Summer Internship Project

(June 11, 2025 – August 11, 2025)

**Flight Booking**

**Management**

**System**

**Submitted By**: Archit Jha (Backend Intern)

**Mentor:** Palak Chawla (Software Developer)

**INDEX**

|  |  |  |
| --- | --- | --- |
| **S.No.** | **Module** | **Subsections** |
|  | **Introduction** | Objective  Problem Statement  Tech Stack & Tools |
|  | **Login & Signup** | Signup Module for New Users  Login Module for Registered Users |
|  | **Database Structure** | Data collection  Data pre-processing  Data insertion  Data updation  (Creation & Retrieval using Kaggle) |
|  | **Data Filtration** | Project folder structure  GET/POST Method on API |
|  | **Submission of Data** | Dummy booking API  Updation of Seat Availability |
|  | **Summary** | -- |

**Objective**

The **Flight Management System** is a backend-only project that allows users to securely sign up, log in, and search for details using API endpoints tested via **Postman**. Instead of using public APIs, the system retrieves flight data from a **MongoDB database** containing manually inserted dummy data. Access to flight search is protected using **JWT authentication**, ensuring only logged-in users can retrieve flight details. Built with **Node.js**, **Express.js**, and **Mongoose**, the project demonstrates key backend concepts like RESTful API design, password hashing with **bcrypt**, and secure route protection.

**Problem Statement**

In real-world airline and travel booking systems, the management of user authentication and real-time flight search requires secure, efficient, and scalable backend systems. However, building such systems using live flight APIs can be expensive, complex, and dependent on third-party services.

Moreover, many beginner and intermediate developers find it challenging to understand how the backend of a flight booking system works — including user login/signup flows, database integration, and secure access to protected data. There is also a need for testing backend APIs in a structured environment without relying on user interfaces or production-level data.

To address these challenges, this project proposes a backend-only **Flight Management System** that simulates a flight booking experience using **predefined (dummy) flight data** stored in **MongoDB**. The system includes **user authentication with secure login/signup**, **JWT-based route protection**, and a **flight search API** — all of which are tested through the **Postman API client**.

This solution provides a controlled, offline-ready environment to practice and demonstrate core backend development skills while simulating how a real flight booking platform might operate behind the scenes.

**Technologies, stack and tools used**

**Backend Stack**

|  |  |  |
| --- | --- | --- |
| **Component** | **Technology Used** | **Purpose** |
| Runtime | **Node.js** | Server-side JavaScript execution |
| Framework | **Express.js** | To create RESTful API endpoints |
| Database | **MongoDB** | To store dummy flight data and user credentials |
| ODM Library | **Mongoose** | To define schemas and interact with MongoDB |
| Security | **bcryptjs** | To hash passwords securely before saving |
| Authentication | **jsonwebtoken (JWT)** | To generate and verify authentication tokens |
| HTTP Client | **axios / node-fetch** | (Optional) For external API simulation if required |

**Testing & Simulation Tools**

|  |  |
| --- | --- |
| **Tool** | **Purpose** |
| **Postman** | Used to test and verify all backend API endpoints (e.g., login, signup, flight search) without a frontend UI |

**Development Tools**

|  |  |
| --- | --- |
| **Tool** | **Purpose** |
| **Visual Studio Code** | Used as the primary code editor for writing and managing the Node.js backend code |
| **MongoDB Compass** | GUI tool to view and manage MongoDB collections (users, flights) |

**Module 1**

**Login & Signup:**

The **Signup and Login Module** of the Flight Management System is responsible for managing user registration and authentication. It ensures that only authorized users can access protected parts of the application, such as the flight search functionality.

**Signup Functionality**

* Method: POST
* Endpoint: 8000/api/register
* During signup, the user is required to provide:
  + **Full name**
  + **Email**
  + **Phone number**
  + **Password**
* **Validation logic** is implemented to ensure data integrity:
  + The **email** must include the '@' symbol to follow a basic email format.
  + The **phone number** must be exactly **10 digits long**.
* If the input fails validation, an appropriate error message is returned.
* Upon passing validation:
  + The password is **hashed using bcrypt** to ensure security.
  + The user data is then stored in the **MongoDB** database using a **Mongoose schema**.

This process ensures that only valid and secure data is accepted into the system.

**Login Functionality**

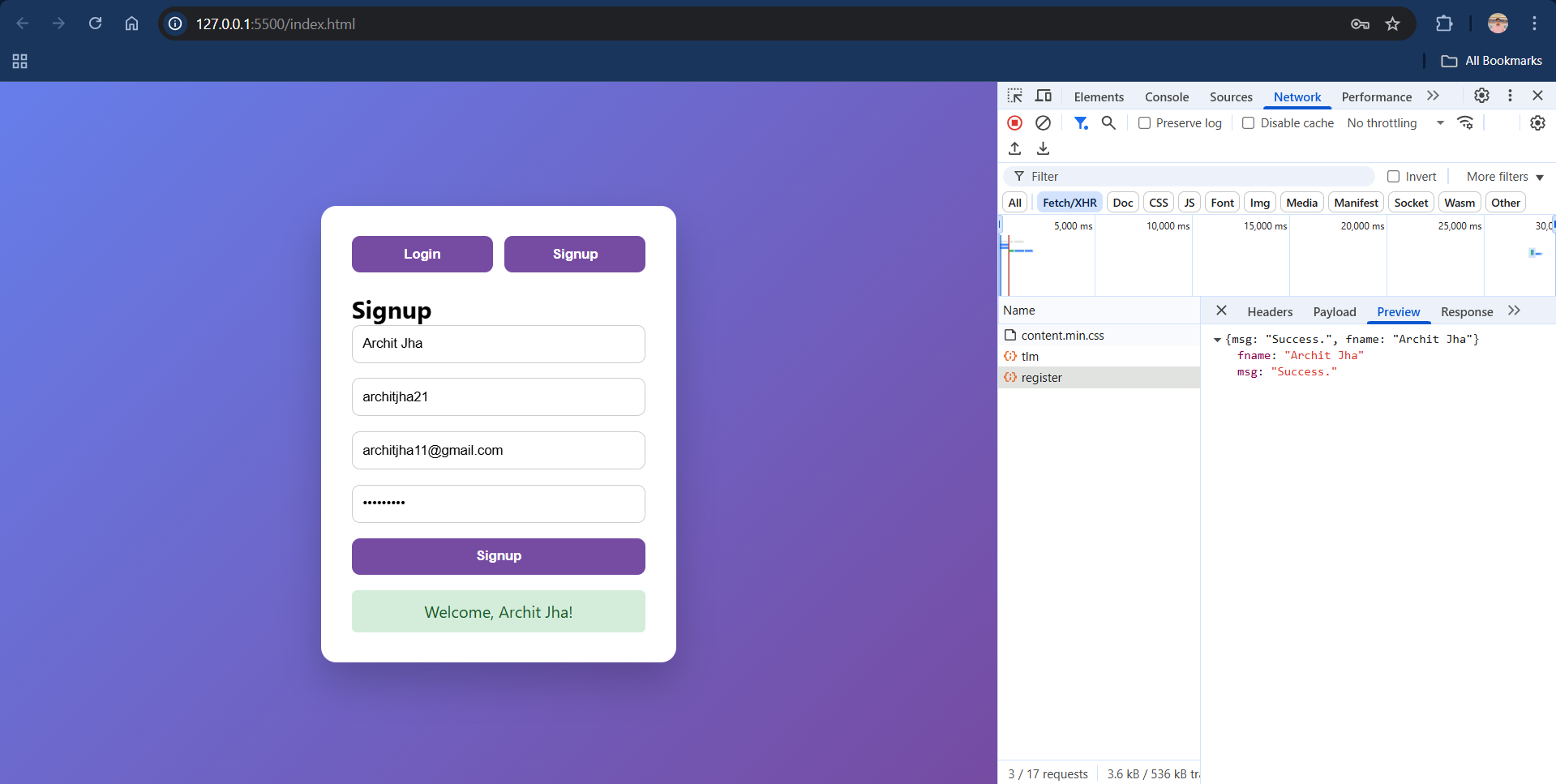
* Method: POST
* Endpoint: 8000/api/login
* The login endpoint verifies a user’s **email and password**.
* If the email exists and the password matches (checked using bcrypt.compare()):
  + A **JWT (JSON Web Token)** is generated and sent back in the response.
  + This token is used to authenticate subsequent requests to protected routes.

**Access Control Using JWT**

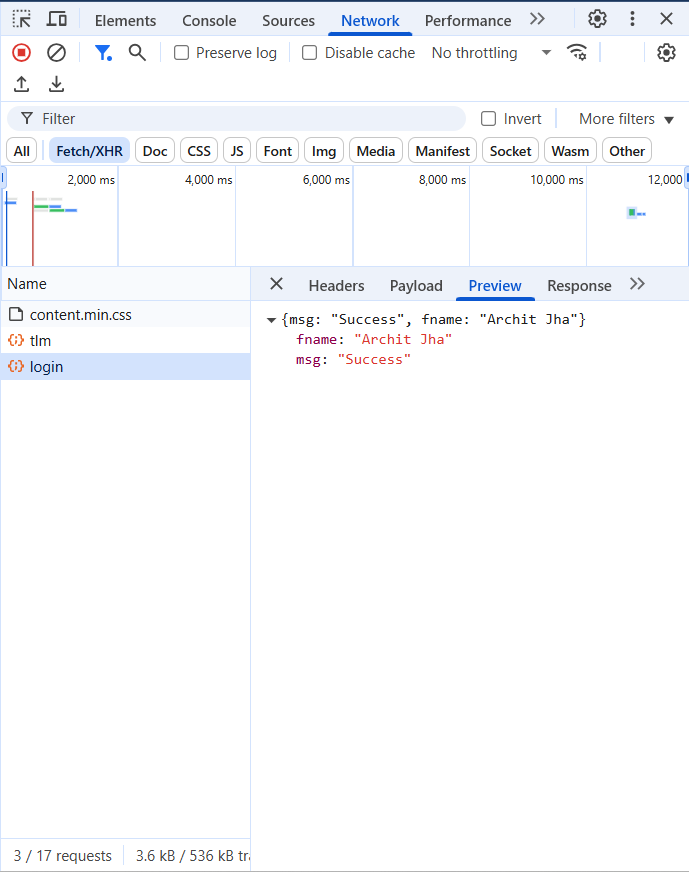
* The system uses **middleware** to verify JWT tokens.
* Only users with valid tokens can access protected routes like /search-flights.
* This ensures that unauthenticated users cannot access sensitive data.

**Testing in Postman**

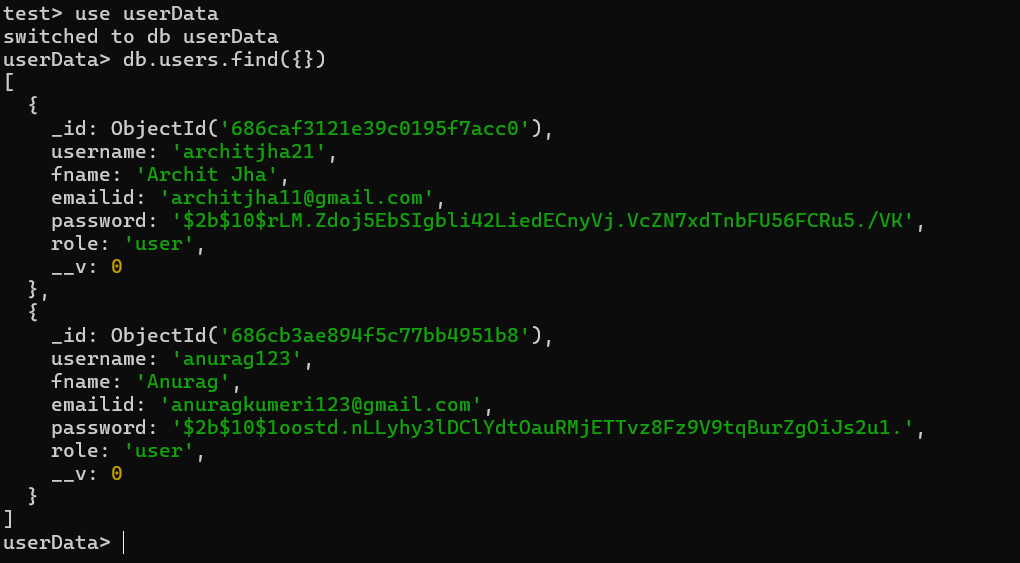
* The module is tested entirely using **Postman**.
* Signup and login requests are sent with JSON payloads.
* After a successful login, the returned JWT token is added in the Authorization header (as Bearer <token>) to test protected APIs like flight search.
* Signup page:



* Login page



* Entry in Database

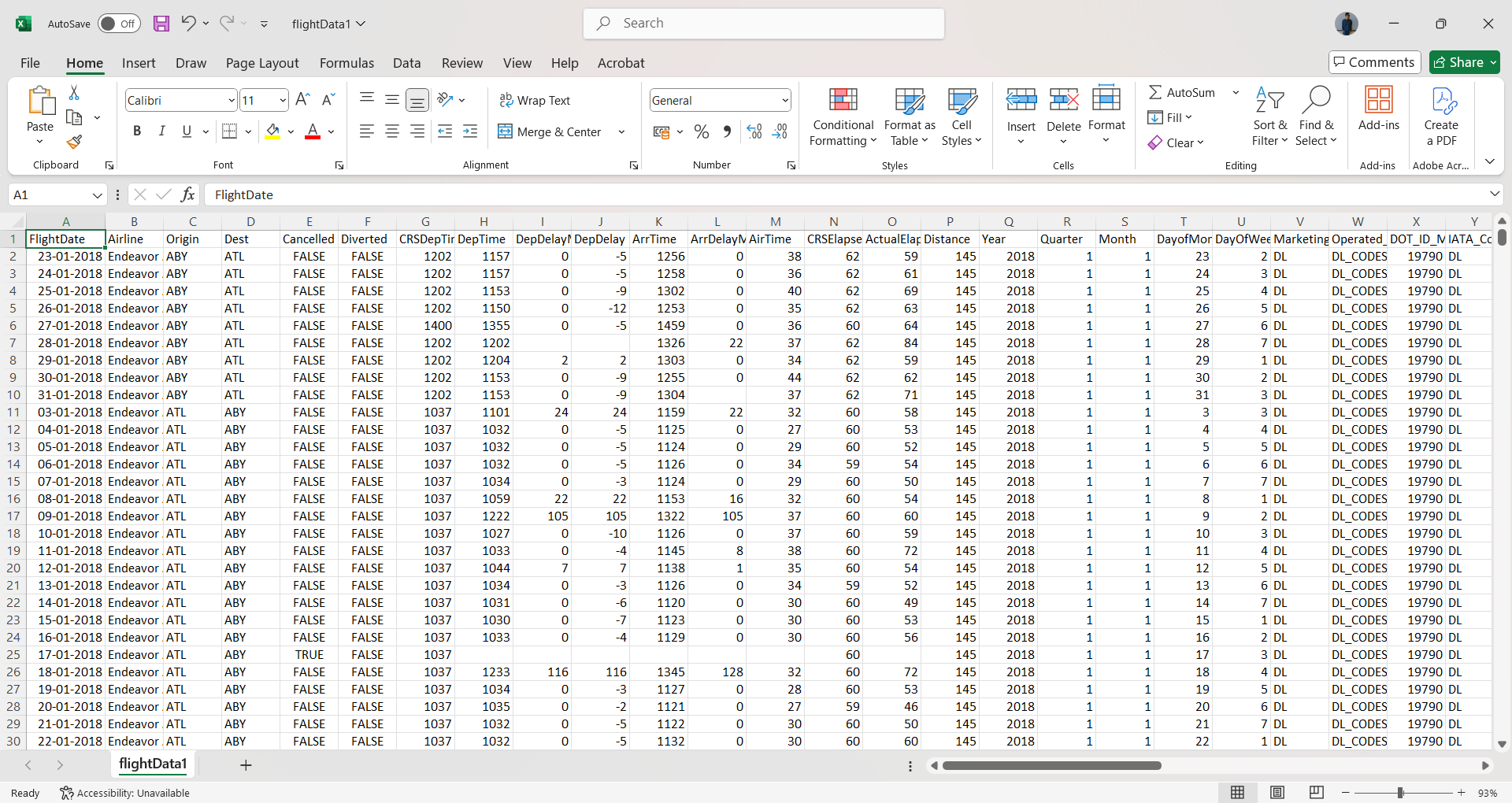


**Module 2**

**Data Collection**

The flight-related datasets used in this project were sourced from **Kaggle**, a popular online platform for datasets and data science competitions. The data was curated and cleaned to match the application's schema requirements, including flight details, pricing, layovers, seat availability, and city information. Some additional fields like **layoverDuration**, **seatCounts**, and **onDate** were generated or randomized using Excel formulas for simulation purposes.

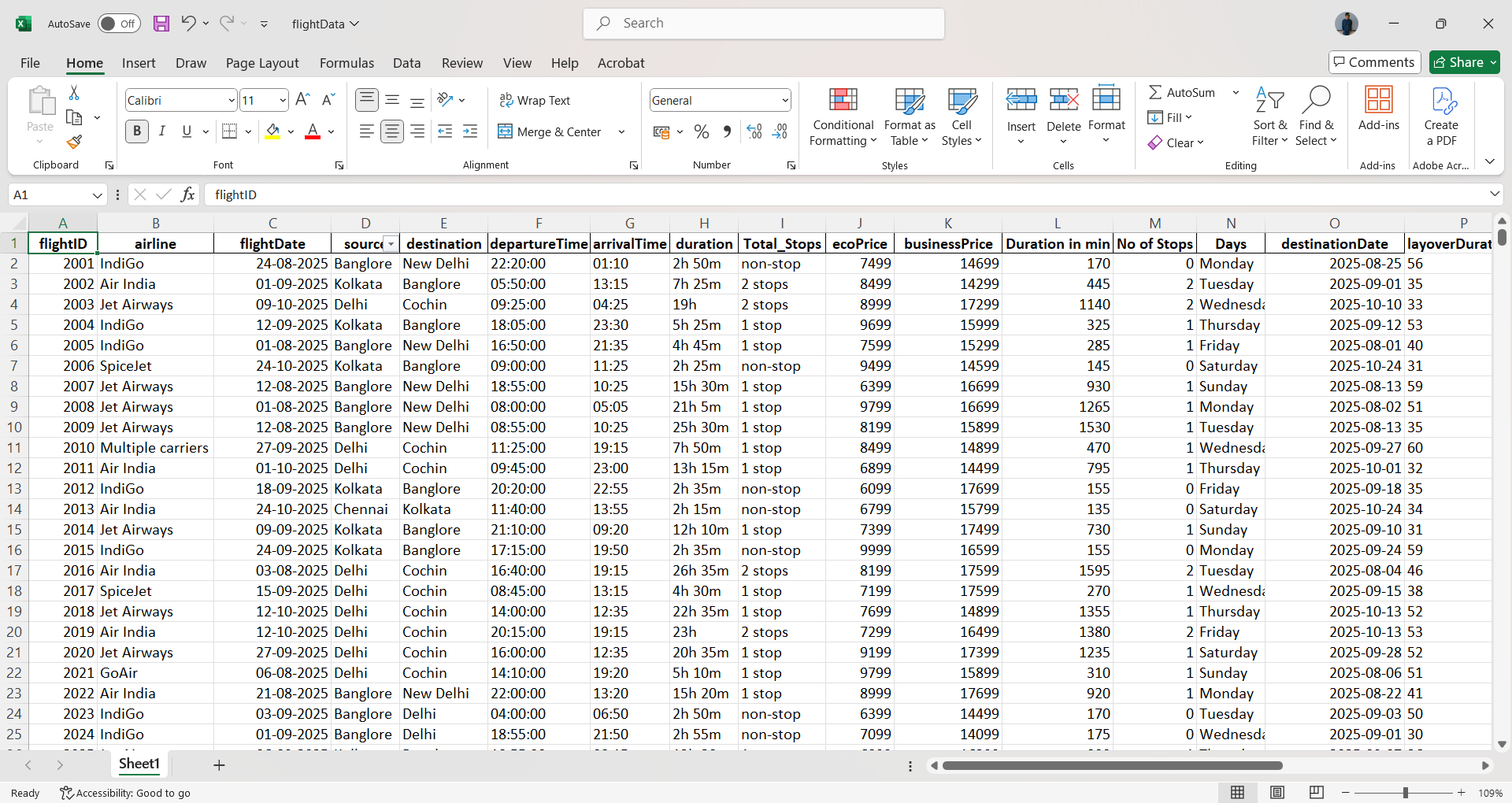
* Raw data:



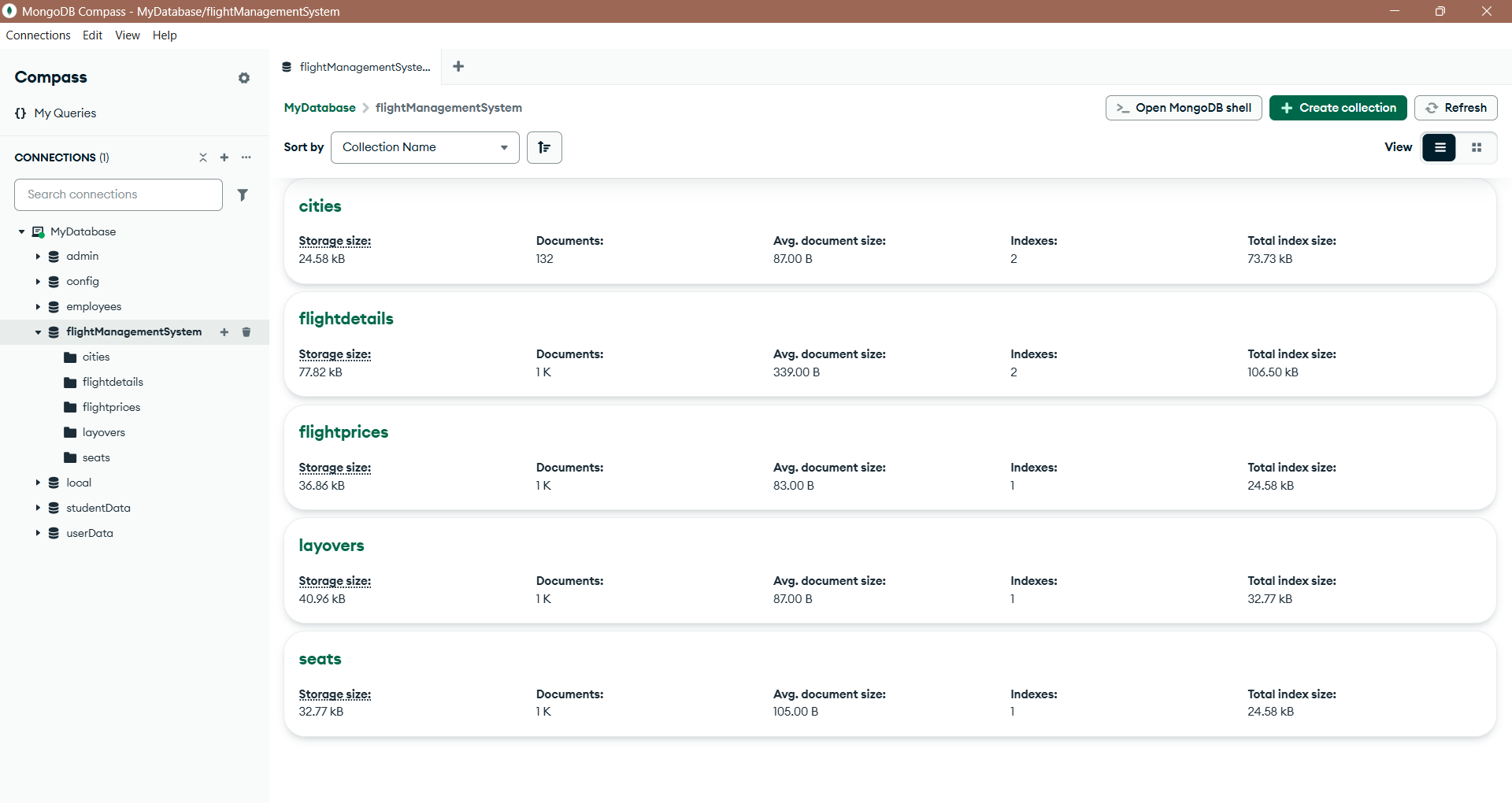
**Data pre-processing**

To align the dataset with the application's backend schema, the data was thoroughly pre-processed. Fields were cleaned, normalized, and transformed as needed to ensure consistency across all modules. The data was organized into five main MongoDB collections:

* **cities** (containing city information)
* **flightdetails** (storing flight schedules and routes)
* **flightprice** (containing pricing information for economy and business classes)
* **seats** (tracking available seats per flight)
* **layovers** (detailing intermediate stop durations). Additional fields such as layoverDuration and onDate were either generated or adjusted to match the application's requirements.
* Pre-processed data



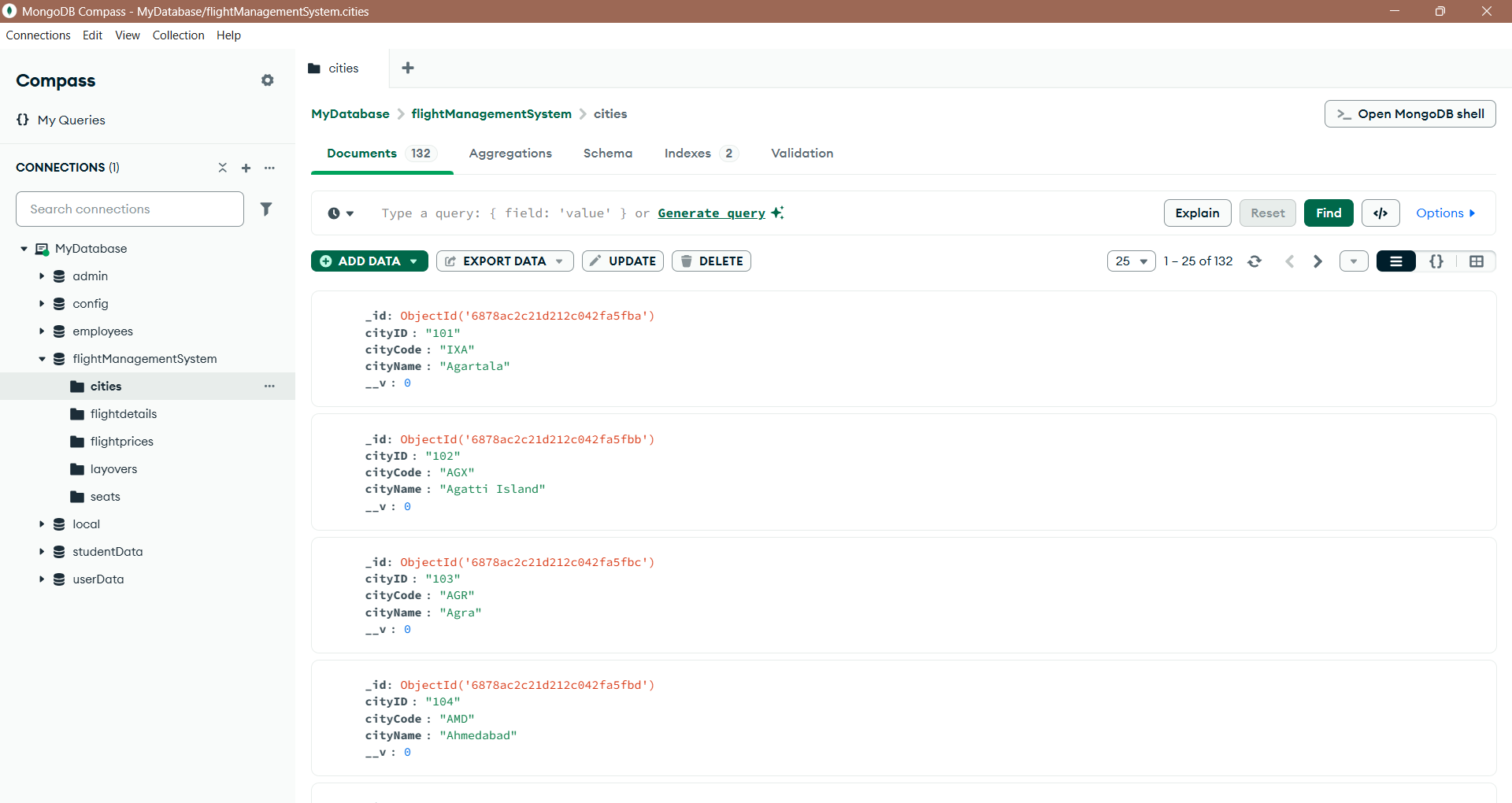
* Collections imported in MongoDB



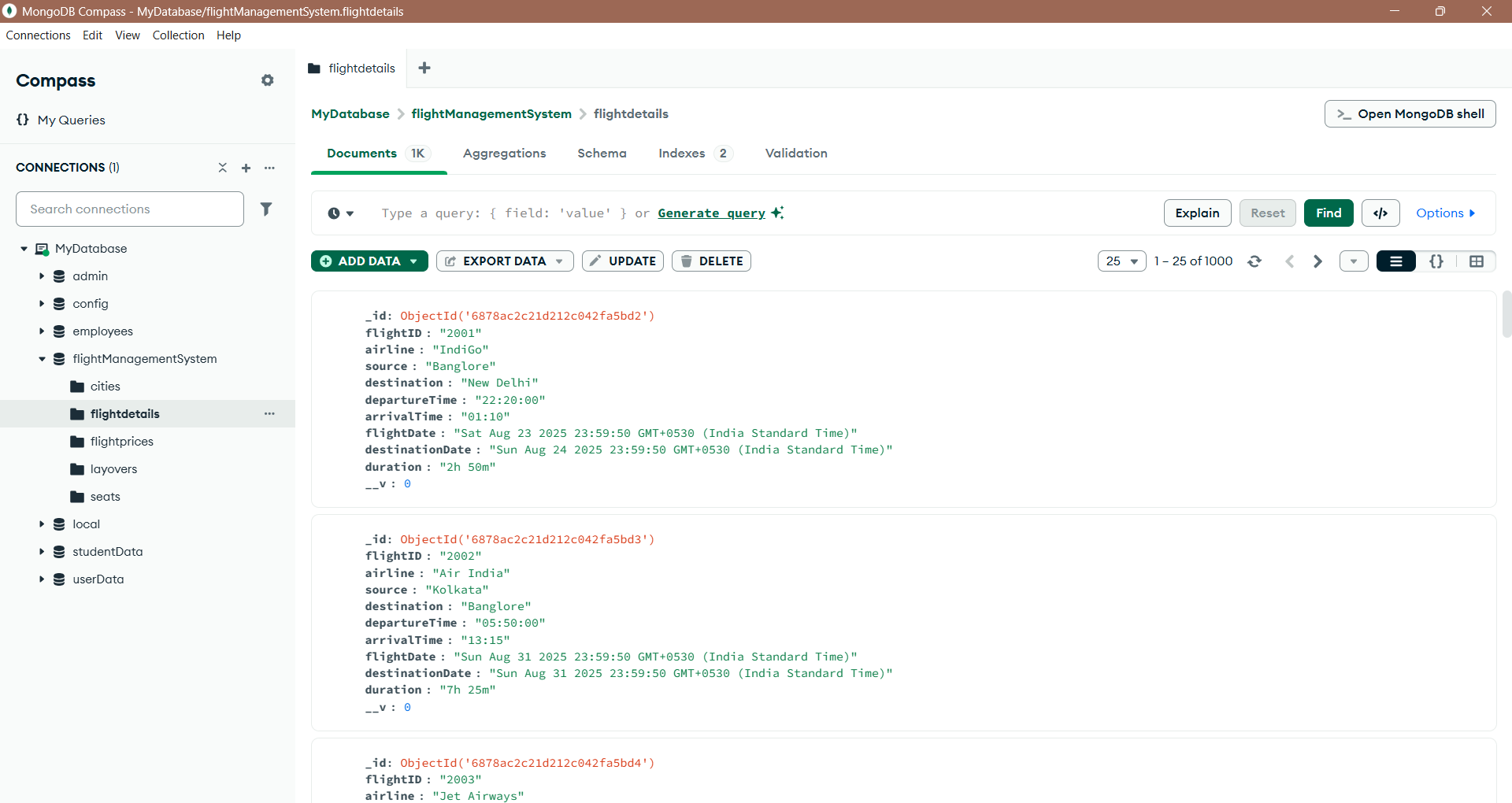
**Data Insertion**

The data used in this project was imported from **pre-processed Excel files** tailored to meet the structural requirements of the flight management system. Using **Node.js** and the **xlsx** library, Excel sheets were read and converted into JSON format. Dedicated **Mongoose schemas** were created for each MongoDB collection to maintain data integrity and validation. The data was then inserted into MongoDB using the insertMany() method for each collection.

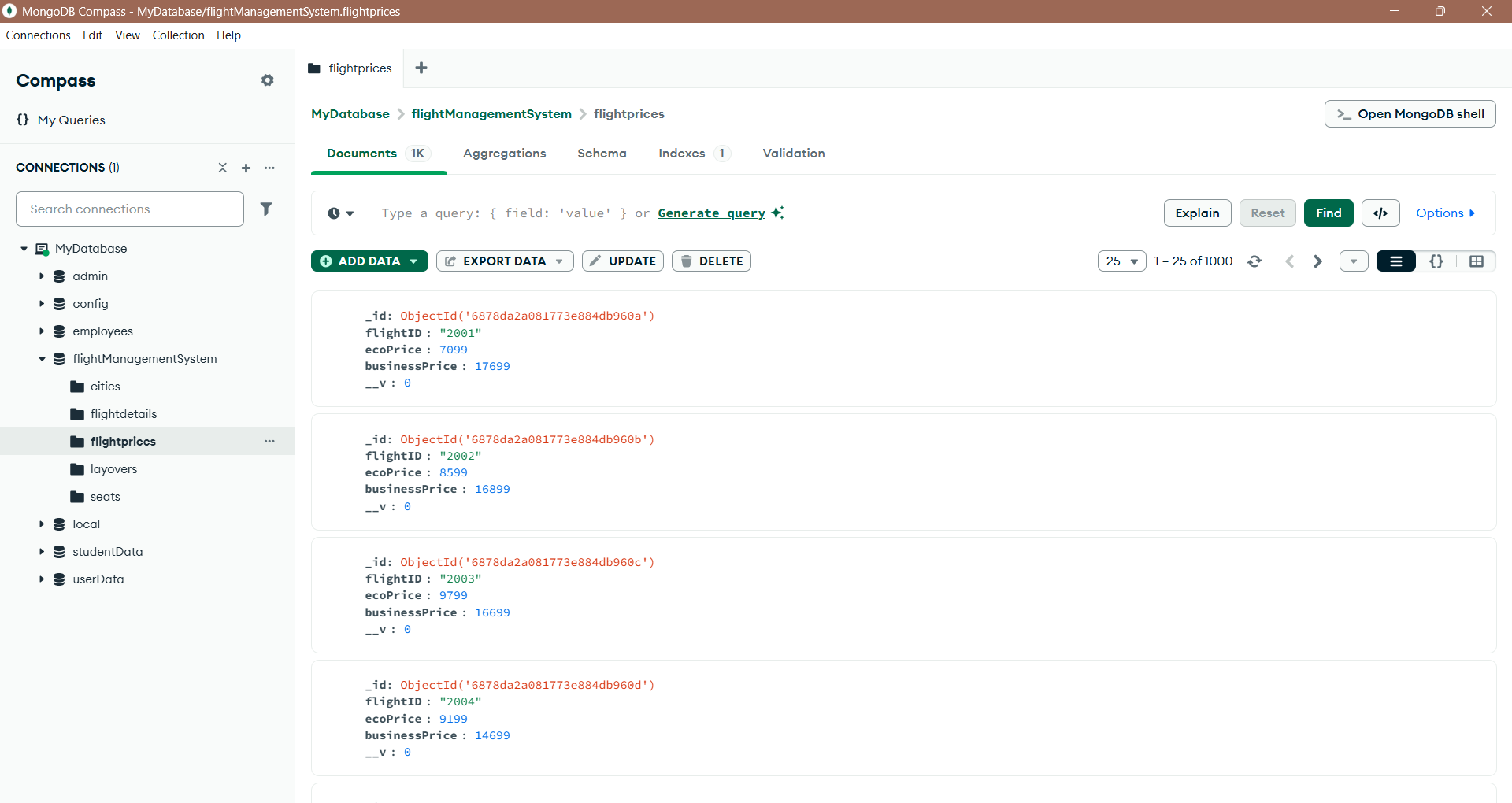
* Cities collection



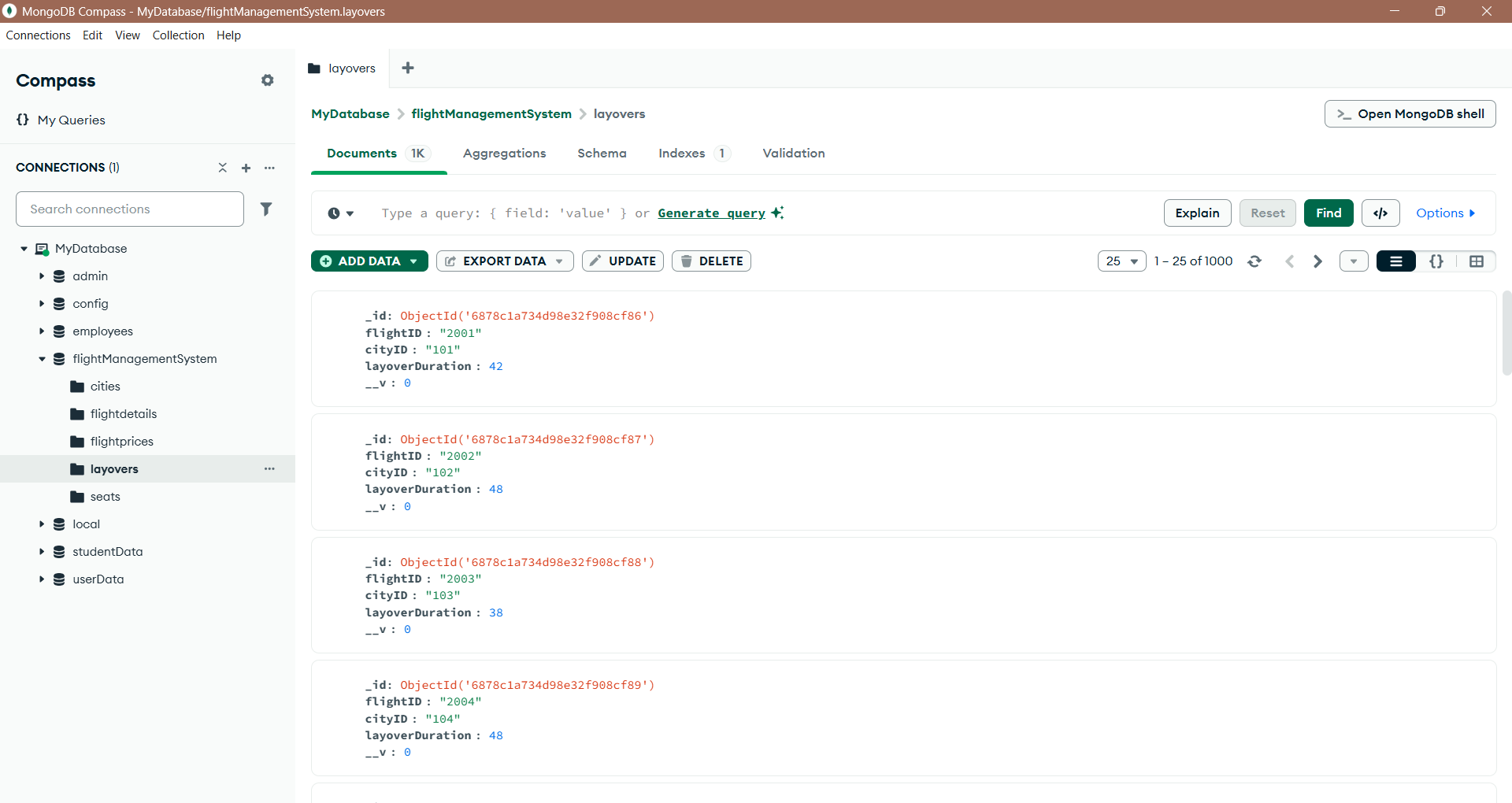
* flightDetails collection



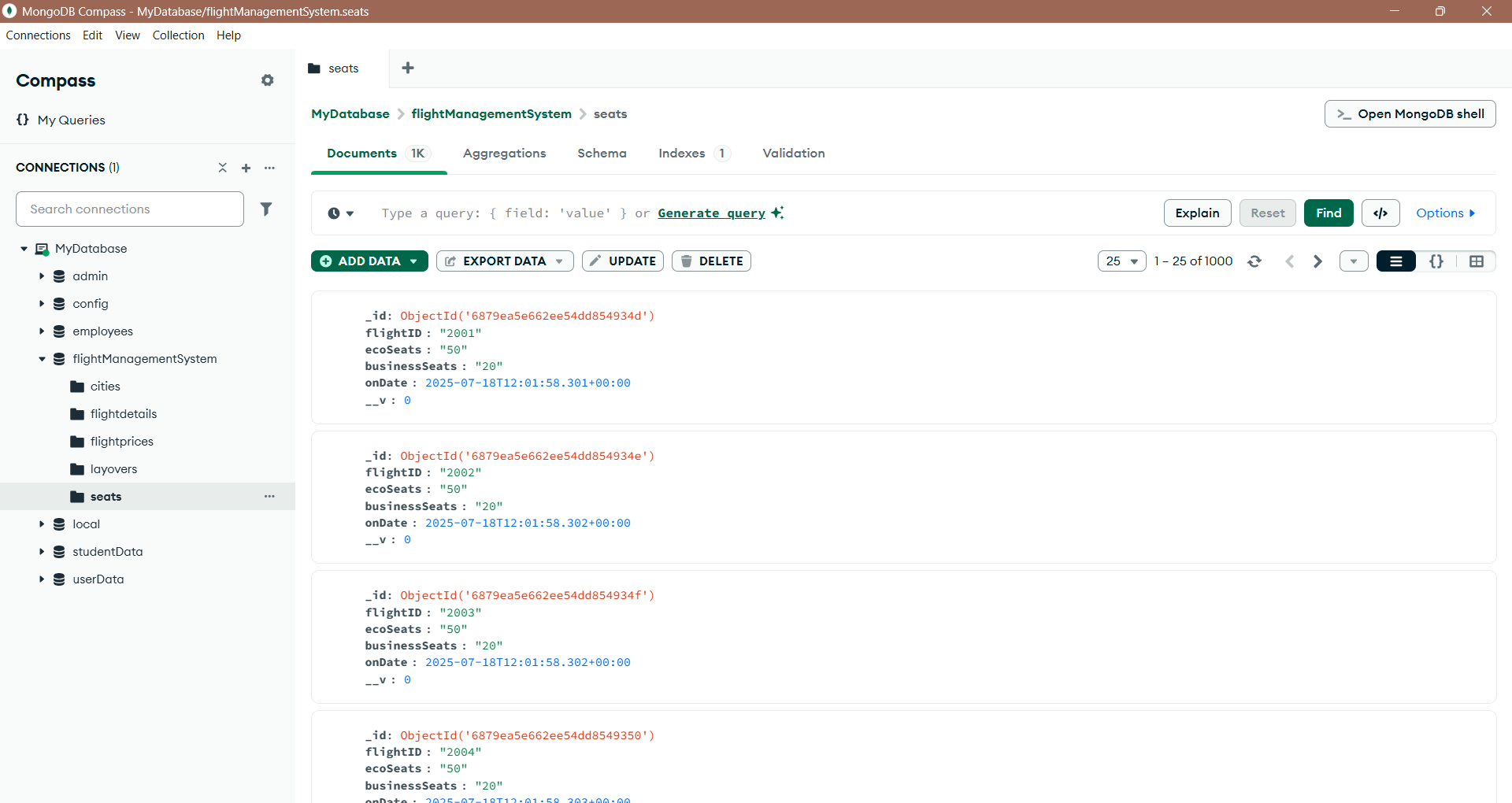
* flightprices collection



* layovers collection



* seats collection



**Data Updation**

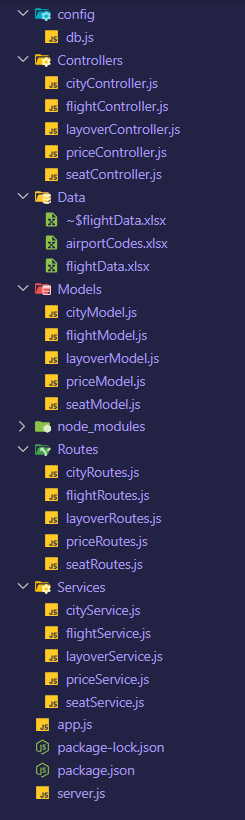
Data updation in the flight management system ensures that key fields remain current and accurate. The onDate field in the seats collection is designed to automatically capture the current date (IST) at the time of insertion, reflecting real-time availability. Other fields such as ecoSeats and **businessSeat** are dynamically updated. whenever a booking is made—deduction of required number of seats per transaction. Similarly, updates can be performed on flightprice, layovers, or flightdetails collections when schedules, pricing, or routing information change. This ensures the system maintains up-to-date records aligned with actual flight operations.

**Module 3**

**Folder Structure:**

I have organized my Node.js project by creating a clean folder structure to improve readability and maintainability. The main entry point is app.js, and I have separated the code into folders like:

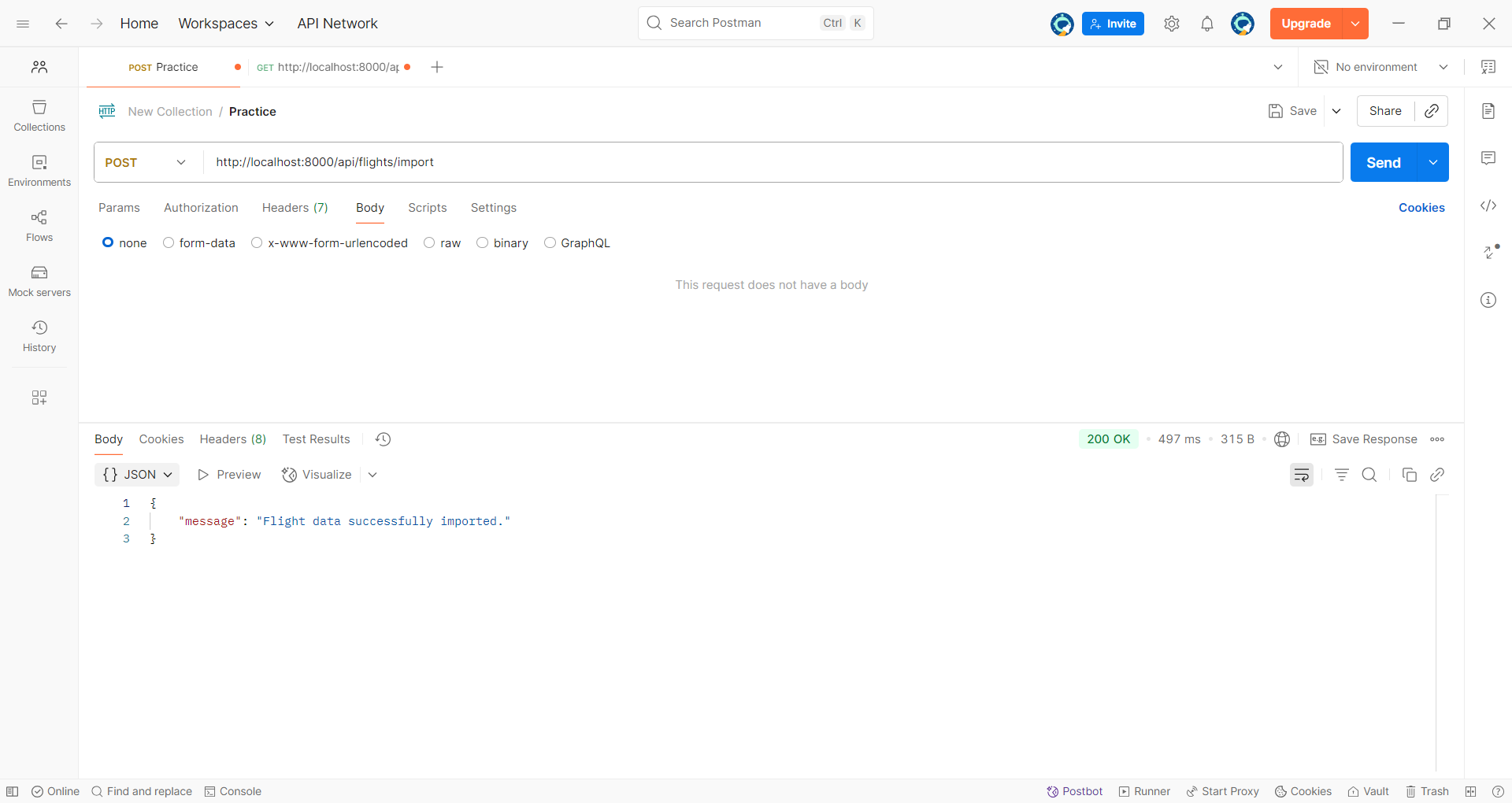
* **routes** for defining API endpoints
* **controllers** for handling request-response logic
* **services** for business logic
* **config** folder to manage the database connection setup



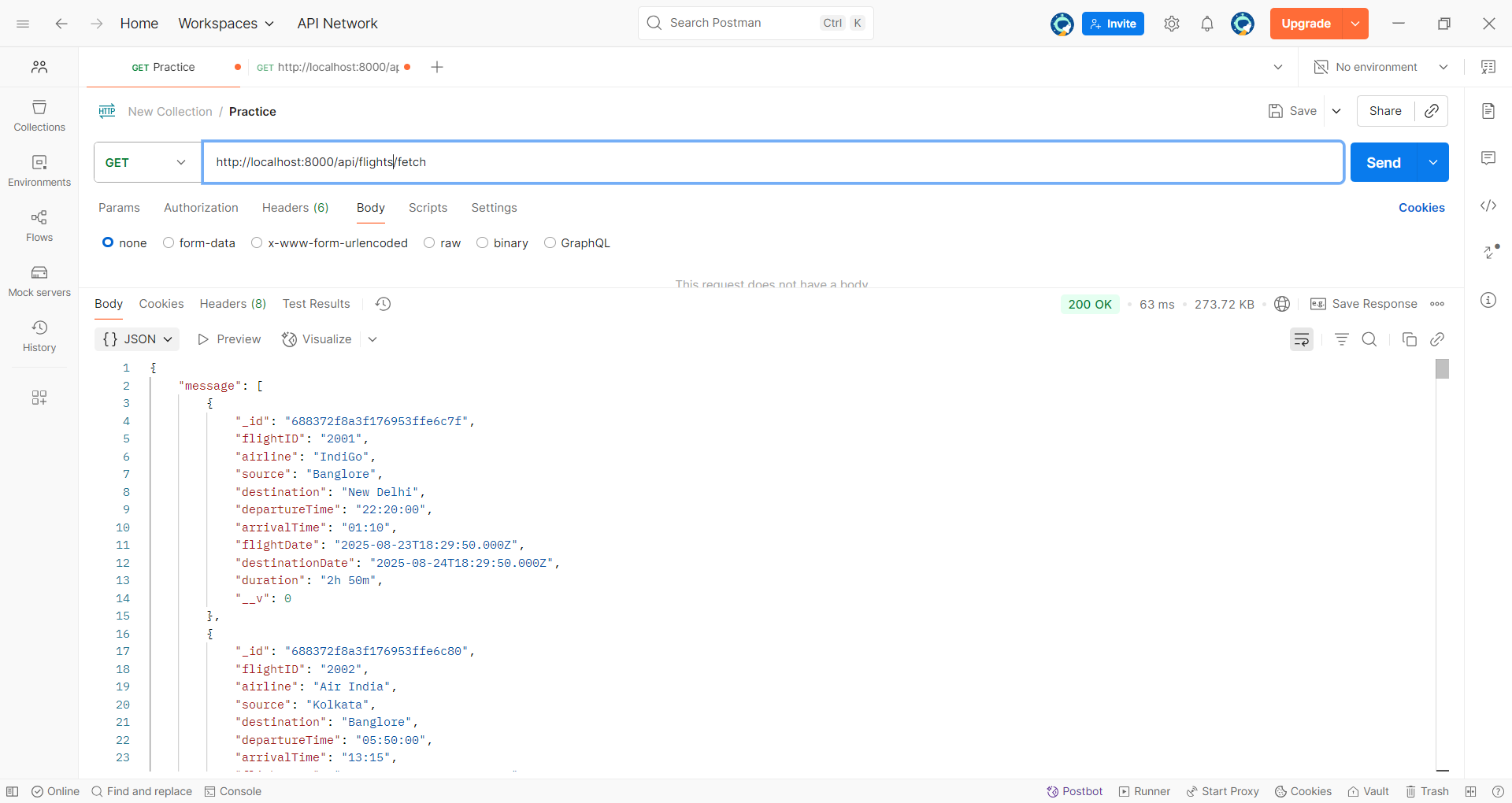
**GET/POST API Method**

I have implemented both **GET** and **POST** APIs in my project. The GET APIs are used to fetch data from the database. The POST APIs allow inserting new data into the respective collections. Each API is connected through routes, handled by **controllers**, and processed with the help of **services** for clean and modular code management.

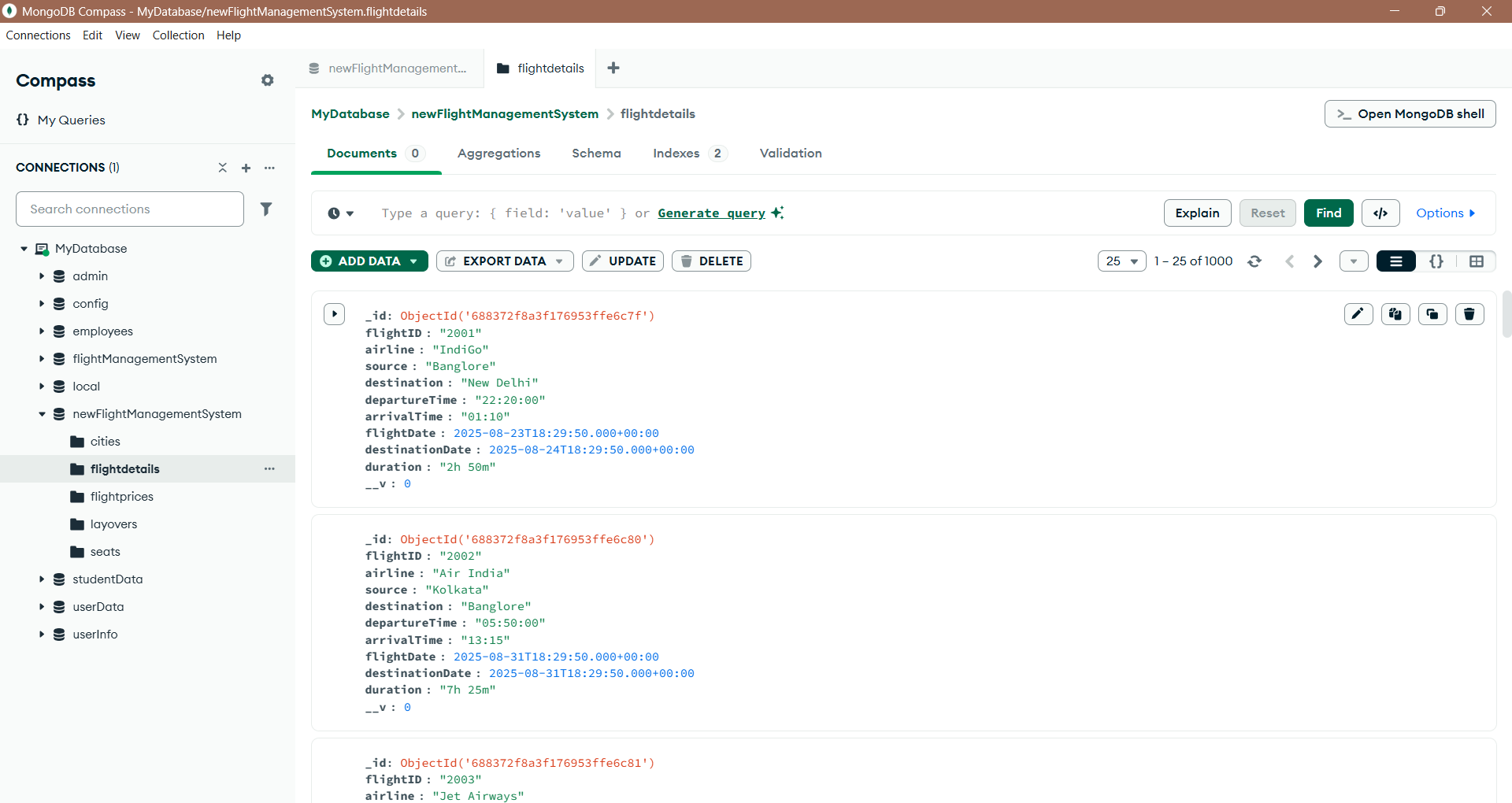
* flightDetails:
  + POST API-

****

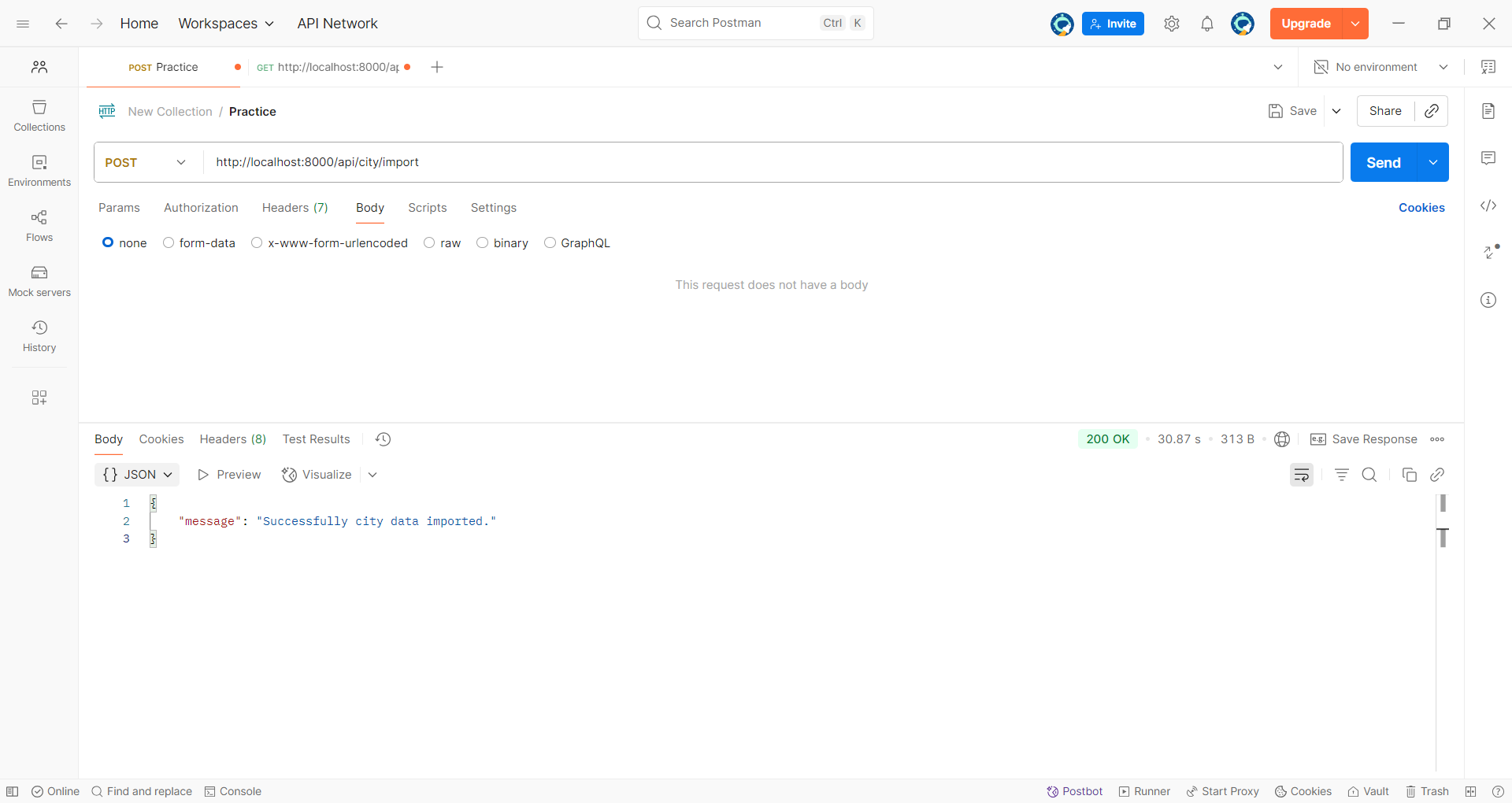
* + GET API-



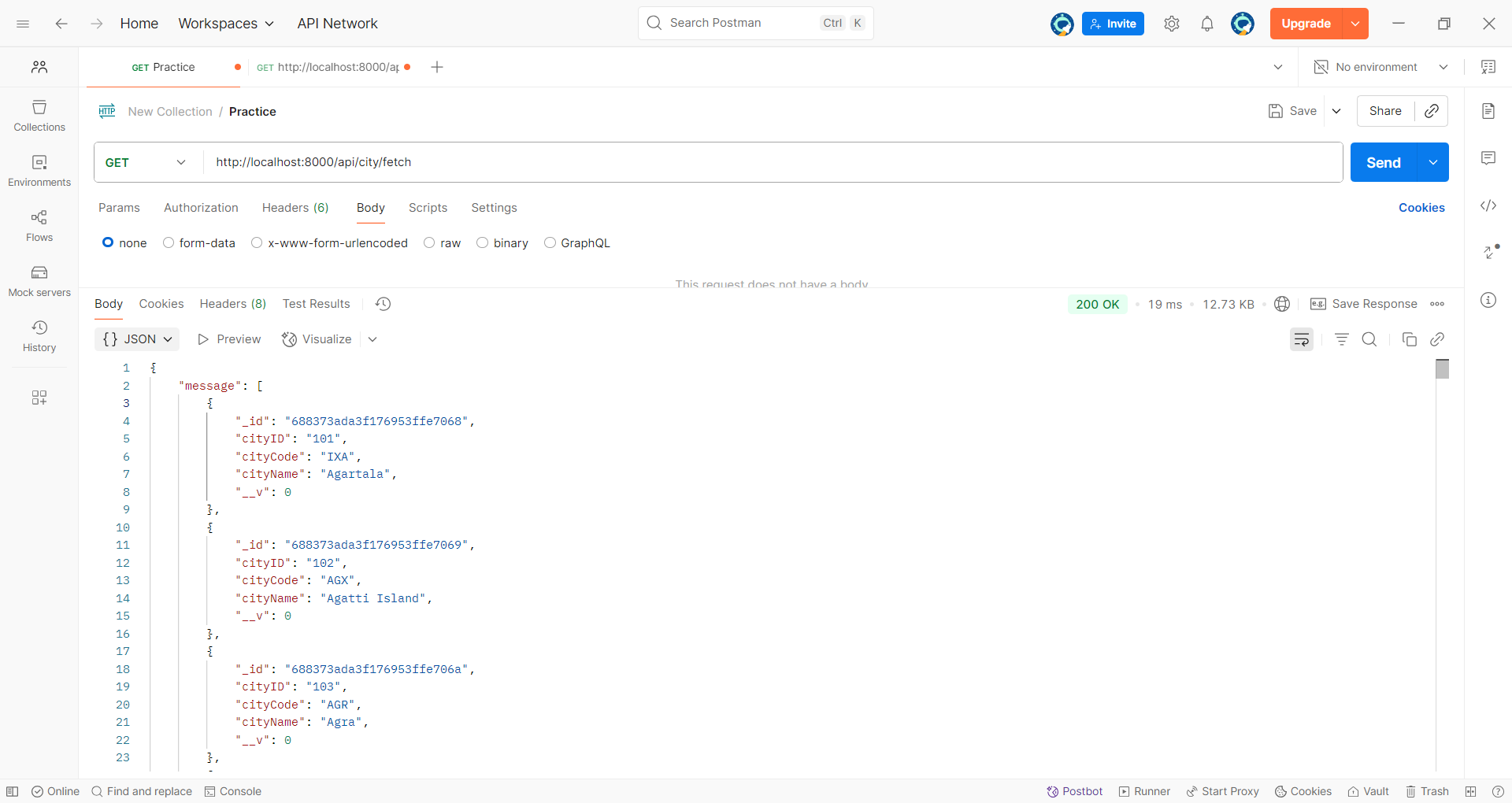
* + DB Collection-

****

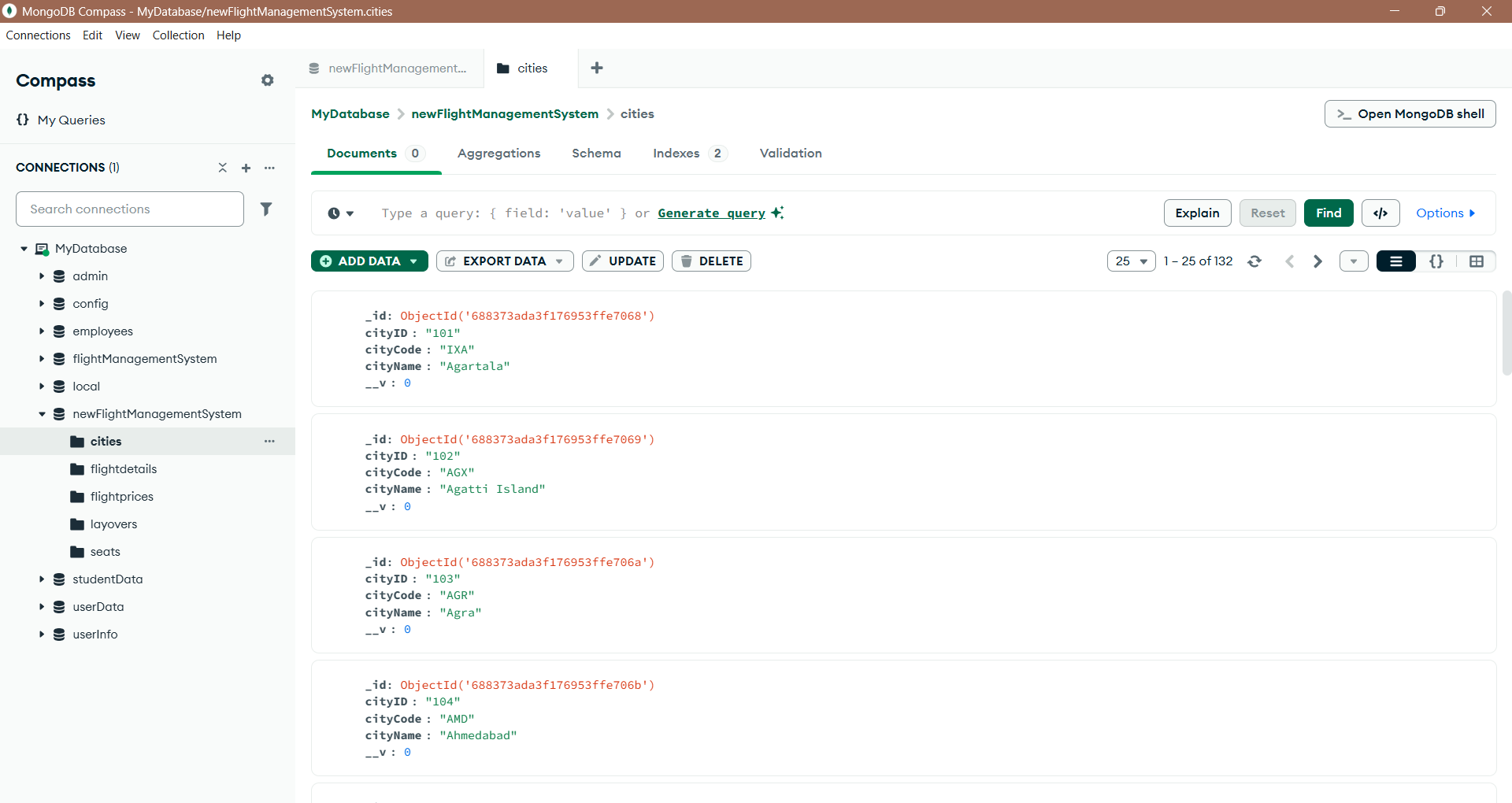
* cities:
  + POST API-

****

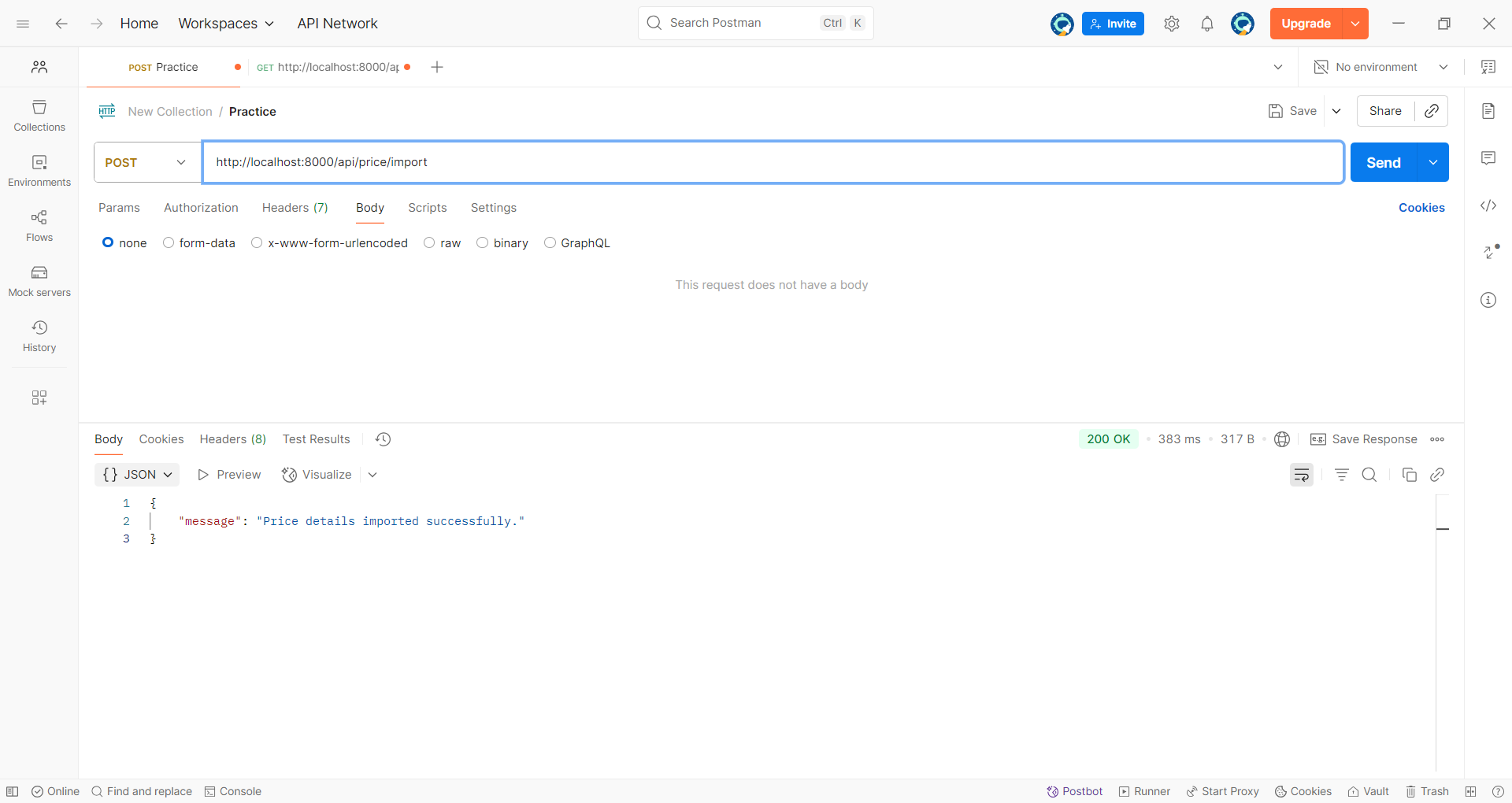
* + GET API-



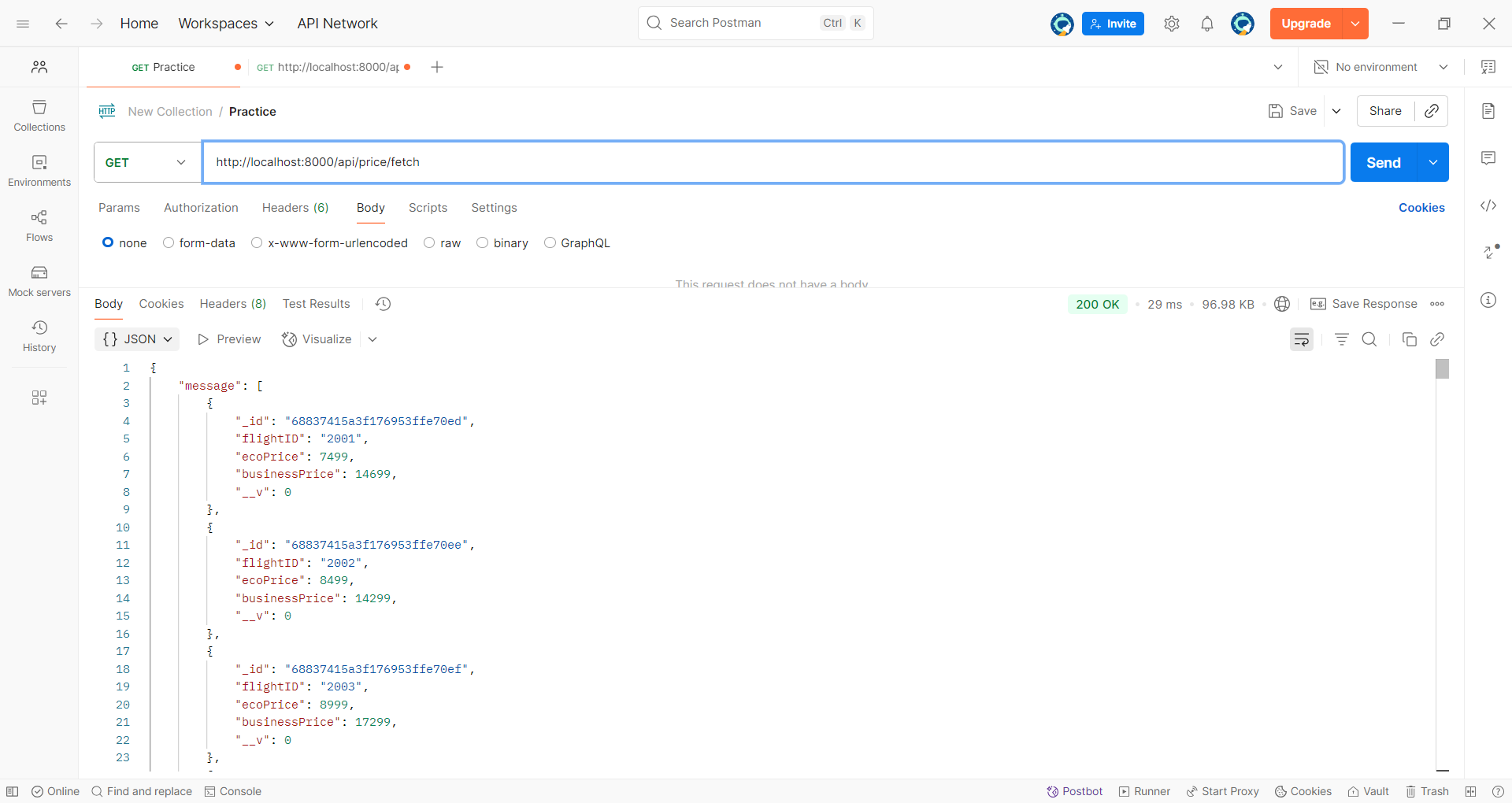
* + DB Collection-

****

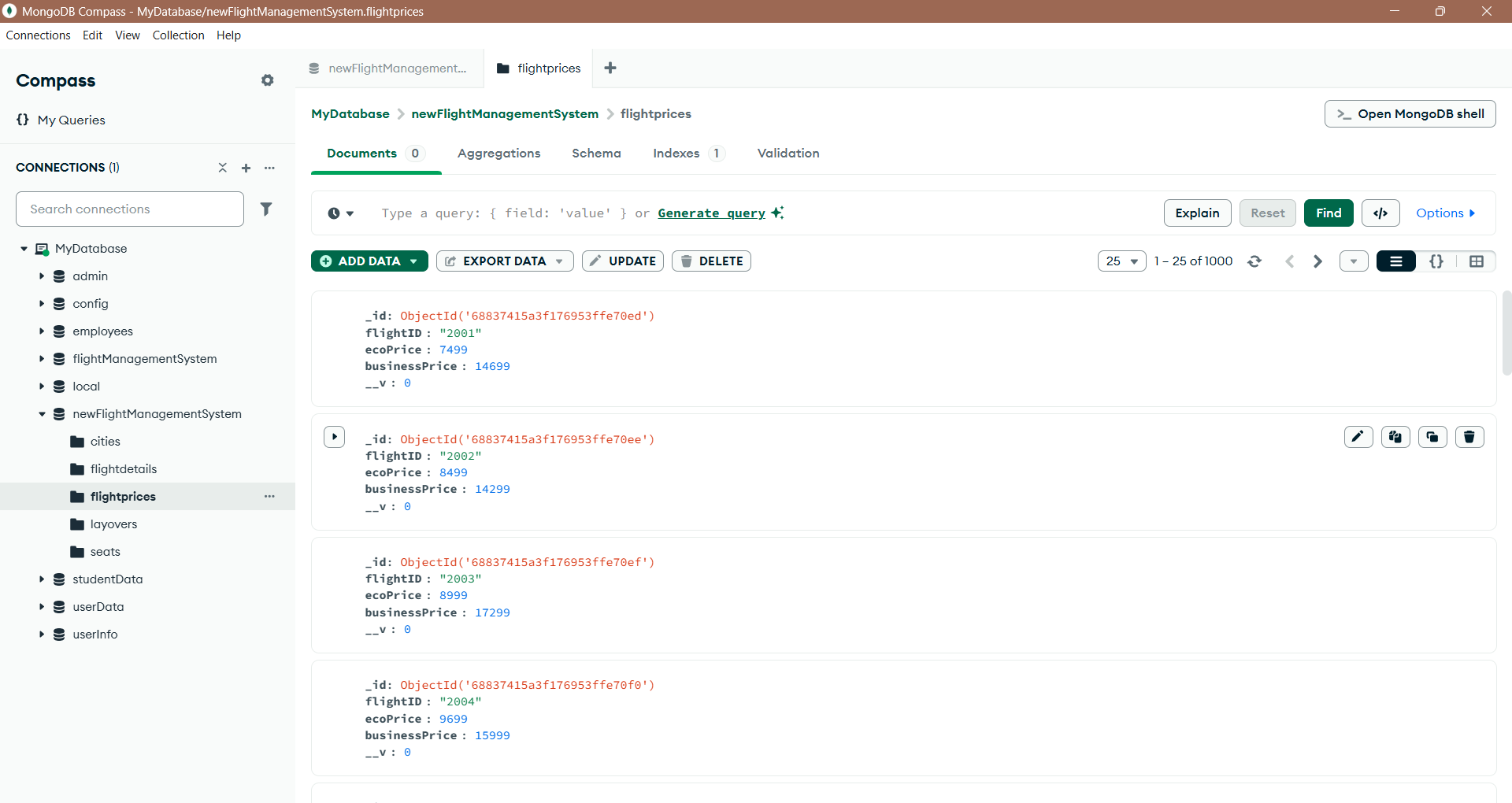
* flightPrices:
  + POST API-



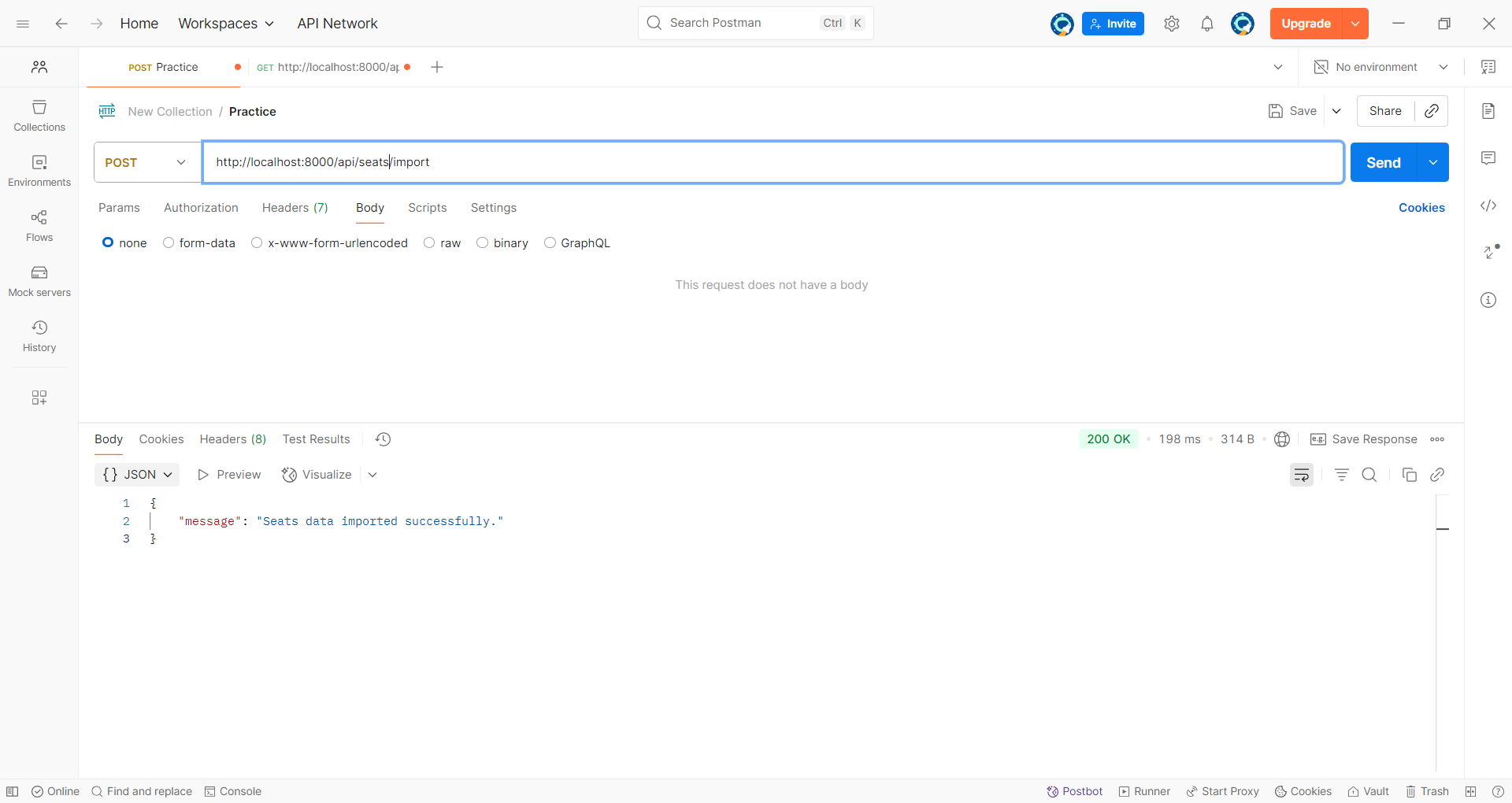
* + GET API-



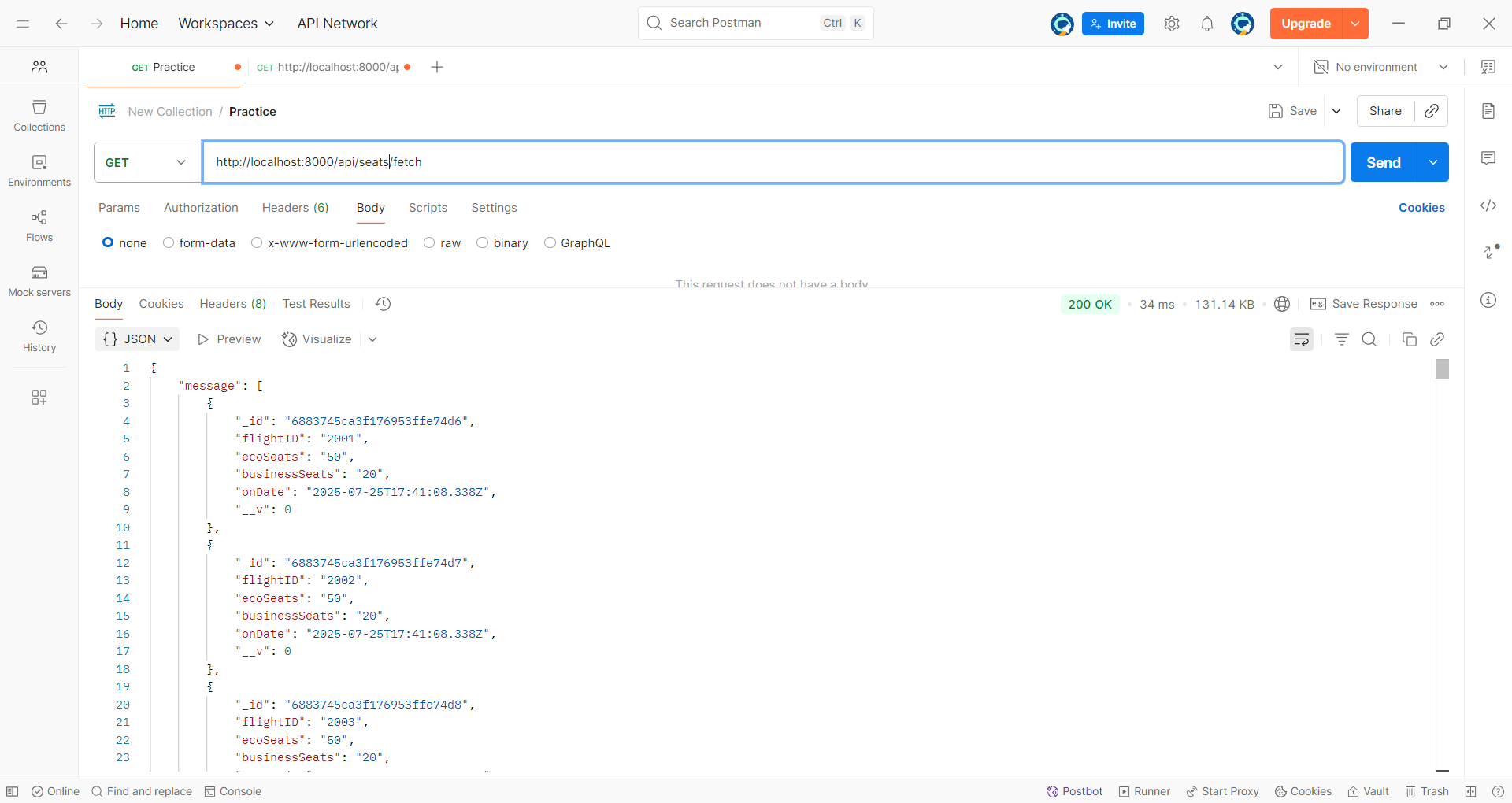
* + DB Collection-



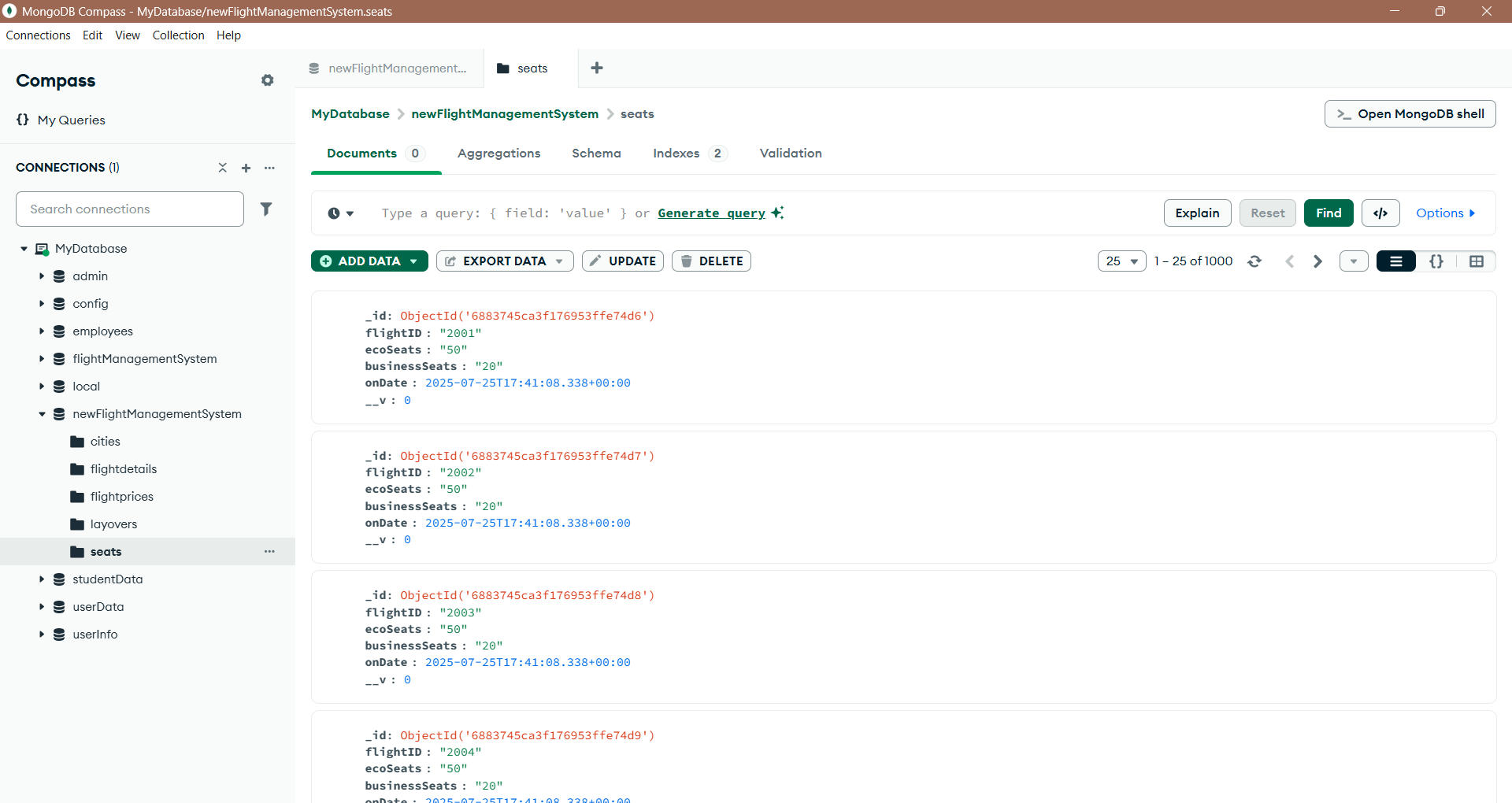
* seats:
  + POST API-



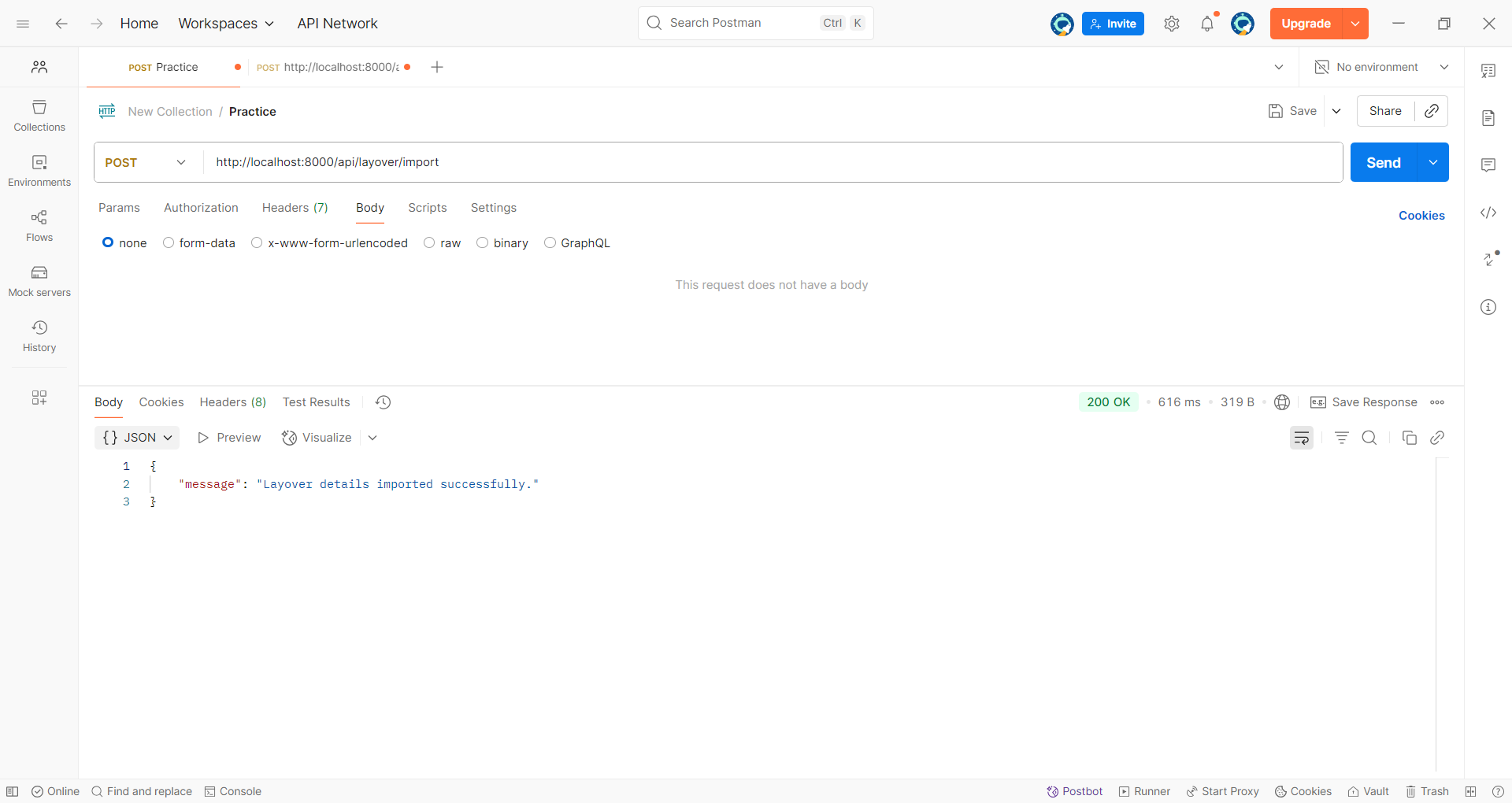
* + GET API-



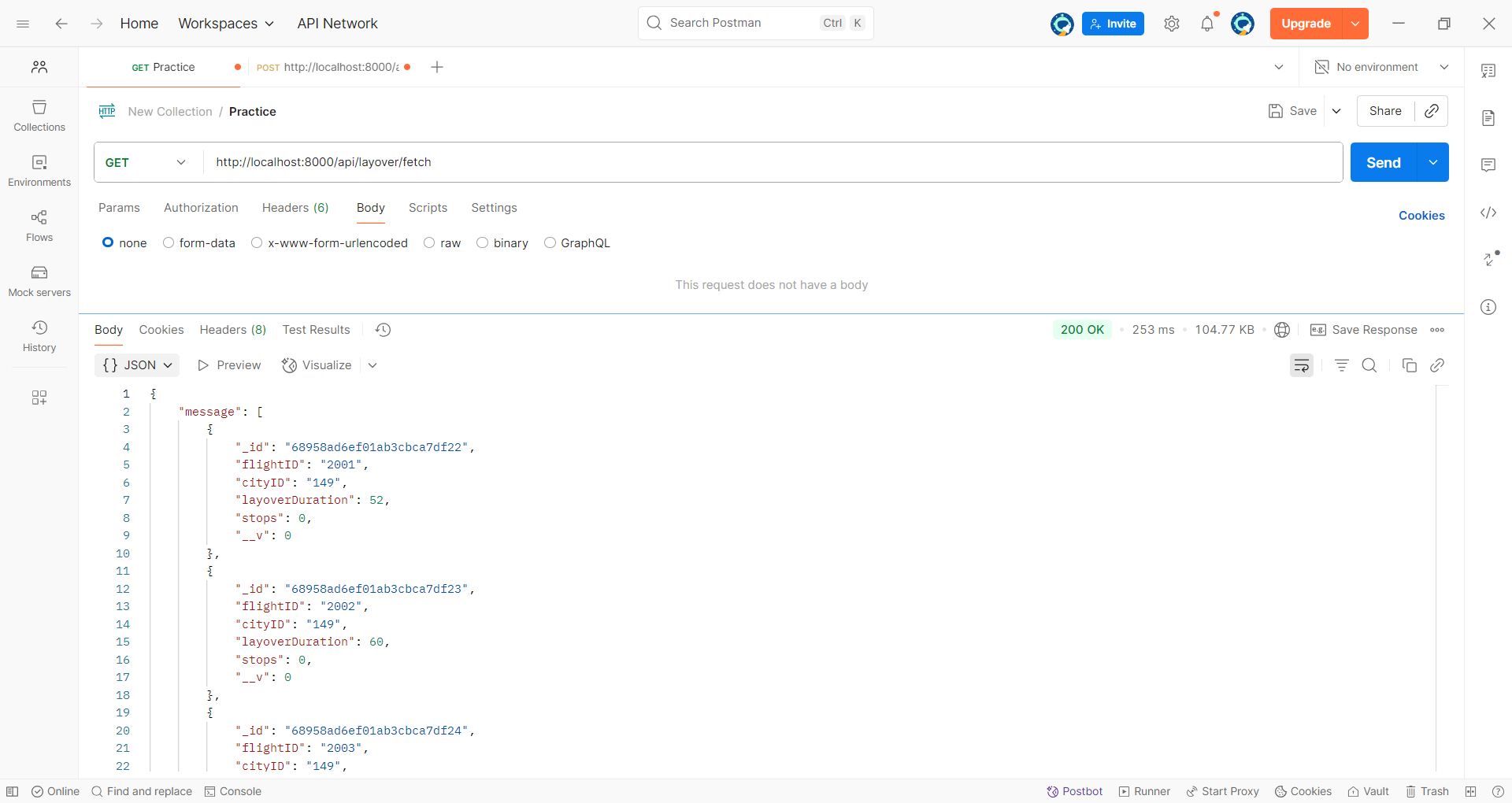
* + DB Collection-



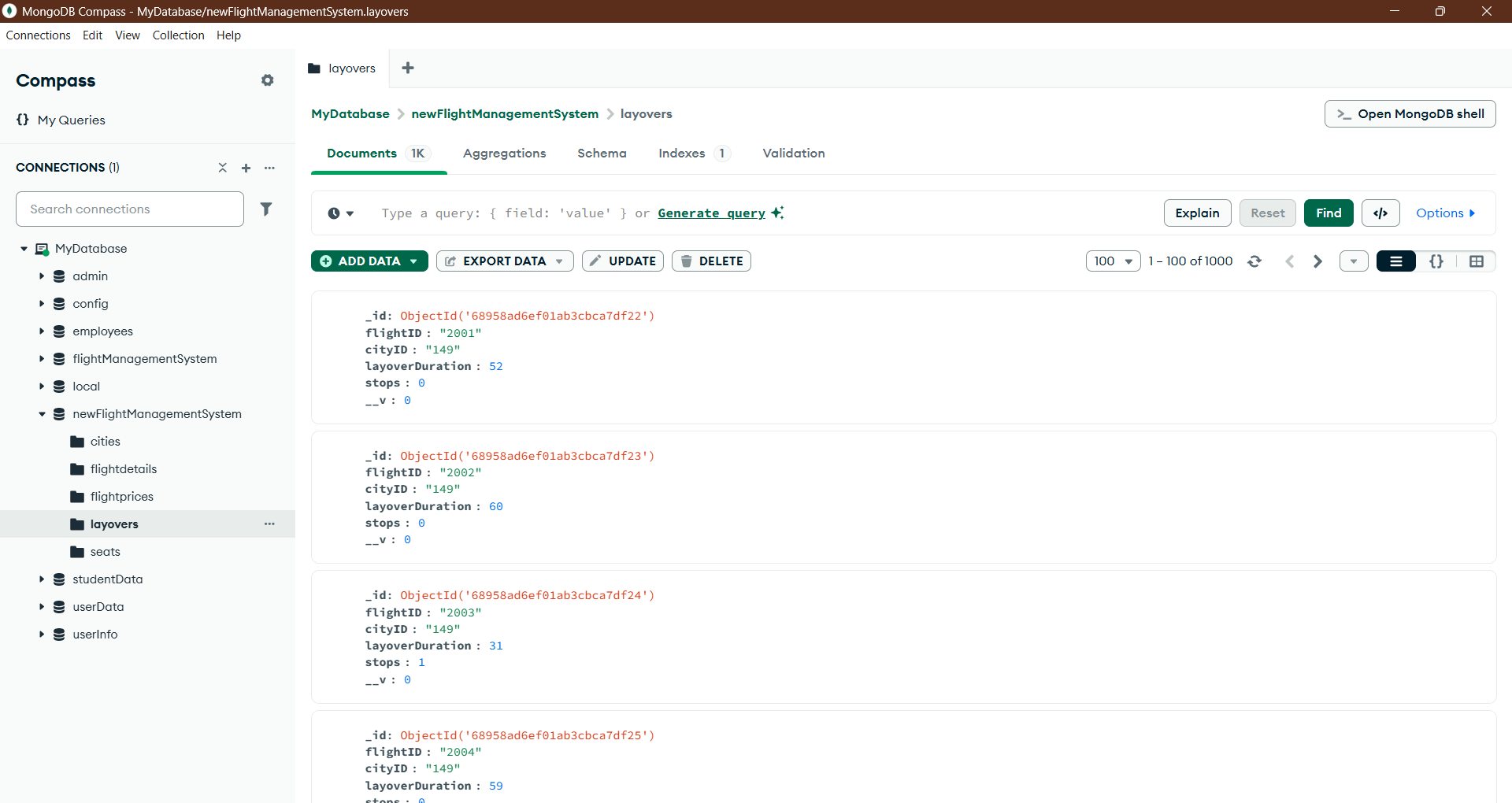
* Layover:
  + POST API-



* + GET API-



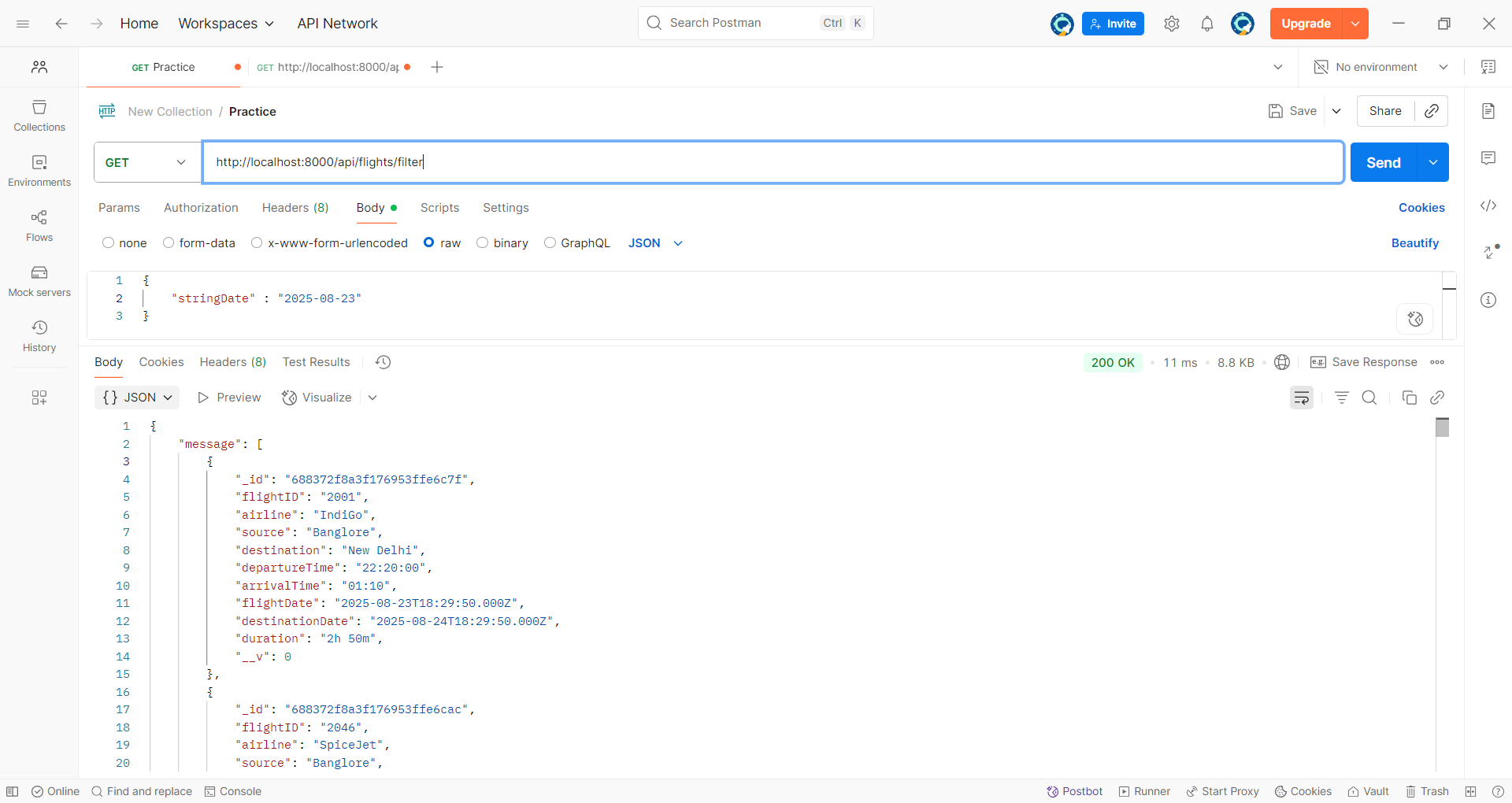
* + DB Collection-



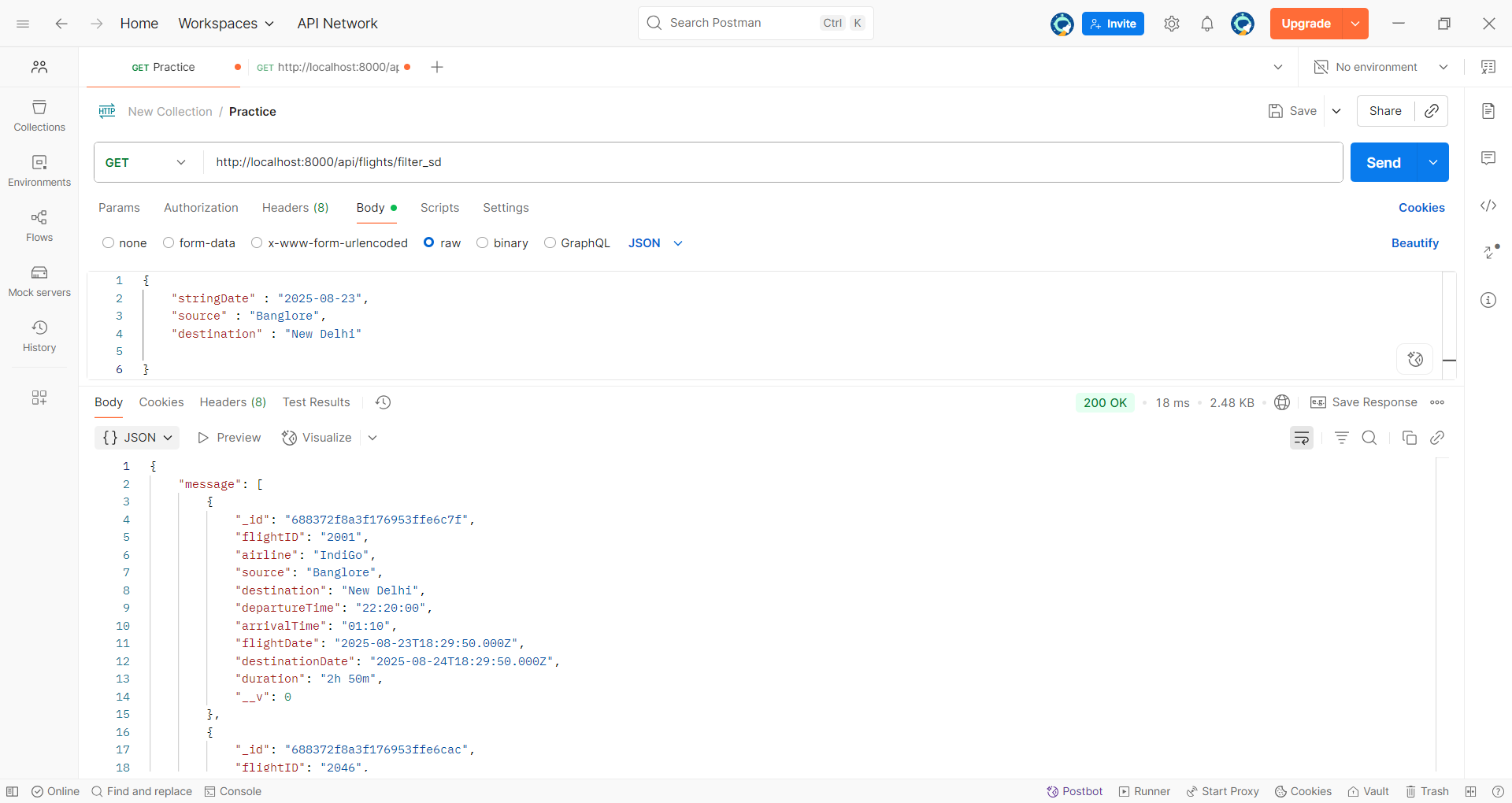
**Filters**

In the program, a date filter is implemented to retrieve flight data based on a specific date provided by the user. The user sends a date as a string (e.g., "2025-08-23"), which is then converted into a JavaScript Date object. The code sets the time range from 00:00:00.000 to 23:59:59.999 for that day to cover the entire date in IST. The converted date range is then used in a MongoDB query using **$gte** (greater than or equal to) and **$lte** (less than or equal to) operators to return all flights scheduled on that particular day. This ensures accurate filtering of records that match the complete date window.

* flightDetails:
  + Date filter-



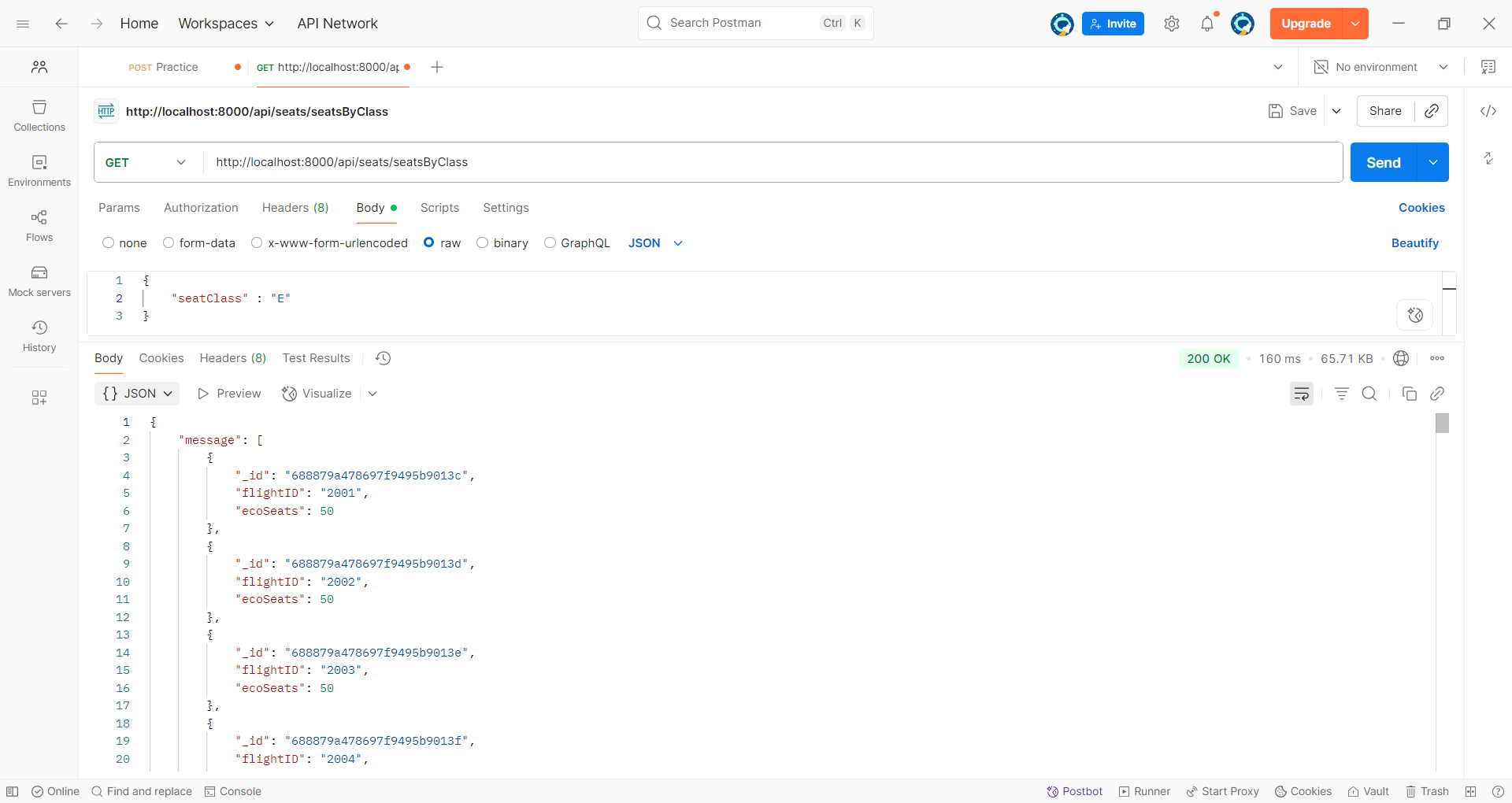
* + Source and Destination filter-



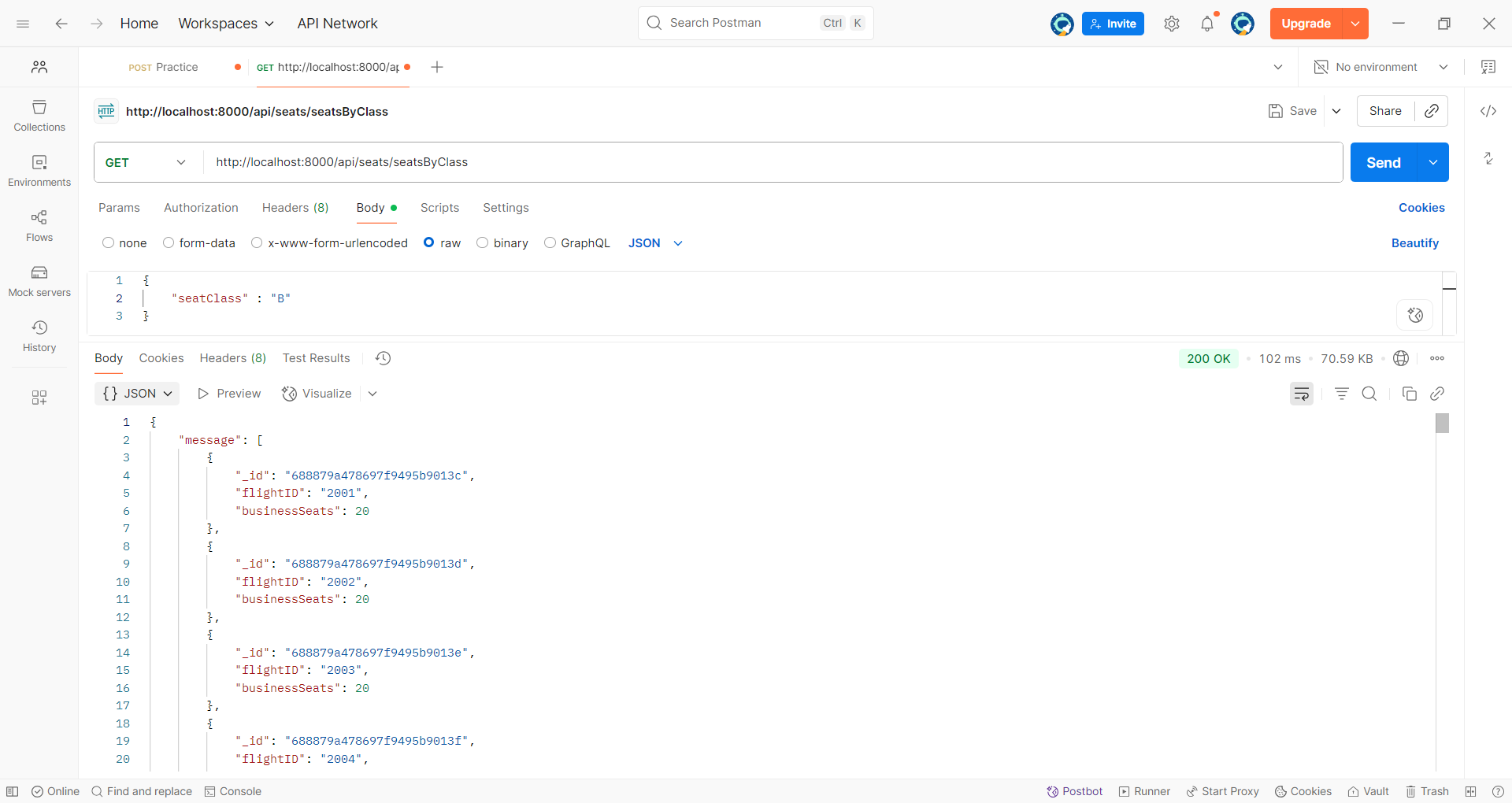
* seats:
  + Seats using flightID-



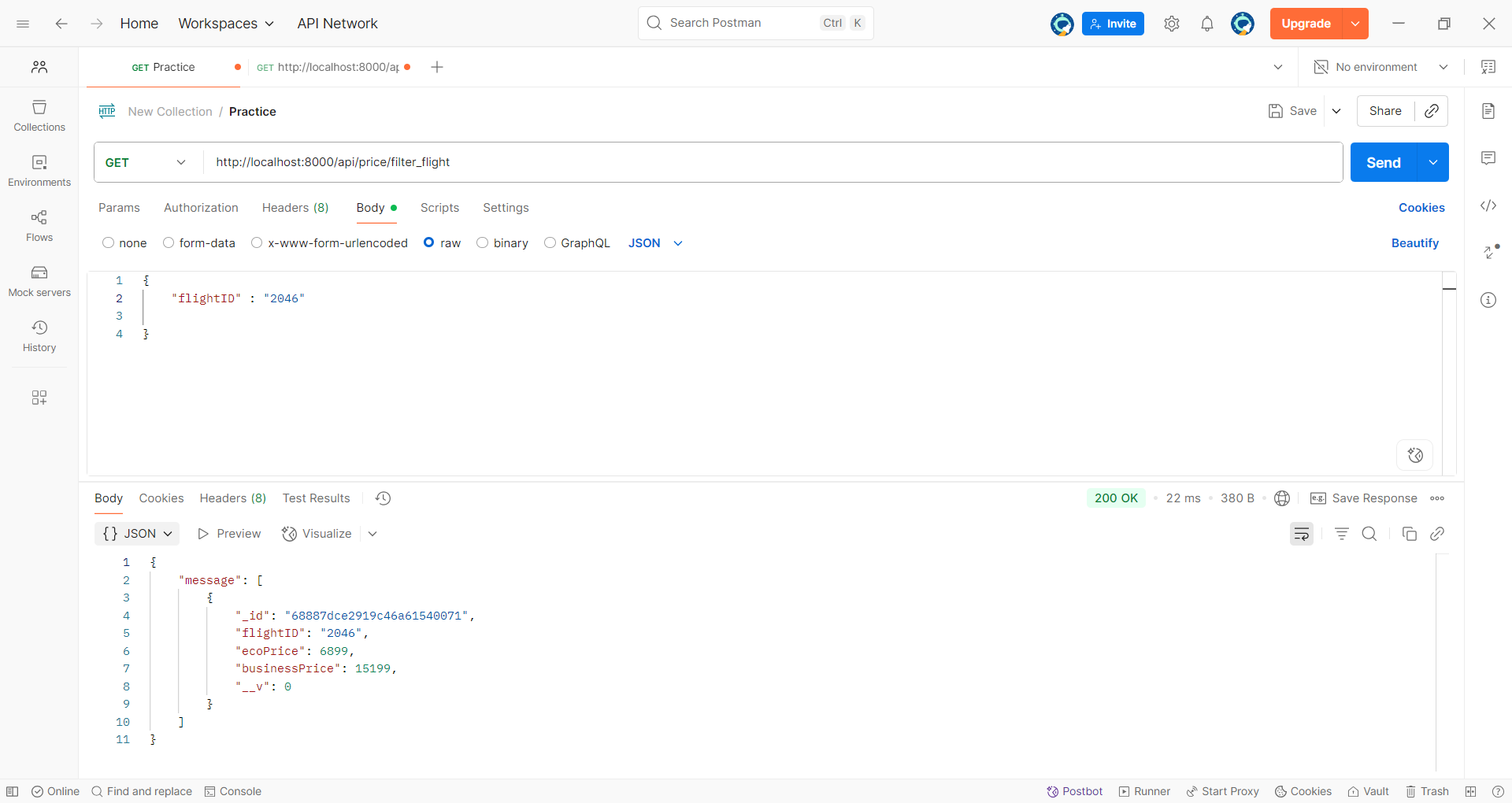
* + Seats by class-
    - Economy seats



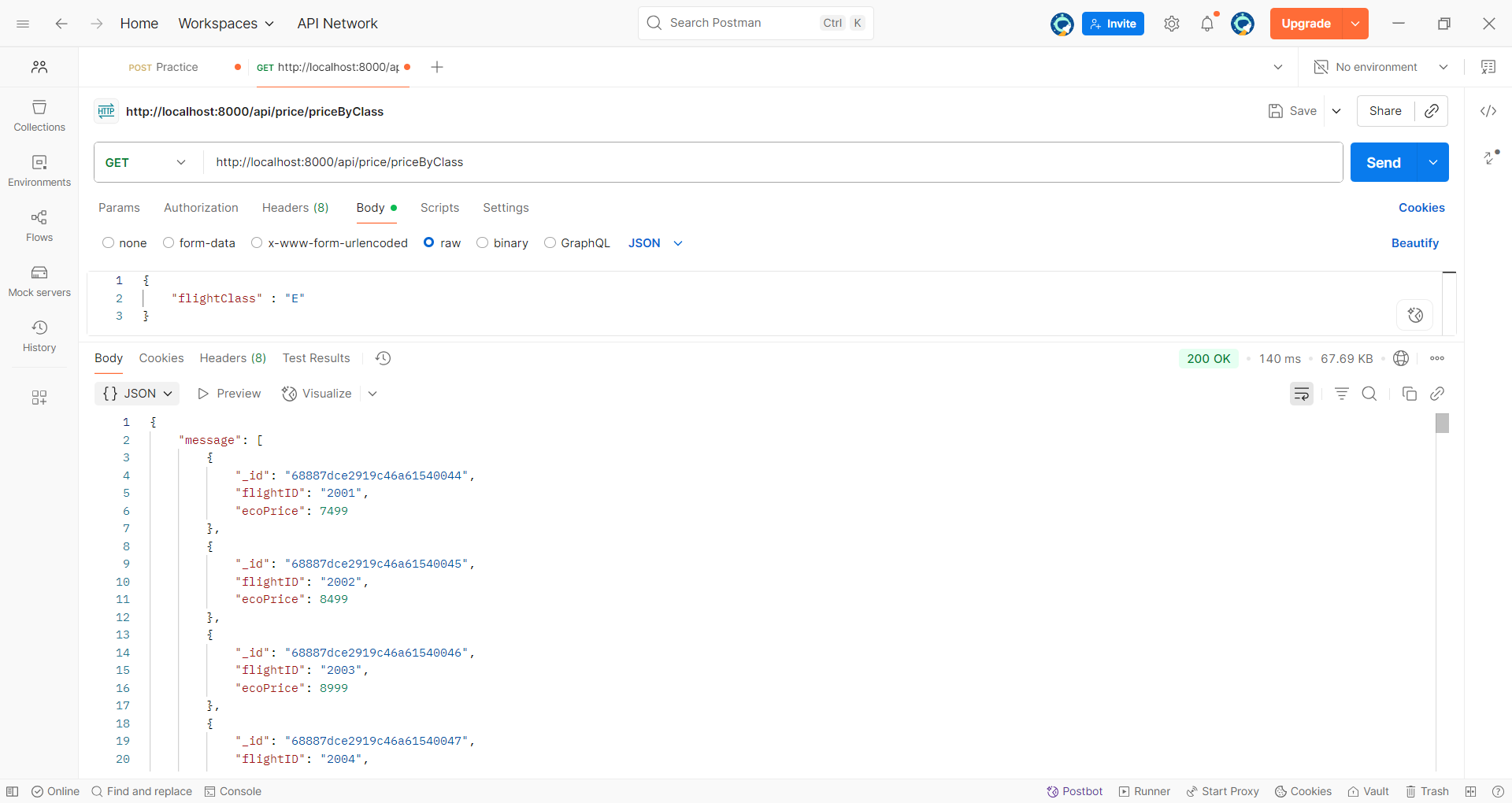
* + - Business seats



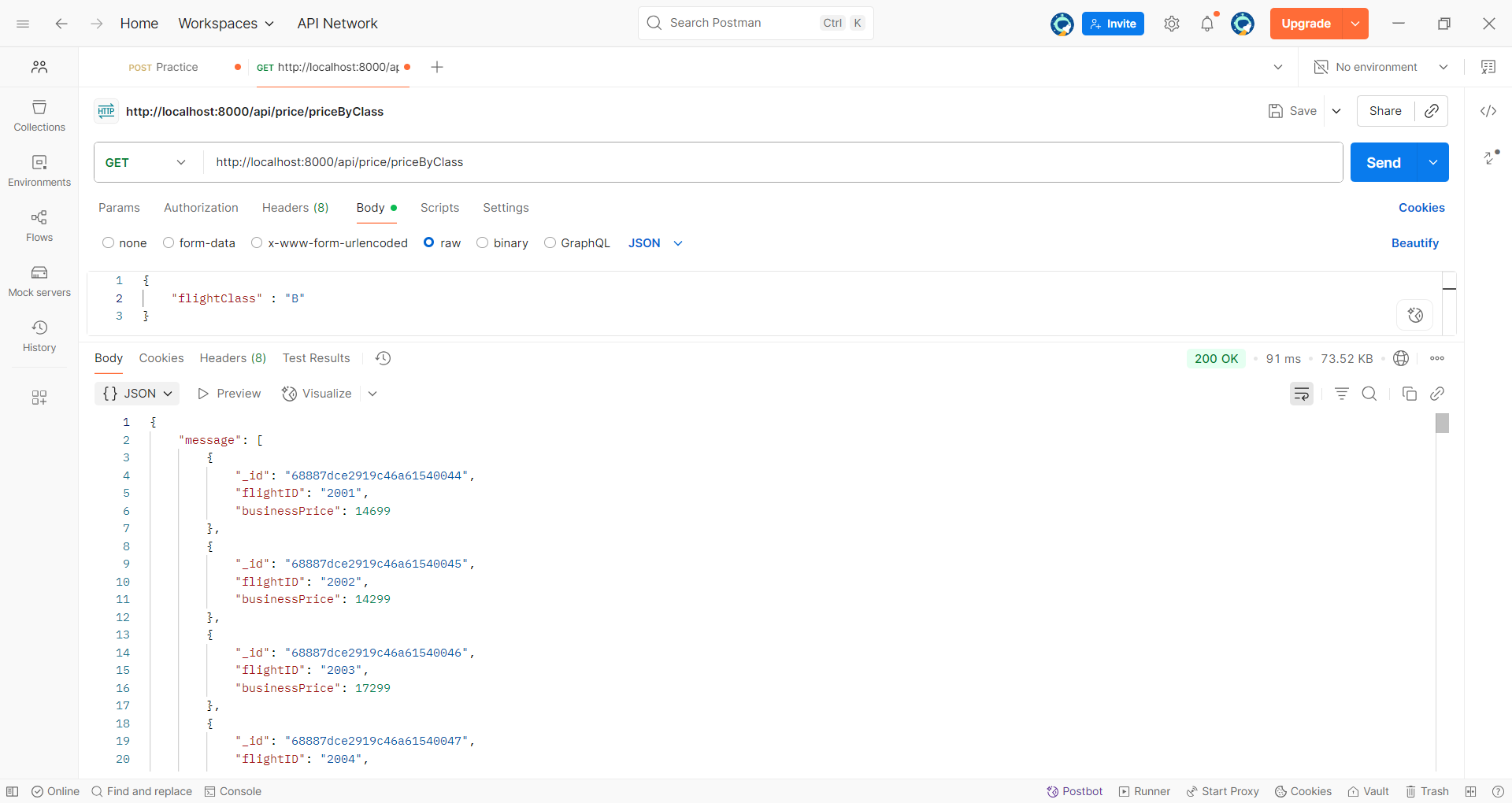
* flightPrices:
  + Price using flightID-



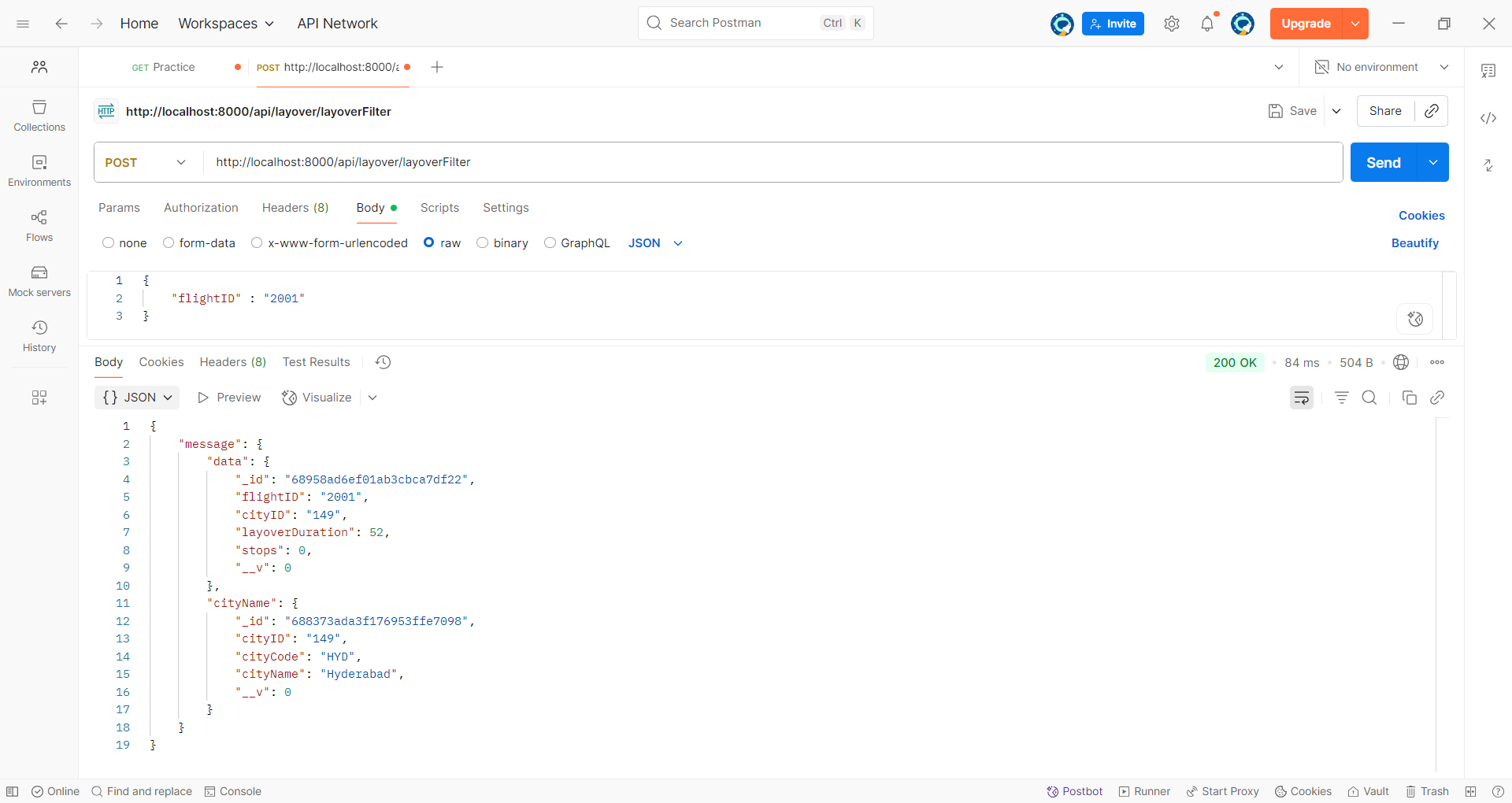
* + Price by Class-
    - Economy-



* + - Business-



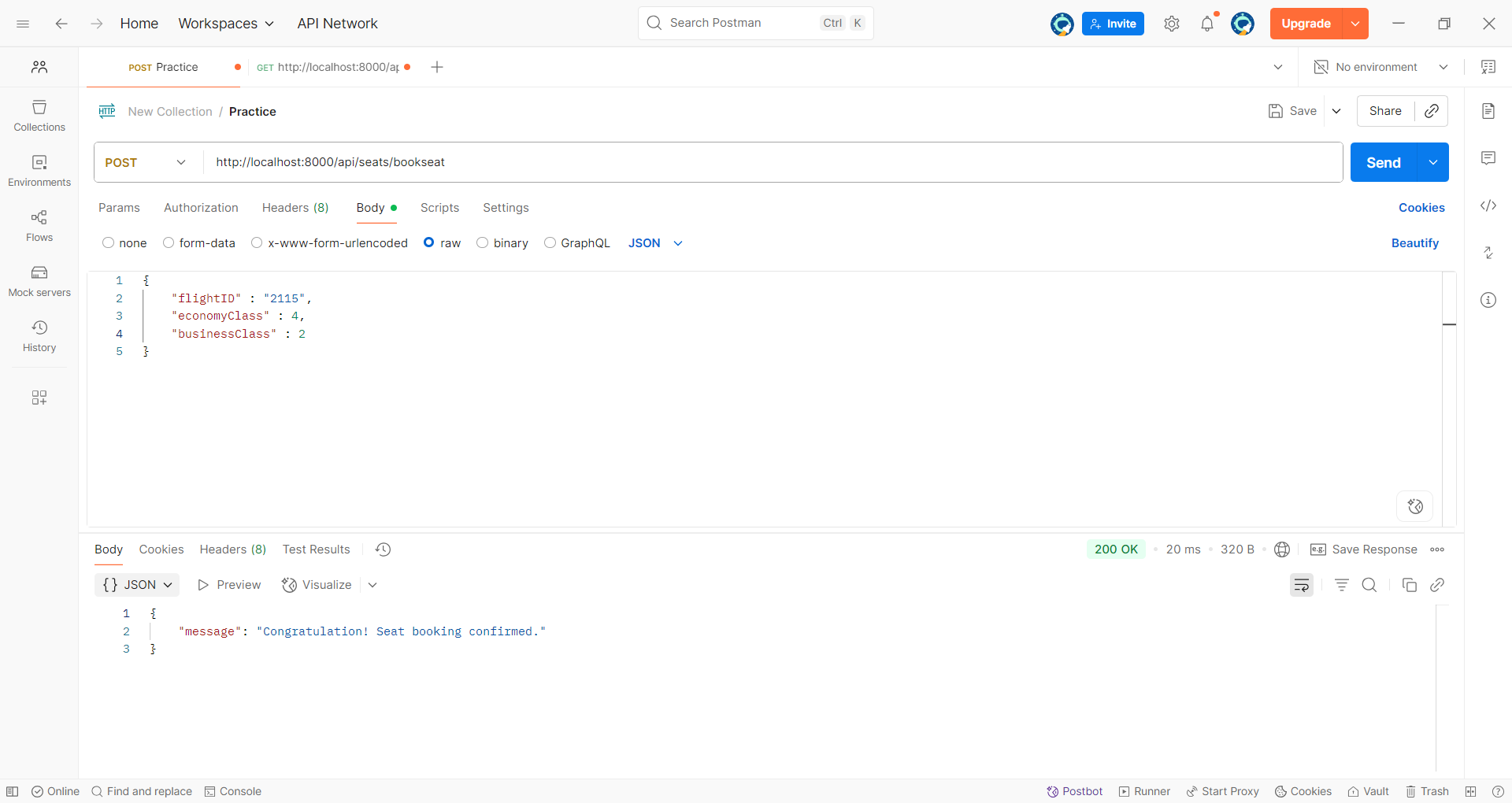
* Layover:
  + Layover by flightID



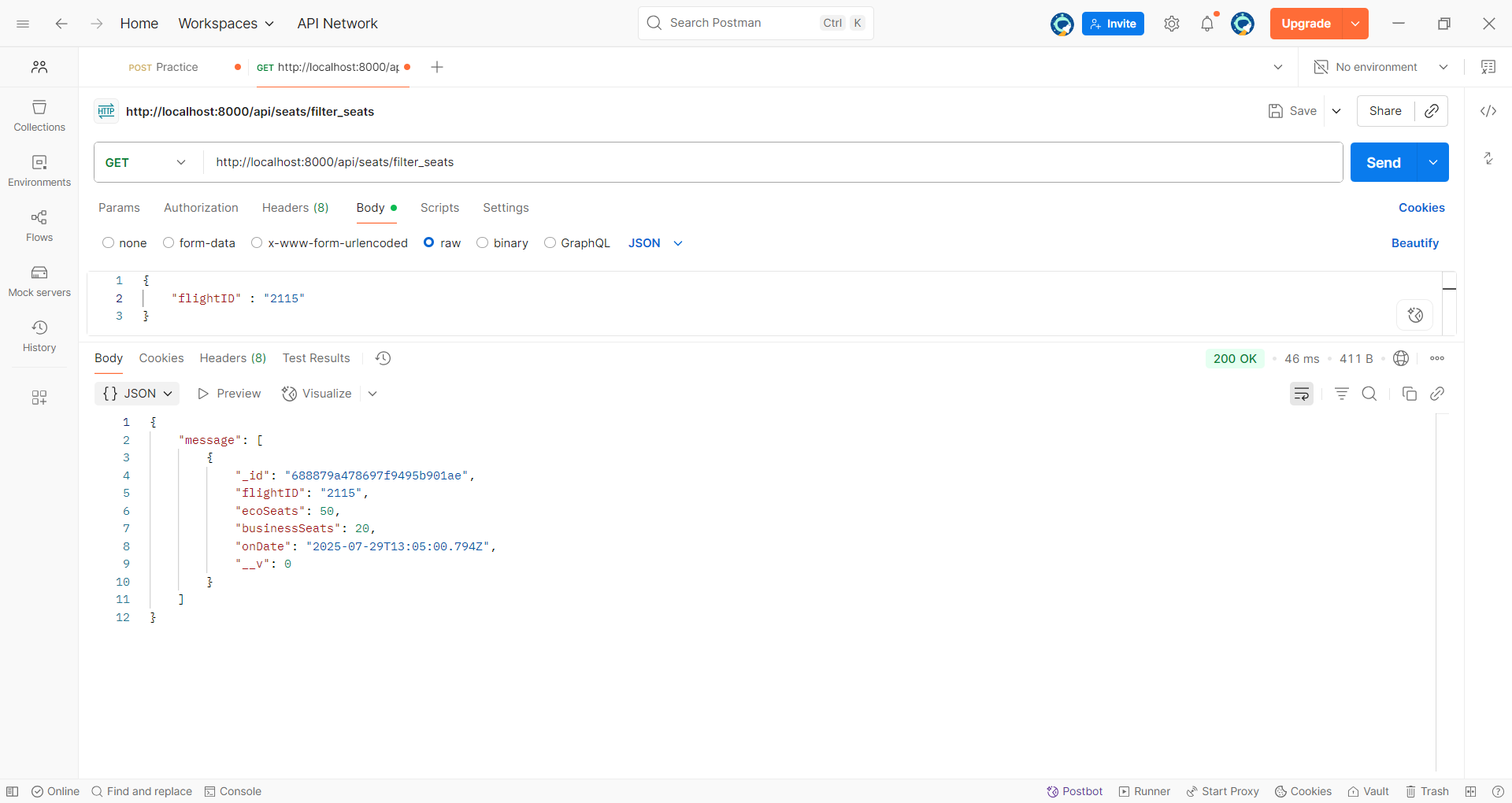
**Module 4**

**Dummy Booking API**

In the dummy booking API of the Flight Management System, the user books a flight by entering the **flightID**, selecting the class (**economy or business**), and specifying the number of **travellers**. The system checks for seat availability based on these inputs, and if the required seats are available, the booking gets **confirmed**.

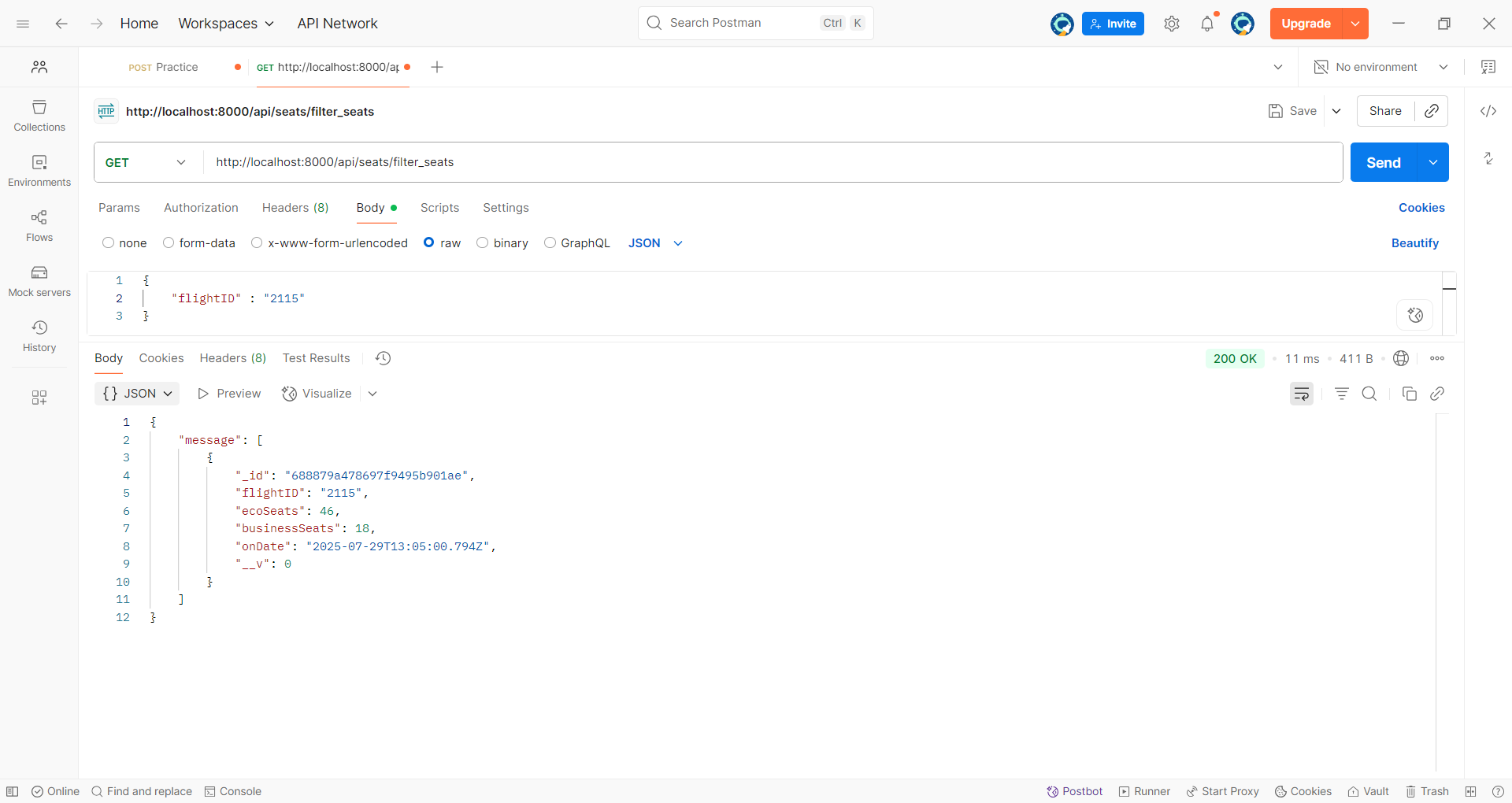


**Before update**



**After update**

After the successful transaction, the available seat count in the selected class is automatically updated in the MongoDB database, reflecting the new seat availability in real time.



**Summary**

The **Flight Management System** is a backend-centric project developed to manage core airline booking operations such as flight search, filtering, and reservation management. Built using **Node.js** and **Express.js**, the system emphasizes performance, scalability, and security while functioning entirely on the server side without a dedicated frontend. **MongoDB** serves as the database, containing structured collections such as flightDetails for storing flight information, aeroLayover for managing stopover details and layover durations, and users for storing registered user credentials. Instead of connecting to live APIs, the database is preloaded with dummy flight data to enable controlled testing of backend logic and query optimizations.

Security and authentication are key aspects of the system. User passwords are securely hashed using **bcrypt** before being stored in the users collection, ensuring that sensitive data remains protected. Authentication and authorization are implemented to restrict flight search and booking features to verified users only. The system also incorporates **CORS** middleware to handle cross-origin requests, ensuring compatibility for future integration with different frontend clients. Flight booking operations are tightly linked to the database, with the seat count in the flightDetails collection automatically decrementing upon successful booking. For flights involving stopovers, the aeroLayover collection stores the number of stops and layover durations, either set to default values or assigned randomly to simulate realistic travel scenarios.

The API endpoints are structured for flexibility and efficiency, supporting advanced filtering such as searching flights based on price, number of stops, or other attributes. All API testing and validation were conducted using the **Postman** application, ensuring that each route functioned as expected under different request scenarios. Postman collections were also created for organized testing of authentication routes, flight search, booking, and layover management endpoints.

By focusing purely on backend development, this project showcases skills in API design, database management, and secure authentication workflows. It demonstrates the ability to build a robust server-side architecture ready for integration with any frontend interface, while maintaining clean, maintainable, and extensible code for future enhancements such as real-time API integration, payment gateways, or notification services.

**Thank You**