**Subway System API Documentation**

### **Project Overview:**

This Subway System API models a subway network where users can create train lines, enter/exit stations with fare cards, and retrieve the shortest routes between stations. It is built using TypeScript, Node.js, Express, PostgreSQL, and Docker. The system supports operations for creating train lines, managing fare cards, processing station entry/exit, and finding optimal routes.

### **Table of Contents:**

1. **System Requirements**
2. **Installation and Setup**
3. **Running the Application**
4. **API Endpoints**
   * **Challenge 1: Train Line and Route APIs**
   * **Challenge 2: Fare Card and Station Entry/Exit APIs**
5. **Extensibility of Code**
6. **Testing**
7. **Conclusion**
8. **Evaluation answers**

**1) System Requirements:**

**->** Run the Subway System API, ensure you have the following installed:

* Node.js (>= v14.0.0)
* PostgreSQL (>= v13.0)
* Docker and Docker Compose

**2) Installation and Setup:**

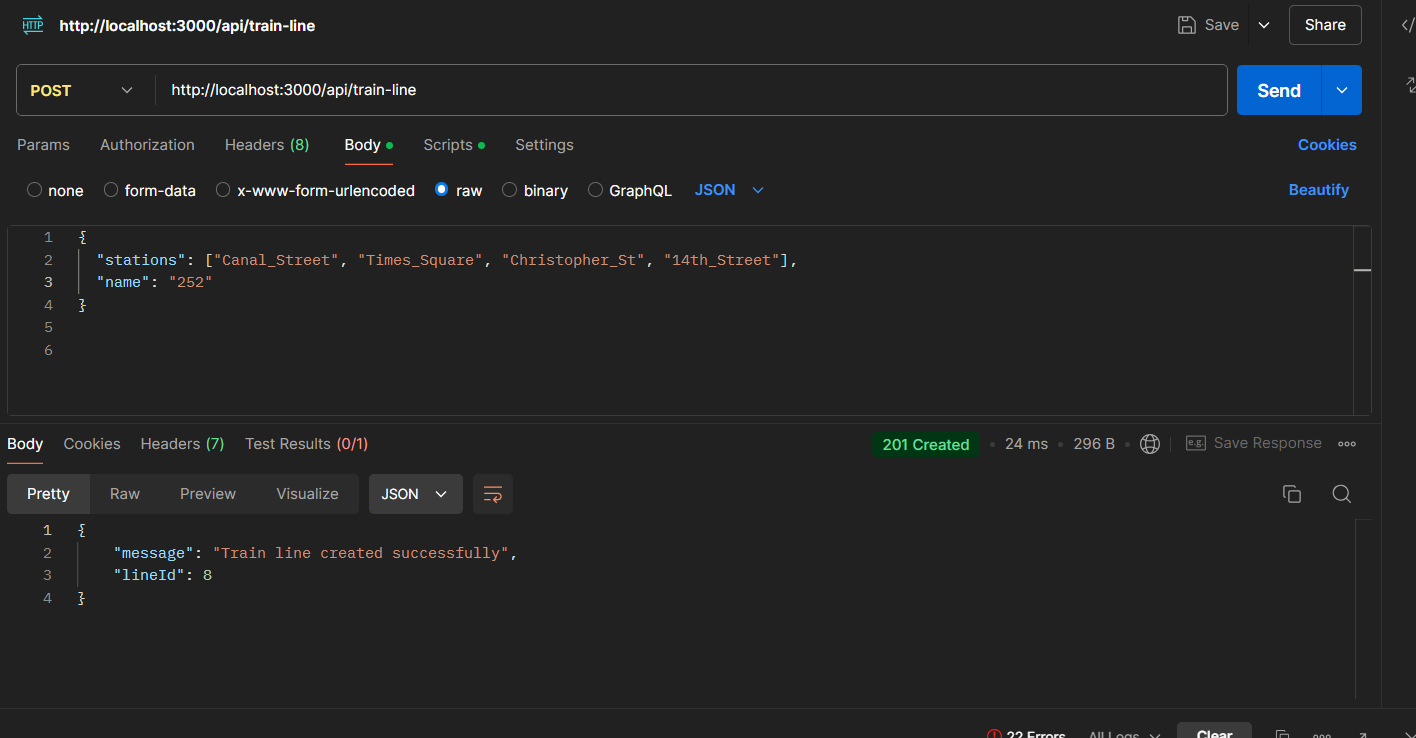
**->** docker-compose up --build.

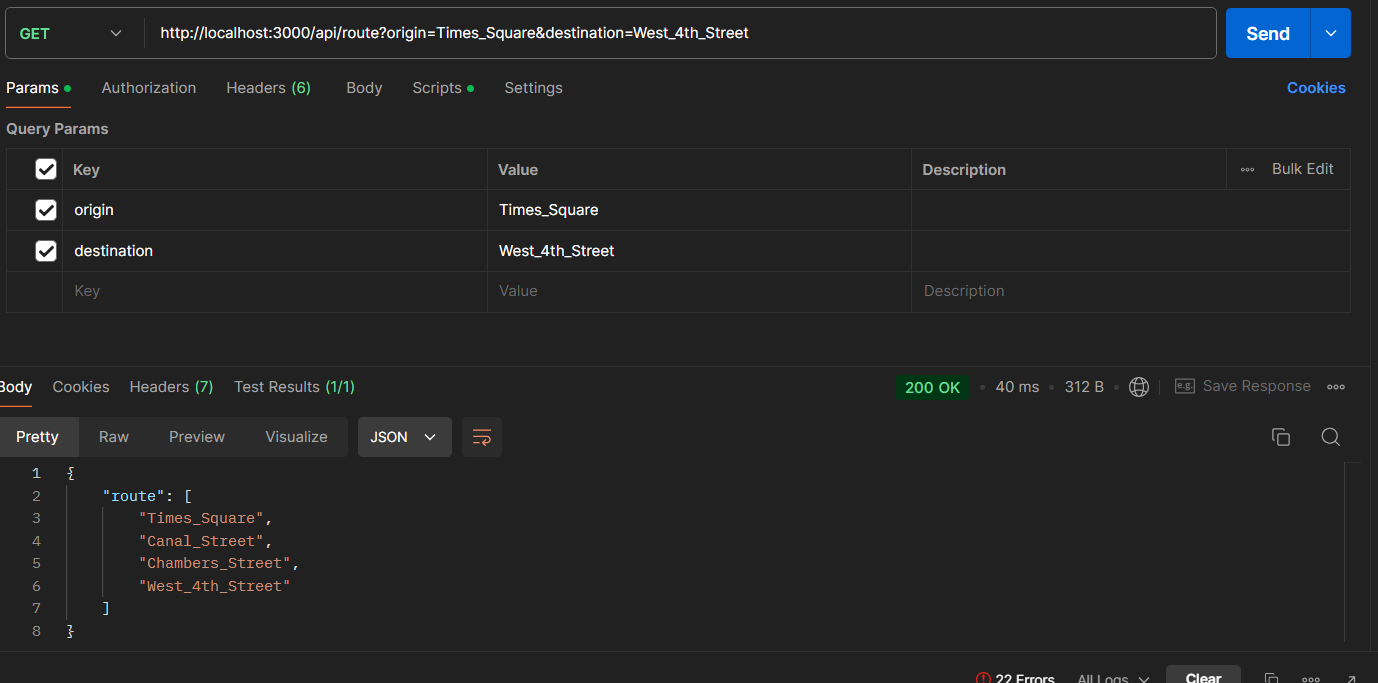
**3) Running the Application:**

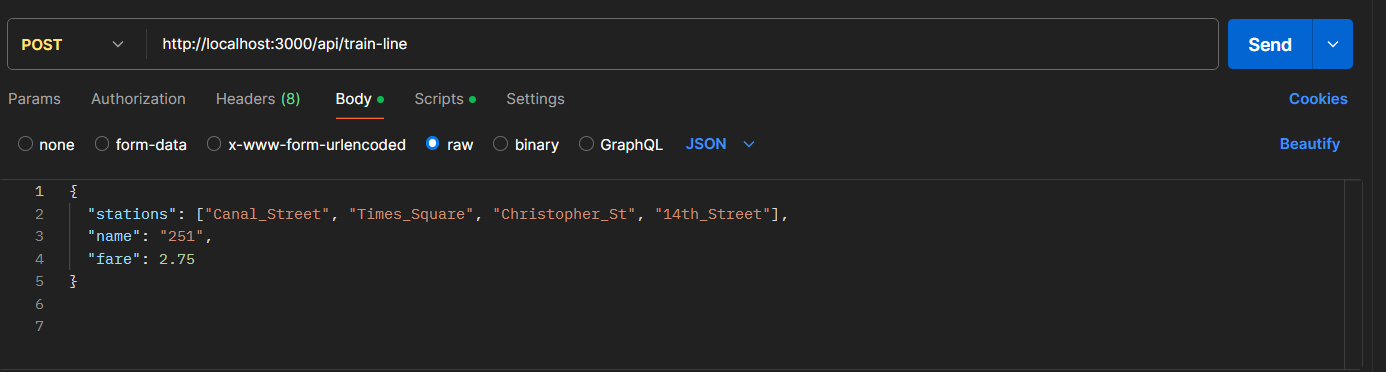
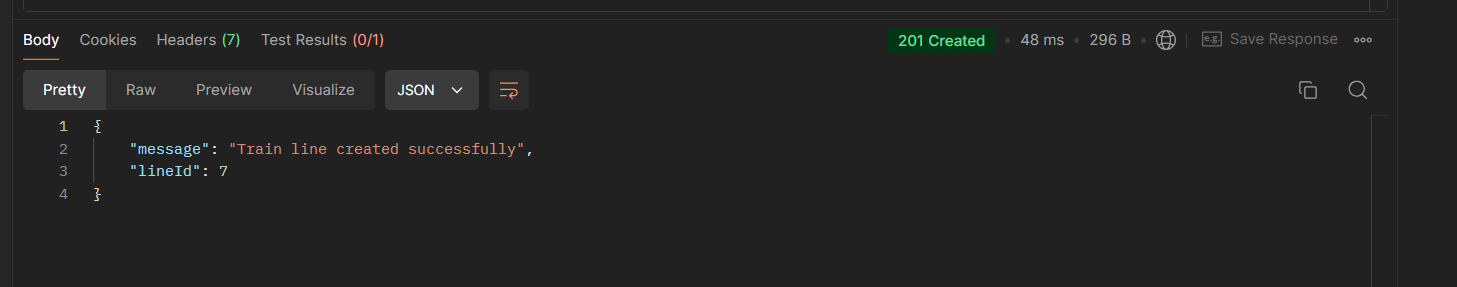
->Using Docker: Once Docker Compose is up, the app will be running at:

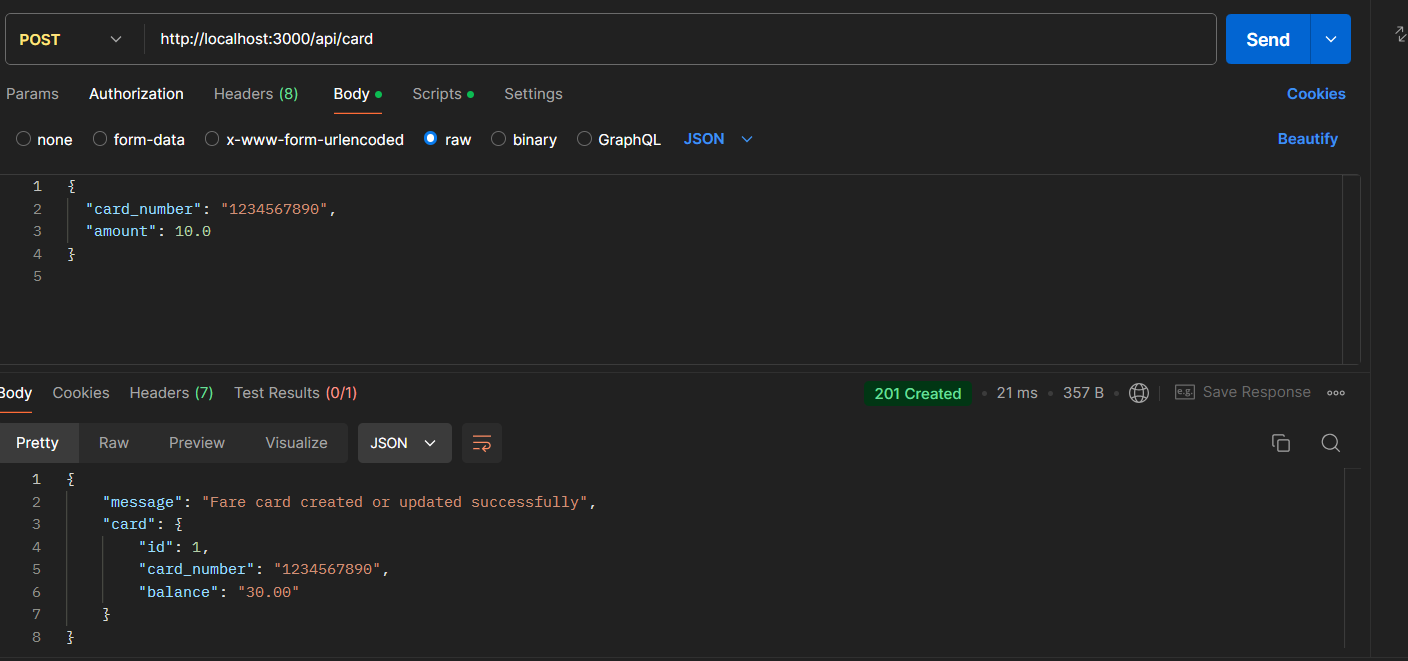
http://localhost:3000

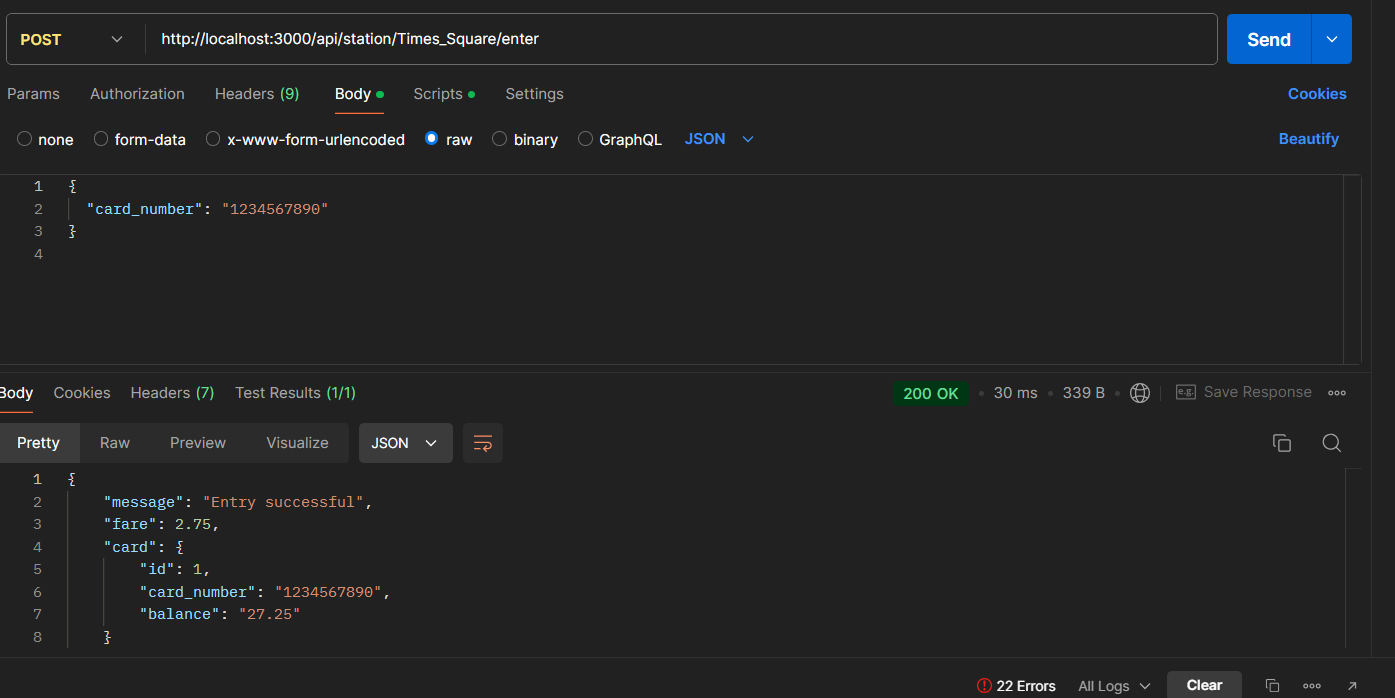
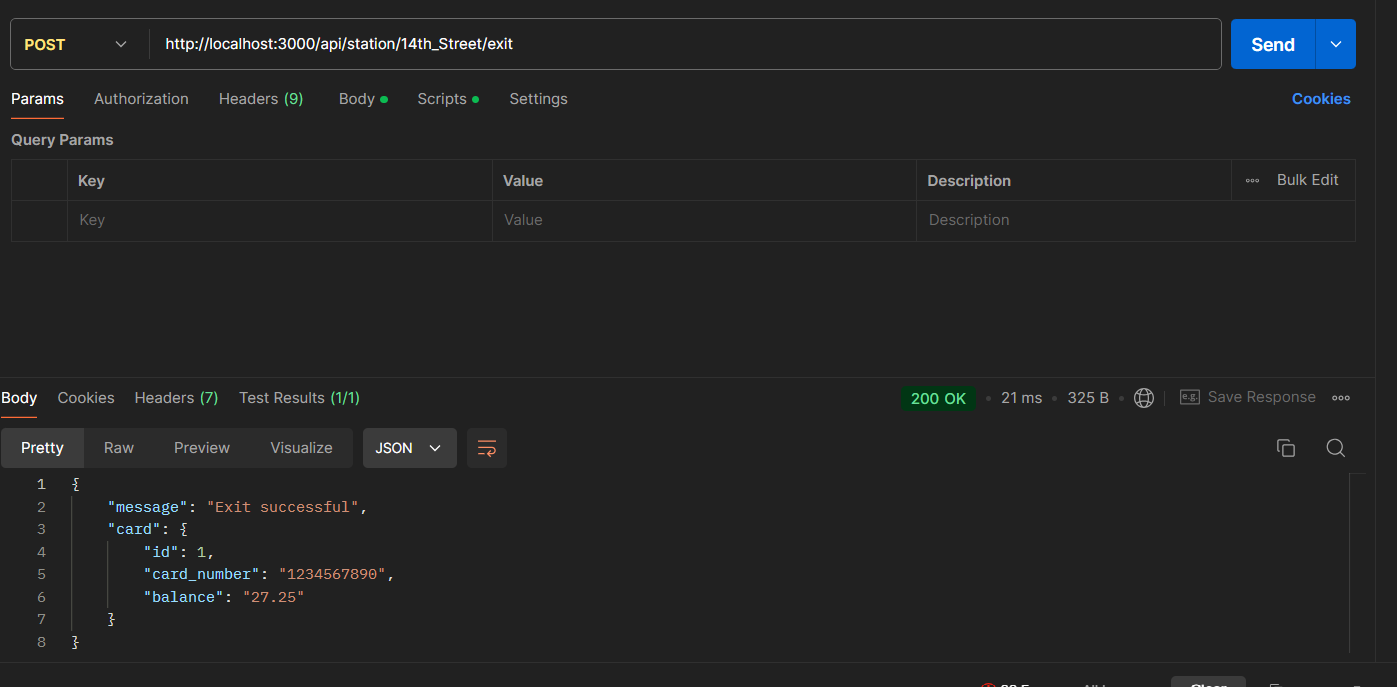
4) **API Endpoints:**

Testing all the Given Challenges:  
  
1) **Challenge 1:**   
   


2)  
  


3)  
  
  
  


4)  
  


5)  
  
  
  
  
6)  
  


## **5) Extensibility of Code:**

The code is organized into different modules to promote extensibility:

* Controllers handle the logic of each API.
* Models manage the database interactions.
* Routes define the API endpoints.
* Services can be extended to introduce more business logic without changing the core structure.
* Testing is separated using jest and supertest, ensuring isolated and scalable unit tests.

The system can easily be extended with additional features such as:

* Dynamic fare pricing.
* Multi-leg train routes with transfer stations.
* Enhanced fare card management with expiration dates or discounted fares.

**6) Testing:**

The application is tested using **Jest** for unit and integration testing to ensure that each part of the code works as expected. The tests are organized into different files corresponding to the various functionalities of the subway system API. Each test covers different parts of the system, ensuring comprehensive coverage and robustness.

#### **Running the Tests**

To run the tests, you can execute the following commands:

1. npx jest src/tests/trainRoutes.test.ts
2. npx jest src/tests/database.test.ts
3. npx jest src/tests/route.test.ts
4. npx jest src/tests/cardRoutes.test.ts

#### **Test Breakdown:**

* **trainRoutes.test.ts**: This test file ensures that the train routes and train line management (such as creating and fetching train lines) are functioning as expected. It validates the correct creation of train lines and checks if routes are correctly returned between stations.
* **database.test.ts**: This test ensures that the PostgreSQL database connections and queries are working properly. It validates that the database schema is correctly defined and that interactions such as inserting and fetching data from tables like stations, train\_lines, and fare\_cards are successful.
* **route.test.ts**: This file tests the functionality of calculating the shortest route between two stations. It validates that the findShortestRoute() function returns the optimal path, covering both valid and edge cases (e.g., when there is no possible route between stations).
* **cardRoutes.test.ts**: This test ensures that the fare card system works as expected. It covers scenarios such as creating a fare card, updating card balances, processing station entries/exits, and logging rides. Additionally, it tests for insufficient balance errors and correct fare deductions.

#### **Testing Strategy:**

* **Unit Testing**: Individual functions such as route finding and database operations are tested in isolation to ensure that each component works as expected.
* **Integration Testing**: The full API endpoints are tested to ensure that different components work together seamlessly, for example, checking that the fare deduction logic is applied correctly when a card is used at a station.

#### **Coverage:**

These tests provide comprehensive coverage for the core functionalities of the subway system, including creating train lines, calculating routes, managing fare cards, and interacting with the PostgreSQL database.

## **7) Conclusion:**

This Subway System API solves both challenges outlined in the task. It allows for efficient management of train lines, route calculation, and fare card handling. The code is modular and well-organized, making it easy to extend with additional functionality in the future.

**Key Features**:

* Efficient database querying using PostgreSQL.
* Route calculation with the shortest path using Breadth-First Search (BFS).
* Transaction management for atomic operations like creating train lines and updating fare card balances.

**8) Evaluation**

Below is an explanation of how my project fulfills the evaluation criteria:

#### **1. Does your code completely solve the questions?**

Yes, the code completely solves both Challenge 1 and Challenge 2 as follows:

* **Challenge 1**: The API allows for creating train lines with associated stations and calculating the shortest route between two stations using Breadth-First Search (BFS). The API correctly responds with the minimal station list required to travel between the origin and destination.
* **Challenge 2**: The API supports fare cards for entering and exiting stations. Fare deductions happen upon entry, and ride logs are created for tracking. The balance is returned after each station entry/exit, fulfilling the requirement for fare tracking.

#### **2. Is your code organized and well thought-out?**

Yes, the code is organized following modern best practices:

* **Controllers** handle the logic for managing train lines and fare cards.
* **Models** abstract database interactions, which makes it easy to modify or replace database logic.
* **Routes** define clean endpoints for different API functionalities.
* **Separation of Concerns** is well-maintained with the use of distinct layers for handling logic, routes, and database interactions.
* **Transaction Management** is utilize dfor atomic operations like creating train lines with stations.

#### **3. Would it be easy to extend your code to a more complex solution with more requirements?**

Yes, the code is highly modular, which makes it easy to extend:

* **Modular Structure**: The use of controllers, services, and models ensures that additional features can be implemented by simply adding new modules or extending existing ones without affecting the core logic.
* **Example Extensions**:
  + You could easily add dynamic fare calculations based on travel distance or peak hours by adjusting the fare card logic.
  + Support for station transfers can be implemented in the route-finding algorithm by modifying the BFS function.
  + Logging can be expanded to include ride history for passengers, which would only require adding new columns or tables to the database schema.

#### **4. How did you test your solution?**

The solution is tested using **Jest** and **Supertest** for unit and integration testing:

* **Unit Tests**: Functions like findShortestRoute() are tested to ensure that the logic returns the shortest path between stations.
* **Integration Tests**: API endpoints are tested for correct responses using Supertest. For example, tests ensure that creating a train line and calculating routes between stations return expected results.
* **Error Handling Tests**: Edge cases such as insufficient balance or a missing fare card are covered to ensure robustness.
* **Database Connection**: Tests ensure that the PostgreSQL database is properly connected and can handle queries like station lookups and fare card updates.