
    - Implement secure communication over HTTPS to protect data during transit.
    - Employ authentication mechanisms, such as token-based authentication using JWT, to ensure that only authorized users can access protected endpoints.
    - Implement schemas for proper input validation and data sanitization techniques to prevent common security vulnerabilities, such as SQL injection or cross-site scripting (XSS).
3. Scalability:
    - Design the system to be scalable, allowing for future growth in the number of stores and items.
    - Ensure that the database architecture and API design can handle increasing amounts of data and user traffic.
4. Error Handling:
    - Implement comprehensive error handling mechanisms to provide meaningful error messages and proper HTTP status codes in case of invalid requests or system failures.
    - Handle exceptions gracefully and log error details for debugging and troubleshooting purposes.

5. Documentation:
    - Provide thorough documentation for the API, including detailed descriptions of endpoints, request/response formats, and authentication procedures.
    - Document any required dependencies and setup instructions for developers who want to use the API.

By defining these system requirements, we establish the functional expectations and performance benchmarks for the REST API. These requirements guide the subsequent stages of design, implementation, and testing, ensuring that the API meets the desired objectives of efficient store and item management.

2 SELECTION OF PROBLEM SOLVING METHODS

In this section, we will discuss the methods chosen to solve the problem of designing and implementing a REST API for store and item management. These methods, including Python, Flask, Flask-Smorest, Flask-JWT-Extended, and the databases used, have been carefully selected to ensure the successful development and deployment of the API. Let's delve into each of these methods.

2.1 Selection of programming languages

Python Programming Language [2]:

Python has been chosen as the primary programming language for developing the REST API. Python's simplicity, readability, and extensive library support make it an ideal choice for web development projects. Its ecosystem of frameworks and tools provides developers with the necessary resources for efficient API development and maintenance.

Advantages of Python Programming Language:

- Simplicity: Python's clean and readable syntax reduces development time and enhances code maintainability.
- Extensive library support: Python offers a vast collection of libraries and frameworks that expedite development tasks and provide ready-to-use functionalities.
- Large community: Python has a large and active community of developers who contribute to its growth, provide support, and share resources.

Disadvantages of Python Programming Language:

- Performance: Python's interpreted nature can result in slower execution speed compared to compiled languages.
- Global Interpreter Lock (GIL): The GIL can limit parallel execution and affect performance in multi-threaded applications.

2.2 Selection a framework for implementation

Flask Web Framework [3]:

Flask, a lightweight and flexible web framework, has been selected as the foundation for building the REST API. Flask's simplicity, scalability, and extensive community support make it a popular choice for developing web applications. Its modular design allows for easy integration of additional functionalities and extensions, making it suitable for building robust APIs.

Advantages of Flask Web Framework:

- Lightweight and flexible: Flask's minimalistic design allows developers to customize components and build APIs according to specific requirements.
- Scalability: Flask's modular architecture enables easy addition of functionalities and extensions as the API grows.
- Extensive community support: Flask has a large community of developers who contribute plugins, extensions, and resources, making it easy to find solutions and get help when needed.

Flask-Smorest [7]:

Flask-Smorest, an extension for Flask, has been utilized to facilitate the development of a well-structured and documented RESTful API. This extension simplifies the process of defining and documenting API endpoints, request/response schemas, pagination, sorting, and filtering capabilities. Flask-Smorest streamlines the API development process and enhances the overall maintainability and readability of the codebase.

Advantages of Flask-Smorest:

- Well-structured and documented APIs: Flask-Smorest simplifies the process of defining and documenting API endpoints, resulting in APIs that are easy to understand and use.
- Built-in support for pagination, sorting, and filtering: Flask-Smorest provides convenient features for handling data manipulation operations in APIs.

Flask-JWT-Extended [6]:

Flask-JWT-Extended, another Flask extension, has been employed to handle authentication and authorization in the REST API. This extension adds support for JSON Web Tokens (JWT) and provides features such as token generation, token validation, and role-based access control. Flask-JWT-Extended enhances the security of the API by ensuring authorized access to protected endpoints and securing communication between the client and server.

Advantages of Flask-JWT-Extended:

- Authentication and authorization support: Flask-JWT-Extended allows for secure authentication using JSON Web Tokens (JWT) and enables role-based access control.

- Token generation and validation: Flask-JWT-Extended provides functionality for generating and validating JWTs, ensuring the security of the API.

2.3 Selection of DBMS

Database Management Systems:

During the testing phase, the SQLite database management system was used for its simplicity and ease of setup. SQLite is a self-contained, serverless, and file-based database engine that is widely used for development and testing purposes. It provides the necessary data storage capabilities required for the initial stages of the project.

Advantages of SQLite:

- Simplicity and ease of setup: SQLite is a lightweight and file-based database system that requires minimal configuration, making it easy to set up and use.
- Suitable for development and testing: SQLite is often used for development and testing purposes due to its simplicity and self-contained nature.

For the final implementation of the project, the PostgreSQL database management system has been chosen. PostgreSQL is a powerful, open-source, and scalable relational database system. It offers advanced features, transactional support, and excellent performance, making it suitable for production environments. The use of PostgreSQL ensures data integrity, reliability, and efficient querying capabilities for the store and item management system.

Advantages of PostgreSQL:

- Advanced features and scalability: PostgreSQL offers advanced database features, including support for complex queries, indexing, and transactional support. It is highly scalable and can handle large volumes of data efficiently.
- Reliability and data integrity: PostgreSQL is known for its robustness and data integrity features, making it suitable for production environments where data consistency is crucial.
- Open-source and community-driven: PostgreSQL is an open-source database system with an active community that provides support, updates, and extensions.

By leveraging Python, Flask, Flask-Smorest, and Flask-JWT-Extended, we benefit from their collective features, community support, and extensive documentation. These methods allow for rapid development, structured API design, enhanced security, and efficient request handling.

Additionally, the choice of SQLite during the testing phase and PostgreSQL for the final implementation reflects the project's progression and the need for a robust and scalable database solution.

Alternatives that could be considered for similar problems:

- For programming languages, alternatives to Python include JavaScript (with Node.js), Ruby, Java, C#, and Go. Each alternative has its own advantages and disadvantages, such as performance, ecosystem support, and community size.
- Alternative frameworks to Flask for API implementation include Django, Express.js, Ruby on Rails, ASP.NET, and Gin. These frameworks offer different features, conventions, and development approaches.
- Alternative database management systems to SQLite and PostgreSQL include MySQL, MongoDB, Microsoft SQL Server, and Oracle Database. The choice of the DBMS depends on specific requirements such as data structure, scalability, and data querying needs.

2.4 Selection of Deployment Environment

For the deployment of the REST API, the Render.com platform was selected as the deployment environment. Render.com is a reliable and scalable hosting platform that simplifies the deployment process and provides infrastructure management.

Advantages of Render.com:

- Reliable and scalable hosting platform: Render.com provides a reliable infrastructure for deploying web applications and offers scalability to handle varying levels of traffic.
- Simplified deployment process: Render.com streamlines the deployment process, making it easier for developers to deploy their applications without worrying about infrastructure management.

To facilitate the deployment, the application was containerized using Docker. Docker allows for easy packaging of the application and its dependencies into a portable container. This ensures consistency and portability across different environments.

Advantages of Docker:

- Easy application packaging and portability: Docker allows applications and their dependencies to be bundled into containers, providing a consistent environment across different systems.
- Consistency and reproducibility: Docker containers ensure that the deployed application runs the same way across development, testing, and production environments.

Alternative deployment environments include cloud platforms like AWS, Google Cloud Platform (GCP), and Microsoft Azure, as well as other hosting services like Heroku and DigitalOcean. Each alternative has its own features, pricing models, and scalability options which should be considered in relation to the solution to be produced.

By leveraging Render.com and Docker, the deployment process was streamlined, enabling easy scaling, monitoring, and management of the deployed API. This provided a reliable hosting environment for seamless interaction with clients.

Additionally, the PostgreSQL database, hosted on the ElephantSQL platform, was integrated with the deployed application. ElephantSQL is a managed PostgreSQL database hosting service that offers scalability and efficient data storage and retrieval.

Overall, the combination of these methods ensures the successful development and deployment of a REST API for store and item management.

3 SOFTWARE IMPLEMENTATION

3.1 Components and architectural

The design and architecture of the REST API for store and item management play a crucial role in ensuring scalability, modularity, and maintainability. This section outlines the key components and architectural decisions that will be employed in the development of the API.

API Structure: The API will follow a resource-oriented design, where resources are represented by URLs and accessed through HTTP methods. The following are some endpoints that will be defined:

1. /stores: This endpoint will be used for creating stores and returning all stores. It will support HTTP methods such as GET and POST for retrieving store information and creating new stores respectively.
2. /stores/<int:store_id>: This endpoint will be used for deleting and returning specific stores(using store id). It will support HTTP methods GET and DELETE.
3. /items: This endpoint will be used for creating items and returning all items. It will support HTTP methods such as GET and POST for retrieving item information and creating new items respectively.
4. /item/<int:item_id>: This endpoint will be used for deleting, updating and returning specific items(using item id). It will support HTTP methods GET,DELETE and PUT.

Data Models: The data models will be designed to represent the entities involved in the store and item management system. The key entities and their attributes will include:

Store:

- ID: A unique identifier for each store.
- Name: The name of the store.

Item:

- ID: A unique identifier for each item.
- Name: The name of the item.
- Price: The price of the item.

User: It will be used for authentication purposes.

Attributes:

- ID: A unique identifier for each user.
- Username
- Password

Tag: It will be used to address various item groups and associate with items and a store.

Attributes:

- ID: A unique identifier for each item.
- Name: The name of the item.
- Store ID: A foreign identifier which relates a tag to a store.

Items-Tags: A model for representing a many to many relationship between items and tags

Attributes:

- ID: A unique identifier for each item.
- Item ID: A foreign identifier which points to the unique item id of an item model.
- Tag ID: A foreign identifier which points to the unique tag id of a tag model.

The relationships between entities will be established using appropriate mechanisms, such as foreign keys or other associations. For example, an item will be associated with a specific store through a foreign key relationship.

Authentication and Security: To ensure secure access to the API, token-based authentication will be implemented using JSON Web Tokens (JWT). When a user successfully logs in or registers, a JWT will be generated and returned as part of the response. Subsequent API requests will require the inclusion of this token in the request headers to authenticate and authorize access to protected endpoints.

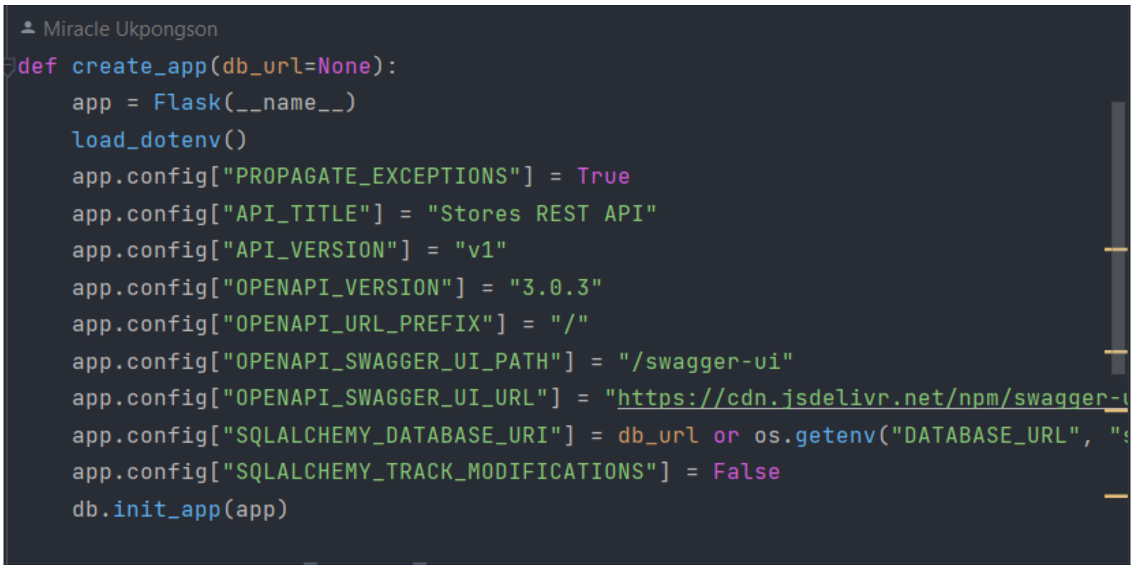
Integration Considerations: If integration with a frontend application is required, considerations such as implementing API authentication on the frontend, and ensuring secure communication between the frontend and the API will be taken into account. These considerations aim to facilitate seamless integration between the API and frontend components.

By adopting this design and architecture, the REST API for store and item management will exhibit modularity, scalability, and maintainability. It will provide a solid foundation for efficient store and item management, empowering businesses to effectively handle their inventory and improve overall operations.

3.2 Implementation of the REST API

The implementation of the REST API using Flask for the store and item management system involved several key components and technologies. The following steps outline the implementation process:

1. Setup and Configuration:
    - The Flask framework was utilized to develop the REST API. The Flask application was initialized, and the necessary configurations were set up. This included defining the application's title, version, and other OpenAPI-related details.



Miracle Ukpongson

```
def create_app(db_url=None):
    app = Flask(__name__)
    load_dotenv()
    app.config["PROPAGATE_EXCEPTIONS"] = True
    app.config["API_TITLE"] = "Stores REST API"
    app.config["API_VERSION"] = "v1"
    app.config["OPENAPI_VERSION"] = "3.0.3"
    app.config["OPENAPI_URL_PREFIX"] = "/"
    app.config["OPENAPI_SWAGGER_UI_PATH"] = "/swagger-ui"
    app.config["OPENAPI_SWAGGER_UI_URL"] = "https://cdn.jsdelivr.net/npm/swagger-
    app.config["SQLALCHEMY_DATABASE_URI"] = db_url or os.getenv("DATABASE_URL", "
    app.config["SQLALCHEMY_TRACK_MODIFICATIONS"] = False
    db.init_app(app)
```

2. Project Structure:
    - A well-organized project structure was created to maintain modularity and separation of concerns. The application code was structured into logical components, such as instance, models, resources and migrations. This structure allows for easy navigation and future scalability.

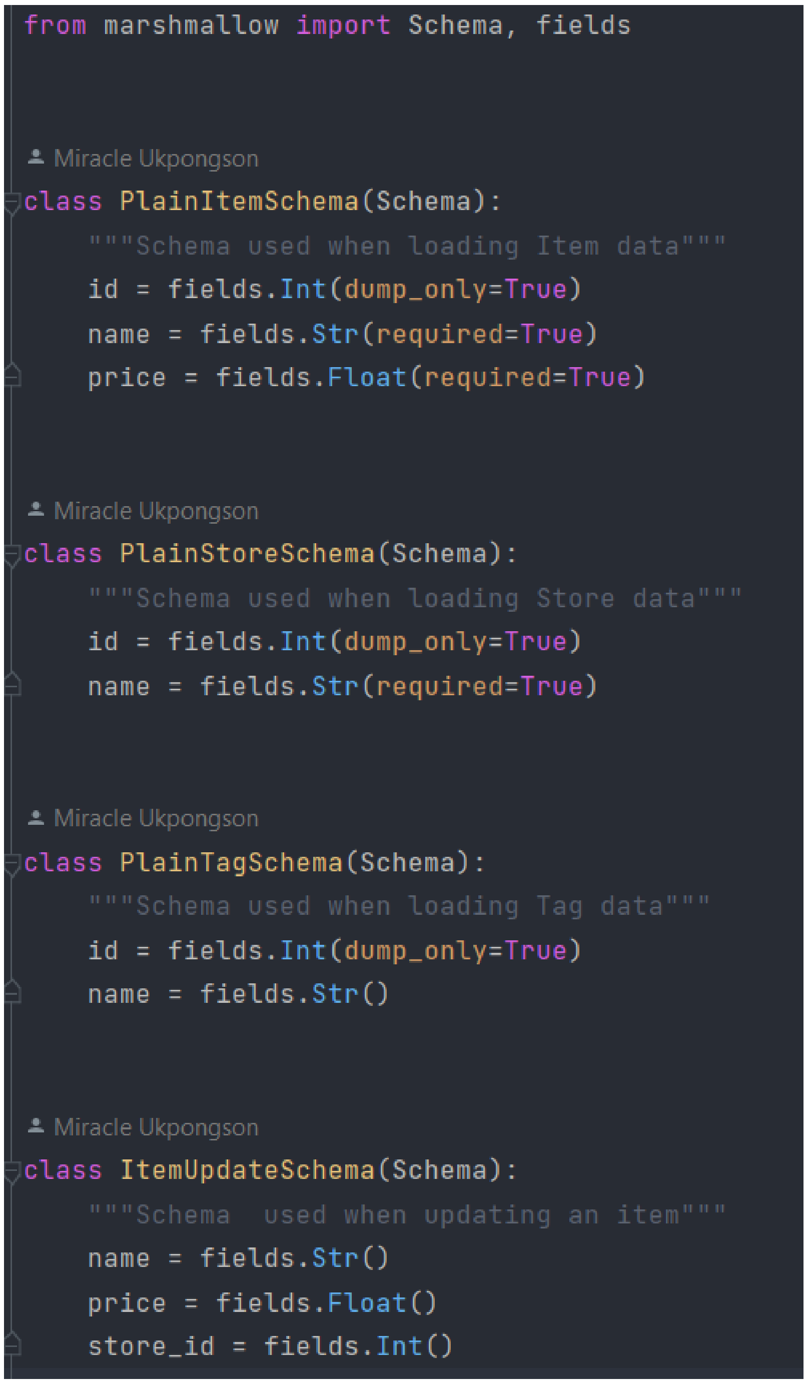
3. Database Configuration:
    - The application utilized a relational database management system (RDBMS) for data storage. During local testing, SQLite was used as the default database, however, for the production environment, a PostgreSQL database was utilized and configured along with access parameters which are loaded as environment

variables in the production environment. The PostgreSQL database was provided by the ElephantSQL [13] platform, which offered a convenient and scalable database hosting solution.

4. Database Migration:
    - To ensure the smooth transition of data from the SQLite database to the PostgreSQL database, database migration techniques were employed. The Flask-Migrate extension was used to generate migration scripts that captured the changes in the database schema. These migration scripts facilitated the transfer of the previously defined tables and their data from the SQLite database to the PostgreSQL database upon deployment.
5. Database Models:
    - The application employed an object-relational mapping (ORM) library called SQLAlchemy to define and interact with the database models. The models module/package contained the model definitions for the store, item, tag, and user entities. These models were mapped to their corresponding database tables in both SQLite and PostgreSQL databases. The migration scripts ensured that the database schemas in both databases remained consistent.
6. Schemas:
    - Schemas were an integral part of the REST API implementation, responsible for serializing and deserializing data between the API and the client. In this project, the Marshmallow library was utilized to define and manage the schemas. Schemas provide a structured representation of the data, enabling validation and transformation of incoming and outgoing data.
    - The project employed various schemas to handle different entities and data operations. These schemas include:
        a) ItemSchema: Used for dumping item data, including information such as the item ID, name, price, associated store, and tags.
        b) StoreSchema: Used for dumping store data, including information such as the store ID, name, and associated items and tags.
        c) TagSchema: Used for dumping tag data, including information such as the tag ID, name, associated store, and items.

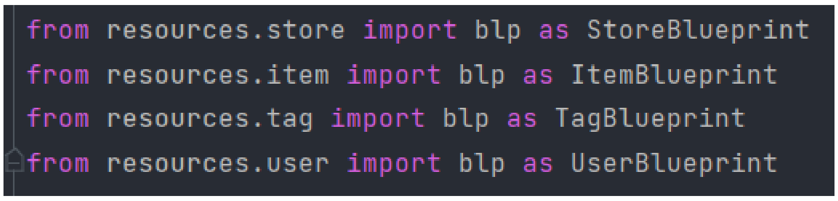
    d) UserSchema: Used for loading and dumping user data, including details like the user ID, username, and password (load-only).
    e) UserRegisterSchema: A specialized schema for user registration, extending the UserSchema to include additional fields such as the user's email.

- These schemas ensure that the data exchanged through the API follows a standardized format and meets the defined validation rules. They facilitate consistent and reliable communication between the client and server components of the system.

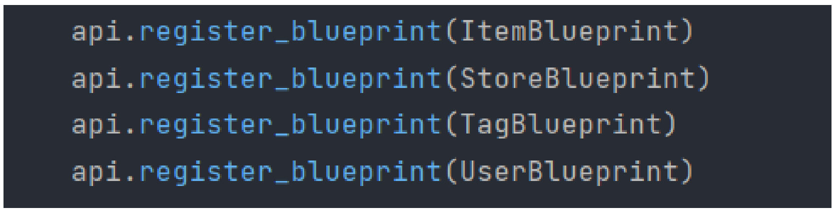


```python
from marshmallow import Schema, fields


Miracle Ukpongson
class PlainItemSchema(Schema):
    """Schema used when loading Item data"""
    id = fields.Int(dump_only=True)
    name = fields.Str(required=True)
    price = fields.Float(required=True)


Miracle Ukpongson
class PlainStoreSchema(Schema):
    """Schema used when loading Store data"""
    id = fields.Int(dump_only=True)
    name = fields.Str(required=True)


Miracle Ukpongson
class PlainTagSchema(Schema):
    """Schema used when loading Tag data"""
    id = fields.Int(dump_only=True)
    name = fields.Str()


Miracle Ukpongson
class ItemUpdateSchema(Schema):
    """Schema  used when updating an item"""
    name = fields.Str()
    price = fields.Float()
    store_id = fields.Int()
```

6. Blueprints and Resource Registration:
    - The application utilized Flask-Smorest to organize the REST API endpoints. Separate blueprints were created for different resources, including stores, items, tags, and users. Each blueprint defined the routes and associated controller functions for CRUD (Create, Read, Update, Delete) operations on the corresponding resource. The blueprints were then registered with the API instance to enable the handling of incoming HTTP requests.

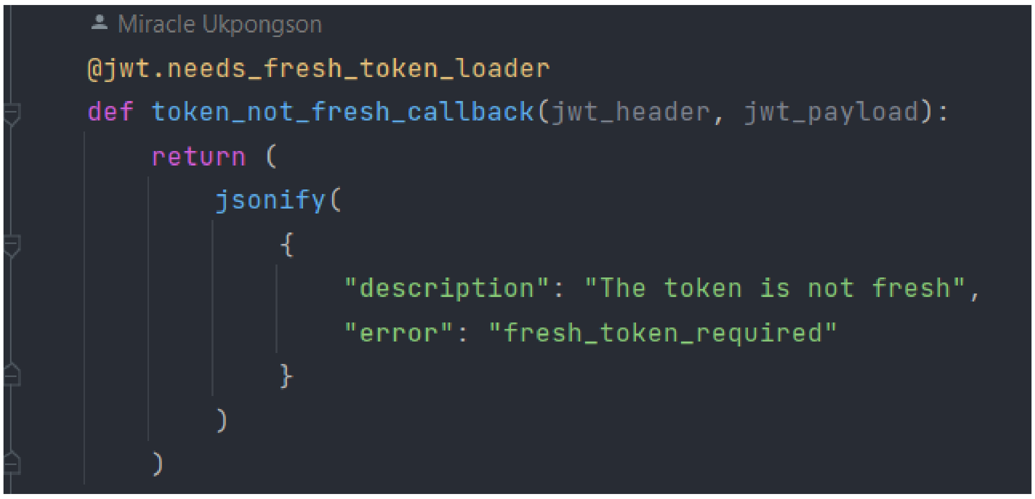


```python
from resources.store import blp as StoreBlueprint
from resources.item import blp as ItemBlueprint
from resources.tag import blp as TagBlueprint
from resources.user import blp as UserBlueprint
```



```python
    api.register_blueprint(ItemBlueprint)
    api.register_blueprint(StoreBlueprint)
    api.register_blueprint(TagBlueprint)
    api.register_blueprint(UserBlueprint)
```

7. Authentication and Authorization:
   - Authentication and authorization functionalities were implemented using Flask-JWT-Extended. The application used JSON Web Tokens (JWT) for user authentication. JWT-based authentication was implemented using the JWTManager instance, which provided various callback functions to handle token-related events such as token expiration, token revocation, and token validation. The callbacks were responsible for verifying the authenticity and validity of the JWT tokens.



```python
    Miracle Ukpongson
    @jwt.needs_fresh_token_loader
    def token_not_fresh_callback(jwt_header, jwt_payload):
        return (
            jsonify(
                {
                    "description": "The token is not fresh",
                    "error": "fresh_token_required"
                }
            )
        )
```

8. API Documentation:
   - The API was documented using OpenAPI specifications. The Swagger UI [11], hosted on a CDN, was integrated into the application to provide an interactive interface for exploring the API endpoints and testing them. The OpenAPI specifications were automatically generated based on the registered

blueprints and route definitions. The Swagger UI documentation can be accessed on: [REST API Swagger UI (store-rest-api-v2-project.onrender.com)](store-rest-api-v2-project.onrender.com)

9. Development and Testing with Postman:
   - During the development of the REST API, the Postman tool [12] was utilized extensively for testing and simulating client interactions. Postman provides a user-friendly interface for sending HTTP requests to API endpoints, allowing developers to validate the functionality and behaviour of the API.
   - By leveraging Postman, I was able to:
     a) Send various types of requests (GET, POST, PUT, DELETE) to different endpoints of the API.
     b) Test the API's response to different scenarios and input data.
     c) Validate the correctness of the API's output and error handling.
     d) Simulate client behaviour by setting request headers, query parameters, and request bodies.
   - Postman played a crucial role in the iterative development and debugging process, ensuring that the API endpoints were functioning as intended and providing the expected responses.

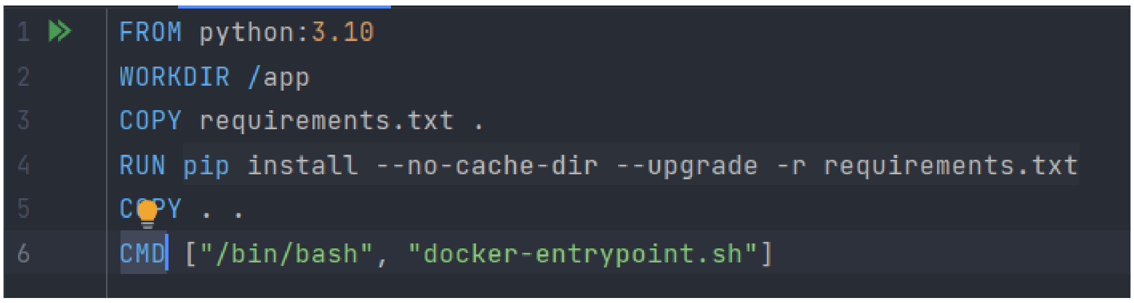
The resulting implementation provided a robust and secure REST API for managing stores and items. The Flask framework, along with Flask-Smorest and Flask-JWT-Extended extensions, facilitated the development of a scalable and efficient API. The use of PostgreSQL for the production environment, hosted on the ElephantSQL platform, ensured reliable and performant data storage.

3.3 Deployment the REST API

The REST API was deployed to the Render.com platform, which facilitated the seamless hosting and management of the application. The deployment process involved containerizing the application using Docker, ensuring portability and easy deployment across different environments.

To containerize the application, a Dockerfile was provided, which specified the necessary dependencies, configurations, and instructions for building the container image. The Dockerfile served as a blueprint for creating a self-contained environment for running the API.

Dockerfile:

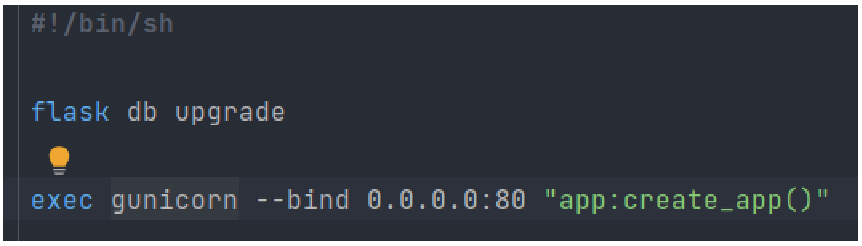


```
FROM python:3.10
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir --upgrade -r requirements.txt
COPY . .
CMD ["/bin/bash", "docker-entrypoint.sh"]
```

Render.com utilized the Dockerfile to automatically build and deploy the containerized application. The Dockerfile contained a shell script which provided additional instructions to be carried out whenever using the Dockerfile as the blueprint.

Shell script:

It simply instructs the platform to carry out necessary migrations changes if available and then create the infrastructure.

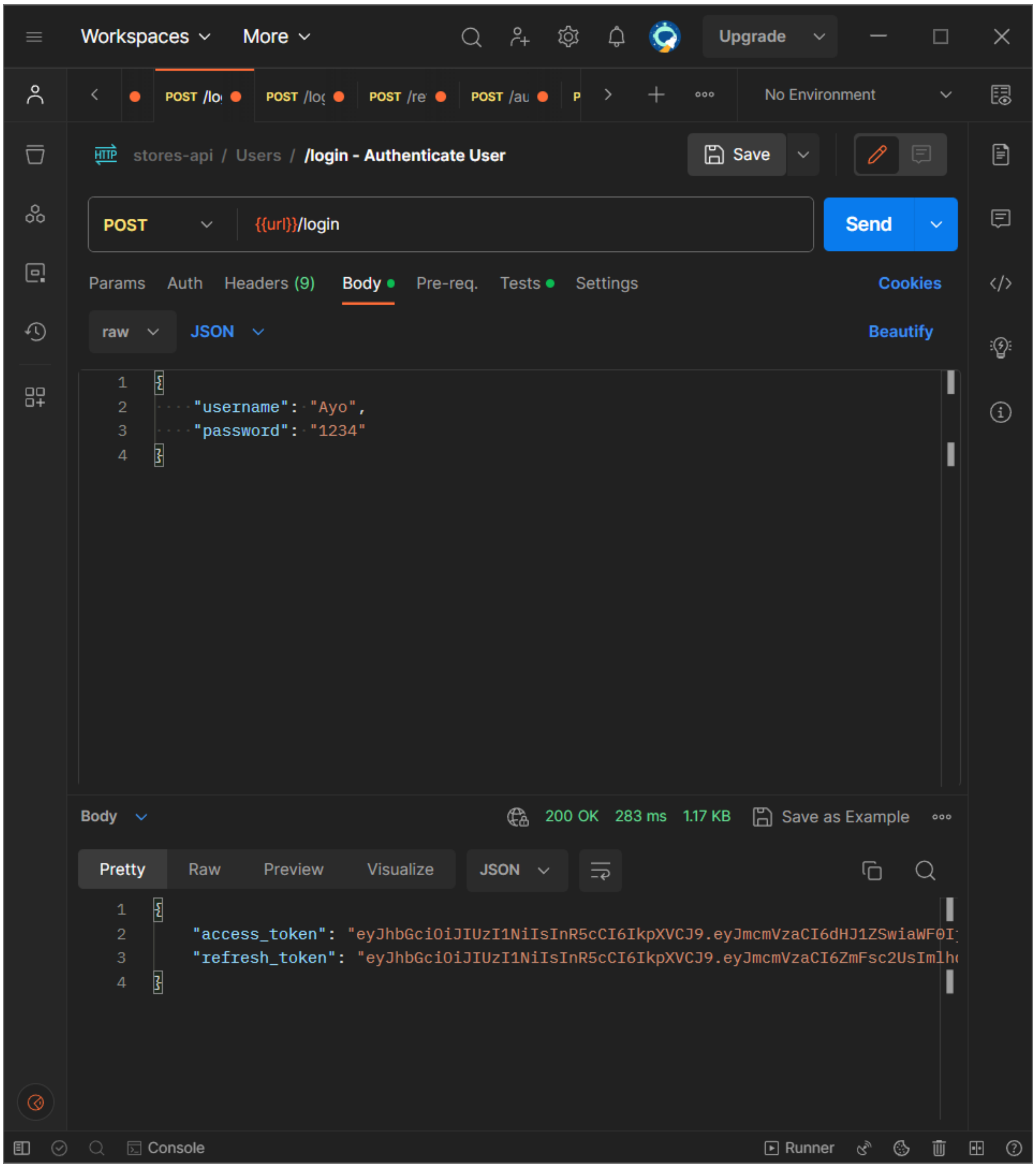


```
#!/bin/sh

flask db upgrade

exec gunicorn --bind 0.0.0.0:80 "app:create_app()"
```

This streamlined the deployment process and ensured consistency across different deployments. The platform leveraged the container image to provision and manage the necessary infrastructure to host the API.

Furthermore, the API relied on a PostgreSQL database to store and manage data. The production environment utilized the ElephantSQL platform to host the PostgreSQL database. During the deployment process, the previously defined tables in the SQLite database were seamlessly migrated to the PostgreSQL database, preserving the existing data and maintaining data integrity.

By leveraging Render.com and ElephantSQL, the deployment process was simplified, allowing for easy scaling, monitoring, and management of the deployed API. This ensured a reliable and robust hosting environment for the REST API, enabling seamless interaction with clients and efficient data storage.



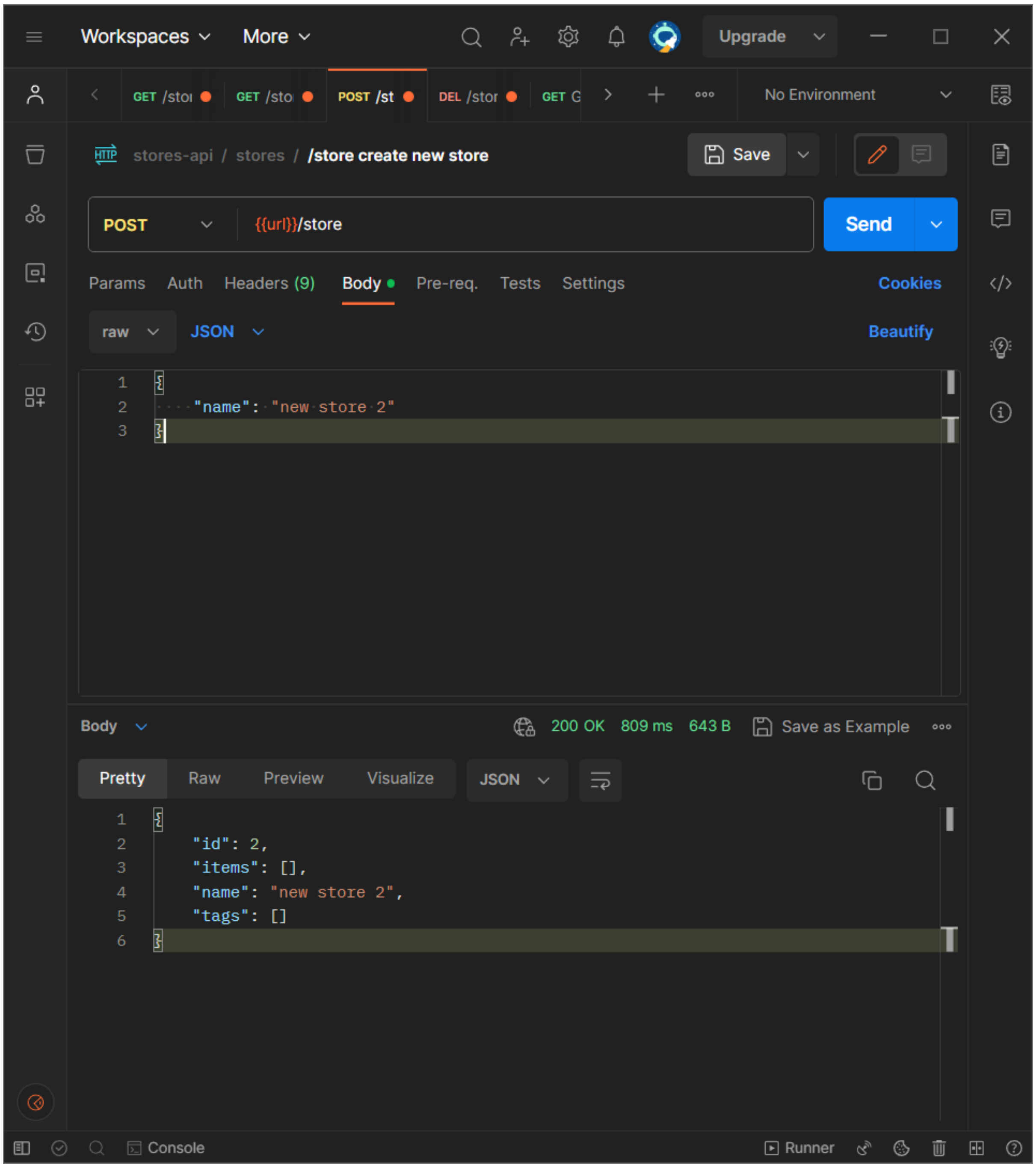
POST {{url}}/login

```
{
    "username": "Ayo",
    "password": "1234"
}
```

Body 200 OK 283 ms 1.17 KB

```
{
    "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJmcmVzaCI6dHJ1ZSwiaWF0I
    "refresh_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJmcmVzaCI6ZmFsc2UsImlh
}
```

With the returned "access_token" we can access all protected endpoints, the "referesh_token" is used to ensure that authentication flow is achieved so the user wont required reauthentication until the current session ends by a logout.



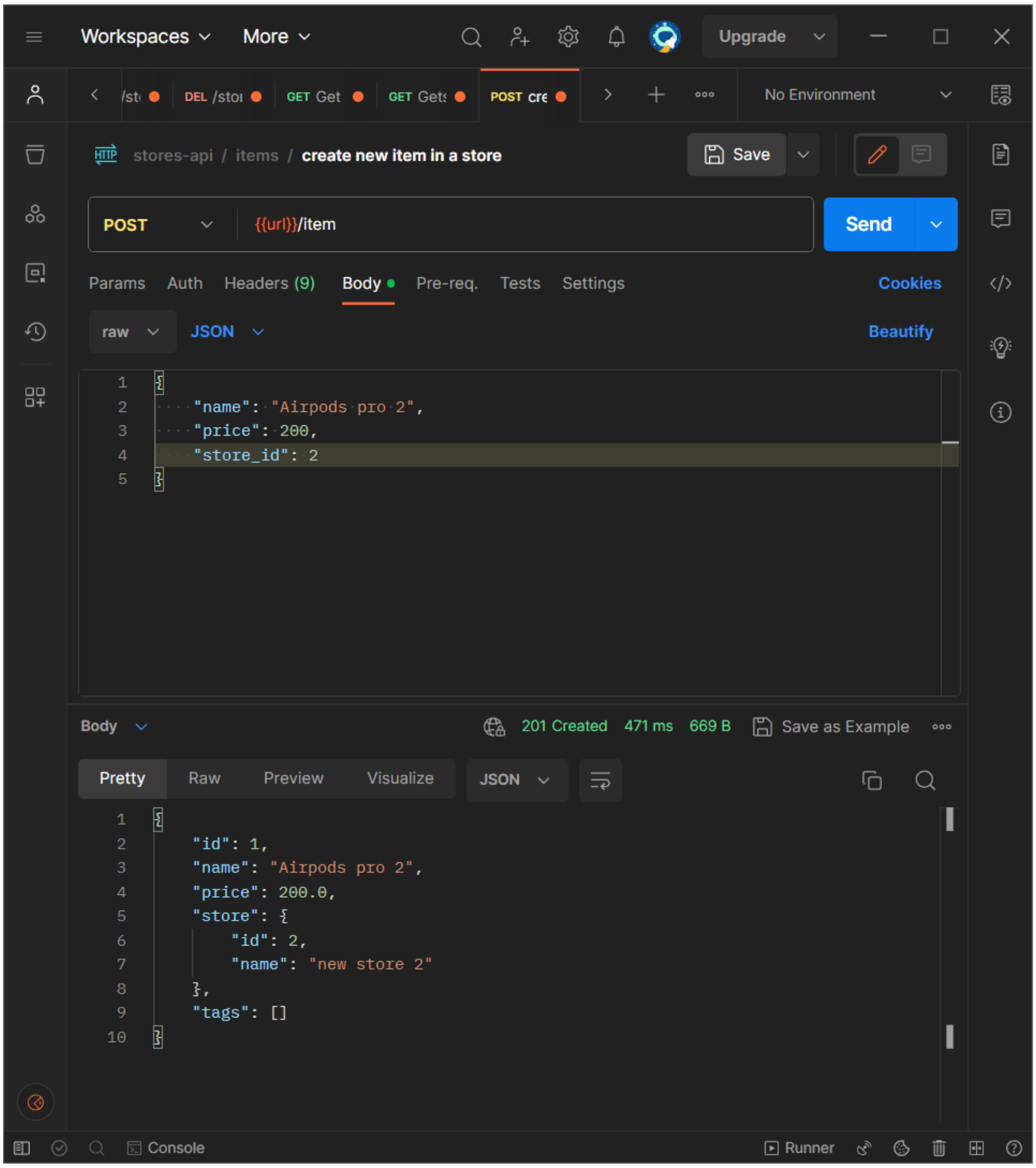
POST {{url}}/store

```
{
    "name": "new store 2"
}
```

200 OK 809 ms 643 B

```
{
    "id": 2,
    "items": [],
    "name": "new store 2",
    "tags": []
}
```

Here a post request is sent along with a payload containing the name of the store to the store endpoint, which creates a store and returns with a response body containing relevant information.

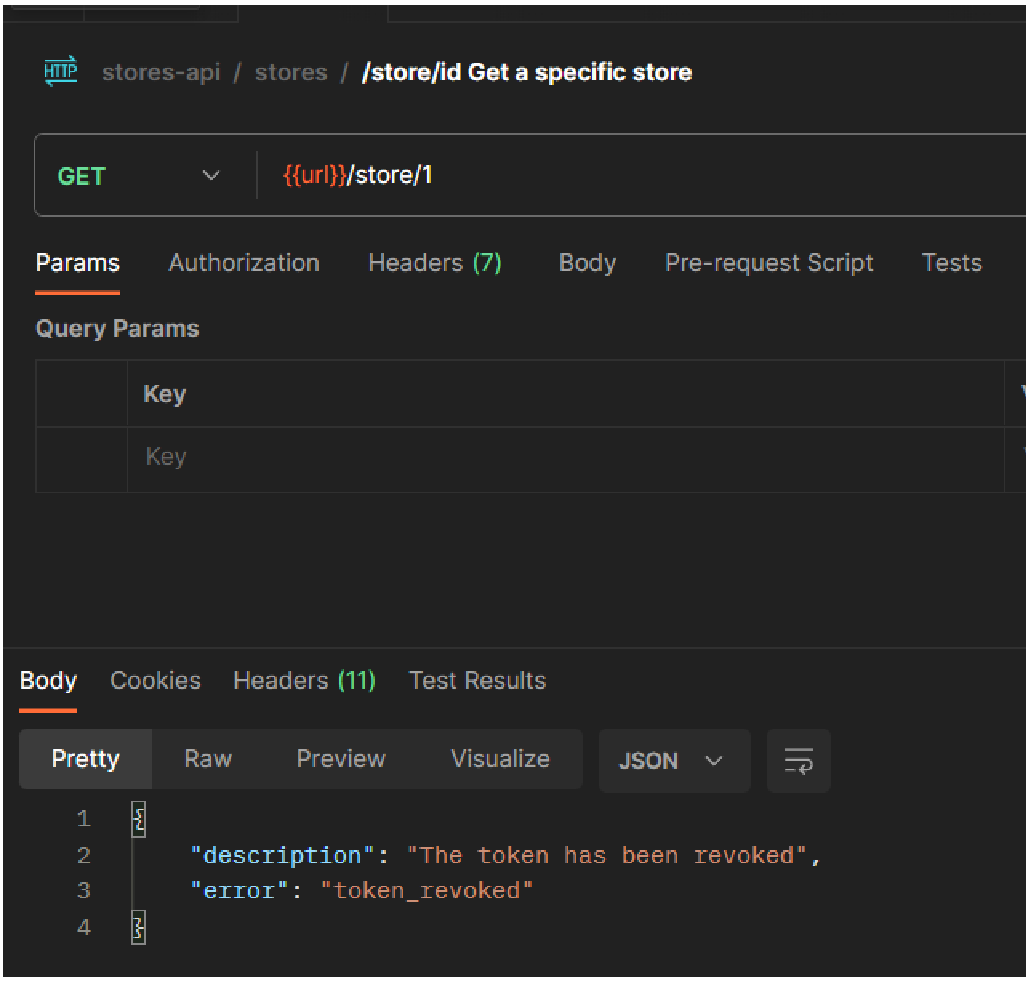


```
{
    "name": "Airpods pro 2",
    "price": 200,
    "store_id": 2
}
```

201 Created 471 ms 669 B

```
{
    "id": 1,
    "name": "Airpods pro 2",
    "price": 200.0,
    "store": {
        "id": 2,
        "name": "new store 2"
    },
    "tags": []
}
```

Here a post request is sent along with a payload containing the name, price and store ID(of the previously created store) to the items endpoint, which creates the item in the store and returns with a response body containing relevant information.



```json
{
    "description": "The token has been revoked",
    "error": "token_revoked"
}
```

Here is the response from the items endpoint when trying to access it after logging out. This simply confirms that authentication flow ends when logout has been made.

CONCLUSION

In conclusion, the design and implementation of the REST API using Flask for store and item management present a robust and practical solution for businesses operating in the e-commerce industry. The API offers a wide range of essential functionalities, scalability, security, and integration potential, effectively addressing the challenges associated with store and item management, enhancing inventory control, and elevating the overall user experience.

The effectiveness and performance of the API have been thoroughly validated through comprehensive testing and evaluation. The use of Postman during the implementation phase allowed for rigorous endpoint testing and simulated client interactions, ensuring the API's functionality and behavior met the intended requirements.

The deployment of the API to the Render.com platform further enhanced its accessibility and availability. Leveraging Docker for containerization facilitated seamless deployment across various environments, ensuring consistency and reliability. The integration with the PostgreSQL database provided by ElephantSQL [13] for the production environment ensured efficient data storage and management.

It is worth noting that the API's versatility and extensibility make it a suitable foundation for future enhancements. Additional functionalities, such as a front-end for user reviews or integration with third-party services, could further extend the capabilities of the API and address evolving business needs.

By providing a well-designed and implemented REST API solution, this study contributes to the field of store management systems. The API empowers businesses to streamline their store operations, improve inventory control, and enhance overall operational efficiency. This project also demonstrates the successful application of Python, Flask, Flask-Smorest, Flask-JWT-Extended, and database technologies like SQLite and PostgreSQL in developing a reliable and scalable REST API solution.

In conclusion, the designed and implemented REST API solution serves as a valuable tool for businesses in the e-commerce industry, enabling efficient store and item management, and paving the way for future advancements in store management systems.

REFERENCES

1. Shopify API reference documentation:
   https://shopify.dev/docs/api
2. Python Documentation:
   https://www.python.org/doc/
3. Flask documentation and user guide:
   Flask Documentation (2.3.x) (palletsprojects.com)
4. IBM cloud:
   What is a REST API? | IBM
5. Interacting With Web Services – Real Python:
   https://realpython.com/api-integration-in-python/
6. Flask-JWT-Extended's Documentation:
   https://flask-jwt-extended.readthedocs.io/en/stable/
7. Flask-Smorest Documentation:
   https://flask-smorest.readthedocs.io/en/latest/
8. WooCommerse REST API reference documentation:
   https://woocommerce.github.io/woocommerce-rest-api-docs/ - introduction
9. Square API Reference:
   https://developer.squareup.com/reference/square
10. HATEOAS – Wikipedia:
    https://en.wikipedia.org/wiki/HATEOAS
11. Swagger-ui CDN and documentation by jsDelivrA:
    https://www.jsdelivr.com/package/npm/swagger-ui-dist
12. Postman API Platform and User guide:
    https://www.postman.com/home
13. ElephantSQL documentation:
    https://www.elephantsql.com/docs/index.html

4. Render.com platform and documentation:
   https://render.com/docs

ACKNOWLEDGMENT

I would like to express my heartfelt gratitude to all those who have contributed to the successful completion of this thesis.

I am sincerely thankful to my supervisor, Miss Galyna for her guidance, support, and valuable insights throughout the entire duration of this research. Their expertise in the field of Computer Science and their dedication to my academic growth has been instrumental in shaping this thesis.

I would like to extend my appreciation to the faculty members of the Computer Science Department at Sumy State University. Their commitment to excellence in education and their passion for imparting knowledge has played a significant role in my intellectual development and the successful completion of this research project.

I am deeply indebted to the participants who willingly participated in the data collection process. Their cooperation and willingness to share their experiences have provided valuable insights and enriched the findings of this study.

I am grateful to my family and friends for their unwavering support, encouragement, and belief in my abilities. Their constant motivation and understanding have been essential in overcoming challenges and staying focused on this academic pursuit.

I would like to express my gratitude to Sumy State University for providing an enriching academic environment and the necessary resources for conducting this research. The opportunities for learning and growth that I have received at this institution have been invaluable.

I would also like to acknowledge the contributions of the open-source community, whose continuous efforts in developing and improving software tools and frameworks have greatly facilitated the implementation and success of this Study. Specifically, I would like to express my gratitude to the contributors of Python, Flask, Flask-Smorest, and Flask-JWT-Extended, which formed the foundation of the development process.

In conclusion, I am deeply grateful to all individuals and institutions that have played a part in this academic journey. Your support, guidance, and encouragement have been invaluable, and I am truly honored to have had the opportunity to work under your mentorship and support.

APPENDIX

The appendix section contains supplementary information and resources that complement the main body of the thesis. These additional materials provide further details, documentation, and supporting evidence for the design, implementation, and evaluation of the REST API for store and item management.

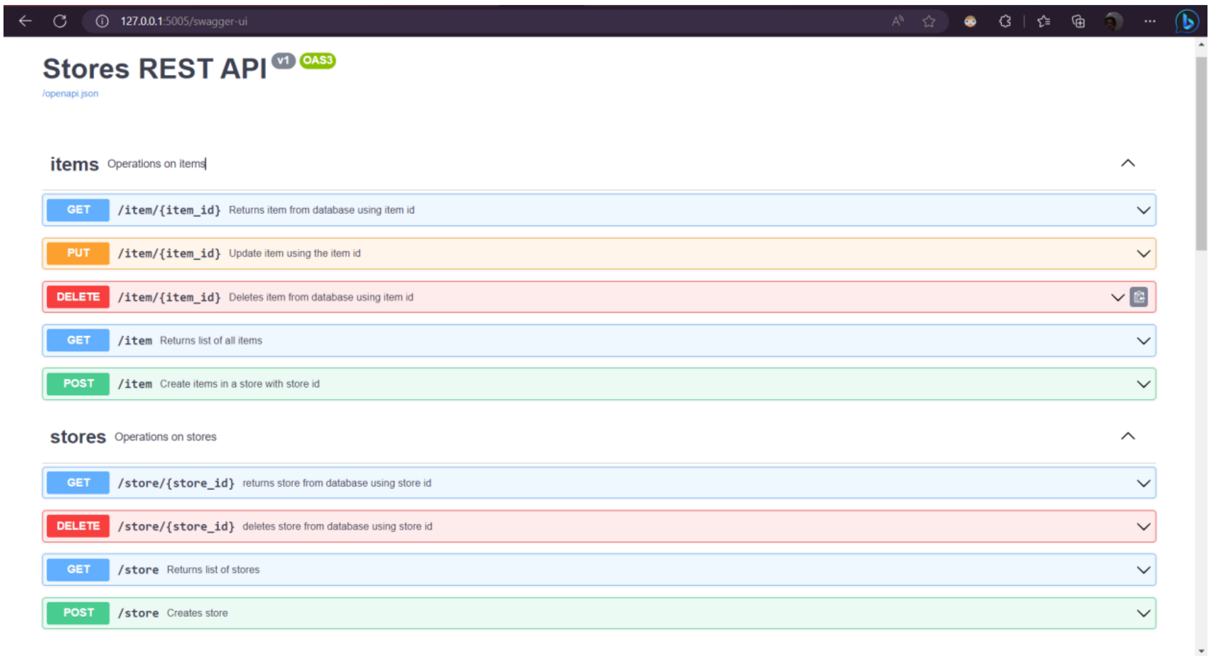
Endpoint of live API: Store REST API endpoint (store-rest-api-v2-project.onrender.com)

GitHub repo for API: pRimeRly/store-rest-api-v2 (github.com)

Swagger UI for API: REST API Swagger UI (store-rest-api-v2-project.onrender.com)

Using the API endpoint and Postman tool for simulating client interactions, the API can be interacted with, additionally, the Swagger UI of the API can be used for the same purpose.

API DOCUMENTATION:



127.0.0.1:5005/swagger-ui

Stores REST API v1 OAS3

/openapi.json

items Operations on items

GET `/item/{item_id}` Returns item from database using item id

PUT `/item/{item_id}` Update item using the item id

DELETE `/item/{item_id}` Deletes item from database using item id

GET `/item` Returns list of all items

POST `/item` Create items in a store with store id

stores Operations on stores

GET `/store/{store_id}` returns store from database using store id

DELETE `/store/{store_id}` deletes store from database using store id

GET `/store` Returns list of stores

POST `/store` Creates store

Code Snippets: This shows some endpoints defined during API Implementation.

Item resource:

```
from flask.views import MethodView
from flask_smorest import Blueprint, abort
from flask_jwt_extended import jwt_required, get_jwt
from sqlalchemy.exc import SQLAlchemyError, IntegrityError

from db import db
from models import ItemModel
from schemas import ItemSchema, ItemUpdateSchema

blp = Blueprint("items", __name__, description="Operations on items")


@blp.route("/item/<int:item_id>")
class Item(MethodView):
    @jwt_required()
    @blp.response(200, ItemSchema)
    def get(self, item_id):
        """Returns item from database using item id"""
        item = ItemModel.query.get_or_404(item_id)
        return item

    @jwt_required(fresh=True)
    def delete(self, item_id):
        """Deletes item from database using item id"""
        jwt = get_jwt()
        if not jwt.get("is_admin"):
            abort(401, message="Admin privilege required")
        item = ItemModel.query.get_or_404(item_id)
        db.session.delete(item)
        db.session.commit()
        return {"message": "Item deleted"}

    @jwt_required()
    @blp.arguments(ItemUpdateSchema)
    @blp.response(200, ItemSchema)
    def put(self, item_data, item_id):
        """Update item using the item id"""
        item = ItemModel.query.get(item_id)
        if item:
            item.price = item_data["price"]
            item.name = item_data["name"]
        else:
            try:
                item = ItemModel(id=item_id, **item_data)
            except IntegrityError:
                abort(400, message="An Item with that name already
exists.")

        db.session.add(item)
        db.session.commit()
        return item


@blp.route("/item")
class ItemList(MethodView):
    @blp.response(200, ItemSchema(many=True))
    def get(self):
        """Returns list of all items"""
        return ItemModel.query.all()

    @jwt_required()
    @blp.arguments(ItemSchema)
    @blp.response(201, ItemSchema)
    def post(self, item_data):
        """Create items in a store with store id"""
        item = ItemModel(**item_data)

        try:
            db.session.add(item)
            db.session.commit()
        except SQLAlchemyError:
            abort(500, message="An error occurred while inserting the
item.")
        return item
```

Store resource:

```
from flask.views import MethodView
from flask_smorest import Blueprint, abort

from db import db
from models import StoreModel
from schemas import StoreSchema
from sqlalchemy.exc import SQLAlchemyError, IntegrityError
from flask_jwt_extended import jwt_required

blp = Blueprint("stores", __name__, description="Operations on stores")


@blp.route("/store/<int:store_id>")
class Store(MethodView):
    @jwt_required()
    @blp.response(200, StoreSchema)
    def get(self, store_id):
        """returns store from database using store id"""
        store = StoreModel.query.get_or_404(store_id)
        return store

    @jwt_required()
    def delete(self, store_id):
        """deletes store from database using store id"""
        store = StoreModel.query.get_or_404(store_id)
        db.session.delete(store)
        db.session.commit()
        return {"message": "Store deleted"}


@blp.route("/store")
class StoreList(MethodView):
    @blp.response(200, StoreSchema(many=True))
    def get(self):
        """Returns list of stores"""
        return StoreModel.query.all()

    @jwt_required()
    @blp.arguments(StoreSchema)
    @blp.response(200, StoreSchema)
    def post(self, store_data):
        """Creates store"""
        store = StoreModel(**store_data)

        try:
            db.session.add(store)
            db.session.commit()
        except IntegrityError:
            abort(400, message="A store with that name already exists.")
        except SQLAlchemyError:
            abort(500, message="An error occurred while creating the
store.")
        return store
```

Database Models: The database Models used in the store and item management system, providing an overview of the tables, their relationships, and the attributes associated with each entity.

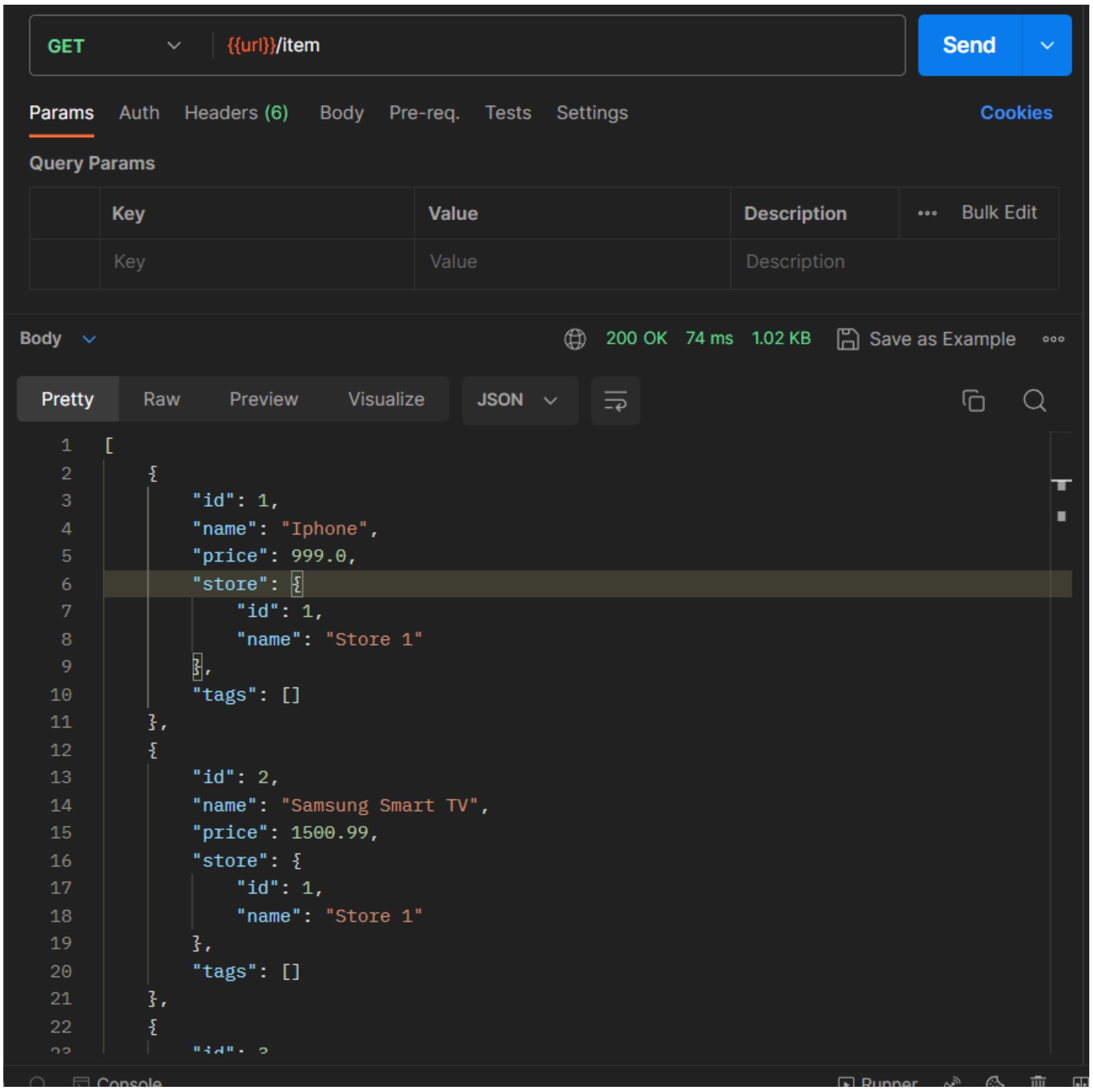
Item Models:

```
"""Model class (ItemModel) for the items table"""
from db import db


class ItemModel(db.Model):
    __tablename__ = "items"

    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(80), unique=True, nullable=False)
    description = db.Column(db.String)
    price = db.Column(db.Float(precision=2), unique=False, nullable=False)
    store_id = db.Column(db.Integer, db.ForeignKey("stores.id"),
unique=False, nullable=False)  # store can have multiple items
    store = db.relationship("StoreModel", back_populates="items")  #
StoreModel object associated with item

    tags = db.relationship("TagModel", back_populates="items",
secondary="items_tags")  # instructs sqlalchemy how to map tags to items
```

Store Model:

```
"""Model class (StoreModel) for the stores table"""
from db import db


class StoreModel(db.Model):
    __tablename__ = "stores"

    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(80), unique=True, nullable=False)
    items = db.relationship("ItemModel", back_populates="store",
lazy="dynamic", cascade="all, delete")
    tags = db.relationship("TagModel", back_populates="store",
lazy="dynamic")
```

API Interaction:



GET {{url}}/item

Send

Params Auth Headers (6) Body Pre-req. Tests Settings Cookies

Query Params

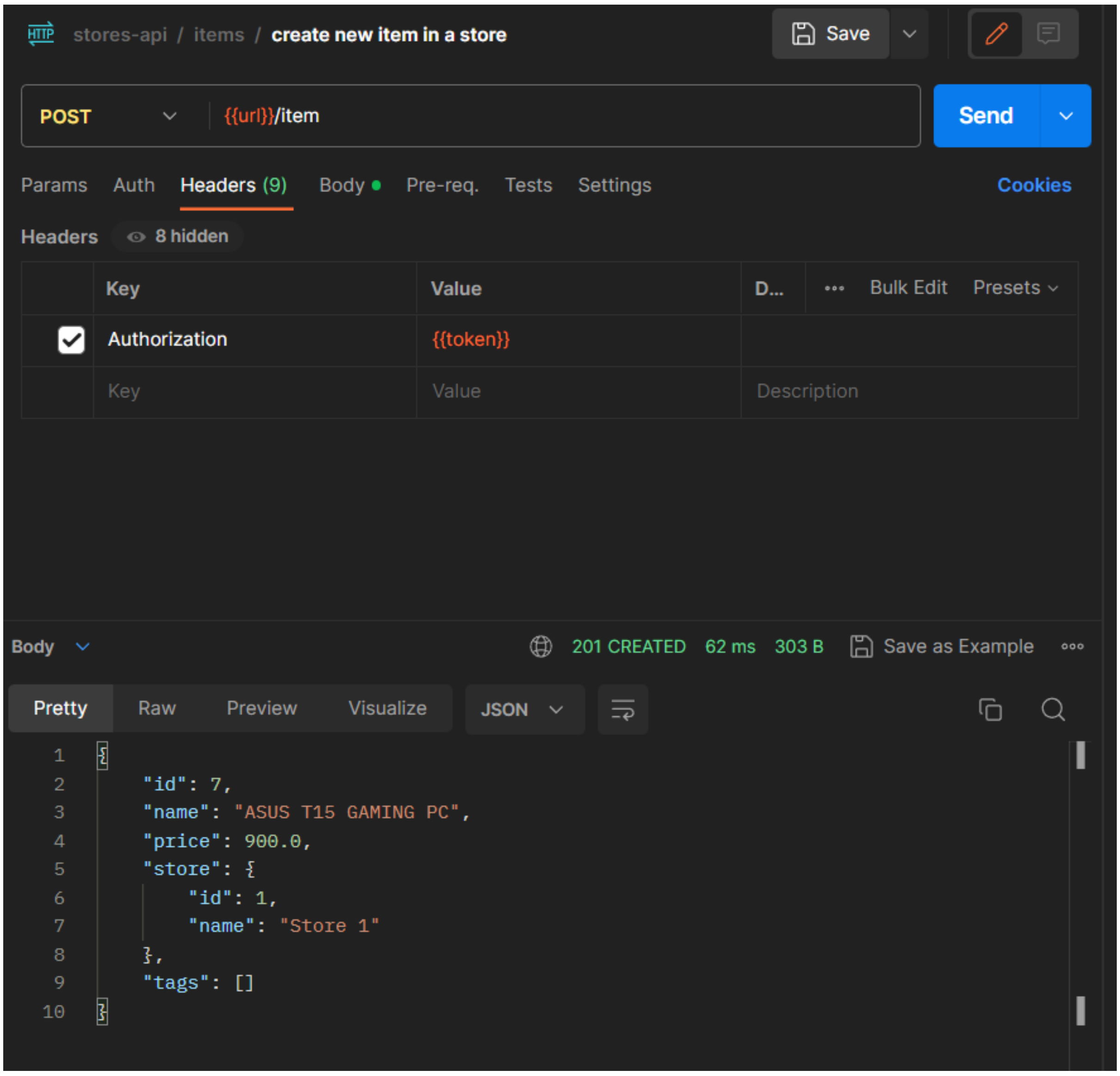
Key	Value	Description
Key	Value	Description

Body 200 OK 74 ms 1.02 KB Save as Example

Pretty Raw Preview Visualize JSON

```
[
    {
        "id": 1,
        "name": "Iphone",
        "price": 999.0,
        "store": {
            "id": 1,
            "name": "Store 1"
        },
        "tags": []
    },
    {
        "id": 2,
        "name": "Samsung Smart TV",
        "price": 1500.99,
        "store": {
            "id": 1,
            "name": "Store 1"
        },
        "tags": []
    },
    {
```

Console Runner



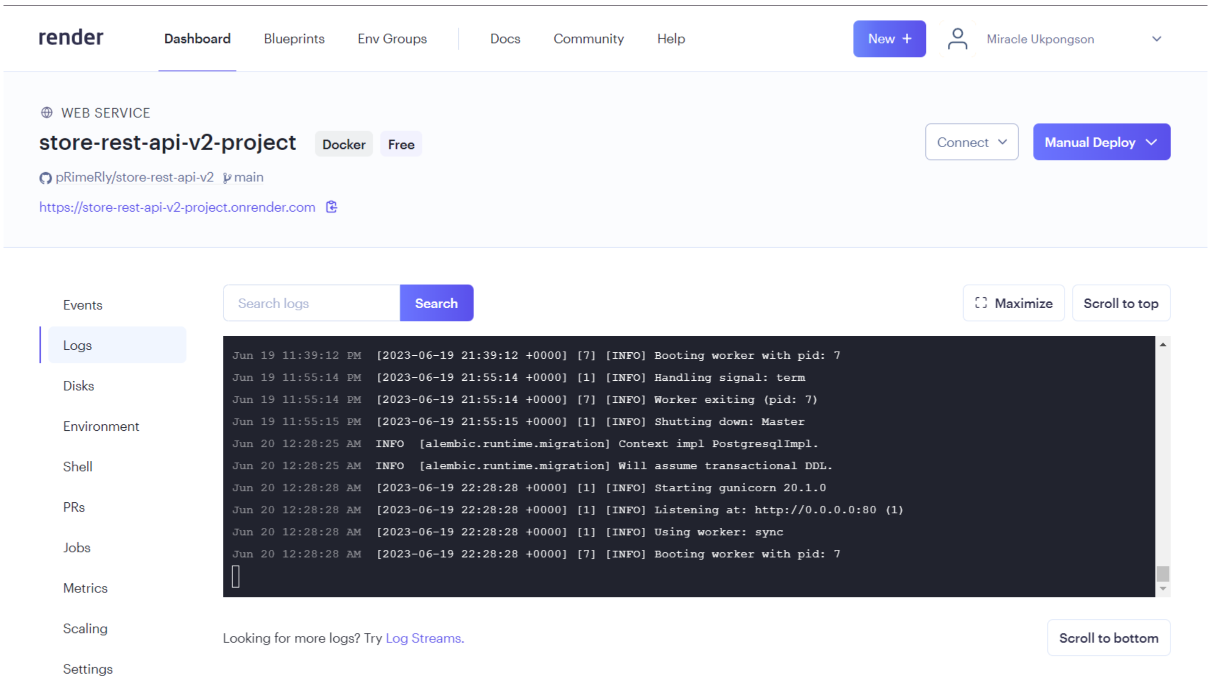
{{url}}/item

	Key	Value	D...
✓	Authorization	{{token}}	
	Key	Value	Description

201 CREATED 62 ms 303 B

```
{
    "id": 7,
    "name": "ASUS T15 GAMING PC",
    "price": 900.0,
    "store": {
        "id": 1,
        "name": "Store 1"
    },
    "tags": []
}
```

Deployment Environment:



```
Jun 19 11:39:12 PM  [2023-06-19 21:39:12 +0000] [7] [INFO] Booting worker with pid: 7
Jun 19 11:55:14 PM  [2023-06-19 21:55:14 +0000] [1] [INFO] Handling signal: term
Jun 19 11:55:14 PM  [2023-06-19 21:55:14 +0000] [7] [INFO] Worker exiting (pid: 7)
Jun 19 11:55:15 PM  [2023-06-19 21:55:15 +0000] [1] [INFO] Shutting down: Master
Jun 20 12:28:25 AM  INFO  [alembic.runtime.migration] Context impl PostgresqlImpl.
Jun 20 12:28:25 AM  INFO  [alembic.runtime.migration] Will assume transactional DDL.
Jun 20 12:28:28 AM  [2023-06-19 22:28:28 +0000] [1] [INFO] Starting gunicorn 20.1.0
Jun 20 12:28:28 AM  [2023-06-19 22:28:28 +0000] [1] [INFO] Listening at: http://0.0.0.0:80 (1)
Jun 20 12:28:28 AM  [2023-06-19 22:28:28 +0000] [1] [INFO] Using worker: sync
Jun 20 12:28:28 AM  [2023-06-19 22:28:28 +0000] [7] [INFO] Booting worker with pid: 7
```

Looking for more logs? Try Log Streams.