Samurai Train Services

In a precise and efficient manner, the Samurai Train service transported many commuters in Bangladesh to and from their destinations. However, as customer demand increased, the demands placed on the service infrastructure also increased. Following several decades of relatively minor enhancements, the backend system needs help to satisfy the ever-increasing customer demand. Fahim is an intelligent software developer who enjoys solving complex challenges. Upon receiving a call for suggestions for a new rail backend service from the country's transportation authorities, he recognized the opportunity to enhance the lives of the locals.

This task is overwhelming. However, Fahim plans to design an efficient system to accomplish these goals by allowing users to purchase tickets, manage train schedules, locate the most efficient route for commuting, handle wallet transactions, and provide the most efficient path to their destination. Fahim pulled together a group of developers who shared his interests and possessed various skills. They must improve the Samurai Train Service system.

Key challenges

Features will include (but are not limited to) listing all trains, all stations, specific user's wallet, adding money to particular user's wallet, purchasing tickets, trip and route planning, optimizing travel time and cost, etc.

- **Wallet Integration:** Integrating wallet functionality for users to make payments, manage balances, view transaction history, and seamlessly conduct financial transactions for ticket purchases and other services.
- **Train Management:** Developing a robust system to manage trains, their schedules, routes, and availability across all relevant platforms.
- **Station Management:** Implementing functionalities to manage stations, including locations, facilities, and associated train services, to ensure accurate routing and scheduling.

Not within the scope of this competition

- There is no interaction with external wallets. Instead, wallet information will be linked to the User data model. To update your wallet information, you must use appropriate APIs.
- User login and authorization are not necessary.
- We anticipate the implementation of an optimized algorithm for calculating routes, although managing concurrency is not included in the current scope.

Data Models

The following section will describe the basic data model given as input to your server application. You must create API endpoints and a suitable database schema to persist these data models. You may add extra fields as you see fit.

User

The `user` object represents the person using the app. The server will receive requests on behalf of an app user identified by `user_id.` The following model represents a user:

Field	Type	Description
user_id	integer	User's ID
user_name	string	User's full name
balance	integer	User's wallet balance

The following section will describe the API specification for adding a user.

Request Specification

- **URL**: /api/users
- **Method**: **POST**
- **Request model**:

```
{
  "user_id":      integer,    # user's numeric id
  "user_name":    string,     # user's full name
  "balance":      integer     # user's wallet balance
}
```

Successful Response

Upon successful operation, your API must return a 201 status code with the saved user object.

- **Response status**: 201 - Created
- **Response model**:

```
{
  "user_id":      integer,    # user's numeric id
  "user_name":    string,     # user's full name
  "balance":      integer     # user's wallet balance
}
```

Examples

Let's look at some example requests and responses.

Example request:

- Request URL: [POST] http://localhost:8000/api/users
- Content Type: application/json
- Request Body:

```
{
   "user_id": 1,
   "user_name": "Fahim",
   "balance": 100
}
```

Example successful response:

- Content Type: application/json
- HTTP Status Code: 201
- Response Body:

```
{
   "user_id": 1,
   "user_name": "Fahim",
   "balance": 100
}
```

Station

The `station` object represents a metro station where trains can stop, and users can get on and off the trains. The following model represents a station:

Field	Type	Description
station_id	integer	Station's ID
station_name	string	Station's name
longitude	float	Longitude coordinate
latitude	float	Latitude coordinate

The following section describes API specification for adding a station.

Request Specification

- **URL**: /api/stations
- **Method**: **POST**
- **Request model**

```
{
  "station_id":       integer,    # station's numeric id
  "station_name":     string,     # station's name
  "longitude":        float,      # coordinate longitude
  "latitude":         float       # coordinate latitude
}
```

Successful Response

Upon successful operation, your API must return a 201 status code with the saved station object.

- **Response status**: 201 - Created
- **Response model**

```
{
  "station_id":       integer,    # station's numeric id
  "station_name":     string,     # station's name
  "longitude":        float,      # coordinate longitude
  "latitude":         float       # coordinate latitude
}
```

Examples

Let's look at some example requests and response.

Example request:

- Request URL: [POST] http://localhost:8000/api/stations
- Content Type: application/json
- Request Body:

```
{
   "station_id": 1,
   "station_name": "Dhaka GPO",
   "longitude": 90.399452,
   "latitude": 23.777176
}
```

Example successful response:

- Content Type: application/json
- HTTP Status Code: 201

- Response Body:

```
{
   "station_id": 1,
   "station_name": "Dhaka GPO",
   "longitude": 90.399452,
   "latitude": 23.777176
}
```

Train

The `train` object represents a metro train. The following model represents a train:

Field	Type	Description
train_id	integer	Train's ID
train_name	string	Train's name
capacity	integer	Seating capacity

However, a train will go through a series of stations. Each train model will contain a nested field `stops` that includes the ordered list of stations where the train will stop for passenger operations. There will be at least two stops for each train. The data model for each stop will be represented as follows:

Field	Type	Description
station_id	integer	Station's ID
arrival_time	string	Arrival time in hh:mm
departure_time	string	Departure time in hh:mm
fare	integer	Fare from previous station

Note: The times here will be in 24-hour format, specifying hours and minutes (hh:mm). So, 1:30 PM will be represented as `13:30`. You can assume that the trains will always maintain the schedule.

The arrival time for the first station will be `null`, and the departure time for the last station will be `null.` A train can visit a station multiple times at different time schedules.

Note: The fare is given as the ticket price from the previous station in the ordered list. The first station in the list will always have a 0 fare.

The following section will describe API specifications for adding a train.

Request Specification

- **URL**: /api/trains
- **Method**: **POST**
- **Request model**

```
{
  "train_id":     integer,    # train's numeric id
  "train_name":   string,     # train's name
  "capacity":     integer,    # seating capacity
  "stops": [                  # list of stops
    {
      "station_id":       integer,    # station's id
      "arrival_time":     string,     # arrives at
      "departure_time"    string,     # leaves at
      "fare":             integer     # ticket cost
    },
    ...
  ]
}
```

Successful Response

Upon successful operation, your API must return a 201 status code with the saved train object.

- **Response status**: 201 - Created
- **Response model**

```
{
  "train_id":         integer,    # train's numeric id
  "train_name":       string,     # train's name
  "capacity":         integer,    # seating capacity
  "service_start":    string,     # service start time
  "service_ends":     string,     # service end time
  "num_stations":     integer     # number of stops
}
```

Here, `service_start` is the start time of the train at the first station, and `service_ends` is the end time of the train at the last station. Time schedule output should follow the same 24-hour time format shown in the input model.

Examples

Let's look at some example requests and response.

Example request:

- Request URL: [POST] http://localhost:8000/api/trains
- Content Type: application/json
- Request Body:

```
{
  "train_id": 1,
  "train_name": "Mahanagar 123",
  "capacity": 200,
  "stops": [
    {
      "station_id": 1,
      "arrival_time": null,
      "departure_time": "07:00",
      "fare": 0
    },
    {
      "station_id": 3,
      "arrival_time": "07:45",
      "departure_time": "07:50",
      "fare": 20
    },
    {
      "station_id": 4,
      "arrival_time": "08:30",
      "departure_time": null,
      "fare": 30
    }
  ]
}
```

Example successful response:

- Content Type: application/json
- HTTP Status Code: 201
- Response Body:

```
{
  "train_id": 1,
  "train_name": "Mahanagar 123",
  "capacity": 200,
  "service_start": "07:00",
  "service_ends": "08:30",
  "num_stations": 3
}
```

Features

The following sections will describe several API endpoints that adds various functionality for the end users.

Station

List All Stations

List down all stations in the ascending order of their IDs.

Request Specification

- **URL: /api/stations**
- **Method: GET**
- **Request model**: None

Successful Response

Upon successful operation, your API must return a 200 status code with the list of all station objects. The output should contain an empty list if there are no stations in the system.

- **Response status**: 200 - Ok
- **Response model**

```
{
  "stations": [
  {
    "station_id":       integer,    # station's numeric id
    "station_name":     string,     # station's name
    "longitude":        float,      # coordinate longitude
    "latitude":         float       # coordinate latitude
    },
    ...
  ]
}
```

If there are no stations, the response will look like the following:

```
{
  "stations": []
}
```

This is still considered a successful response.

Failed Response

For this endpoint, you do not have to handle failure responses.

Examples

Let's look at some example requests and responses.

Example request:

- Request URL: [GET] http://localhost:8000/api/stations

Example successful response:

- Content Type: application/json
- HTTP Status Code: 200
- Response Body:


```
{
  "stations": [
    {
      "station_id": 1,
      "station_name": "Dhaka GPO",
      "longitude": 90.399452,
      "latitude": 23.777176
    },
    {
      "station_id": 2,
      "station_name": "Motijheel",
      "longitude": 90.417458,
      "latitude": 23.733330
    },
    {
      "station_id": 3,
      "station_name": "Rajarbagh",
      "longitude": 90.4166667,
      "latitude": 23.7333333
    }
  ]
}
```


List All Train

List down all trains that have a stop at a given station. Sort them in the ascending order of departure time. If trains have the exact departure time, sort them according to their arrival time in ascending order. In sorting order, `null` values should appear before a time value for the respective fields.

If a tie between entries still exists, sort them based on their `train_id`s ascending order.

Request Specification

- **URL: /api/stations/{station_id}/trains #** `station_id` will be part of the path parameter.
- **Method: GET**
- **Request model**: None

Successful Response

Upon successful operation, your API must return a 200 status code with the list of all station objects.

- **Response status**: 200 - Ok
- **Response model**

```
{
  "station_id":               integer,    # the station's id
  "trains": [
    {
      "train_id":         integer,    # train's id
      "arrival_time":     string,     # arrives at
      "departure_time":   string      # leaves at
    },
    ...
  ]
}
```

If no train passes through the station in question, the response will look like the following:

```
{
  "station_id":               integer,    # the station's id
  "trains": []
}
```

This is still considered a successful response.

Failed Response

If a station does not exist with the given ID, your API must return a 404 status code with a message.

- **Response status**: 404 - Not Found

- **Response model**

```
{
  "message": "station with id: {station_id} was not found"
}
```

Replace `{station_id}` with the station ID from request.

Examples

Let's look at some example requests and responses.

Example request:

- Request URL: [GET] http://localhost:8000/api/stations/1/trains

Example successful response:

- Content Type: application/json
- HTTP Status Code: 200
- Response Body:

```
{
  "station_id": 1,
  "trains": [
    {
      "train_id": 1,
      "arrival_time": null,
      "departure_time": "07:00"
    },
    {
      "train_id": 2,
      "arrival_time": "06:55",
      "departure_time": "07:00"
    },
    {
      "train_id": 3,
      "arrival_time": "07:30",
      "departure_time": "08:00"
    }
  ]
}
```

Wallet

For this problem, you will not have to maintain a separate wallet. However, we will simulate a basic wallet functionality with the `user` object. The wallet ID of a user will be the same as its user's ID.

Get Wallet Balance

Find the wallet balance of a user from the wallet ID.

Request Specification

- **URL: /api/wallets/{wallet_id}** # Wallet ID is part of the path parameter.
- **Method: GET**
- **Request model**: None

Successful Response

Upon successful operation, your API must return a 200 status code with the wallet objects.

- **Response status**: 200 - Ok
- **Response model**

```
{
  "wallet_id":    integer,        # user's wallet id
  "balance":      integer,        # user's wallet balance
  "wallet_user":
  {
    "user_id":      integer,    # user's numeric id
    "user_name":    string      # user's full name
  }
}
```

Failed Response

If a wallet does not exist with the given ID, your API must return a 404 status code with a message.

- **Response status**: 404 - Not Found
- **Response model**

```
{
  "message": "wallet with id: {wallet_id} was not found"
}
```

Replace `{wallet_id}` with the wallet ID from request.

Examples

Let's look at some example requests and responses.

Example request:

- Request URL: [GET] http://localhost:8000/api/wallets/1

Example successful response:

- Content Type: application/json
- HTTP Status Code: 200
- Response Body:


```
{
  "wallet_id": 1,
  "balance": 100,
  "wallet_user": {
    "user_id": 1,
    "user_name": "Fahim"
  }
}
```


Example request:

- Request URL: [GET] http://localhost:8000/api/wallets/67

Example failed response:

- Content Type: application/json
- HTTP Status Code: 404
- Response Body:


```
{
  "message": "wallet with id: 67 was not found"
}
```


Add Wallet Balance

Add funds to the user's wallet. Added fund must be an integer within range 100 - 10000 Taka. Any value out of range should be responded with appropriate errors.

Request Specification

- **URL: /api/wallets/{wallet_id}** # Wallet ID is part of the path parameter.

- **Method: PUT**
- **Request model**:

```
{
  "recharge":     integer         # Fund to be added to the wallet
}
```

Successful Response

Upon successful operation, your API must return a 200 status code with the updated wallet objects.

- **Response status**: 200 - Ok
- **Response model**

```
{
  "wallet_id":    integer,        # user's wallet id
  "balance":      integer,        # user's wallet balance
  "wallet_user":
  {
    "user_id":      integer,    # user's numeric id
    "user_name":    string      # user's full name
  }
}
```

Failed Response

If a wallet does not exist with the given ID, your API must return a 404 status code with a message.

- **Response status**: 404 - Not Found
- **Response model**

```
{
  "message": "wallet with id: {wallet_id} was not found"
}
```

Replace `{wallet_id}` with the wallet ID from request.

If the recharge amount is out of range (not between 100 and 10000), then return a 400 status code with a message.

- **Response status**: 400 - Bad Request
- **Response model**

```
{
    "message": "invalid amount: {recharge_amount}"
}
```

Replace `{recharge_amount}` with the amount that was sent with the request body.

Examples

Let's look at some example requests and response.

Example request:

- Request URL: [PUT] http://localhost:8000/api/wallets/1
- Content Type: application/json
- Request Body:

```
{
  "recharge": 150
}
```

Example success response:

- Content Type: application/json
- HTTP Status Code: 200
- Response Body:

```
{
  "wallet_id": 1,
  "wallet_balance": 250,
  "wallet_user": {
    "user_id": 1,
    "user_name": "Fahim"
  }
}
```

Example request when wallet does not exist:

- Request URL: [PUT] http://localhost:8000/api/wallets/67
- Content Type: application/json
- Request Body:

```
{
  "recharge": 150
}
```

Example failed response:

- Content Type: application/json
- HTTP Status Code: 404
- Response Body:

```
{
  "message": "wallet with id: 67 was not found"
}
```

Example request when amount is out of range:

- Request URL: [PUT] http://localhost:8000/api/wallets/1
- Content Type: application/json
- Request Body:

```
{
  "recharge": 3
}
```

Example failed response:

- Content Type: application/json
- HTTP Status Code: 400
- Response Body:

```
{
  "message": "invalid amount: 3"
}
```

Ticketing

Purchase Ticket

Users can use their wallet balance to purchase tickets from stations A to B. The cost is calculated as the sum of the fares for each pair of consecutive stations along the route. Upon successful purchase, your API should respond with the station in order visited by one or more trains and the remaining wallet balance. If the wallet does not have sufficient funds, respond with an error specifying the shortage amount. If it is impossible to reach station B from station A within the day or due to a lack of trains, respond with an error specifying that no tickets are available.

Note: The user may want to change the train at a particular station for a cheaper trip. Partial scoring will be awarded if your API fails to find an optimal route. You can assume that the start and destination stations will always differ, and the user must complete the trip within the current day.

Request Specification

- **URL: /api/tickets**
- **Method: POST**
- **Request model**:

```
{
  "wallet_id":    int,            # user's wallet id (same as user id)
  "time_after":   string,         # time (24 hours hh:mm) after which user
wants to purchase a ticket
  "station_from": int,            # station_id for the starting station
  "station_to":   int             # station_id for the destination station
}
```

Successful Response

Upon successful operation, your API must return a 201 status code with the generated ticket ID, remaining balance, wallet ID, and a list of all stations in order of visits. You should also include each station's train ID and arrival and departure schedules in the output object. Departure time should follow the same time format as in the input model. For the first station, the arrival time should be `null,` and for the last station, the departure time should be `null.`

- **Response status**: 201 - Created
- **Response model**

```
{
  "ticket_id":                int,        # generate a unique integer ticket
ID
  "wallet_id":                int,        # user's wallet id (same as user
id)
  "balance":                  integer,    # remaining balance
  "stations": [
    {
      "station_id":       integer,    # station's numeric id
```

```
      "train_id":         integer,    # train's id user is riding
      "arrival_time":     string,     # arrival time
      "departure_time":   string      # departure time
    },
    ...
  ]
}
```

Failed Response

Insufficient balance

If the wallet has insufficient balance for purchasing the ticket, respond with HTTP 402 - Payment Required and a message showing the shortage amount.

- **Response status**: 402 - Payment Required
- **Response model**

```
{
  "message": "recharge amount: {shortage_amount} to purchase the ticket"
}
```

Replace `{shortage_amount}` with the amount the user is short of the ticket's cost.

Note: This amount may vary depending on whether you can find an optimal-cost route for the user. Sub-optimal solutions may be awarded with partial scores.

Unreachable station

If it is impossible to reach the destination station from the starting station, output a message with HTTP 403 - Forbidden and a message for the user.

- **Response status**: 403 - Forbidden
- **Response model**

```
{
  "message": "no ticket available for station: {station_from} to station:
{station_to}"
}
```

Replace `{station_from}` and `{station_to}` as specified in the input model.

Examples

Let's look at some example requests and responses.

Example request:

- Request URL: [POST] http://localhost:8000/api/tickets
- Content Type: application/json
- Request Body:

```
{
  "wallet_id":    3,
  "time_after":   "10:55",
  "station_from": 1,
  "station_to":   5
}
```

Example successful response:

- Content Type: application/json
- HTTP Status Code: 201
- Response Body:

```
{
  "ticket_id": 101,
  "balance": 43,
  "wallet_id": 3,
  "stations": [
    {
      "station_id": 1,
      "train_id": 3,
      "departure_time": "11:00",
      "arrival_time": null,
    },
    {
      "station_id": 3,
      "train_id": 2,
      "departure_time": "12:00",
      "arrival_time": "11:55"
    },
    {
      "station_id": 5,
      "train_id": 2,
      "departure_time": null,
      "arrival_time": "12:25"
    }
```

```
  ]
}
```

Example request for no tickets:

- Request URL: [POST] http://localhost:8000/api/tickets
- Content Type: application/json
- Request Body:

```
{
  "wallet_id":    3,
  "time_after":   "10:55",
  "station_from": 1,
  "station_to":   5
}
```

Example failed response:

- Content Type: application/json
- HTTP Status Code: 403
- Response Body:

```
{
  "message":  "no ticket available for station: 1 to station: 5"
}
```

Example request for insufficient funds:

- Request URL: [POST] http://localhost:8000/api/tickets
- Content Type: application/json
- Request Body:

```
{
  "wallet_id":    3,
  "time_after":   "10:55",
  "station_from": 1,
  "station_to":   5
}
```

Example failed response:

- Content Type: application/json
- HTTP Status Code: 402
- Response Body:

```
{
  "message":  "recharge amount: 113 to purchase the ticket"
}
```

Planning

Plan Optimal Routes

Users can request the API to generate an optimal route between two stations. Here, the user is just planning a trip and not using any wallet balance—the optimal path calculated is based on the shortest time to complete or the cheapest route.

Note: You can assume that the start and destination stations will always differ, and the user must complete the trip within the same day.

Request Specification

- **URL: /api/routes?from={station_from}&to={station_to}&optimize={cost|time}**
- **Method: GET**
- **Request model**: None

Here, `station_from` and `station_to` are station IDs, and optimize will be either `cost` or `time`.

Successful Response

Upon successful operation, your API must return a 200 status code with total time, total cost, and a list of all stations in order of visits. You should also include each station's train ID and arrival and departure schedules in the output object. Departure time should follow the same time format as in the input model. For the first station, the arrival time should be `null,`; for the last station, the departure time should be `null.`

- **Response status**: 200 - OK
- **Response model**

```
{
  "total_cost":               int,        # total cost
  "total_time":               int,        # total time in minutes
  "stations": [
    {
      "station_id":       integer,    # station's numeric id
      "train_id":         integer,    # train's id user is riding
      "arrival_time":     string,     # arrival time
```

```
      "departure_time":   string      # departure time
    },
    ...
  ]
}
```

Failed Response

Unreachable station

If it is not possible to reach the destination station from the starting station, output a message with HTTP 403 - Forbidden and a message for the user.

- **Response status**: 403 - Forbidden
- **Response model**

```
{
  "message": "no routes available from station: {station_from} to station:
{station_to}"
}
```

Replace `{station_from}` and `{station_to}` as specified in the input model.

Examples

Let's look at some example requests and response.

Example request:

- Request URL: [GET] http://localhost:8000/api/routes?from=1&to=5&optimize=cost

Example successful response:

- Content Type: application/json
- HTTP Status Code: 200
- Response Body:

```
{
  "total_cost": 101,
  "total_time": 85,
  "stations": [
    {
      "station_id": 1,
      "train_id": 3,
```

```
      "departure_time": "11:00",
      "arrival_time": null,
    },
    {
      "station_id": 3,
      "train_id": 2,
      "departure_time": "12:00",
      "arrival_time": "11:55"
    },
    {
      "station_id": 5,
      "train_id": null,
      "departure_time": null,
      "arrival_time": "12:25"
    }
  ]
}
```

Example request for no routes:

- Request URL: [GET] http://localhost:8000/api/routes?from=1&to=5&optimize=cost

Example failed response:

- Content Type: application/json
- HTTP Status Code: 403
- Response Body:

```
{
  "message":  "no routes available from station: 1 to station: 5"
}
```

Dos and Don'ts:

- The zip file should contain a readme file (readme.md) including the team name (column A), Institution name, and the list of emails of all the team members. You can add instructions there, but we will not use them in the judging process. This is useful in case some team uses the incorrect name of zip.
- Make sure your docker file and docker-compose file exist in the `root path` of your folder
- Make sure your dockerfile launches the application with an empty database. Store your environment variables in a `.env` file.
- Add a simple `dockerfile` to build an image from your solution. Remember to mention the environment variables you use in the dockerfile with proper values.
- Add a `compose.yaml` file to add all local dependencies of your solution.

- Optionally, add a `.dockerignore` file to tell the docker which files or folders you do not want to copy.
- We will use port `8000`. Make sure you are exposing the correct port in your dockerfile.
- **DO NOT** explicitly declare a custom network in your docker compose file.