Database Project 2

Harrold Tok Kwan Hang 12212025, Saruulbuyan Munkthur 12212643

June 6, 2024

Basic Information of Your Group

1. Group Members: Harrold Tok 12212025, Saruulbuyan Munkhtur 12212643
2. Tuesday 2pm 507 Lab Session
   (a) **Saruulbuyan Munkhtur (Student ID: 12212643)(50%)**
   - Frontend Development
   - Frontend-Backend Connectivity
   - High-Concurrency Pressure Test
   - Views, Triggers, user permissions, data visualization

   (b) **Harrold Tok (Student ID: 12212025) (50%)**
   - Basic Functions
   - Advance Functions
   - Frontend Development
   - Price Import

1 Introduction

Our API follows REST principles with resource-oriented URLs, JSON requests/responses, and standard HTTP methods.

2 Station Management Module

This module provides functions for managing stations in a transportation system. It interacts with the `Station`, `Lines_Station`, and `Line` models using the Sequelize ORM. Copy

2.1 Functions

- **createStation(lineName, station_english_name, district, intro, chinese_name, position, status):** Creates a new station with the provided details and associates it with a line at the specified position and status.
  - **Endpoint**: `/stations`
  - **Method**: `POST`

  - **URL Example**: `http://localhost:3000/stations`

- **getAllStations():** Retrieves all stations from the database.
  - **Endpoint**: `/stations`
  - **Method**: `GET`
  - **URL Example**: `http://localhost:3000/stations`

- **getStationById(stationId):** Retrieves a station by its ID.
  - **Endpoint**: `/stations/:id`
  - **Method**: `GET`
  - **URL Example**: `http://localhost:3000/stations/1`

- **updateStation(lineName, station_english_name, district, intro, chinese_name, position, status):** Updates the details of a station and modifies its position and status on the associated line.
  - **Endpoint**: `/stations/:id`
  - **Method**: `PUT`
  - **URL Example**: `http://localhost:3000/stations/1`

- **deleteStation(stationId):** Deletes a station by its English name and updates the positions of the remaining stations on the line.
  - **Endpoint**: `/stations/:id`
  - **Method**: `DELETE`
  - **URL Example**: `http://localhost:3000/stations/1`

- **updatePositionsAfterInsertion(lineName, lastInsertedPosition, x):** Updates the positions of stations on a line after a new station is inserted, incrementing the positions of stations at or after the inserted position by `x`.
  - **Endpoint**: `/stations/updatePositionsAfterInsertion`
  - **Method**: `POST`
  - **URL Example**: `http://localhost:3000/stations/updatePositionsAfterInsertion`

2.2 Dependencies

The module requires the following dependencies:

- `Station` model: Represents a station in the database.
- `Lines_Station` model: Represents the association between a station and a line.
- `Line` model: Represents a line in the transportation system.
- `Sequelize` library: Provides an ORM for interacting with the database.
- `modifyPosition`, `modifyStationStatus`, `placeStationOnLine` functions: Helper functions for modifying station positions and statuses.

3 Lines Management Module

This module provides functions for managing lines in a transportation system. It interacts with the `Line` and `Station` models using the Sequelize ORM.

- **getLines():** Retrieves all lines from the database.
  - **Endpoint**: `/lines`
  - **Method**: `GET`
  - **URL Example**: `http://localhost:3000/lines`
- **getLineById(id):** Retrieves a line by its ID.
  - **Endpoint**: `/lines/:id`
  - **Method**: `GET`
  - **URL Example**: `http://localhost:3000/lines/1`
- **createLine(lineName, intro, mileage, color, first_opening, url, start, end):** Creates a new line with the provided details.
  - **Endpoint**: `/lines`
  - **Method**: `POST`
  - **URL Example**: `http://localhost:3000/lines`
- **updateLine(id, lineData):** Updates the details of a line.
  - **Endpoint**: `/lines/:id`
  - **Method**: `PUT`
  - **URL Example**: `http://localhost:3000/lines/1`
- **deleteLine(id):** Deletes a line by its ID.
  - **Endpoint**: `/lines/:id`
  - **Method**: `DELETE`
  - **URL Example**: `http://localhost:3000/lines/1`
- **findNthStation(lineName, stationNameInput, position):** Finds the nth station on a line based on the provided inputs.
  - **Endpoint**: `/lines/find-nth-position`
  - **Method**: `POST`
  - **URL Example**: `http://localhost:3000/lines/find-nth-position`
- **deleteLineStation(lineName, stationName):** Deletes a station from a line.
  - **Endpoint**: `/lines/delete-from-station`
  - **Method**: `DELETE`
  - **URL Example**: `http://localhost:3000/lines/delete-from-station`
- **verifyStation(lineName, stationName):** Verifies if a station exists on a line.
  - **Endpoint**: `/lines/verify-station`
  - **Method**: `POST`
  - **URL Example**: `http://localhost:3000/lines/verify-station`

- **placeStationsOnLine(lineName, stationNames, position, status):** Places multiple stations on a line with the specified positions and status.
  - **Endpoint**: `/lines/place-stations-on-line`
  - **Method**: `POST`
  - **URL Example**: `http://localhost:3000/lines/place-stations-on-line`

3.1 Dependencies

The module requires the following dependencies:

- `api` module: Provides the API for making HTTP requests to the server.

4 Ride Management Module

This module provides functions for managing rides in a transportation system. It interacts with the ride-related endpoints of an API.

4.1 Functions

- **getAllRides():** Retrieves all rides from the database.
  - **Endpoint**: `/rides`
  - **Method**: `GET`
  - **URL Example**: `http://localhost:3000/rides`
- **getAllRidesP():** Retrieves all rides for a passenger from the database.
  - **Endpoint**: `/rides/get-all-rides-passenger`
  - **Method**: `GET`
  - **URL Example**: `http://localhost:3000/rides/get-all-rides-passenger`
- **registerRideUsingCard(ID, StartStation, StartTime):** Registers a new ride using a card with the provided details.
  - **Endpoint**: `/rides/register-ride-using-card`
  - **Method**: `POST`
  - **URL Example**: `http://localhost:3000/rides/register-ride-using-card`
- **exitRideUsingCard(id, ID, EndStation, EndTime):** Marks the end of a ride using a card with the provided details.
  - **Endpoint**: `/rides/exit-using-card/:ride_id`
  - **Method**: `PUT`
  - **URL Example**: `http://localhost:3000/rides/exit-using-card/1`
- **registerRideUsingPassenger(ID, StartStation, StartTime):** Registers a new ride using passenger details.
  - **Endpoint**: `/rides/register-ride-using-passenger`
  - **Method**: `POST`
  - **URL Example**: `http://localhost:3000/rides/register-ride-using-passenger`

- **exitRideUsingPassenger(id, EndStation, EndTime):** Marks the end of a ride using passenger details.
  - **Endpoint**: `/rides/exit-using-passenger/:ride_id`
  - **Method**: `PUT`
  - **URL Example**: `http://localhost:3000/rides/exit-using-passenger/1`
- **nthParamSearch(startStation, endStation, minStartTime, maxStartTime, minEndTime, maxEndTime, minPrice, maxPrice, status):** Searches for rides based on multiple parameters.
  - **Endpoint**: `/rides/nth-param-search`
  - **Method**: `POST`
  - **URL Example**: `http://localhost:3000/rides/nth-param-search`
- **reloadCard(Code, amount):** Reloads a card with a specified amount.
  - **Endpoint**: `/rides/reload-card/:Code`
  - **Method**: `PUT`
  - **URL Example**: `http://localhost:3000/rides/reload-card/1234`

5 Graph Service Module

This module provides functions for managing the graph-related operations in a transportation system. It interacts with the graph-related endpoints of an API.

5.1 Functions

- **getShortestPath(startNodeName, endNodeName):** Retrieves the shortest path between two nodes.
  - **Endpoint**: `/graph/shortest-path`
  - **Method**: `GET`
  - **URL Example**: `http://localhost:3000/graph/shortest-path?startNodeName=StationA&endNodeName=StationB`
- **getShortestPathWithBus(startNodeName, endNodeName):** Retrieves the shortest path including bus routes between two nodes.
  - **Endpoint**: `/graph/shortest-path-bus`
  - **Method**: `GET`
  - **URL Example**: `http://localhost:3000/graph/shortest-path-bus?startNodeName=StationA&endNodeName=Statio`
- **getAdjacencyList(stationName):** Retrieves the adjacency list of a station.
  - **Endpoint**: `/graph/adjacency-list/:stationName`
  - **Method**: `GET`
  - **URL Example**: `http://localhost:3000/graph/adjacency-list/StationA`
- **updateStationStatus(stationName, updatedStatus):** Updates the status of a station.
  - **Endpoint**: `/graph/update-status/:stationName`
  - **Method**: `PUT`
  - **URL Example**: `http://localhost:3000/graph/update-status/StationA`

- **getBusesAtStations(station1, station2):** Retrieves the buses between two stations.
  - **Endpoint**: `/graph/get-buses`
  - **Method**: `GET`
  - **URL Example**: `http://localhost:3000/graph/get-buses?station1=StationA&station2=StationB`

5.2 Dependencies

The module requires the following dependencies:

- `api` module: Provides the API for making HTTP requests to the server.

6 Error Handling

All functions in this project use try-catch blocks to handle errors. If an error occurs during the execution of a function, an appropriate error message is thrown.

7 Advance Requirements

7.1 Database Configuration with Sequelize

This module sets up a connection to a database using Sequelize, an ORM (Object-Relational Mapping) library for Node.js. It uses environment variables to configure the database connection details.

7.2 Source Code

```
require('dotenv').config();
const Sequelize = require('sequelize');

const sequelize = new Sequelize(process.env.DB_NAME, process.env.DB_USER,
    ↪ process.env.DB_PASSWORD, {
  host: process.env.DB_HOST,
  port: process.env.DB_PORT,
  dialect: process.env.DB_DIALECT,
  pool: {
    max: 15,
    min: 0,
    acquire: 30000,
    idle: 10000
  },
});

module.exports = sequelize;
```

7.3 Explanation

- **Environment Variables:** The `dotenv` package is used to load environment variables from a `.env` file into `process.env`. This allows sensitive information such as database credentials to be stored securely outside of the source code.

- `DB_NAME`: The name of the database.
- `DB_USER`: The username to connect to the database.
- `DB_PASSWORD`: The password for the database user.
- `DB_HOST`: The hostname of the database server.
- `DB_PORT`: The port number on which the database server is running.
- `DB_DIALECT`: The database dialect (e.g., `mysql`, `postgres`, `sqlite`, `mssql`).

- **Sequelize Initialization:** An instance of `Sequelize` is created using the database connection details.
  - `host`: The hostname where the database is hosted.
  - `port`: The port number where the database server is listening.
  - `dialect`: The type of database being used.
  - `pool`: An optional configuration object to control the pool settings.
    - `max`: The maximum number of connections in the pool.
    - `min`: The minimum number of connections in the pool.
    - `acquire`: The maximum time, in milliseconds, that pool will try to get a connection before throwing an error.
    - `idle`: The maximum time, in milliseconds, that a connection can be idle before being released.
- **Module Export:** The configured `sequelize` instance is exported, making it available for use in other parts of the application.

7.4 MySQL

This API uses MySQL as it's database. The java import script uses:

`url="jar://$USER_HOME$/Downloads/mysql-connector-j-8.4.0/mysql-connector-j-8.4.0.jar!/".`

7.5 Path Query

API Name: getShortestPath

Purpose

This API is used to perform a path query between two stations in the subway system.

Parameters

- `start_station`: [string] The starting station for the path query.
- `end_station`: [string] The destination station for the path query.

Return Value

```
    exports.getShortestPath = async (startNodeName, endNodeName) => {
  try {
    const path = await shortestPath(startNodeName, endNodeName);
    return path;
  } catch (error) {
    throw new Error('Failed to find shortest path');
  }
};
```

[list of strings] A list of stations representing the path from the start station to the end station.

Description

For each station in the `lines_station` table, we treat it as a node and create a graph. Then, we run Dijkstra's algorithm to find the shortest path between two nodes.

7.6 Station Status

API Name: updateStatus

Purpose

This API is used to update the status of a station.

Parameters

- `start_station`: [string] Station Name
- `status`: [string] Status

Return Value

Alter the status of a station in the stations table.

Description

We use sequelize findAll function to get all stations with the same name as the input. Then we use sequelize .update function to modify the status.

7.7 Business Carriage

When registering for a ride, the user is able to pick business carriage, where the price will double. In our database, we simply add a 'Class' Attribute, which holds either 'ECONOMY' or 'BUSINESS'.

7.8 Integration of Buses and Subways

API Name: getBusesAtStations

Purpose

This API is used to find all bus connections between two stations. Furthermore, if no path is found, we include edges that are connected by the same bus and run our shortest path algorithm again, to create a path with stations and buses.

Parameters

- `start_station`: [string] The starting station.
- `end_station`: [string] The destination station.

Return Value

[list of strings] A list of stations joined by a bus

Description

We run the query to return buses that connect two stations

```
const query = `
  SELECT DISTINCT t1.entrance,  t1.bus_info FROM station_buses AS t1
  JOIN station_buses AS t2 ON t1.bus_info = t2.bus_info
  WHERE t1.station_name = :station1 AND t2.station_name = :station2;
`;
```

7.9 Multi-parameter Search

In the interface, you can input multiple fields to be passed to the backend. The initial query returns all the rides. Parameters include start station, end station, start time, end time, price, and status.

```
let query1 = `SELECT * FROM cardid_rides cr JOIN cards c on cr.user_code = c.code WHERE ride_id > 0 `;
let query2 = `SELECT * FROM userid_rides u JOIN passengers p ON u.user_id = p.id_number WHERE ride_id > 0 `;
```

If the field is not empty, we append to the query. For example, if the user fills in start station as 'Luohu', we append to query1 and query2 the string 'AND start_station = 'Luohu".

7.10 Back-end Server

The back-end server of the project is implemented using Node.js and Express.js. The server is responsible for handling HTTP requests, interacting with the database, and providing API endpoints for the front-end application. The main components of the back-end server include:

- **Express.js:** Express.js is used as the web application framework for building the server. It provides a robust set of features for handling routes, middleware, and request/response processing.
- **Sequelize:** Sequelize is used as the Object-Relational Mapping (ORM) library for interacting with the PostgreSQL database. It provides a convenient way to define models, perform database queries, and manage database transactions.
- **API Routes:** The server defines various API routes for handling different functionalities of the application. These routes include endpoints for user authentication, station management, line management, and more. Each route is mapped to a specific controller function that handles the request and sends the appropriate response.
- **Controllers:** The controller functions are responsible for processing the incoming requests, interacting with the database through Sequelize models, and sending the response back to the client.
- **Services:** The service functions contain the business logic of the application and performs operations such as creating, reading, updating, and deleting records in the database.
- **Error Handling:** The server implements error handling middleware to catch and handle any errors that occur during the request processing. It sends appropriate error responses to the client and logs the errors for debugging purposes.

Here's an example of a route definition in the backend server:

```
1 const express = require('express');
2 const router = express.Router();
3 const stationController = require('../controllers/stationController');
4 router.get('/', stationController.getAllStations);
5 router.get('/:id', stationController.getStationById);
```

```
router.post('/', stationController.createStation);
router.put('/:id', stationController.updateStation);
router.delete('/:id', stationController.deleteStation);
module.exports = router;
```

In this example, the `stationController` is imported, and various routes are defined for handling station-related operations. The routes are mapped to the corresponding controller functions, which handle the requests and send the responses.

```
    exports.createStation = async (req, res) => {
  try {
    const { lineName, station_english_name, district, intro, chinese_name,
        ↪ position, status } = req.body;
    const newStation = await stationService.createStation(lineName,
        ↪ station_english_name, district, intro, chinese_name, position,
        ↪ status);
    res.status(201).json(newStation);
  } catch (error) {
    res.status(500).json({ error: 'Failed to create station' });
  }
};
```

The controller functions have try and catch functions ensuring error handling. The back-end server is tested thoroughly to ensure the correctness of the API endpoints, error handling, and database interactions. Integration tests are performed to verify the end-to-end functionality of the server. By implementing a robust and scalable back-end server using Node.js, Express.js, and Sequelize, the project achieves efficient request handling, seamless database integration, and provides a reliable API for the front-end application to consume.

7.11 Page Display Design

The front-end of the project is built using React, the most popular JavaScript library for building user interfaces. The page display design focuses on creating an intuitive and visually appealing user experience. The main components of the page display design include:

- **React Components:** The user interface is divided into reusable React components. Each component represents a specific part of the page and encapsulates its own structure, styling, and behavior. This modular approach allows for easier maintenance and reusability of code.
- **Navigation:** The application includes a navigation menu or header component that allows users to easily switch between different pages or sections of the application. The navigation is implemented using React Router, which enables client-side routing and provides a seamless navigation experience.
- **Form Handling:** Forms are used for user input, such as creating new stations or updating line information. React's form handling capabilities, along with libraries like Formik or React Hook Form, are utilized to manage form state, validation, and submission.
- **Data Visualization:** The application incorporates data visualization components to present information in a visually appealing and understandable manner. Charts, graphs, and maps are used to display station details, line routes, and other relevant data. Libraries such as Chart.js or Leaflet can be used for creating interactive visualizations.
- **Error Handling and Feedback:** The page display design includes proper error handling and user feedback mechanisms. Error messages are displayed to the user when input validation fails or when server-side errors occur. Loading indicators and success messages are used to provide feedback to the user about the status of their actions.

- **Styling and Theming:** The application follows a consistent visual style and theme throughout all the pages. CSS stylesheets are used to define the styling of components, ensuring a cohesive and visually appealing design.

7.12 Users and triggers

Users can log on as simple_customer or superusers. As a simple_customer, you are granted SELECT, INSERT, UPDATE on Rides and Graph operations and only allowed to view other pages. As a superuser, you are equipped with all GRANTS, allowing a full reconstruction of the subway system if so wish.

Triggers are placed in tables which undergoes rapid inserts such as rides table. The trigger ensures no field in the tuple is empty before insertion and will throw an error otherwise.

7.13 Big Data Management and Views

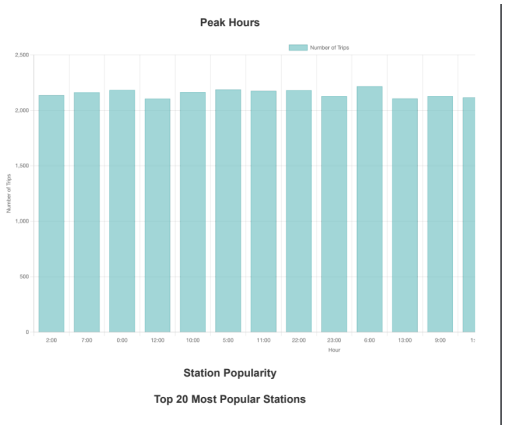

Figure 1: Average travel time

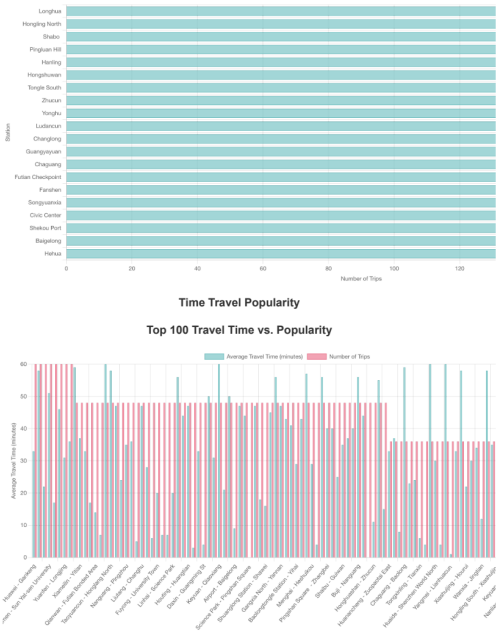

Figure 2: Station to Station Popularity

Figure 3: Station Popularity

Our viewsService.js functions getStationPopularity, getAvgTravelTime, getPeakHours, and getStationToStationPopularity fetch various datasets from an API, which are derived from views in a database managed using DataGrip. These views aggregate and summarize transit system data to provide insights into station popularity, average travel times, peak hours, and station-to-station travel patterns. Utilizing JavaScript graph libraries such as Chart.js, D3.js, or Highcharts, these datasets are visualized in the form of bar charts, line graphs, and time series charts. These visualizations transform complex data into easily interpretable formats, enhancing the understanding of transit usage trends and facilitating informed decision-making.

7.14 High-Concurrency Support

To ensure the application can handle a large number of concurrent users and requests, high-concurrency support was implemented and tested using Postman's performance testing tool. The critical endpoints and API routes that are likely to experience high traffic were identified. These endpoints include frequently accessed routes such as get all stations, line from station information retrieval, and ride information retrieval.

1. Summary

Total requests sent	Throughput	Average response time	Error rate
11,723	173.90 requests/second	261 ms	0.06 %

1.1 Response time

Response time trends during the test duration.

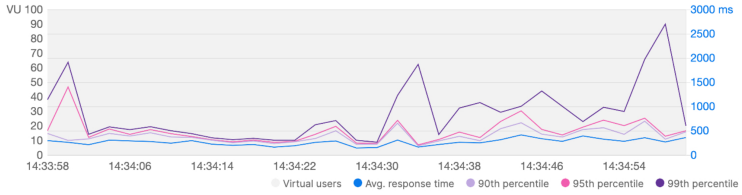

Figure 4: Response time chart

2. Metrics for each request

The requests are shown in the order they were sent by virtual users.

Request	Total requests	Requests/s	Min (ms)	Avg (ms)	90th (ms)	Max (ms)	Error %
GET Get Station lines http://localhost:3000/lines/station/Gangxia?stationName=Gangxia	2,956	43.85	19	291	450	3,020	0.2
GET Get All Stations http://localhost:3000/stations	2,933	43.51	15	254	405	1,530	0.03
GET Get line from station http://localhost:3000/stations	2,922	43.34	12	249	390	1,874	0
GET Get all rides http://localhost:3000/stations	2,912	43.20	8	249	386	1,954	0

Figure 5: Metrics

Figure 6: Performance test of common get requests