

Summer Internship Project

(June 11, 2025 – August 11, 2025)

# Flight Booking Management System

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

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



– Innovate • Integrate • Transform –

Singapore | Delhi | Mumbai | Pune  
Bangalore | Lucknow | Jaipur | Goa



# INDEX

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

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

### Testing & Simulation Tools

Tool	Purpose
<b>Postman</b>	Used to test and verify all backend API endpoints (e.g., login, signup, flight search) without a frontend UI

### Development Tools

Tool	Purpose
<b>Visual Studio Code</b>	Used as the primary code editor for writing and managing the Node.js backend code
<b>MongoDB Compass</b>	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

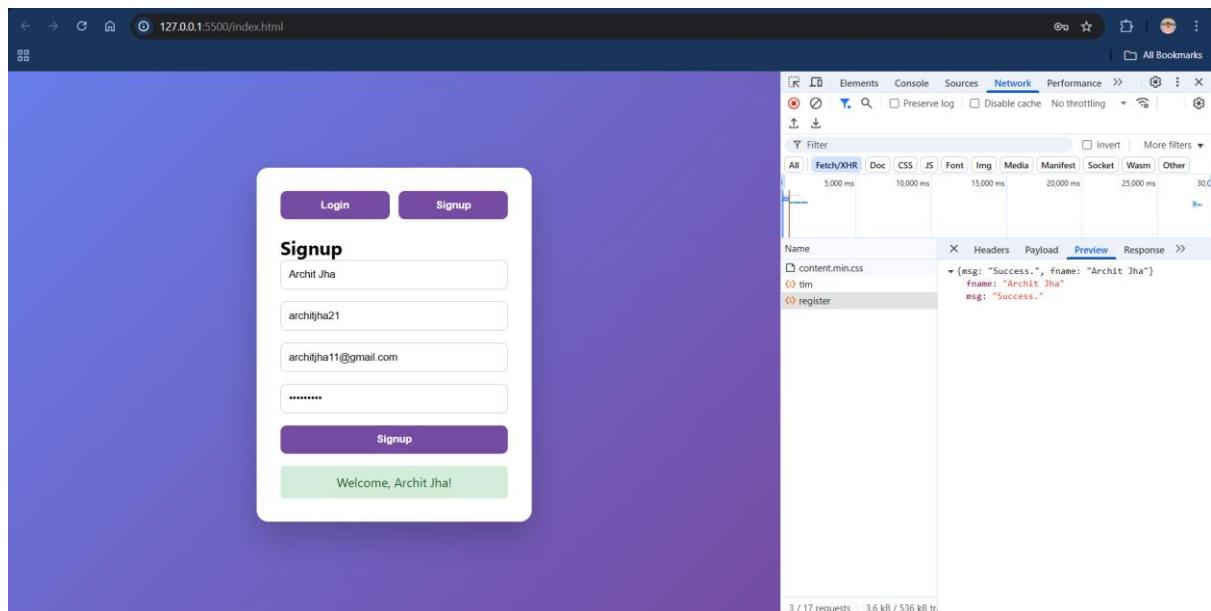
- Method: POST
  - Endpoint: 8000/api/login
  - The login endpoint verifies a user's **email and password**.
  - If the email exists and the password matches (checked using `bcrypt.compare()`):
    - A **JWT (JSON Web Token)** is generated and sent back in the response.
    - This token is used to authenticate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

## Access Control Using JWT

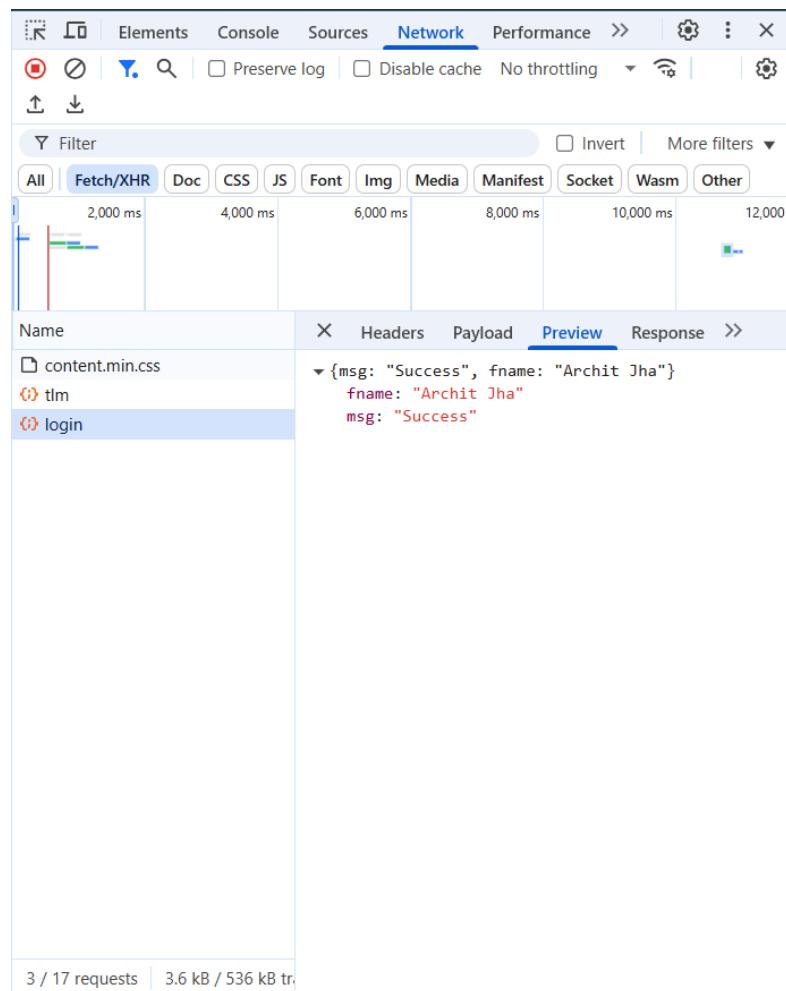
- The system uses **middleware** to verify JWT tokens.
  - Only users with valid tokens can access protected routes like /search-flights.
  - This ensures that unauthenticated users cannot access sensitive data.
- 

## 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

```
test> use userData
switched to db userData
userData> db.users.find({})
[
  {
    _id: ObjectId('686caf3121e39c0195f7acc0'),
    username: 'architjha21',
    fname: 'Archit Jha',
    emailid: 'architjhalla@gmail.com',
    password: '$2b$10$rLM.Zdoj5EbSIgbli42LiedECnyVj.VcZN7xdTnbFU56FCRu5./VK',
    role: 'user',
    __v: 0
  },
  {
    _id: ObjectId('686cb3ae894f5c77bb4951b8'),
    username: 'anurag123',
    fname: 'Anurag',
    emailid: 'anuragkumeri123@gmail.com',
    password: '$2b$10$loostd.nLLyhy3lDC1YdtOauRMjETTvz8Fz9V9tqBurZg0iJs2u1.',
    role: 'user',
    __v: 0
  }
]
userData> |
```

# 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:

A screenshot of Microsoft Excel showing a spreadsheet titled "flightData1". The table has 30 rows of data and approximately 25 columns. The columns include: FlightDate, Airline, Origin, Dest, Cancelled, Diverted, CRSDepTir, DepTime, DepDelay, DepDelay, ArrTime, ArrDelay, AirTime, CRSElapse, ActualElap, Distance, Year, Quarter, Month, DayoffMor, DayOffWer, Marketing, Operated, DOT\_ID, IATA\_C, and DL\_CODES. The data shows various flight records with specific details like arrival and departure times, distances, and operational status.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	
1	FlightDate	Airline	Origin	Dest	Cancelled	Diverted	CRSDepTir	DepTime	DepDelay	DepDelay	ArrTime	ArrDelay	AirTime	CRSElapse	ActualElap	Distance	Year	Quarter	Month	DayoffMor	DayOffWer	Marketing	Operated	DOT_ID	IATA_C	DL_CODES
2	23-01-2018	Endeavor	ABY	ATL	FALSE	FALSE	1202	1157	0	-5	1256	0	38	62	59	145	2018	1	1	23	2 DL	DL_CODES	19790	DL		
3	24-01-2018	Endeavor	ABY	ATL	FALSE	FALSE	1202	1157	0	-5	1258	0	36	62	61	145	2018	1	1	24	3 DL	DL_CODES	19790	DL		
4	25-01-2018	Endeavor	ABY	ATL	FALSE	FALSE	1202	1153	0	-9	1302	0	40	62	69	145	2018	1	1	25	4 DL	DL_CODES	19790	DL		
5	26-01-2018	Endeavor	ABY	ATL	FALSE	FALSE	1202	1150	0	-12	1253	0	35	62	63	145	2018	1	1	26	5 DL	DL_CODES	19790	DL		
6	27-01-2018	Endeavor	ABY	ATL	FALSE	FALSE	1400	1355	0	-5	1459	0	36	60	64	145	2018	1	1	27	6 DL	DL_CODES	19790	DL		
7	28-01-2018	Endeavor	ABY	ATL	FALSE	FALSE	1202	1202			1326	22	37	62	84	145	2018	1	1	28	7 DL	DL_CODES	19790	DL		
8	29-01-2018	Endeavor	ABY	ATL	FALSE	FALSE	1202	1204	2	2	1303	0	34	62	59	145	2018	1	1	29	1 DL	DL_CODES	19790	DL		
9	30-01-2018	Endeavor	ABY	ATL	FALSE	FALSE	1202	1153	0	-9	1255	0	44	62	62	145	2018	1	1	30	2 DL	DL_CODES	19790	DL		
10	31-01-2018	Endeavor	ABY	ATL	FALSE	FALSE	1202	1153	0	-9	1304	0	37	62	71	145	2018	1	1	31	3 DL	DL_CODES	19790	DL		
11	03-02-2018	Endeavor	ATL	ABY	FALSE	FALSE	1037	1101	24	24	1159	22	32	60	58	145	2018	1	1	3	3 DL	DL_CODES	19790	DL		
12	04-02-2018	Endeavor	ATL	ABY	FALSE	FALSE	1037	1037	0	-5	1125	0	27	60	53	145	2018	1	1	4	4 DL	DL_CODES	19790	DL		
13	05-02-2018	Endeavor	ATL	ABY	FALSE	FALSE	1037	1032	0	-5	1124	0	29	60	52	145	2018	1	1	5	5 DL	DL_CODES	19790	DL		
14	06-02-2018	Endeavor	ATL	ABY	FALSE	FALSE	1037	1032	0	-5	1126	0	34	59	54	145	2018	1	1	6	6 DL	DL_CODES	19790	DL		
15	07-02-2018	Endeavor	ATL	ABY	FALSE	FALSE	1037	1034	0	-3	1124	0	29	60	50	145	2018	1	1	7	7 DL	DL_CODES	19790	DL		
16	08-02-2018	Endeavor	ATL	ABY	FALSE	FALSE	1037	1059	22	22	1153	16	32	60	54	145	2018	1	1	8	1 DL	DL_CODES	19790	DL		
17	09-02-2018	Endeavor	ATL	ABY	FALSE	FALSE	1037	1222	105	105	1322	105	37	60	60	145	2018	1	1	9	2 DL	DL_CODES	19790	DL		
18	10-02-2018	Endeavor	ATL	ABY	FALSE	FALSE	1037	1027	0	-10	1106	0	37	60	59	145	2018	1	1	10	3 DL	DL_CODES	19790	DL		
19	11-02-2018	Endeavor	ATL	ABY	FALSE	FALSE	1037	1033	0	-4	1145	8	38	60	72	145	2018	1	1	11	4 DL	DL_CODES	19790	DL		
20	12-02-2018	Endeavor	ATL	ABY	FALSE	FALSE	1037	1044	7	7	1138	1	35	60	54	145	2018	1	1	12	5 DL	DL_CODES	19790	DL		
21	13-02-2018	Endeavor	ATL	ABY	FALSE	FALSE	1037	1034	0	-3	1126	0	34	59	52	145	2018	1	1	13	6 DL	DL_CODES	19790	DL		
22	14-02-2018	Endeavor	ATL	ABY	FALSE	FALSE	1037	1031	0	-6	1120	0	30	60	49	145	2018	1	1	14	7 DL	DL_CODES	19790	DL		
23	15-02-2018	Endeavor	ATL	ABY	FALSE	FALSE	1037	1030	0	-7	1123	0	30	60	53	145	2018	1	1	15	1 DL	DL_CODES	19790	DL		
24	16-02-2018	Endeavor	ATL	ABY	FALSE	FALSE	1037	1033	0	-4	1129	0	30	60	54	145	2018	1	1	16	2 DL	DL_CODES	19790	DL		
25	17-02-2018	Endeavor	ATL	ABY	TRUE	FALSE	1037	1037			1129	0	30	60	60	145	2018	1	1	17	3 DL	DL_CODES	19790	DL		
26	18-02-2018	Endeavor	ATL	ABY	FALSE	FALSE	1037	1233	116	116	1245	128	22	60	72	145	2018	1	1	18	4 DL	DL_CODES	19790	DL		
27	19-02-2018	Endeavor	ATL	ABY	FALSE	FALSE	1037	1034	0	-3	1127	0	28	60	53	145	2018	1	1	19	5 DL	DL_CODES	19790	DL		
28	20-02-2018	Endeavor	ATL	ABY	FALSE	FALSE	1037	1035	0	-2	1121	0	27	59	46	145	2018	1	1	20	6 DL	DL_CODES	19790	DL		
29	21-02-2018	Endeavor	ATL	ABY	FALSE	FALSE	1037	1032	0	-5	1122	0	30	60	50	145	2018	1	1	21	7 DL	DL_CODES	19790	DL		
30	22-02-2018	Endeavor	ATL	ABY	FALSE	FALSE	1037	1032	0	-5	1132	0	30	60	60	145	2018	1	1	22	1 DL	DL_CODES	19790	DL		

## 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

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1	flightID	airline	flightDate	source	destination	departureTime	arrivalTime	duration	totalStops	ecoPrice	businessPrice	durationInMin	No of Stops	Days	destinationDate	layoverDurat
2	2001	IndiGo	24-08-2025	Banglore	New Delhi	22:20:00	01:10	2h 50m	non-stop	7499	14699	170	0	Monday	2025-08-25	56
3	2002	Air India	01-09-2025	Kolkata	Banglore	05:50:00	13:15	7h 25m	2 stops	8499	14299	445	2	Tuesday	2025-09-01	35
4	2003	Jet Airways	09-10-2025	Delhi	Cochin	09:25:00	04:25	19h	2 stops	8999	17299	1140	2	Wednesday	2025-10-10	33
5	2004	IndiGo	12-09-2025	Kolkata	Banglore	18:05:00	23:30	5h 25m	1 stop	9699	15999	325	1	Thursday	2025-09-12	53
6	2005	IndiGo	01-08-2025	Banglore	New Delhi	16:50:00	21:35	4h 45m	1 stop	7599	15299	285	1	Friday	2025-08-01	40
7	2006	Spicelet	24-10-2025	Kolkata	Banglore	09:00:00	11:25	2h 25m	non-stop	9499	14599	145	0	Saturday	2025-10-24	31
8	2007	Jet Airways	12-08-2025	Banglore	New Delhi	18:55:00	10:25	15h 30m	1 stop	6399	16699	930	1	Sunday	2025-08-13	59
9	2008	Jet Airways	01-08-2025	Banglore	New Delhi	08:00:00	05:05	21h 5m	1 stop	9799	16699	1265	1	Monday	2025-08-02	51
10	2009	Jet Airways	12-08-2025	Banglore	New Delhi	08:55:00	10:25	25h 30m	1 stop	8199	15899	1530	1	Tuesday	2025-08-13	35
11	2010	Multiple carriers	27-09-2025	Delhi	Cochin	11:25:00	19:15	7h 50m	1 stop	8499	14899	470	1	Wednesday	2025-09-27	60
12	2011	Air India	01-10-2025	Delhi	Cochin	09:45:00	23:00	13h 15m	1 stop	6899	14499	795	1	Thursday	2025-10-01	32
13	2012	IndiGo	18-09-2025	Kolkata	Banglore	20:20:00	22:55	2h 35m	non-stop	6099	17699	155	0	Friday	2025-09-18	35
14	2013	Air India	24-10-2025	Chennai	Kolkata	11:40:00	13:55	2h 15m	non-stop	6799	15799	135	0	Saturday	2025-10-24	34
15	2014	Jet Airways	09-09-2025	Kolkata	Banglore	21:10:00	09:20	12h 10m	1 stop	7399	17499	730	1	Sunday	2025-09-10	31
16	2015	IndiGo	24-09-2025	Kolkata	Banglore	17:15:00	19:50	2h 35m	non-stop	9999	16599	155	0	Monday	2025-09-24	59
17	2016	Air India	03-08-2025	Delhi	Cochin	16:40:00	19:15	26h 35m	2 stops	8199	17599	1595	2	Tuesday	2025-08-04	46
18	2017	Spicelet	15-09-2025	Delhi	Cochin	08:45:00	13:15	4h 30m	1 stop	7199	17599	270	1	Wednesday	2025-09-15	38
19	2018	Jet Airways	12-10-2025	Delhi	Cochin	14:00:00	12:35	22h 35m	1 stop	7699	14899	1355	1	Thursday	2025-10-13	52
20	2019	Air India	12-10-2025	Delhi	Cochin	20:15:00	19:15	23h	2 stops	7299	16499	1380	2	Friday	2025-10-13	53
21	2020	Jet Airways	27-09-2025	Delhi	Cochin	16:00:00	12:35	20h 35m	1 stop	9199	17399	1235	1	Saturday	2025-09-28	52
22	2021	GoAir	06-08-2025	Delhi	Cochin	14:10:00	19:20	5h 10m	1 stop	9799	15899	310	1	Sunday	2025-08-06	51
23	2022	Air India	21-08-2025	Banglore	New Delhi	22:00:00	13:20	15h 20m	1 stop	8999	17699	920	1	Monday	2025-08-22	41
24	2023	IndiGo	03-09-2025	Banglore	Delhi	04:00:00	06:50	2h 50m	non-stop	6399	14499	170	0	Tuesday	2025-09-03	50
25	2024	IndiGo	01-09-2025	Banglore	Delhi	18:55:00	21:50	2h 55m	non-stop	7099	14099	175	0	Wednesday	2025-09-01	30

- Collections imported in MongoDB

Collection	Storage size	Documents	Avg. document size	Indexes	Total index size
cities	24.56 kB	132	87.00 B	2	73.73 kB
flightdetails	77.82 kB	1K	339.00 B	2	106.50 kB
flightprices	36.86 kB	1K	83.00 B	1	24.58 kB
layovers	40.96 kB	1K	87.00 B	1	32.77 kB
seats	32.77 kB	1K	105.00 B	1	24.58 kB

## 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

```
_id: ObjectId('6878ac2c21d212c042fa5fb1')
cityID: "101"
cityCode: "IXA"
cityName: "Agartala"
__v: 0

_id: ObjectId('6878ac2c21d212c042fa5fb2')
cityID: "102"
cityCode: "AGX"
cityName: "Agatti Island"
__v: 0

_id: ObjectId('6878ac2c21d212c042fa5fb3')
cityID: "103"
cityCode: "AGR"
cityName: "Agra"
__v: 0

_id: ObjectId('6878ac2c21d212c042fa5fb4')
cityID: "104"
cityCode: "AMD"
cityName: "Ahmedabad"
__v: 0
```

- flightDetails collection

```
_id: ObjectId('6878ac2c21d212c042fa5bd2')
flightID: "2001"
airline: "Indigo"
source: "Banglore"
destination: "New Delhi"
departureTime: "23:20:00"
arrivalTime: "00:11:18"
flightDate: "Sat Aug 23 2025 23:59:50 GMT+0530 (India Standard Time)"
destinationDate: "Sun Aug 24 2025 23:59:50 GMT+0530 (India Standard Time)"
duration: "2h 50m"
__v: 0

_id: ObjectId('6878ac2c21d212c042fa5bd3')
flightID: "2002"
airline: "Air India"
source: "Kolkata"
destination: "Banglore"
departureTime: "00:50:00"
arrivalTime: "13:15"
flightDate: "Sun Aug 31 2025 23:59:50 GMT+0530 (India Standard Time)"
destinationDate: "Sun Aug 31 2025 23:59:50 GMT+0530 (India Standard Time)"
duration: "7h 25m"
__v: 0

_id: ObjectId('6878ac2c21d212c042fa5bd4')
flightID: "2003"
airline: "Jet Airways"
```

- flightprices collection

The screenshot shows the MongoDB Compass interface with the following details:

- Database:** MyDatabase
  - Collection:** flightManagementSystem.flightprices
- Documents:** 1K (1 document shown)
- Fields:** Type a query: { field: 'value' } or [Generate query](#)
- Buttons:** ADD DATA, EXPORT DATA, UPDATE, DELETE, Explain, Reset, Find, Options
- Results:** Four documents are listed, each with a unique \_id and flight details:
  - Flight ID: "2001", ecoPrice: 7099, businessPrice: 17699
  - Flight ID: "2002", ecoPrice: 8599, businessPrice: 16899
  - Flight ID: "2003", ecoPrice: 9799, businessPrice: 16699
  - Flight ID: "2004", ecoPrice: 9199, businessPrice: 14699

- layovers collection

The screenshot shows the MongoDB Compass interface with the following details:

- Database:** MyDatabase
  - Collection:** flightManagementSystem.layovers
- Documents:** 1K (1 document shown)
- Fields:** Type a query: { field: 'value' } or [Generate query](#)
- Buttons:** ADD DATA, EXPORT DATA, UPDATE, DELETE, Explain, Reset, Find, Options
- Results:** Four documents are listed, each with a unique \_id and layover details:
  - Flight ID: "2001", cityID: "101", layoverDuration: 42
  - Flight ID: "2002", cityID: "102", layoverDuration: 48
  - Flight ID: "2003", cityID: "103", layoverDuration: 38
  - Flight ID: "2004", cityID: "104", layoverDuration: 48

- seats collection

```

_id: ObjectId('6879ea5e662ee54dd854934d')
flightID: "2001"
ecoSeats: "50"
businessSeats: "20"
onDate: 2025-07-18T12:01:58.301+00:00
__v: 0

_id: ObjectId('6879ea5e662ee54dd854934e')
flightID: "2002"
ecoSeats: "50"
businessSeats: "20"
onDate: 2025-07-18T12:01:58.302+00:00
__v: 0

_id: ObjectId('6879ea5e662ee54dd854934f')
flightID: "2003"
ecoSeats: "50"
businessSeats: "20"
onDate: 2025-07-18T12:01:58.303+00:00
__v: 0

_id: ObjectId('6879ea5e662ee54dd8549350')
flightID: "2004"
ecoSeats: "50"
businessSeats: "20"
onDate: 2025-07-18T12:01:58.303+00:00
__v: 0

```

## Data Updation

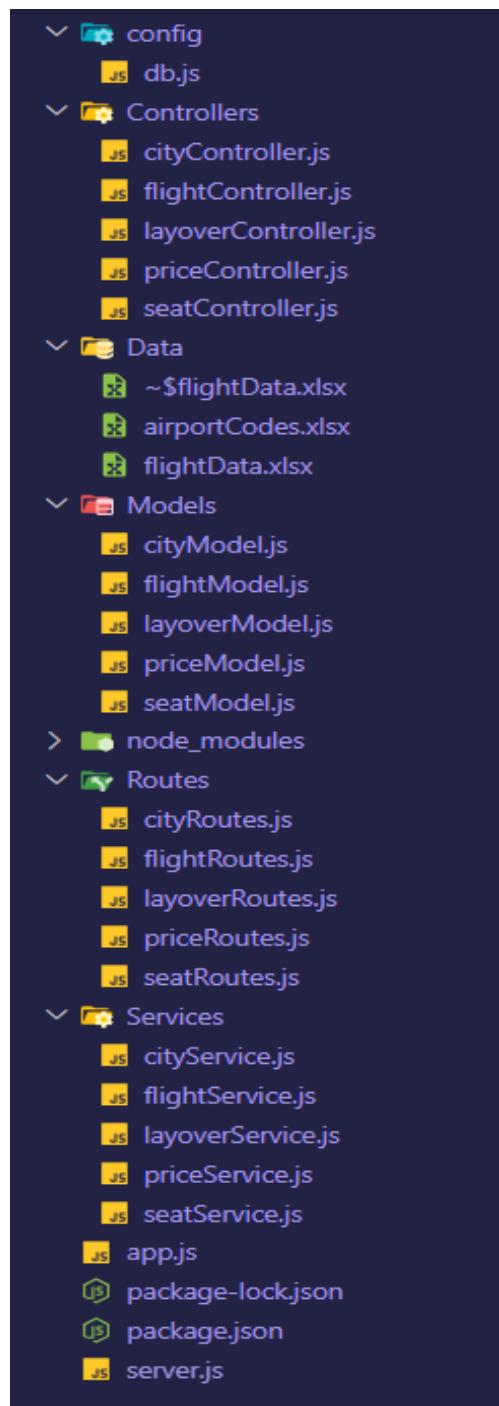
Data updation in the flight management system ensures that key fields remain current and accurate. The `onDate` field in the `seats` collection is designed to automatically capture the current date (IST) at the time of insertion, reflecting real-time availability. Other fields such as `ecoSeats` and `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-

The screenshot shows the Postman interface with a successful POST request. The URL is `http://localhost:8000/api/flights/import`. The response status is `200 OK` with a response time of `497 ms` and a size of `315 B`. The response body is a JSON object with a single key `message` containing the value `"Flight data successfully imported."`.

- GET API-

The screenshot shows the Postman interface with a successful GET request. The URL is `http://localhost:8000/api/flights/fetch`. The response status is `200 OK` with a response time of `63 ms` and a size of `273.72 KB`. The response body is a JSON array containing two flight records. Each record includes fields like `_id`, `flightID`, `airline`, `source`, `destination`, `departureTime`, `arrivalTime`, `flightDate`, `destinationDate`, `duration`, and `_v`.

```
1 {
2   "message": [
3     {
4       "_id": "688372f8a3f176953fffe6c7f",
5       "flightID": "2001",
6       "airline": "Indigo",
7       "source": "Banglore",
8       "destination": "New Delhi",
9       "departureTime": "22:20:00",
10      "arrivalTime": "01:10",
11      "flightDate": "2025-08-23T18:29:50.000Z",
12      "destinationDate": "2025-08-24T18:29:50.000Z",
13      "duration": "2h 50m",
14      "_v": 0
15    },
16    {
17      "_id": "688372f8a3f176953fffe6c80",
18      "flightID": "2002",
19      "airline": "Air India",
20      "source": "Kolkata",
21      "destination": "Banglore",
22      "departureTime": "06:50:00",
23      "arrivalTime": "13:15",
24    }
  ]
```

## ○ DB Collection-

The screenshot shows the MongoDB Compass interface with the following details:

- Connections:** MyDatabase, newFlightManagementSystem, flightdetails.
- Documents:** 0 (under Aggregations), Schema, Indexes 2, Validation.
- Query Bar:** Type a query: { field: 'value' } or [Generate query](#).
- Buttons:** ADD DATA, EXPORT DATA, UPDATE, DELETE.
- Document 1:**

```
_id: ObjectId('688372f8a3f176953ffe6c7f')
flightID : "2001"
airline : "IndiGo"
source : "Banglore"
destination : "New Delhi"
departureTime : "22:00:00"
arrivalTime : "01:10"
flightDate : 2025-08-23T18:29:50.000+00:00
destinationDate : 2025-08-24T18:29:50.000+00:00
duration : "2h 50m"
__v : 0
```
- Document 2:**

```
_id: ObjectId('688372f8a3f176953ffe6c80')
flightID : "2002"
airline : "Air India"
source : "Kolkata"
destination : "Banglore"
departureTime : "05:50:00"
arrivalTime : "13:15"
flightDate : 2025-08-31T18:29:50.000+00:00
destinationDate : 2025-08-31T18:29:50.000+00:00
duration : "7h 25m"
__v : 0
```
- Document 3:**

```
_id: ObjectId('688372f8a3f176953ffe6c81')
flightID : "2003"
airline : "Jet Airways"
```

## ● cities:

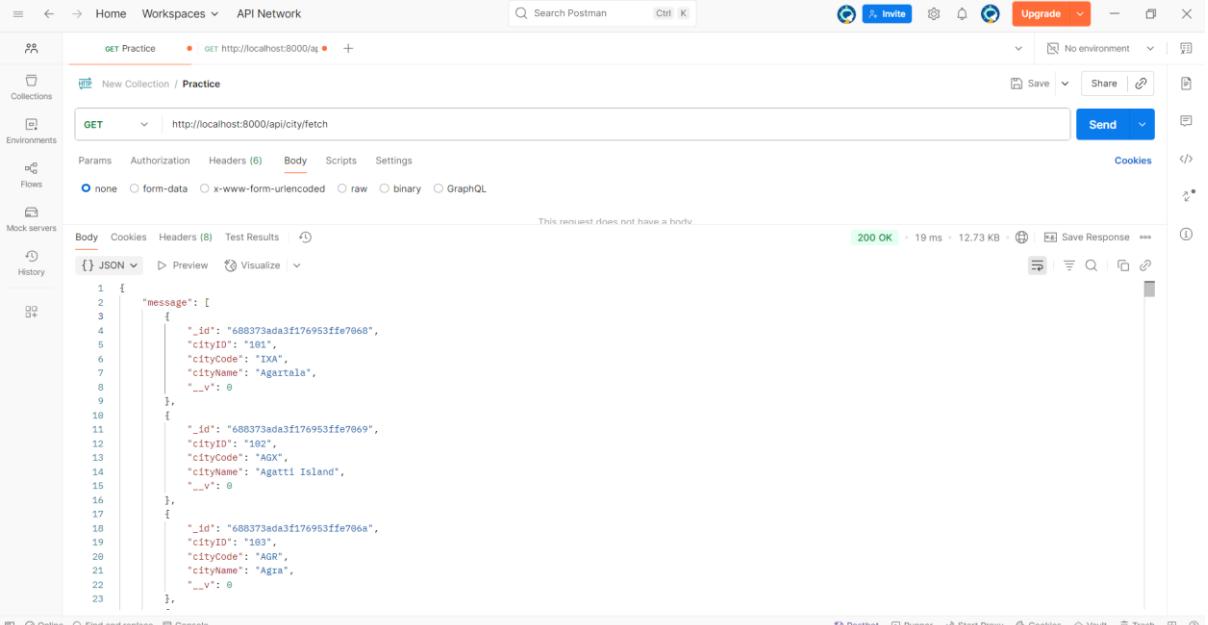
## ○ POST API-

The screenshot shows the Postman interface with the following details:

- Request:** POST Practice → http://localhost:8000/api/city/import
- Body:** Params, Authorization, Headers (7), Body, Scripts, Settings. Body type: none.
- Response:** 200 OK, 30.87 s, 313 B. Response body:

```
{
  "message": "Successfully city data imported."
}
```

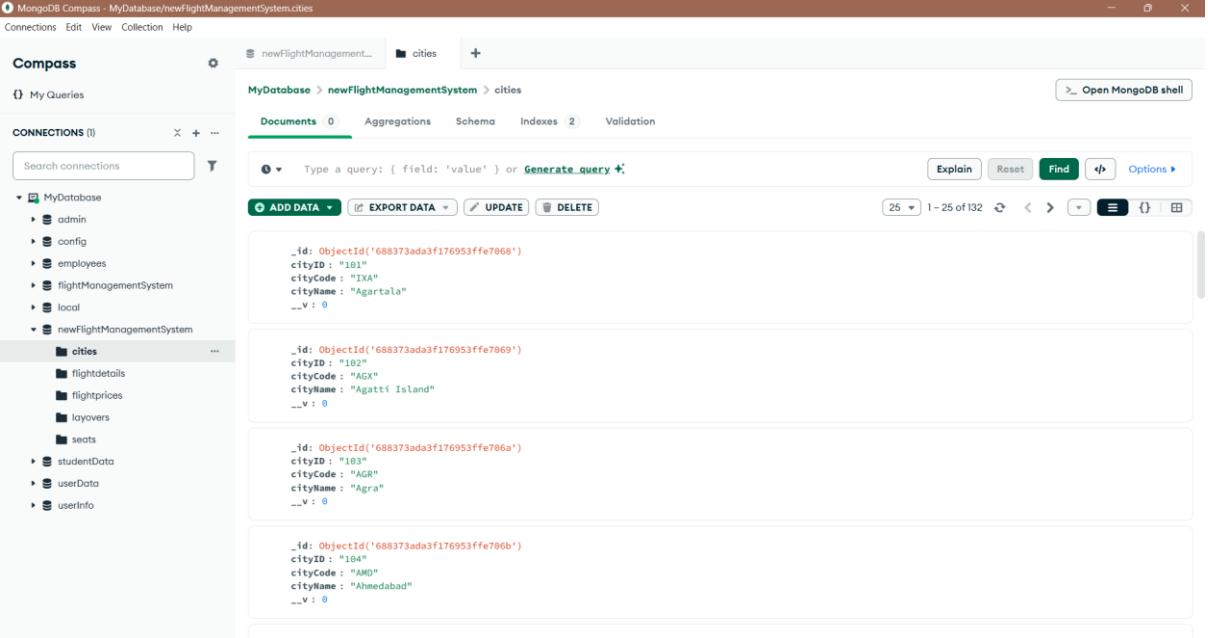
## ○ GET API-



The screenshot shows the Postman interface with a successful API call. The URL is `http://localhost:8000/api/city/fetch`. The response status is `200 OK` with a response time of 19 ms and a size of 12.73 KB. The response body is a JSON array of city documents:

```
[{"_id": "688373ada3f176953ffe7068", "cityID": "101", "cityCode": "IXA", "cityName": "Agartala", "__v": 0}, {"_id": "688373ada3f176953ffe7069", "cityID": "102", "cityCode": "AGX", "cityName": "Agatti Island", "__v": 0}, {"_id": "688373ada3f176953ffe706a", "cityID": "103", "cityCode": "AGR", "cityName": "Agra", "__v": 0}, {"_id": "688373ada3f176953ffe706b", "cityID": "104", "cityCode": "AMD", "cityName": "Ahmedabad", "__v": 0}]
```

## ○ DB Collection-



The screenshot shows the MongoDB Compass interface with the `cities` collection from the `newFlightManagementSystem` database. The collection contains four documents, each representing a city with fields: `_id`, `cityID`, `cityCode`, `cityName`, and `__v`.

- `_id: ObjectId('688373ada3f176953ffe7068')`  
`cityID: "101"`  
`cityCode: "IXA"`  
`cityName: "Agartala"`  
`__v: 0`
- `_id: ObjectId('688373ada3f176953ffe7069')`  
`cityID: "102"`  
`cityCode: "AGX"`  
`cityName: "Agatti Island"`  
`__v: 0`
- `_id: ObjectId('688373ada3f176953ffe706a')`  
`cityID: "103"`  
`cityCode: "AGR"`  
`cityName: "Agra"`  
`__v: 0`
- `_id: ObjectId('688373ada3f176953ffe706b')`  
`cityID: "104"`  
`cityCode: "AMD"`  
`cityName: "Ahmedabad"`  
`__v: 0`

- flightPrices:

- POST API-

The screenshot shows the Postman interface with a successful POST request. The URL is `http://localhost:8000/api/price/import`. The response status is 200 OK, with a response time of 383 ms and a body size of 317 B. The response body is JSON, containing the message: "Price details imported successfully."

```
[{"message": "Price details imported successfully."}]
```

- GET API-

The screenshot shows the Postman interface with a successful GET request. The URL is `http://localhost:8000/api/price/fetch`. The response status is 200 OK, with a response time of 29 ms and a body size of 96.98 KB. The response body is JSON, containing multiple price records. Each record includes an \_id, flightID, ecoPrice, and businessPrice.

```
[{"_id": "68837415a3f176953ffe70ed", "flightID": "2001", "ecoPrice": 7499, "businessPrice": 14699, "__v": 0}, {"_id": "68837415a3f176953ffe70ee", "flightID": "2002", "ecoPrice": 8499, "businessPrice": 14299, "__v": 0}, {"_id": "68837415a3f176953ffe70ef", "flightID": "2003", "ecoPrice": 8999, "businessPrice": 17299, "__v": 0}]
```

## ○ DB Collection-

The screenshot shows the MongoDB Compass application interface. On the left, the sidebar lists databases and collections under 'MyDatabase'. The 'flightprices' collection is selected. The main area shows four documents in the 'flightprices' collection. Each document contains the following fields:

```

_id: ObjectId('68837415a3f176953ffe70ed')
flightID: "2801"
ecoPrice: 7499
businessPrice: 14699
__v: 0

```

```

_id: ObjectId('68837415a3f176953ffe70ee')
flightID: "2802"
ecoPrice: 8499
businessPrice: 14299
__v: 0

```

```

_id: ObjectId('68837415a3f176953ffe70ef')
flightID: "2803"
ecoPrice: 8999
businessPrice: 17299
__v: 0

```

```

_id: ObjectId('68837415a3f176953ffe70f0')
flightID: "2804"
ecoPrice: 9699
businessPrice: 15999
__v: 0

```

## ● seats:

## ○ POST API-

The screenshot shows the Postman application interface. A new collection named 'Practice' is selected. A POST request is defined with the URL <http://localhost:8000/api/seats/import>. The 'Body' tab is selected, showing the request body is empty. The 'Test Results' tab shows the response details:

200 OK    198 ms    314 B    Save Response

```

{
  "message": "Seats data imported successfully."
}

```

## ○ GET API-

The screenshot shows the Postman interface with a successful API call. The URL is `http://localhost:8000/api/seats/fetch`. The response is a JSON array containing three seat objects, each with fields: `_id`, `flightID`, `ecoSeats`, `businessSeats`, and `onDate`.

```

1  [
2    {
3      "message": [
4        {
5          "_id": "6883745ca3f176953ffe74d6",
6          "flightID": "2001",
7          "ecoSeats": "50",
8          "businessSeats": "20",
9          "onDate": "2025-07-25T17:41:00.338Z",
10         "__v": 0
11       },
12       {
13         "_id": "6883745ca3f176953ffe74d7",
14         "flightID": "2002",
15         "ecoSeats": "50",
16         "businessSeats": "20",
17         "onDate": "2025-07-25T17:41:00.338Z",
18         "__v": 0
19       },
20       {
21         "_id": "6883745ca3f176953ffe74d8",
22         "flightID": "2003",
23         "ecoSeats": "50",
24         "businessSeats": "20",
25         "onDate": "2025-07-25T17:41:00.338Z",
26         "__v": 0
27     }
28   ]
29 ]

```

## ○ DB Collection-

The screenshot shows the MongoDB Compass interface with the `seats` collection from the `newFlightManagementSystem` database. Each document contains fields: `_id`, `flightID`, `ecoSeats`, `businessSeats`, and `onDate`.

```

1 _id: ObjectId('6883745ca3f176953ffe74d6')
2 flightID: "2001"
3 ecoSeats: "50"
4 businessSeats: "20"
5 onDate: 2025-07-25T17:41:00.338+00:00
6 __v: 0
7
8 _id: ObjectId('6883745ca3f176953ffe74d7')
9 flightID: "2002"
10 ecoSeats: "50"
11 businessSeats: "20"
12 onDate: 2025-07-25T17:41:00.338+00:00
13 __v: 0
14
15 _id: ObjectId('6883745ca3f176953ffe74d8')
16 flightID: "2003"
17 ecoSeats: "50"
18 businessSeats: "20"
19 onDate: 2025-07-25T17:41:00.338+00:00
20 __v: 0
21
22 _id: ObjectId('6883745ca3f176953ffe74d9')
23 flightID: "2004"
24 ecoSeats: "50"
25 businessSeats: "20"
26 onDate: 2025-07-25T17:41:00.338+00:00
27 __v: 0

```

- Layover:
  - POST API-

POST Practice POST http://localhost:8000/layover/import

Params Authorization Headers (7) Body Scripts Settings

Body Cookies Headers (8) Test Results

```
{
  "message": "Layover details imported successfully."
}
```

200 OK 616 ms 319 B

- GET API-

GET Practice GET http://localhost:8000/api/layover/fetch

Params Authorization Headers (6) Body Scripts Settings

Body Cookies Headers (8) Test Results

```
[
  {
    "message": [
      {
        "_id": "68958ad6ef01ab3cbc7df22",
        "flightID": "2001",
        "cityID": "149",
        "layoverDuration": 52,
        "stops": 0,
        "__v": 0
      },
      {
        "_id": "68958ad6ef01ab3cbc7df23",
        "flightID": "2002",
        "cityID": "149",
        "layoverDuration": 60,
        "stops": 0,
        "__v": 0
      },
      {
        "_id": "68958ad6ef01ab3cbc7df24",
        "flightID": "2003",
        "cityID": "149",
        "layoverDuration": 60,
        "stops": 0,
        "__v": 0
      }
    ]
  }
]
```

200 OK 253 ms 104.77 KB

- DB Collection-

```

_id: ObjectId('68958ad6ef01ab3cbca7df22')
flightID: "2801"
cityID: "149"
layoverDuration: 52
stops: 0
__v: 0

_id: ObjectId('68958ad6ef01ab3cbca7df23')
flightID: "2802"
cityID: "149"
layoverDuration: 60
stops: 0
__v: 0

_id: ObjectId('68958ad6ef01ab3cbca7df24')
flightID: "2803"
cityID: "149"
layoverDuration: 31
stops: 1
__v: 1

_id: ObjectId('68958ad6ef01ab3cbca7df25')
flightID: "2804"
cityID: "149"
layoverDuration: 59
__v: 0
  
```

## 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-

```

1 {
2   "stringDate": "2025-08-23"
3 }
  
```

```

1 {
2   "message": [
3     {
4       "_id": "688372fa3f176953ffe6c7f",
5       "flightID": "2801",
6       "airline": "Indigo",
7       "source": "Banglore",
8       "destination": "New Delhi",
9       "departureTime": "22:20:00",
10      "arrivalTime": "01:10",
11      "flightToDate": "2025-08-23T10:29:50.000Z",
12      "destinationDate": "2025-08-24T10:29:50.000Z",
13      "duration": "2h 50m",
14      "__v": 0
15    },
16    {
17      "_id": "688372fa3f176953ffe6cac",
18      "flightID": "2846",
19      "airline": "Scoot",
20      "source": "Banglore",
  
```

- Source and Destination filter-

GET Practice | http://localhost:8000/api/flights/filter\_sd

```

1 {
2   "stringDate": "2025-08-23",
3   "source": "Banglore",
4   "destination": "New Delhi"
5 }

```

Body Cookies Headers (8) Test Results

```

1 {
2   "message": [
3     {
4       "_id": "688372f0a3f176953ffec7f",
5       "flightID": "2001",
6       "airline": "Indigo",
7       "source": "Banglore",
8       "destination": "New Delhi",
9       "departureTime": "22:00:00",
10      "arrivalTime": "01:18",
11      "flightDate": "2025-08-23T18:29:56.000Z",
12      "destinationDate": "2025-08-24T18:29:56.000Z",
13      "duration": "2h 56m",
14      "__v": 0
15    },
16    {
17      "_id": "688372f0a3f176953ffecac",
18      "flightID": "2046".

```

200 OK - 18 ms - 2.48 KB - Save Response

- seats:

- Seats using flightID-

GET Practice | http://localhost:8000/api/seats/filter.seats

```

1 {
2   "flightID": "2046"
3 }

```

Body Cookies Headers (8) Test Results

```

1 {
2   "message": [
3     {
4       "_id": "688879a478697f9495b99169",
5       "flightID": "2046",
6       "ecoSeats": 50,
7       "businessSeats": 20,
8       "onDate": "2025-07-29T13:05:00.790Z",
9       "__v": 0
10    ]
11  ]
12 }

```

200 OK - 16 ms - 411 B - Save Response

- Seats by class-
  - Economy seats

The screenshot shows the Postman interface with a successful API call. The URL is `http://localhost:8000/api/seats/seatsByClass`. The request method is `GET`. The body parameter is set to `{ "seatClass": "E" }`. The response status is `200 OK` with a response time of 160 ms and a size of 65.71 KB. The response body is a JSON array of messages, each containing flight ID, eco seats count, and ID.

```

1 {
2   "message": [
3     {
4       "_id": "688879a478697f9495b9913c",
5       "flightID": "2001",
6       "ecoSeats": 50
7     },
8     {
9       "_id": "688879a478697f9495b9913d",
10      "flightID": "2002",
11      "ecoSeats": 50
12    },
13    {
14      "_id": "688879a478697f9495b9913e",
15      "flightID": "2003",
16      "ecoSeats": 50
17    },
18    {
19      "_id": "688879a478697f9495b9913f",
20      "flightID": "2004",
21    }
22  ]
23}
  
```

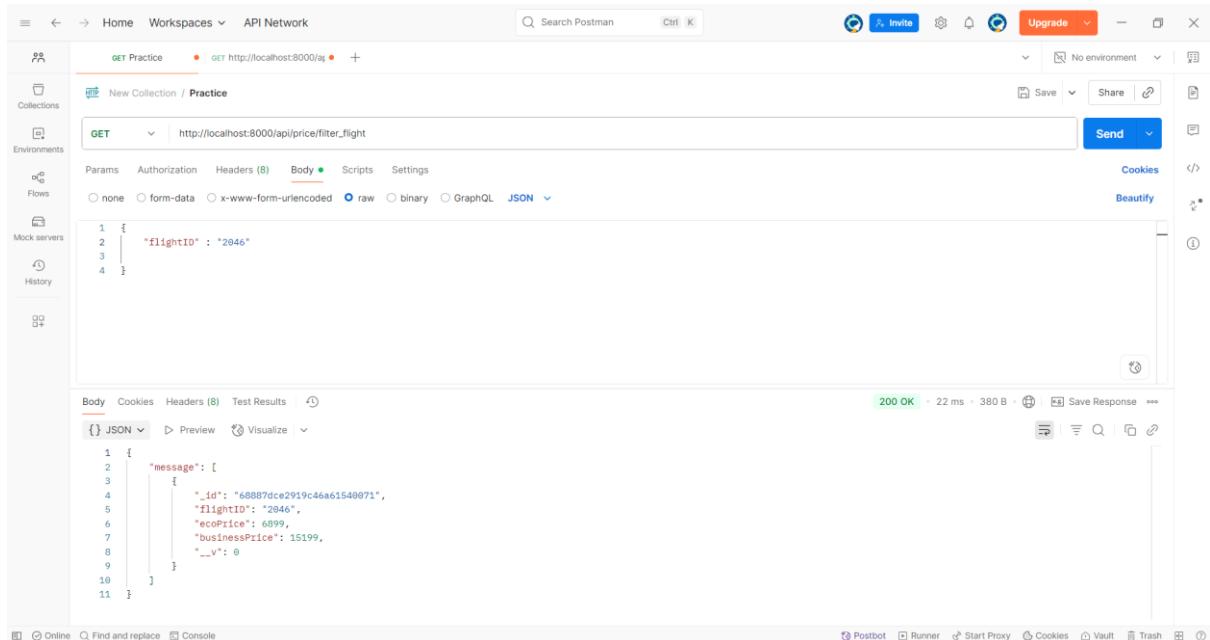
- Business seats

The screenshot shows the Postman interface with a successful API call. The URL is `http://localhost:8000/api/seats/seatsByClass`. The request method is `GET`. The body parameter is set to `{ "seatClass": "B" }`. The response status is `200 OK` with a response time of 102 ms and a size of 70.59 KB. The response body is a JSON array of messages, each containing flight ID, business seats count, and ID.

```

1 {
2   "message": [
3     {
4       "_id": "688879a478697f9495b9913c",
5       "flightID": "2001",
6       "businessSeats": 20
7     },
8     {
9       "_id": "688879a478697f9495b9913d",
10      "flightID": "2002",
11      "businessSeats": 20
12    },
13    {
14      "_id": "688879a478697f9495b9913e",
15      "flightID": "2003",
16      "businessSeats": 20
17    },
18    {
19      "_id": "688879a478697f9495b9913f",
20      "flightID": "2004",
21    }
22  ]
23}
  
```

- flightPrices:
  - Price using flightID-



The screenshot shows the Postman interface with a collection named "Practice". A new request is being made to `http://localhost:8000/api/price/filter_flight`. The "Body" tab is selected, showing the following JSON payload:

```

1 {
2   "flightID": "2046"
3 }
4

```

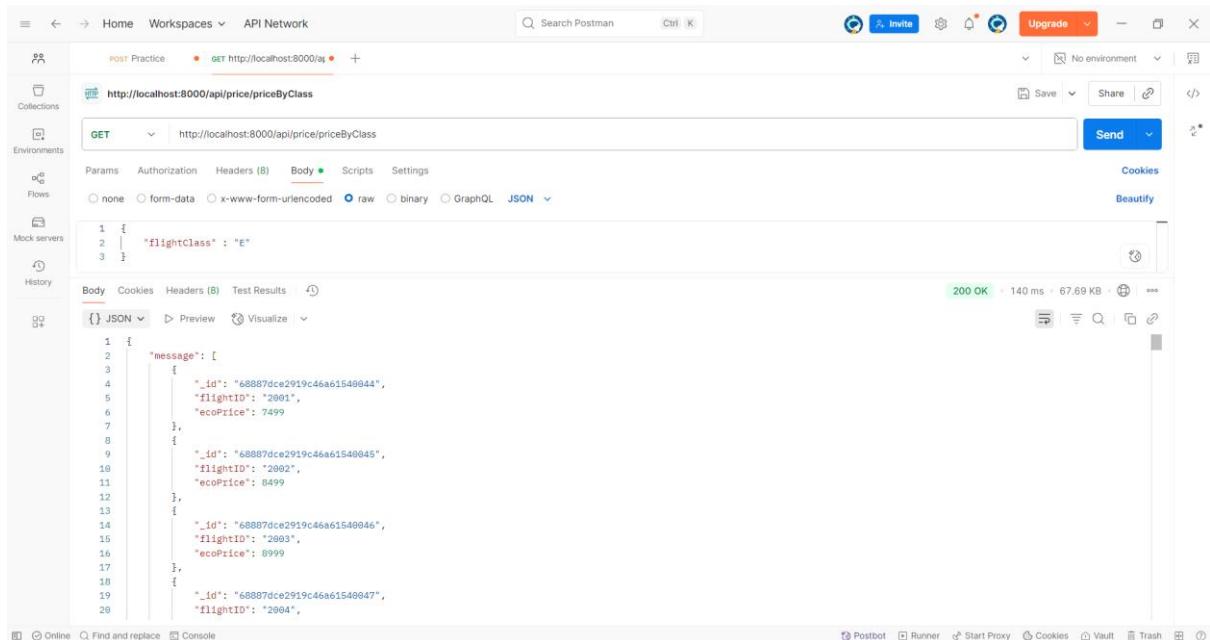
The response status is `200 OK` with a response time of 22 ms, 380 B. The response body is displayed as JSON:

```

1 {
2   "message": [
3     {
4       "_id": "68887dce2919c46a61540871",
5       "flightID": "2046",
6       "ecoPrice": 6899,
7       "businessPrice": 15199,
8       "__v": 0
9     }
10   ]
11 }

```

- Price by Class-
  - Economy-



The screenshot shows the Postman interface with a collection named "Practice". A new request is being made to `http://localhost:8000/api/price/priceByClass`. The "Body" tab is selected, showing the following JSON payload:

```

1 {
2   "flightClass": "E"
3 }

```

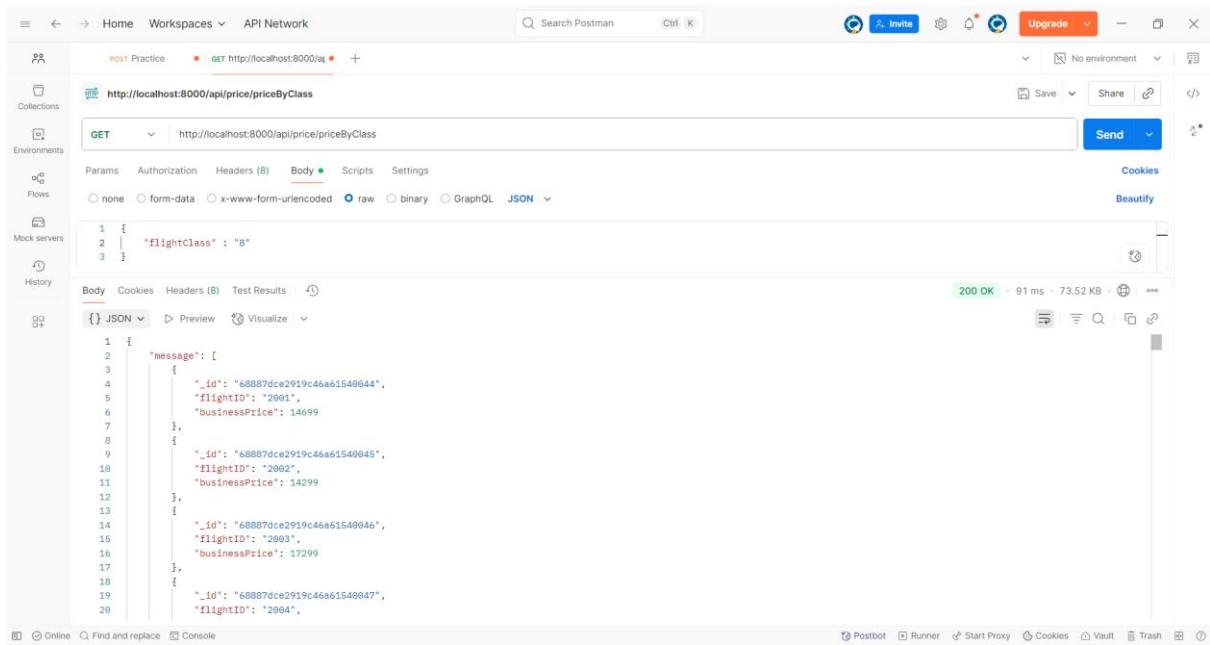
The response status is `200 OK` with a response time of 140 ms, 67.69 KB. The response body is displayed as JSON:

```

1 {
2   "message": [
3     {
4       "_id": "68887dce2919c46a61540844",
5       "flightID": "2001",
6       "ecoPrice": 7499
7     },
8     {
9       "_id": "68887dce2919c46a61540845",
10      "flightID": "2002",
11      "ecoPrice": 8499
12    },
13    {
14      "_id": "68887dce2919c46a61540846",
15      "flightID": "2003",
16      "ecoPrice": 8999
17    },
18    {
19      "_id": "68887dce2919c46a61540847",
20      "flightID": "2004",
21      "ecoPrice": 9499
22    }
23  ]
24 }

```

## ■ Business-



POST Practice ● GET http://localhost:8000/aj ● +

http://localhost:8000/api/price/priceByClass

GET http://localhost:8000/api/price/priceByClass

Params Authorization Headers (8) Body Scripts Settings

none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
1 {  
2   "flightClass" : "B"  
3 }
```

Body Cookies Headers (8) Test Results

{} JSON Preview Visualize

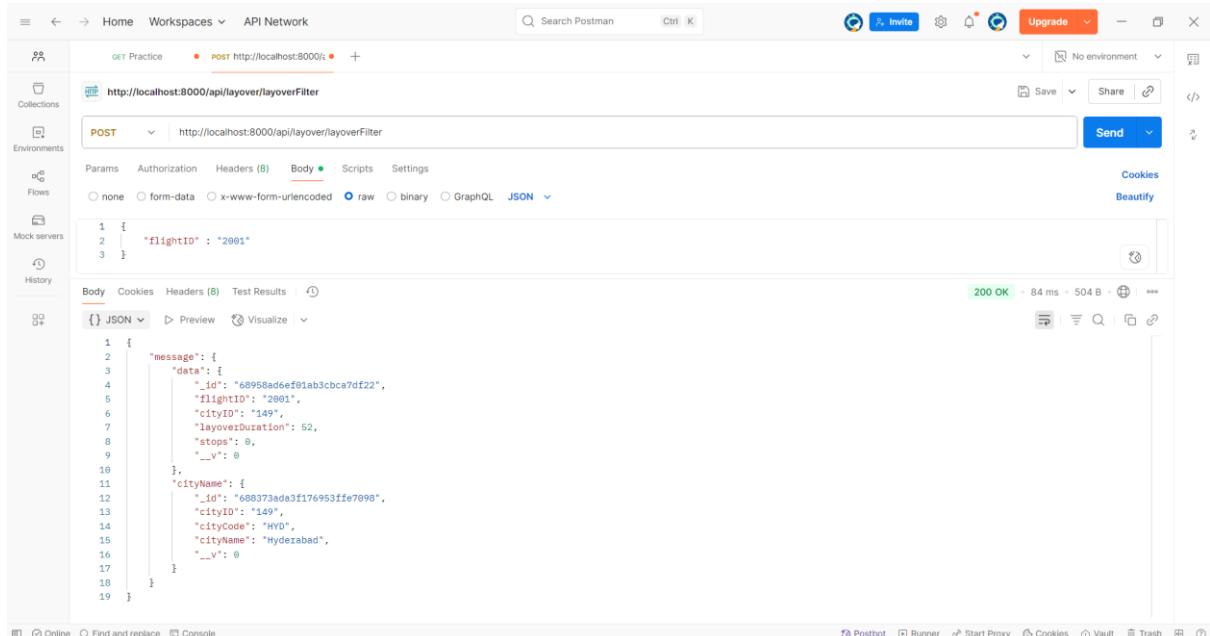
```
1 {  
2   "message": [  
3     {  
4       "_id": "68887dce2919c46a61548044",  
5       "flightID": "2001",  
6       "businessPrice": 14699  
7     },  
8     {  
9       "_id": "68887dce2919c46a61548045",  
10      "flightID": "2002",  
11      "businessPrice": 14299  
12    },  
13    {  
14      "_id": "68887dce2919c46a61548046",  
15      "flightID": "2003",  
16      "businessPrice": 17299  
17    },  
18    {  
19      "_id": "68887dce2919c46a61548047",  
20      "flightID": "2004",  
21    }  
22  ]  
23}
```

200 OK 91 ms 73.52 KB

Online Find and replace Console Postbot Runner Start Proxy Cookies Vault Trash

## ● Layover:

- Layover by flightID



GET Practice ● POST http://localhost:8000/aj ● +

http://localhost:8000/api/layover/layoverFilter

POST http://localhost:8000/api/layover/layoverFilter

Params Authorization Headers (8) Body Scripts Settings

none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
1 {  
2   "flightID" : "2001"  
3 }
```

Body Cookies Headers (8) Test Results

{} JSON Preview Visualize

```
1 {  
2   "message": {  
3     "data": [  
4       {  
5         "_id": "68988ad6ef01ab3cbc7df22",  
6         "flightID": "2001",  
7         "cityID": "149",  
8         "layoverDuration": 52,  
9         "stops": 0,  
10        "_v": 0  
11      },  
12      {  
13        "_id": "688373ada3f176953ffe7098",  
14        "cityID": "149",  
15        "cityCode": "HYD",  
16        "cityName": "Hyderabad",  
17        "_v": 0  
18      }  
19    ]  
20}
```

200 OK 84 ms 504 B

Online Find and replace Console Postbot Runner Start Proxy Cookies Vault Trash

# 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**.

The screenshot shows the Postman interface with a collection named "Practice". A POST request is being made to `http://localhost:8000/api/seats/bookseat`. The request body is set to JSON and contains the following data:

```
1 {
2   "flightID": "2115",
3   "economyClass": 4,
4   "businessClass": 2
5 }
```

The response status is 200 OK, with a response time of 20 ms and a size of 320 B. The response body is:

```
1 {
2   "message": "Congratulations! Seat booking confirmed."
3 }
```

## Before update

The screenshot shows the Postman interface with a collection named "Practice". A GET request is being made to `http://localhost:8000/api/seats/filter_seats`. The request body is set to JSON and contains the following data:

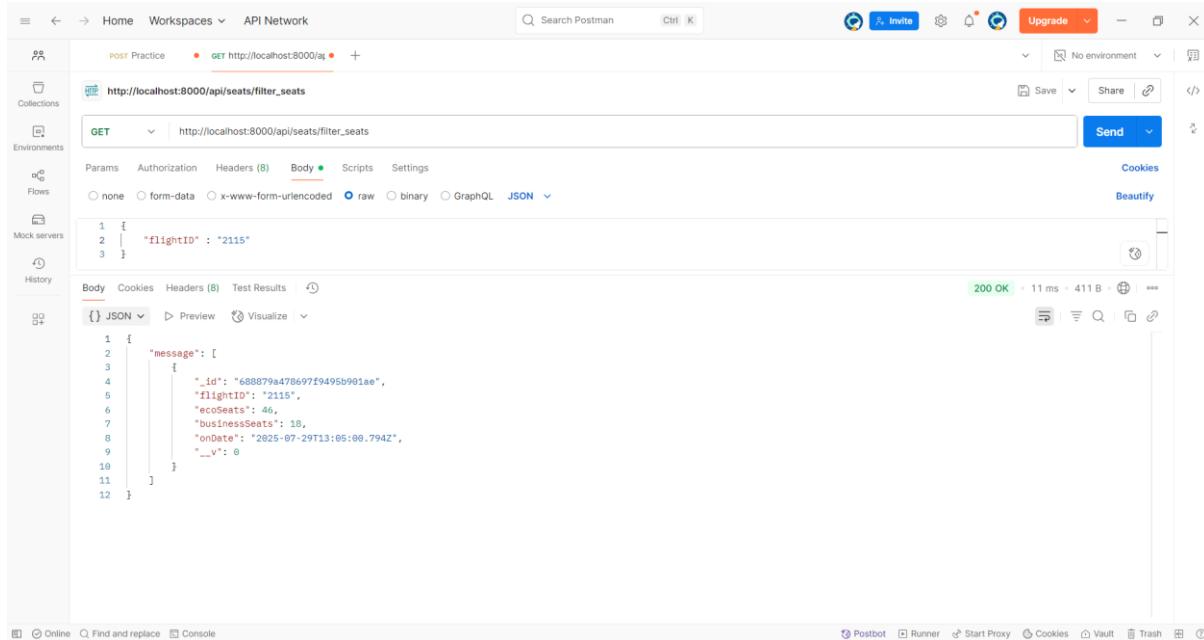
```
1 {
2   "flightID": "2115"
3 }
```

The response status is 200 OK, with a response time of 46 ms and a size of 411 B. The response body is:

```
1 {
2   "message": [
3     {
4       "_id": "688879a478697f9495b991ae",
5       "flightID": "2115",
6       "economySeats": 50,
7       "businessSeats": 20,
8       "onDate": "2025-07-29T13:05:00.794Z",
9       "__v": 0
10    }
11  ]
12 }
```

## 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.



The screenshot shows the Postman application interface. On the left, there's a sidebar with 'Collections', 'Environments', 'Flows', 'Mock servers', and 'History'. The main area has tabs for 'POST Practice' and 'GET http://localhost:8000/api/seats/filter\_seats'. The 'GET' tab is active, showing the URL 'http://localhost:8000/api/seats/filter\_seats'. Below the URL, under 'Body', there is a JSON payload: 

```
1 {  
2   "flightID": "2115"  
3 }
```

. The 'Test Results' section shows a successful response with status code 200 OK, 11 ms duration, and 411 B size. The response body is displayed as JSON: 

```
1 {  
2   "message": [  
3     {  
4       "_id": "688879a478697f9495b901ae",  
5       "flightID": "2115",  
6       "ecoSeats": 46,  
7       "businessSeats": 18,  
8       "onDate": "2025-07-29T13:05:00.794Z",  
9       "_v": 0  
10    }  
11  ]  
12 }
```

## **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**