Personal Report on Secure API Gateway Project

Introduction

For this project, I developed a **Secure API Gateway** using Flask. The goal was to create a backend system that supports user registration, login with JWT authentication, role-based access control, and security measures like rate limiting and IP blocking.

Challenges Faced

1. Module Import Errors

Initially, I struggled with importing modules across multiple files (`models_module.py`, `auth.py`, `config.py`) which caused frequent `ImportError`s. This was mainly due to circular imports and incorrect file/module naming.

How I solved it:
I consolidated the entire code into a single file (`app.py`) to simplify dependencies and avoid import conflicts.

---

2. Flask Decorator Usage

I encountered an `AttributeError` related to `before_first_request` not being recognized. I learned that the correct usage is with `@app.before_first_request` but also considered alternative approaches like `@app.before_request` or running initialization code before the app starts.

---

3. Token Handling and Authentication

Implementing JWT authentication with token generation, expiration, and role checking required careful attention to header formatting (e.g., parsing the `Authorization` header correctly).

How I solved it:
I standardized the header to accept `Bearer <token>` and wrapped authentication logic in decorators for easy reuse.

---

4. Rate Limiting and IP Blocking

Integrating Redis for rate limiting and blocking IPs after failed login attempts was new to me. Setting up Redis and ensuring correct increment and expiration of keys took some trial and error.

How I solved it:
Used Redis commands like `incr`, `expire`, `sadd`, and `sismember` to track attempts and block IPs temporarily. Also, designed a rate limiter decorator that counts requests per user.

Testing and Tools Used

I tested the API mainly with **Postman**, crafting requests to:

- Register new users with different roles.
- Log in and receive JWT tokens.
- Access protected endpoints with and without tokens.
- Verify role-based restrictions.
- Trigger rate limits by rapid requests.

I also used **curl** commands for quick checks and debugging.

Learning Outcome

Through this project, I deepened my understanding of:

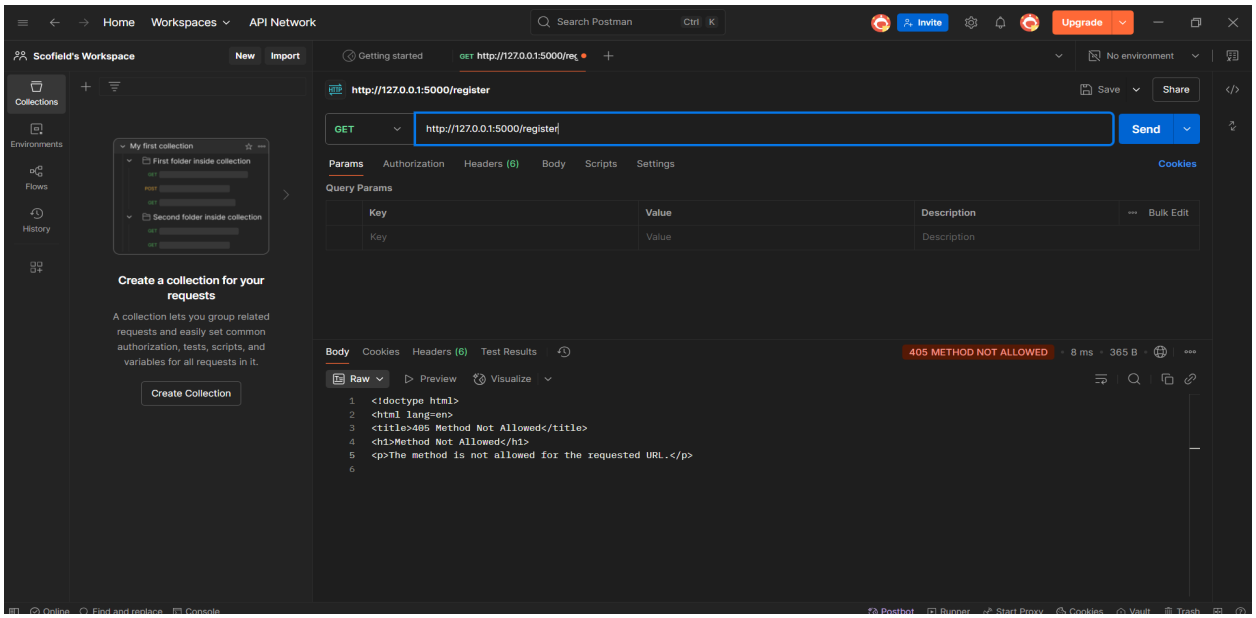
- Flask application structure and decorators.

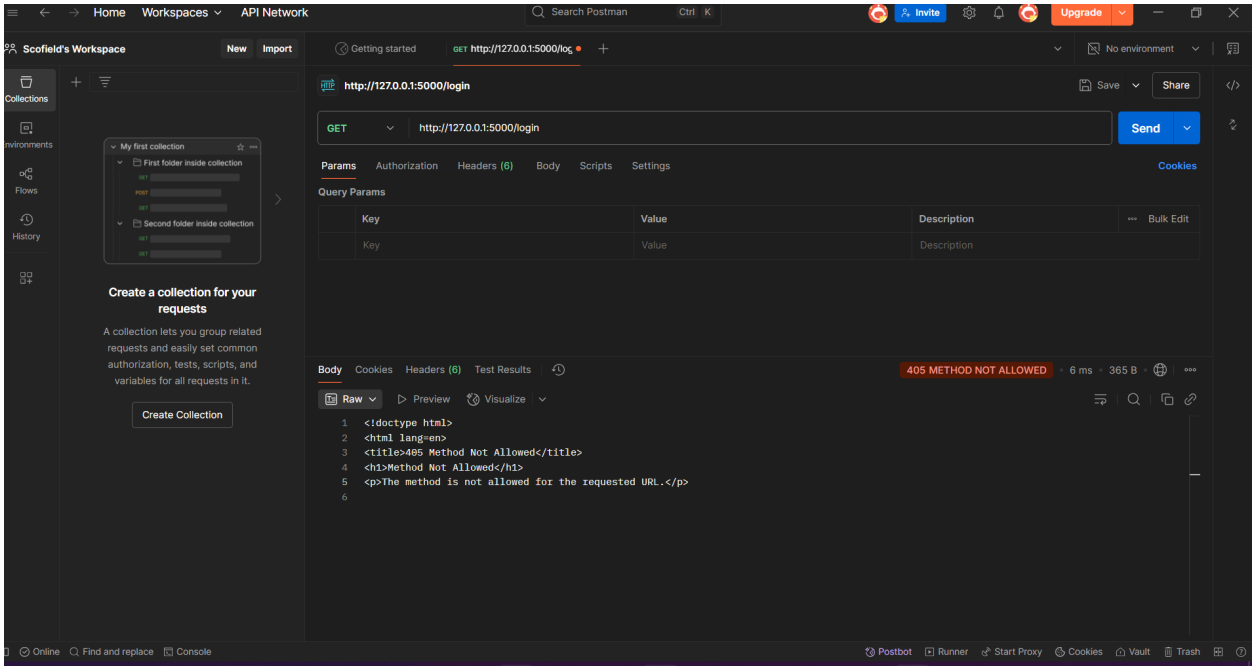
- JWT token-based authentication flows.
- Rate limiting and security best practices.
- Redis integration for stateful rate limiting.
- Debugging import and module issues in Python.

Future Improvements

- Add password reset and email verification flows.
- Enhance rate limiting with dynamic limits per endpoint.
- Integrate logging and monitoring.
- Dockerize the app with Redis for easier deployment.

Photographic Evidence





Example:

- User registration request and response
- Login request and returned JWT token
- Access denied due to insufficient role
- Rate limit exceeded error response