RESTful API for a Bookstore Management System

main.py

This file contains the main Flask application with endpoints for managing books and user authentication using JWT tokens.

```
from flask import Flask, request, jsonify
from flask_sqlalchemy import SQLAlchemy
from flask_marshmallow import Marshmallow
import os
from flask_jwt_extended import JWTManager, jwt_required, create_access_token,
get_jwt_identity

app = Flask(__name__)
basedir = os.path.abspath(os.path.dirname(__file__))
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///'+os.path.join(basedir,
'db.sqlite')
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
app.config['JWT_SECRET_KEY'] = 'ooo'
app.config['JWT_ACCESS_TOKEN_EXPIRES'] = 3600

db = SQLAlchemy(app)
ma = Marshmallow(app)
jwt = JWTManager(app)

class Book(db.Model):
    id=db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(100))
    author = db.Column(db.String(100))
    isbn = db.Column(db.String(100), unique=True)
    price = db.Column(db.Float(100))
    quantity = db.Column(db.Integer)

    def __init__(self,title,author,isbn,price,quantity):
        self.title = title
        self.author = author
        self.isbn = isbn
        self.price = price
        self.quantity = quantity

class BookSchema(ma.Schema):
    class Meta:
        fields = ('id','title','author','isbn','price','quantity')
Book_schema = BookSchema()
Books_schema = BookSchema(many=True)
app.app_context().push()

@app.route('/login', methods=['POST'])
```

```python
def login():
    username = request.json.get('username')
    password = request.json.get('password')
    if username=='admin' and password=='balaji':
        access_token = create_access_token(identity=username)
        return jsonify(access_token=access_token),200
    else:
        return jsonify({"message":"Invalid username or password"}), 401

@app.route('/book',methods=['POST'])
@jwt_required()
def add_books():
    title = request.json['title']
    author = request.json['author']
    isbn = request.json['isbn']
    price = request.json['price']
    quantity = request.json['quantity']
    new_Book = Book(title, author, isbn, price, quantity)
    db.session.add(new_Book)
    db.session.commit()
    return Book_schema.jsonify(new_Book)

@app.route('/book',methods=['GET'])
@jwt_required()
def show_all_book():
    all_Books = Book.query.all()
    result = Books_schema.dump(all_Books)
    return jsonify(result)

@app.route('/book/<id>', methods=['GET'])
@jwt_required()
def getBookByid(id):
    book = Book.query.get(id)
    result = Book_schema.dump(book)
    return jsonify(result)

@app.route('/book/<author>',methods=['PUT'])
@jwt_required()
def updateUserByauthor(author):
    book = Book.query.filter_by(author=author).first()
    if book:
        title = request.json.get('title')
        price = request.json.get('price')
        quantity = request.json.get('quantity')
        if title is not None:
            book.title = title
        if price is not None:
            book.price = price
```

```
        if quantity is not None:
            book.quantity = quantity
    db.session.commit()
    return Book_schema.jsonify(book)

@app.route('/book/<id>', methods=['DELETE'])
@jwt_required()
def delete_book(id):
    book = Book.query.get(id)
    if book:
        db.session.delete(book)
        db.session.commit()
        return jsonify({"message": "Book deleted successfully"}), 200
    else:
        return jsonify({"error": "Book not found"}), 404

if __name__ == '__main__':
    app.run(debug=True, port=2002)
```

Dependencies:

Flask: Web framework for building APIs.

Flask-SQLAlchemy: Flask extension for interacting with SQL databases.

Flask-Marshmallow: Flask extension for serialization/deserialization.

Flask-JWT-Extended: Flask extension for JWT token-based authentication.

Endpoints:

/login: POST endpoint for user authentication. It accepts a JSON payload containing username and password, and it returns a JWT token upon successful authentication.

/book:

POST: Add a new book to the database. Requires a JWT token for authentication.

GET: Retrieve all books from the database. Requires a JWT token for authentication.

/book/<id>:

GET: Retrieve a book by its ID from the database. Requires a JWT token for authentication.

DELETE: Delete a book by its ID from the database. Requires a JWT token for authentication.

/book/<author>:

PUT: Update a book by its author. Requires a JWT token for authentication.

test.py

```python
import unittest
import json,os
from main import app, db, Book

class TestBookAPI(unittest.TestCase):

    def setUp(self):
        app.config['TESTING'] = True
        basedir = os.path.abspath(os.path.dirname(__file__))
        app.config['SQLALCHEMY_DATABASE_URI'] =
'sqlite:///'+os.path.join(basedir, 'db.sqlite')
        self.app = app.test_client()
        db.create_all()
    def tearDown(self):
        db.session.remove()
        db.drop_all()
    def test_valid_login(self):
        response = self.app.post('/login', json={'username': 'admin',
'password': 'balaji'})
        data = json.loads(response.data.decode('utf-8'))
        self.assertEqual(response.status_code, 200)
        self.assertIn('access_token', data)
        print(data)
    def test_invalid_login(self):
        response = self.app.post('/login', json={'username': 'invalid',
'password': 'invalid'})
        self.assertEqual(response.status_code, 401)
        data = json.loads(response.data.decode('utf-8'))
        self.assertEqual(data['message'], 'Invalid username or password')

    def test_add_books(self):
        access_token = self.get_access_token()
        response = self.app.post('/book', json={'title': 'The Great Gatsby',
'author': 'F. Scott Fitzgerald', 'isbn': '9780743273565', 'price': 9.99,
'quantity': 25}, headers={'Authorization': 'Bearer ' + access_token})
        self.assertEqual(response.status_code, 200)
        data = json.loads(response.data.decode('utf-8'))
        self.assertEqual(data['title'], 'Book1')

    def test_show_all_book(self):
        access_token = self.get_access_token()
        response = self.app.get('/book', headers={'Authorization': 'Bearer ' +
access_token})
        self.assertEqual(response.status_code, 200)
        data = json.loads(response.data.decode('utf-8'))
        self.assertEqual(len(data), 0)

    def test_get_book_by_id(self):
```

```
        access_token = self.get_access_token()
        response = self.app.get('/book/2', headers={'Authorization': 'Bearer '
+ access_token})
        self.assertEqual(response.status_code, 404)

    def test_update_user_by_author(self):
        access_token = self.get_access_token()
        response = self.app.put('/book/Harper_Lee', json={'title': 'Go Set a
Watchman', 'price': 20.0, 'quantity': 10}, headers={'Authorization': 'Bearer '
+ access_token})
        self.assertEqual(response.status_code, 404)

    def test_delete_book(self):
        access_token = self.get_access_token()
        response = self.app.delete('/book/1', headers={'Authorization':
'Bearer ' + access_token})
        self.assertEqual(response.status_code, 404)

    def get_access_token(self):
        response = self.app.post('/login', json={'username': 'admin',
'password': 'balaji'})
        data = json.loads(response.data.decode('utf-8'))
        return data['access_token']
if __name__ == '__main__':
    unittest.main()
```

This file contains unit tests for the Flask application defined in main.py.

Dependencies:

unittest: Python's built-in unit testing framework.

Test Cases:

test_valid_login: Test the /login endpoint with valid credentials.

test_invalid_login: Test the /login endpoint with invalid credentials.

test_add_books: Test the /book POST endpoint to add a new book.

test_show_all_book: Test the /book GET endpoint to retrieve all books.

test_get_book_by_id: Test the /book/<id> GET endpoint to retrieve a book by its ID.

test_update_user_by_author: Test the /book/<author> PUT endpoint to update a book by its author.

test_delete_book: Test the /book/<id> DELETE endpoint to delete a book by its ID.

Running the Tests

To run the tests, execute the test.py file. Ensure that the Flask app is not running on the same port while running the tests. The tests will make HTTP requests to the Flask app to simulate user interactions and API calls.

Output: