Project Documentation: E-commerce Platform

Jazz Records
Abdilmazhit Bakhytzhan SE-2317

1. Introduction

- Project Overview: This project implements a basic e-commerce platform demonstrating core functionalities like product listing, user registration/login, order placement, and admin management. It serves as a practical example for learning and applying MongoDB concepts within a Python Flask application.
- Purpose of this Document: This document provides a comprehensive overview of the project's architecture, functionality, and implementation details, with a particular emphasis on the use of MongoDB as the primary database.
- Target Audience: Developers, students, and instructors interested in learning how to build web applications with Flask and MongoDB.
- Key Technologies: Python, Flask, MongoDB, Flask-JWT-Extended, bcrypt.

2. Technology Stack

- Python: The primary programming language.
- Flask: A micro web framework used for building the web application.
- MongoDB: A NoSQL document database used for storing application data.
- Flask-JWT-Extended: Used for handling JSON Web Token (JWT) based authentication.
- bcrypt: Used for password hashing and security.

3. MongoDB: A NoSQL Database

3.1 Introduction to NoSQL and MongoDB

- What is NoSQL? NoSQL (Not Only SQL) databases offer an alternative to traditional relational databases (SQL). They are designed to handle large volumes of unstructured or semi-structured data, providing scalability and flexibility advantages.
- Why MongoDB? MongoDB is a popular document-oriented NoSQL database that stores data in flexible, JSON-like documents. This approach aligns well with modern development practices and simplifies data modeling for many applications.

3.2 Key Features and Benefits of MongoDB

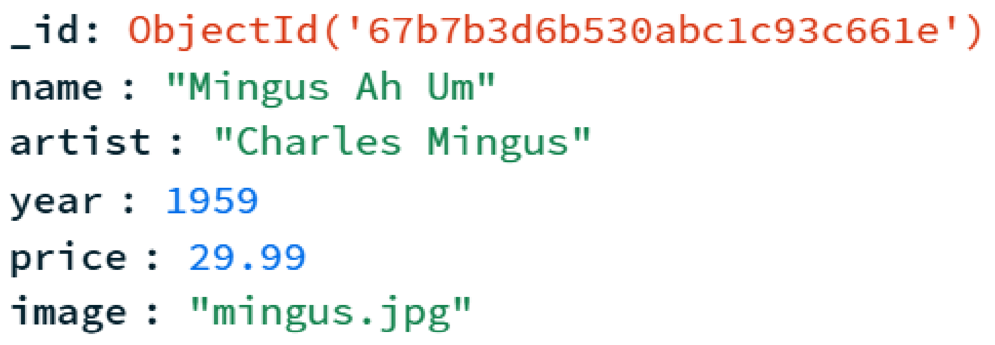
- Document-Oriented: Data is stored in BSON (Binary JSON) documents, allowing for complex data structures to be represented naturally. This simplifies data access and manipulation within the application code.
- Flexible Schema: MongoDB does not enforce a rigid schema, allowing for easy evolution of the data model as application requirements change. New fields can be added to documents without requiring schema migrations.
- Scalability: MongoDB can scale horizontally by sharding data across multiple servers. This enables the application to handle increasing data volumes and user traffic.
- High Performance: MongoDB provides indexing capabilities and query optimization techniques for fast read and write operations.
- Replication and High Availability: MongoDB's replica set feature provides data redundancy and automatic failover, ensuring high availability and data durability.
- Aggregation Framework: MongoDB's powerful aggregation framework allows for complex data transformations and analysis, enabling the generation of reports, data mining, and real-time dashboards.

3.3 MongoDB in this Project

- Database and Collection Design:
    - The database name is `ecommerce`.
    - The project uses the following collections:
        - `products`: Stores product information (name, artist, year, price, image).
        - `users`: Stores user information (name, email, password, role).
        - `orders`: Stores order information (user_email, products, status).

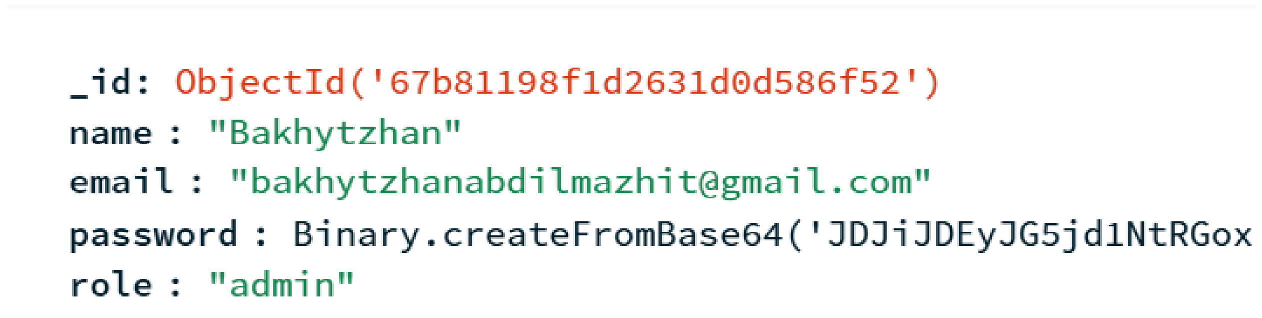
Data Modeling:

- Data is modeled using JSON-like documents. Here are examples of documents in each collection.
- Products:



```
_id: ObjectId('67b7b3d6b530abc1c93c661e')
name : "Mingus Ah Um"
artist : "Charles Mingus"
year : 1959
price : 29.99
image : "mingus.jpg"
```

- Users:



```
_id: ObjectId('67b81198f1d2631d0d586f52')
name : "Bakhytzhan"
email : "bakhytzhanabdilmazhit@gmail.com"
password : Binary.createFromBase64('JDJiJDEyJG5jd1NtRGox
role : "admin"
```

Connection Details:

- The application connects to the MongoDB database using the `pymongo` driver. The connection string is:

  ```
  MONGO_URI =
  "mongodb+srv://bakhytzhanabdilmazhit:6FcpfAMoAP95tOXK@cluster0.mlqaf
  .mongodb.net/?retryWrites=true&w=majority&appName=Cluster0"
  ```

4. Project Architecture

- Overall Structure: The application follows a typical client-server architecture. The Flask backend handles API requests and serves HTML templates. The client-side (browser) interacts with the backend via HTTP requests.
- Components:
  - Backend (Flask):
    - Handles routing and logic for API endpoints related to products, users, authentication, and orders.
    - Renders HTML templates for the user interface.
  - Database (MongoDB):
    - Stores all application data in collections as described above.
- Data Flow: A user interacts with the application through their web browser. When a user performs an action (e.g., viewing products, logging in, placing an order), the browser sends an HTTP request to the Flask backend. The Flask backend processes the request, interacts with the MongoDB database to retrieve or update data, and then sends a response back to the browser, often in the form of HTML or JSON.

5. API Endpoints

Endpoint	Method	Purpose	Request Parameters	Response	Authentication
`/`	`GET`	Retrieves all products for display on the home page.	None	HTML page with a list of products.	None
`/products`	`GET`	Retrieves a list of all products as JSON.	None	JSON array of product objects.	None
`/product/`	`GET`	Retrieves a specific product by ID and displays it on a product page.	`product_id` in the URL	HTML page with product details.	None

`/register`	`GET`, `POST`	Handles user registration.	`name`, `email`, `password`, `role` (optional, defaults to "customer") in JSON body	`GET`: Registration form (HTML). `POST`: JSON with success/error message.	None
`/login`	`GET`, `POST`	Handles user login.	`email`, `password` in JSON body	`GET`: Login form (HTML). `POST`: JSON with JWT token and user data on success, error message on failure.	None
`/logout`	`GET`	Logs out the user.	None	Redirects to the home page (`/`).	None
`/admin`	`GET`	Displays the admin dashboard with a list of users.	None	HTML page with a list of users.	None

`/admin/users`	`GET`	Retrieves a list of all users as JSON.	None	JSON array of user objects.	None
`/admin/update-user/`	`PUT`	Updates a user's role (Admin Only).	`user_id` in the URL, `role` in JSON body	JSON with a success message.	Admin Only
`/admin/delete-user/`	`DELETE`	Deletes a user (Admin Only).	`user_id` in the URL	JSON with a success message.	Admin Only
`/admin/add-product`	`POST`	Adds a new product (Admin Only).	Product details (name, artist, year, price, image) in JSON body	JSON with a success message and the product ID.	Admin Only
`/admin/update-product/`	`PUT`	Updates a product's information (Admin Only).	`product_id` in the URL, product details to update in JSON body	JSON with a success message.	Admin Only

`/admin/delete-product/`	`DELETE`	Deletes a product (Admin Only).	`product_id` in the URL	JSON with a success message.	Admin Only
`/order`	`POST`	Places an order (Customer Only).	`products` (list of product names) in JSON body	JSON with a success message and the order ID.	JWT Required
`/my-orders`	`GET`	Retrieves a list of the current user's orders (Customer Only).	None	JSON array of order objects.	JWT Required
`/test-db`	`GET`	Tests the database connection.	None	JSON with a success or error message.	None

6. Authentication and Authorization

- Authentication: Users are authenticated using JSON Web Tokens (JWTs).
    - The login process:
        1. The user submits their email and password via the `/login` endpoint.
        2. The Flask backend verifies the credentials against the `users` collection in MongoDB.
        3. If the credentials are valid (email exists and the password matches the stored hash), the server generates a JWT using `Flask-JWT-Extended`. This JWT contains the user's email and role.
        4. The JWT is returned to the client in the JSON response.
    - The client is responsible for storing the JWT (typically in local storage or a cookie).
    - For subsequent requests to protected resources (e.g., placing an order, viewing their orders), the client includes the JWT in the `Authorization` header of the HTTP request, using the `Bearer` scheme: `Authorization: Bearer <JWT_TOKEN>`.
- Authorization: User roles are used to authorize access to specific resources.
    - The application defines two roles: "customer" and "admin".
    - Admin users have access to the `/admin/...` endpoints for managing users and products.
    - The `@jwt_required()` decorator and `get_jwt_identity()` function (from `Flask-JWT-Extended`) are used to enforce authorization. For example:
- python

```
@app.route('/admin/update-user/', methods=['PUT'])
@jwt_required()
def update_user(user_id):
    current_user = get_jwt_identity()
    if current_user["role"] != "admin":
        return jsonify({"message": "Unauthorized"}), 403
    data = request.json
    db.users.update_one({"_id": ObjectId(user_id)}, {"$set":
{"role": data.get("role")}})
    return jsonify({"message": "User updated successfully!"})
```

-

- This code snippet first checks if the current user (obtained from the JWT) has the "admin" role. If not, it returns a 403 (Unauthorized) error.
- [ADD A SCREENSHOT SHOWING HOW JWT TOKENS ARE USED IN THE APPLICATION (E.G., USING THE BROWSER'S DEVELOPER TOOLS TO INSPECT THE `Authorization` HEADER IN A REQUEST TO A PROTECTED ENDPOINT)]

7. Code Explanation (Key Sections)

Database Connection: The application connects to the MongoDB database using the `pymongo` driver.

```
MONGO_URI =
"mongodb+srv://bakhytzhanabdilmazhit:6FcpfAMoAP95tOXK@cluster0.m
lqaf.mongodb.net/?retryWrites=true&w=majority&appName=Cluster0"
client = pymongo.MongoClient(MONGO_URI)
db = client.get_database("ecommerce")
```

- This code establishes a connection to the MongoDB Atlas cluster specified by the `MONGO_URI` and retrieves the `ecommerce` database.
- User Registration: New users are registered through the `/register` endpoint.

```
hashed_pw = bcrypt.hashpw(password.encode('utf-8'),
bcrypt.gensalt())
user = {
    "name": name,
    "email": email,
    "password": hashed_pw,
    "role": role
}
db.users.insert_one(user)
```

- This code hashes the user's password using `bcrypt` before storing it in the `users` collection. Hashing passwords is crucial for security.
- User Login: Users are authenticated through the `/login` endpoint.

```
user = db.users.find_one({"email": email})
if user:
    stored_password = user["password"]
    if isinstance(stored_password, str):
        stored_password = stored_password.encode('utf-8')
    if bcrypt.checkpw(password.encode('utf-8'),
stored_password):
        token = create_access_token(identity={"email":
user["email"], "role": user["role"]})
        return jsonify({
            "token": token,
            "user": {
                "name": user["name"],
                "email": user["email"],
                "role": user["role"]
            }
        })
return jsonify({"error": "Invalid credentials"}), 401
```

- This code retrieves the user from the `users` collection based on their email address, verifies the password using `bcrypt.checkpw()`, and generates a JWT using `create_access_token()` if the credentials are valid.
- Product Retrieval: Products are retrieved from the `products` collection using the `find()` method.

```
products = list(db.products.find({}, {"_id": 1, "name": 1,
"artist": 1, "year": 1, "price": 1, "image": 1}))
for product in products:
    product["_id"] = str(product["_id"])
```

- This code retrieves all products from the `products` collection, specifying which fields to include in the results. The `_id` field (which is an `ObjectId` in MongoDB) is converted to a string for easier handling in the application.
- Order Placement: Orders are placed through the `/order` endpoint.

```
@app.route('/order', methods=['POST'])
@jwt_required()
def place_order():
    current_user = get_jwt_identity()
    data = request.json
    order = {
        "user_email": current_user["email"],
        "products": data.get("products"),
        "status": "Pending"
    }
    order_id = db.orders.insert_one(order).inserted_id
    return jsonify({"message": "Order placed!", "order_id": 
str(order_id)})
```

- This code retrieves the user's email from the JWT, creates a new order document with the user's email, the list of products, and a status of "Pending", and inserts the order into the `orders` collection.

8. Potential Improvements

- Frontend Development: Develop a more sophisticated frontend using a JavaScript framework like React, Vue.js, or Angular. This would improve the user experience and allow for more dynamic interactions.
- Improved Security: Implement more robust security measures, such as input validation, output encoding, and protection against Cross-Site Scripting (XSS) and Cross-Site Request Forgery (CSRF) vulnerabilities. Consider using HTTPS for all communication.
- Advanced Features:
    - Add product search, filtering, and sorting.
    - Implement a shopping cart.
    - Allow users to write product reviews.
- Payment Gateway Integration: Integrate with a payment gateway (e.g., Stripe, PayPal) to enable real-world transactions.
- Testing: Implement unit tests and integration tests using a testing framework like `pytest` to ensure the quality and reliability of the application.
- Dockerization: Containerize the application using Docker for easier deployment and management. This would allow you to easily deploy the application to different environments (e.g., development, staging, production) without worrying about dependency conflicts.
- Deployment: Deploy the application to a production environment (e.g., Heroku, AWS, Google Cloud).

9. Conclusion

This project demonstrates the fundamental principles of building an e-commerce platform with Flask and MongoDB. By leveraging MongoDB's flexible schema and scalability, the application can efficiently store and manage data for products, users, and orders. The use of JWT authentication provides a secure way to protect API endpoints and manage user sessions. This project provides a solid foundation for building more complex and feature-rich e-commerce applications.

Abdilmazhit Bakhytzhan
@231012