Service Oriented Architecture Project

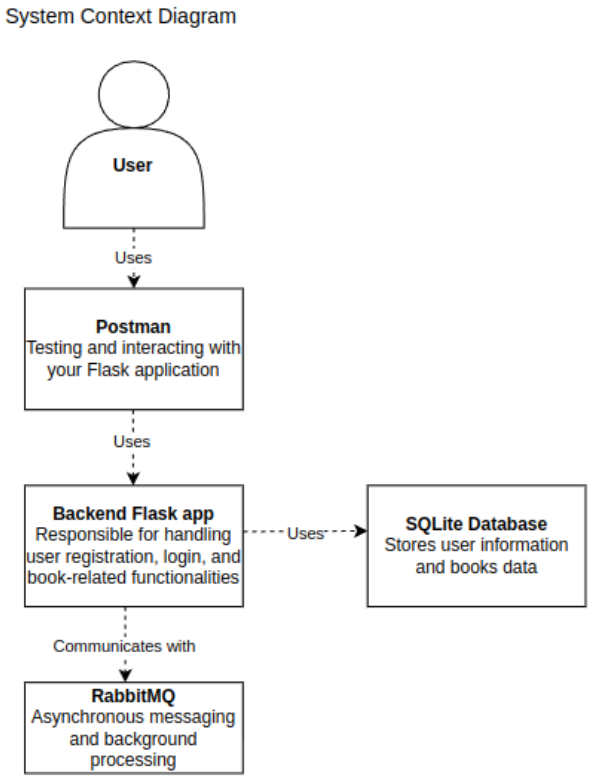
1. Description:

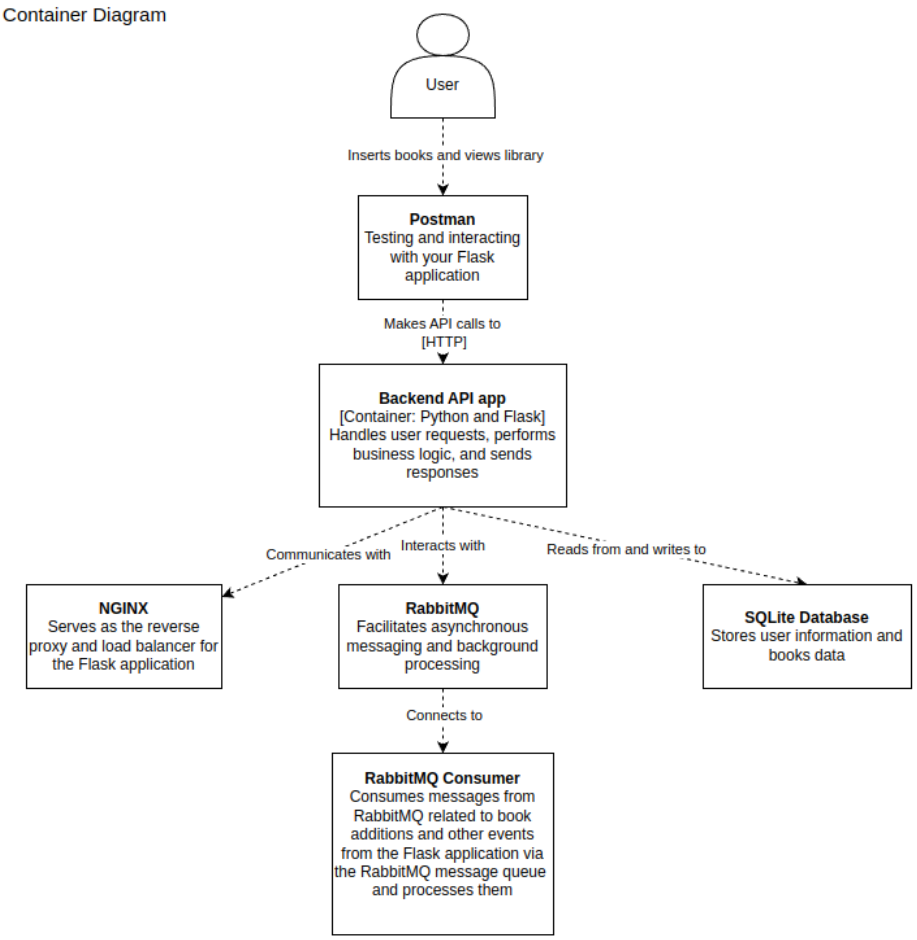
The project consists of an application that allows users to register into a digital reading list. There, they can insert new books with authors and titles, or they can visualize all the books they have read.

Components:

- Postman: used as a tool to test and interact with the backend API
- Python and Flask microframework: used as the foundation for building the REST API to manage routing, request handling, the response generation and integration with SQL Alchemy
- SQLite Database: to store information about users and books
- JWT (JSON Web Tokens) : used for user authentication and authorization
- Nginx: as a high-performance web server and reverse proxy server, it was used as a reverse proxy server in front of the Flask application to handle incoming HTTP requests and load balancing
- Gunicorn: is a Python WSGI HTTP server for UNIX-like operating systems and it was used as the production web server to serve the Flask application.
- RabbitMQ: as a message broker, it was used for asynchronous communication, message queuing and background processing
- Docker: containerization and deployment of the project

2. Diagrams:





3. Tutorial on securing a REST API:

- Installing Flask and the required dependencies:

```
from flask import Flask, request, jsonify, make_response
from flask_sqlalchemy import SQLAlchemy
from werkzeug.security import generate_password_hash, check_password_hash
import uuid
import jwt
import datetime
import pika
from functools import wraps
```

- Initialising Flask app and its configuration:

```
app = Flask(__name__)

app.config['SECRET_KEY'] = 'thisisthesecretkey'
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///site.db'
db = SQLAlchemy(app)
```

The secret key app.config[‘SECRET_KEY’] is used to sign JWT (JSON Web Token) tokens generated during user authentication.

- Defining database models:
  In the case of this project, I created User and Books models using SQL Alchemy.
  The user information in the form of username and password will be required for registration and login.

```
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    public_id = db.Column(db.String(50), unique=True)
    username = db.Column(db.String(50), unique=True)
    password = db.Column(db.String(80))

class Book(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    author = db.Column(db.String(50))
    title = db.Column(db.String(100))
```

- Implementing user authentication:

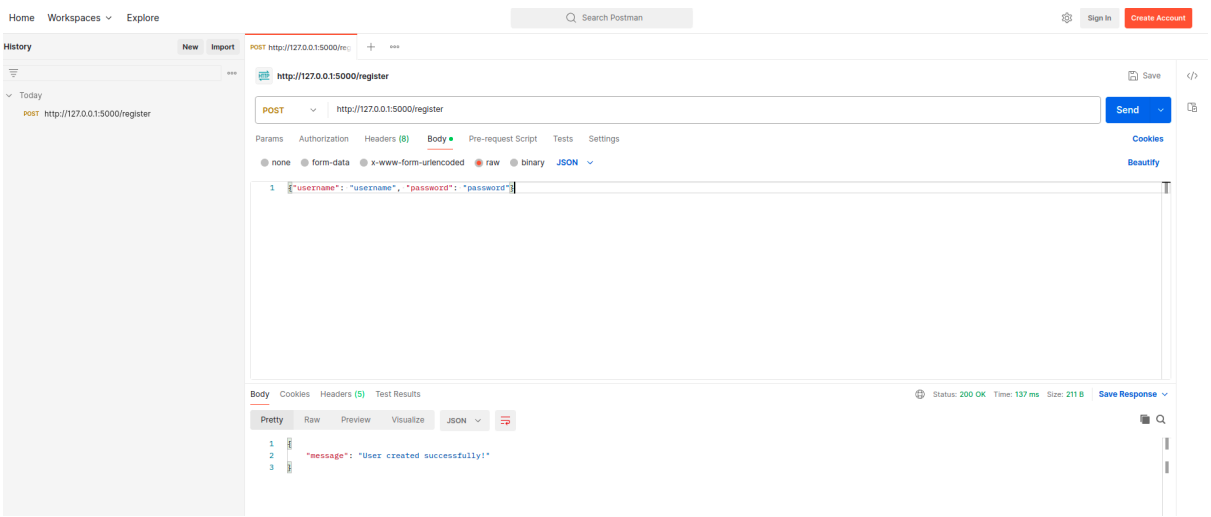
  ➢ User Registration and Login Endpoints: Implement endpoints for user registration and login. These endpoints handle HTTP requests from clients to register new users and authenticate existing users.
  ➢ Password Hashing: Hash user passwords using the generate_password_hash function from werkzeug.security.
  ➢ Token Generation: Generate JWT tokens upon successful login using the jwt.encode() function. Tokens contain user information and expiration time.
  ➢ Token Verification: Implement a token required decorator token_required to protect routes that require authentication.

```
def token_required(f):
    @wraps(f)
    def decorated(*args, **kwargs):

        token = None

        if 'x-access-tokens' in request.headers:
            token = request.headers['x-access-tokens']

        if not token:
            return jsonify({'message': 'Token is missing'}), 403
        try:
            data = jwt.decode(token, app.config['SECRET_KEY'], algorithms=["HS256"])
            current_user = User.query.filter_by(public_id=data['public_id']).first()
        except:
            return jsonify({'message': 'Token is invalid!'}), 401

        return f(current_user, *args, **kwargs)

    return decorated
```

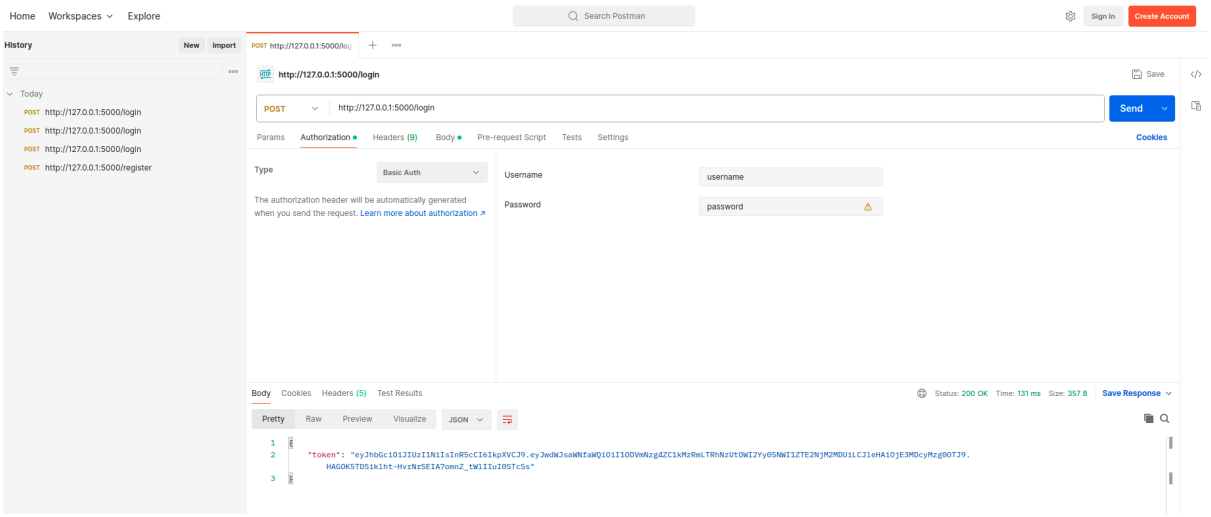
```python
@app.route('/protected_authors', methods=['POST'])
@token_required
def add_book(current_user):
    data = request.get_json()

    new_book = Book(author=data['author'], title=data['title'])
    db.session.add(new_book)
    db.session.commit()

    message = f"New book added by {current_user.username}: {data['title']} by {data['author']}"
    send_message_to_queue(message)

    return jsonify({'message': 'Book added successfully!'})

@app.route('/all_books', methods=['GET'])
@token_required
def get_all_books(current_user):
    books = Book.query.all()

    output = []
    for book in books:
        book_data = {'author': book.author, 'title': book.title}
        output.append(book_data)

    return jsonify({'books': output})
```

- Implementing API endpoints:
  - ➢ Book Endpoint: Implement an endpoint for adding books to the database. This endpoint requires authentication and adds a new book to the database.
  - ➢ Get All Books Endpoint: Implement an endpoint for retrieving all books from the database. This endpoint also requires authentication.

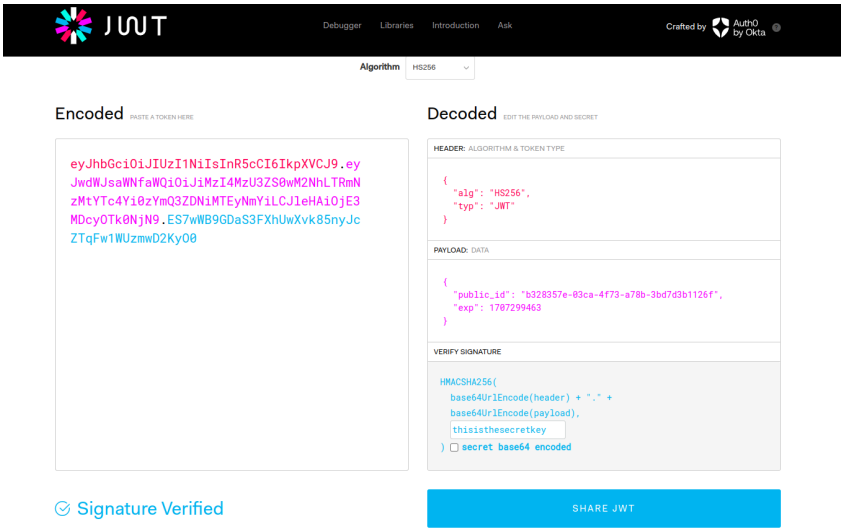
- Running the Flask app
- Testing with Postman:
  - ➢ Test User Registration: Use Postman to send a POST request to the /register endpoint with JSON data containing a username and password. Verify that a new user is created successfully.



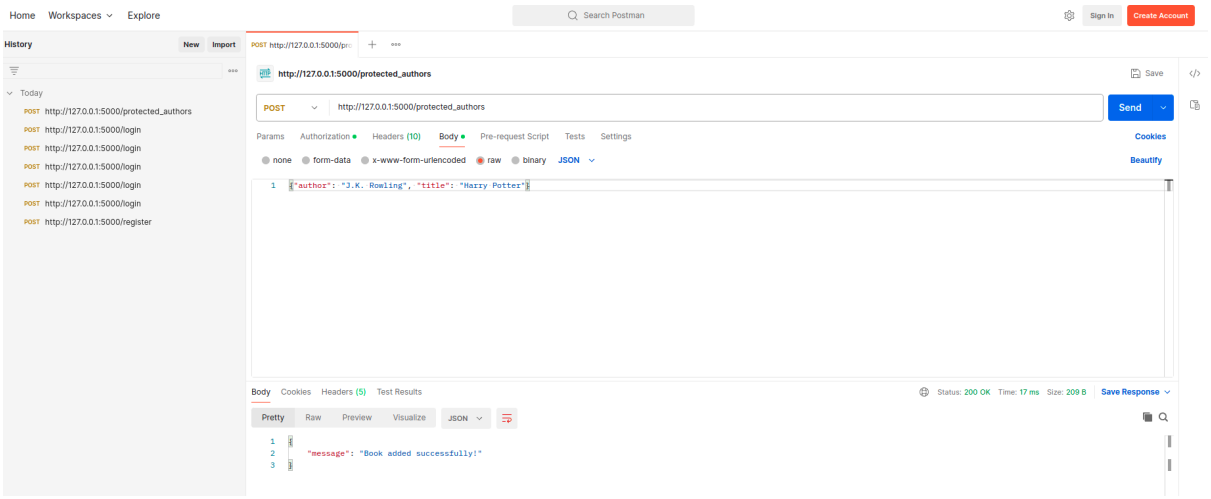
  - ➢ Test User Login: Send a POST request to the /login endpoint with the credentials of the newly registered user. Verify that a JWT token is returned upon successful login.

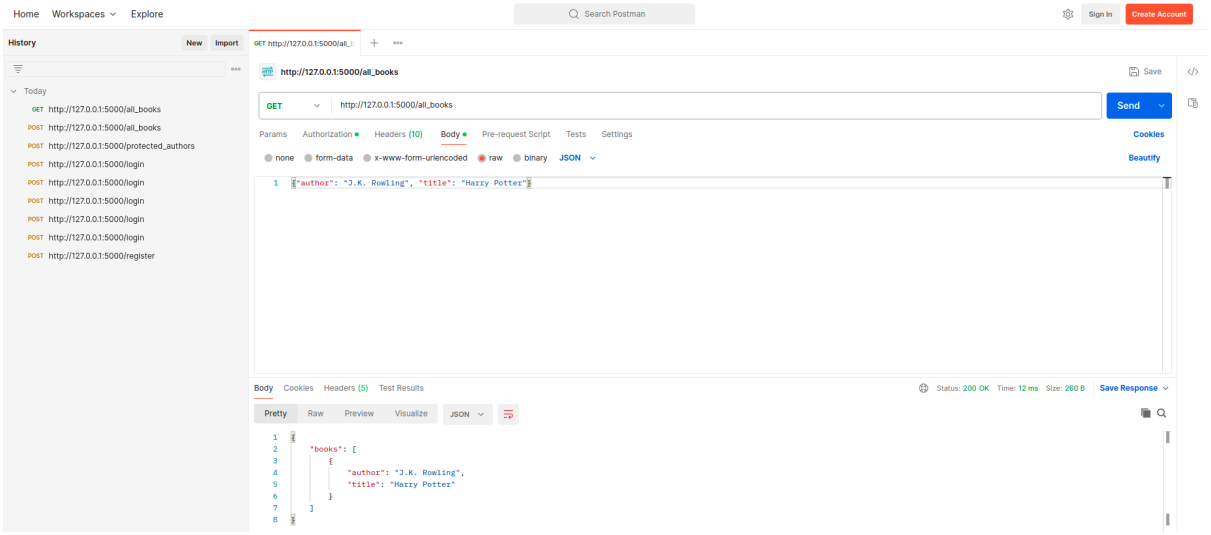


➢ Optional: The token can be verified using jwt.io:



➢ Test Protected Endpoints: Send requests to the protected endpoints (/protected_authors, /all_books) with the JWT token included in the request headers. Verify that access is granted only to authenticated users.




Home
Workspaces
Explore
Search Postman
Sign In
Create Account
History
New
Import
GET http://127.0.0.1:5000/all_
Today
GET http://127.0.0.1:5000/all_books
POST http://127.0.0.1:5000/all_books
POST http://127.0.0.1:5000/protected_authors
POST http://127.0.0.1:5000/login
POST http://127.0.0.1:5000/login
POST http://127.0.0.1:5000/login
POST http://127.0.0.1:5000/login
POST http://127.0.0.1:5000/login
POST http://127.0.0.1:5000/register
http://127.0.0.1:5000/all_books
Save
GET
http://127.0.0.1:5000/all_books
Send
Params
Authorization
Headers (10)
Body
Pre-request Script
Tests
Settings
Cookies
none
form-data
x-www-form-urlencoded
raw
binary
JSON
Beautify
1 {"author": "J.K. Rowling", "title": "Harry Potter"}
Body
Cookies
Headers (5)
Test Results
Status: 200 OK
Time: 12 ms
Size: 260 B
Save Response
Pretty
Raw
Preview
Visualize
JSON
1 {
2 "books": [
3 {
4 "author": "J.K. Rowling",
5 "title": "Harry Potter"
6 }
7]
8 }