The Establishment of Shenzhen Metro Management System

> The Report for the 2nd Project of 24Spring CS307, by Group1 in Lab Thursday 3-4 Zhu

Basic Information

Group Member	Student ID	Lab Session	Contribution Rate
李天宇	12212824	Thursday 3-4 Zhu	50%
李怡萱	12212959	Thursday 3-4 Zhu	50%

Contribution of work

李天宇：Basic API Design Task 5, 6, 7; Price data import; Advanced API Task (2), (3); Ultilization of index; Parts of Webpage design

李怡萱：Basic API Task 1，2，3，4; User Interface design; Advanced API Design Task (4), (5); Backend Server building; Main Webpage design; Ultilization of view; Pressure tests conduction

1 Project Structure

In this project , we have developed a front-end and back-end separated Shenzhen Metro management system, which allows for front-end requests to be sent to the back-end subway information database for operations.

We introduce the **Springboot Gradle back-end framework** and add dependencies of **MyBatis, Thymeleaf, JDBC and JPA**. Our programming language is **JAVA, HTML and SQL**. We perform user operations on the **front-end web pages**, and achieve data transmission from database in PostgreSQL by sending **HTTP requests**.

The basic tree structure of our project is as follows.

```
.
└── CS307-24Spring-Project2
    ├── build.gradle
    ├── settings.gradle
    └── src
        ├── main
        │   ├── java
        │   │   └── cs307
        │   │       └── cs30724springproject2
        │   │           ├── Cs30724SpringProject2Application.java
        │   │           ├── controller
        │   │           ├── entity
        │   │           ├── mapper
        │   │           └── service
        │   └── resources
        │       ├── application.properties
        │       ├── mapper (XML files)
        │       └── templates (HTML files)
        └── test
            └── java
                └── cs307
                    └── cs30724springproject2
                        └── Cs30724SpringProject2ApplicationTests.java
```

The details of the main parts in `src` for our project are as follows:

- Java
  - `Cs30724SpringProject2Application.java`: Run this to start your application.
  - `entity`: The entities of all the tables in the database. Consist of all the parameters and relevant construction and getter setter methods.
  - `mapper (dao)`: The data access layer, which is primarily responsible for interacting with the database.
  - `service`: The Service layer operates the database by calling the interfaces of the Mapper layer.
  - `controller`: Receives requests from front-end, schedules methods at the Service layer, and returns the results to clients.
- Resources
  - `mapper`: XML files to dynamically construct specific SQL statements based on the parameters transmitted by the front-end, with the dependency of MyBatis.

    - `templates`: HTML files for front-end web page construction, which are carried out under the dependency of Thymeleaf.
    - `application.properties`: The setting file for web access ports, database connection and all the dependencies.

2 Basic functions

2.1 Task achievement

- [x] **API Specification**
    - [x] **Task1 Add, modify, delete a station**
    - [x] **Task2 Add, modify, delete a new line**
    - [x] **Task3 Place and remove stations at a specified location on a line**
    - [x] **Task4 Search the name of the station that is the n-th station ahead/behind a specific station on a line**
    - [x] **Task5 Boarding functionality for a passenger or card**
    - [x] **Task6 Exit functionality for a passenger or card**
    - [x] **Task7 View all information about passengers or cards who have boarded but have not yet exited at the current time**
- [x] **Functional Requirements:**
    - [x] **Web Interface Building**
    - [x] **Preparing testing data: import data in `price.xlsx` and all data from project 1.**

2.2 API Specification

1. **Add, modify, delete a station**

- URL: http://localhost:8080/station/

```
@PostMapping("/insert")   // add a station
public ResponseEntity<String> insertStation(@RequestBody station stationT) {...}

@GetMapping(value = "/selectById",produces = "application/json;charset=utf-8")
public String selectById(@RequestParam int id, Model model) {...}
//  search for a station

@PostMapping("/delete")   //  delete a station
public ResponseEntity<String> deleteStation(@RequestParam("id") int id) {...}

@PostMapping("/updateStation")  //  modify a station
public ResponseEntity<String> updateStation(@RequestBody station stationT) {...}
```

2. **Add, modify, delete a new line**

- URL: http://localhost:8080/line/

```
// search for a line
@GetMapping(value = "/selectByNameInfo",produces = "application/json;charset=utf-8")
public String selectByNameInfo(@RequestParam("name") String name, Model model) {...}
@ResponseBody
@GetMapping(value = "/selectByName", produces = "application/json;charset=utf-8")
public line selectByName(@RequestParam String name) {...}

@PostMapping("/insert")  // insert a line
public ResponseEntity<String> insertLine(@RequestBody line lineT) {...}

@PostMapping("/delete")  // delete a line
public ResponseEntity<String> deleteLine(@RequestParam("name") String name) {...}

@PostMapping("/updateLine")  // modify a line
public ResponseEntity<String> updateLine(@RequestBody line lineT) {...}

```

3. **Place one or more stations at a specified location on a line**

- URL: http://localhost:8080/linedetail/

```
@PostMapping("/updateInsertStationNoBehind")
public void updateInsertStationNoBehind(@RequestBody Map<String, Object> parameters) {...}

@PostMapping("/insertMultipleStationsBehind")
public ResponseEntity<String> insertMultipleStationsBehind(@RequestBody Map<String, Object> request) {...}

@PostMapping("/updateInsertStationNoFront")
public void updateInsertStationNoFront(@RequestBody Map<String, Object> parameters) {...}

@PostMapping("/insertMultipleStationsFront")
public ResponseEntity<String> insertMultipleStationsFront(@RequestBody Map<String, Object> request) {...}

@PostMapping("/updateStationFromLine")
public ResponseEntity<String> updateStationFromLine(@RequestBody Map<String, String> request) {...}
```

4. **Remove a station from a line**

- URL: http://localhost:8080/linedetail/

```java
@PostMapping("/deleteStationFromLine")
public ResponseEntity<String> deleteStationFromLine(@RequestBody Map<String, String> request) {...}
```

5. **Search the name of the station that is the n-th station ahead/behind a specific station on a line**

- URL: http://localhost:8080/linedetail/

```java
@GetMapping("/getNBefore")
public List<String> getStationsBefore(@RequestParam String lineName,
@RequestParam String stationName, @RequestParam int number) {...}
@GetMapping("/getNAfter")
public List<String> getStationsAfter(@RequestParam String lineName,
@RequestParam String stationName, @RequestParam int number) {...}
```

6. **Boarding functionality for a passenger or card** (Take passenger as example)

- URL: http://localhost:8080/passengerOnboard/

```java
@PostMapping("/insert")  // insert the passenger ID, start station, and start time into the table
public ResponseEntity<String> addPassengerInfo(@RequestBody passengerOnboard passengerOnboard){...}
```

7. **Exit functionality for a passenger or card** (Take passenger as example)

- URL: http://localhost:8080/passengerOnboard/

```java
@ResponseBody  // query about the information of a passenger onboard (start time, carriage type)
@GetMapping(value = "/selectByIdStation")
public passengerOnboard selectByIdStation(@RequestParam String passengerId, @RequestParam String startStation){...}
@PostMapping("/deleteByIdStation")  // remove the passenger from the table storing onboard passengers
public ResponseEntity<String> deleteByIdStation(@RequestParam String passengerId, @RequestParam String startStation) {...}
```

- URL: http://localhost:8080/passengerRide/

```java
@PostMapping("/insert")  // add the ride record to the corresponding table
public ResponseEntity<String> insert(@RequestBody passengerRide passengerRide) {...}
```

- URL: http://localhost:8080/price

```java
@ResponseBody  // search for the corresponding price according to the start and end stations
@GetMapping(value = "/selectByStations", produces = "application/json;charset=utf-8")
public price selectByStations(@RequestParam String start, @RequestParam String end, Model model){...}
```

8. **View all information about passengers or cards who have boarded but have not yet exited at the current time**

- URL: http://localhost:8080/passengerOnboard/

```java
@GetMapping(value = "/selectAllPassengerOnboard", produces = "application/json;charset=utf-8")
public List<passengerOnboard> getAllPassengerOnboard(){return passengerService.getAllPassengerOnboard();}
```

2.3 Functional Requirements

- We use **Java** as the general programming language to interact with the database. Besides this, we use **HTML** and **CSS** to build the web pages, and **PostgreSQL** to write `mapper.xml` files.
- We provide a **webpage-based interface** for users to interact with our program.
- We prepare all the data in project 1 and import `price.xlsx` as the testing data and store them in the database for project 2.

3 Advanced functions

3.1 Task achievement

- [x] **Task2 Further enhance the usability of the APIs**
  - [x] **(2) Business carriage in the subway**
  - [x] **(3) Add and appropriately utilize the condition of stations**
  - [x] **(4) Establish a system to integrate buses and subways**
  - [x] **(5) Implement a multi-parameter search for ride records**
- [x] **Task3 Implement a real back-end server, with the technology of package management, using HTTP/RESTful Web, connection pools and implementing backend frameworks with ORM mapping**
- [x] **Task4 Elegant and useful page display design**
- [x] **Task5 Appropriately utilize index and view**
- [x] **Task7 Support high-concurrency with proper pressure tests**

3.2 Details Explanation

3.2.1 Advanced API Design

- **(2) Business carriage**

If a passenger/card intends to take the business carriage, then press the `商务车厢进站` button, then its carriage type will be recorded as `商务` in `passenger_onboard`/`card_onboard`.

Meanwhile, we found data on prices for business carriage rides in 深圳市发展和改革委员会关于公布深圳市轨道交通2023年新线开通线网票价表的通知-服务价格-深圳市发展和改革委员会网站 (sz.gov.cn), created the table `business_price` (similar with table `price`), and inserted the found data into the newly created table with `ReadVIP.java`.

```sql
create table business_price (
    id            serial       primary key,
    start_station varchar(200) not null,
    end_station   varchar(200) not null,
    price         integer
);
```

When the passenger/card taking the business carriage is exiting, the price of the ride will be selected in `business_price`, which is higher than that in `price`:

URL: http://localhost:8080/businessPrice

```java
@ResponseBody
@GetMapping(value = "/selectByStations", produces = "application/json;charset=utf-8")
public businessPrice selectByStations(@RequestParam String start, @RequestParam String end, Model model){...}
```

- **(3) Conditions of the stations**

We altered the table `station` by adding a column `condition` and setting the values of already inserted columns as "运营中":

```sql
alter table station add column condition varchar(20);
update station set condition = '运营中';
```

When entering/exiting the station, information about it will be selected from the table according to the input station name. If the station's condition is not “运营中”, alert message "始发站当前不在运营中！"/"终到站当前不在运营中！" will pop up and the passenger/card cannot enter/exit the station successfully.

URL: http://localhost:8080/station/

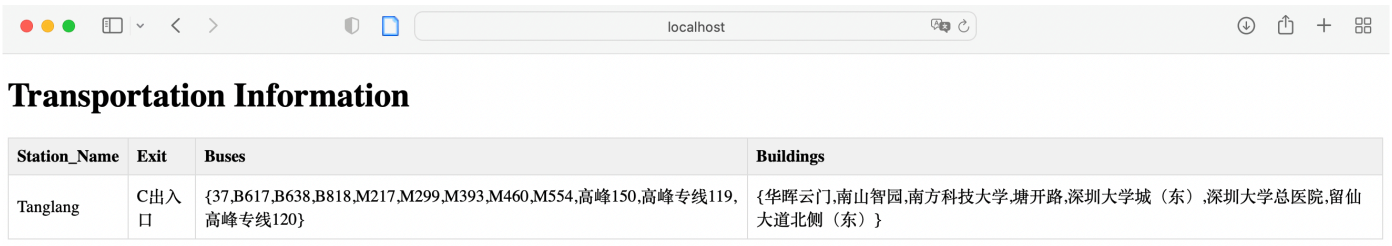
```java
@ResponseBody
@GetMapping(value = "/selectByChineseName", produces =
"application/json;charset=utf-8")
public station selectByChineseName(@RequestParam String chineseName) {...}
```

- **(4) View information about surrounding bus lines and buildings (utilizing views)**

When assembling subway stations, exits, and buses, we use `view` to create a `TransportationView` to query bus information, taking into account that the data may be changed on the fly. Then, we can query the buses information when clicking the button on webpage.

URL: http://localhost:8080/

```java
@GetMapping(value = "/transportationView", produces = "application/json;charset=utf-8")
public String findAllTransInfo(@RequestParam("stationName") String stationName,
@RequestParam("exit") String exit, Model model) {...}
```



Transportation Information

Station_Name	Exit	Buses	Buildings
Tanglang	C出入口	{37,B617,B638,B818,M217,M299,M393,M460,M554,高峰150,高峰专线119,高峰专线120}	{华晖云门,南山智园,南方科技大学,塘开路,深圳大学城（东）,深圳大学总医院,留仙大道北侧（东）}

- **(5) Multi-parameter search for ride records**

We use the dynamic construction SQL statement provided by MyBatis to query multi-parameter ride records.

```xml
    <!-- 多参数查询card_ride:设参start/endStation, cardCode, price, timeInterval -->
    <select id="selectCardRideByParameters" resultType="cardRide">
        SELECT * FROM card_ride
        <where>
            <if test="cardCode != null">
                AND card_code = #{cardCode}
            </if>
            <if test="startStation != null">
                AND start_station = #{startStation}
            </if>
            <if test="endStation != null">
                AND end_station = #{endStation}
            </if>
            <if test="price != null">
                AND price = #{price}
            </if>
            <if test="startTime != null">
                AND start_time &gt;= #{startTime}
            </if>
            <if test="endTime != null">
                AND end_time &lt;= #{endTime}
            </if>
        </where>
    </select>
```

3.2.2 Back-end Server Implementation

- **Using backend frameworks and ORM mapping**

We used **Springboot Gradle** integrates with **MyBatis** as our backend framework.

Our project can use Spring's dependency injection mechanism to manage instances of MyBatis' SQLSessionFactory and Mapper interfaces, as well as Spring's transaction management to manage database transactions.

With MyBatis, we can define the mapping between Java entities and database tables by writing mapping files (XML), and use SQL query statements to perform database operations and map the results back to Java objects. The following is an example of `cardMapper.xml`.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<!-- 定义了一个名为 “cardMapper” 的映射器，返回一个entity中card类型的java对象 -->
<mapper namespace="cs307.cs30724springproject2.mapper.cardMapper">
    <select id="selectAll" resultType="cs307.cs30724springproject2.entity.card">
        SELECT * FROM card
    </select>

</mapper>
```

- **Package management**

We divide the packages into two categories: `Java` and `Resources`, of which the Java package stores the `Controller`, `Service`, `Entity`, and `Mapper` packages, and Resources stores the `Mapper` package based on the MyBatis XML file, and the `Templates` HTML file package. So, our project organizes the code into different packages according to the functional modules and business logic set by the SpringBoot project, aiming for easy management and maintenance.

- **Using HTTP/RESTful Web**

On the front-end Webpage, we use HTTP requests and url link redirection to achieve front-end and back-end communication, in which HTTP requests are divided into different methods such as `Get` and `Post` according to the backend API type.

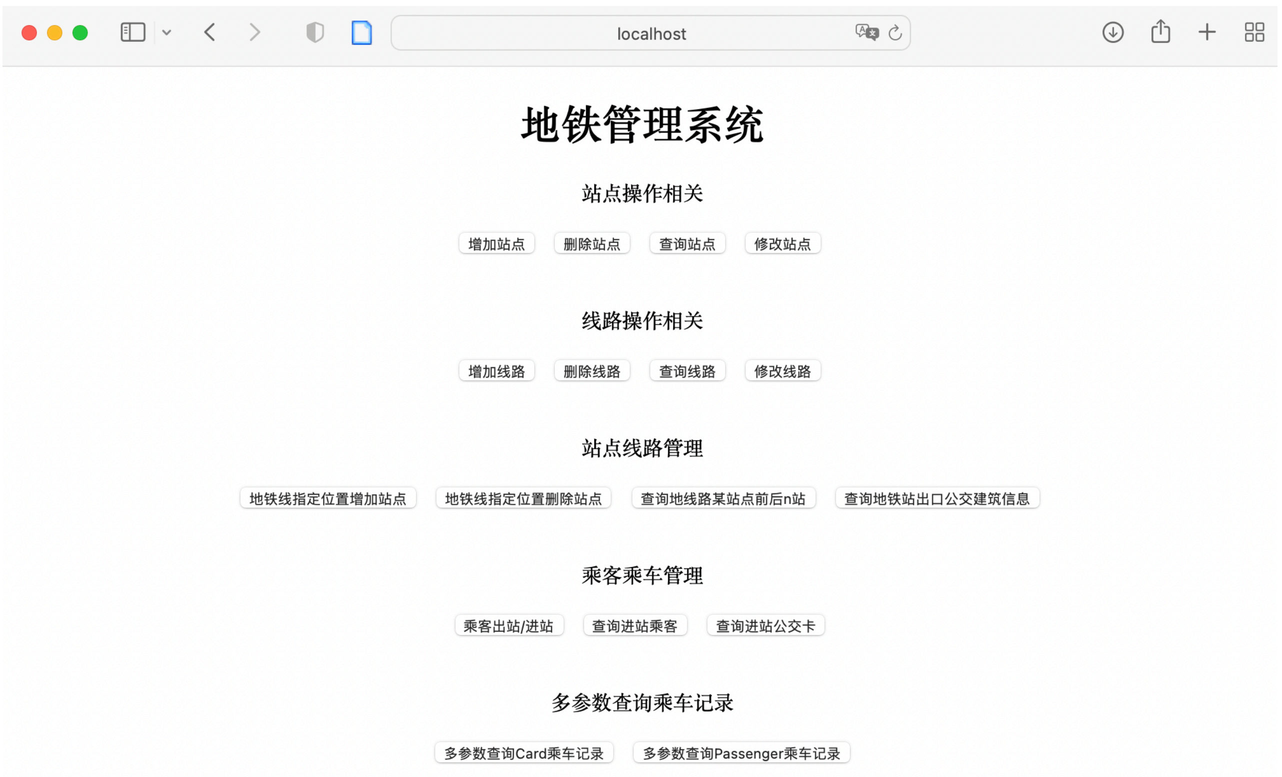
In the `Controller` class, we use `@RestController` and `@Controller` annotations to render the returned data as an entity type (e.g. json) or an html web page, and use annotations such as `@RequestMapping` and `@GetMapping` to connect URLs.

- **Using connection pools**

Since we add `spring-boot-starter-jdbc` and `spring-boot-starter-data-jpa` as dependencies , the database is automatically configured with the HikariCP connection pool.

3.2.3 Webpages Design

Based on the dependencies of `spring-boot-starter-web` and `spring-boot-starter-thymeleaf` , we design a useful webpage using **HTML** and **CSS**. Also, the pop-up dialogs and page separation are very clear, while the pages to show our data is elegant. The following graphs are our webpage examples.



地铁管理系统

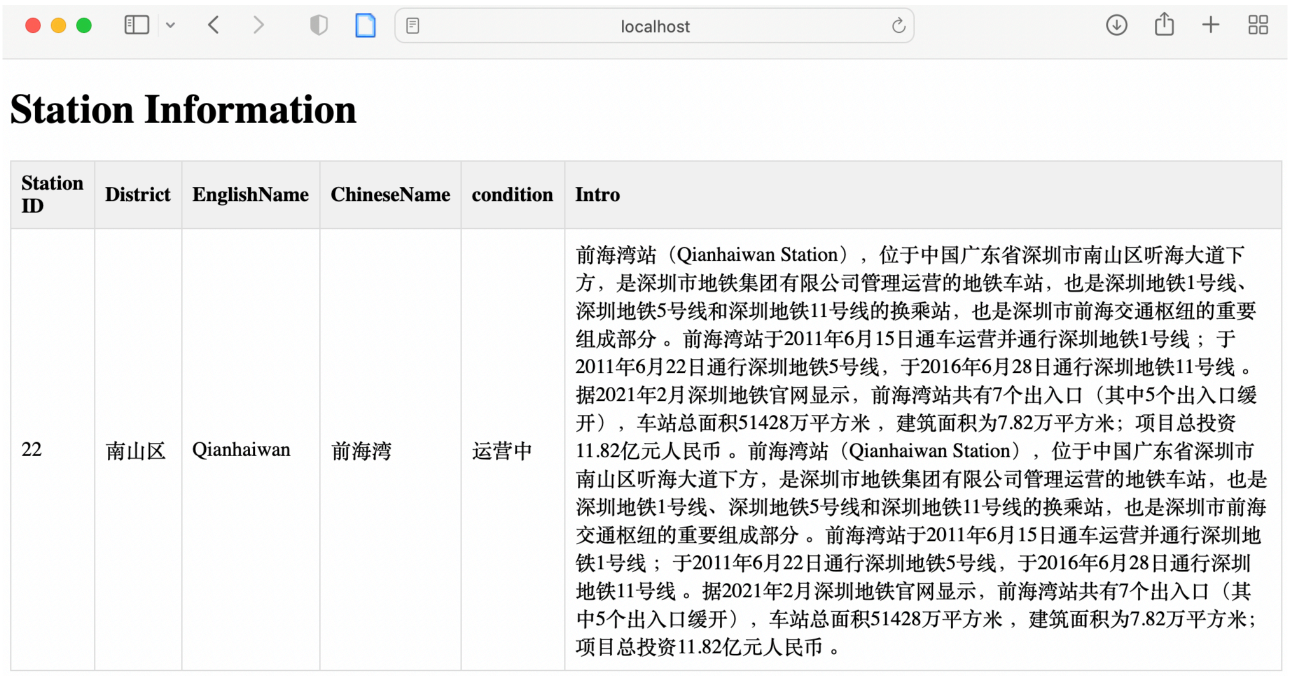
站点操作相关

线路操作相关

站点线路管理

乘客乘车管理

多参数查询乘车记录



Station Information

Station ID	District	EnglishName	ChineseName	condition	Intro
22	南山区	Qianhaiwan	前海湾	运营中	前海湾站（Qianhaiwan Station），位于中国广东省深圳市南山区听海大道下方，是深圳市地铁集团有限公司管理运营的地铁车站，也是深圳地铁1号线、深圳地铁5号线和深圳地铁11号线的换乘站，也是深圳市前海交通枢纽的重要组成部分 。前海湾站于2011年6月15日通车运营并通行深圳地铁1号线；于2011年6月22日通行深圳地铁5号线，于2016年6月28日通行深圳地铁11号线 。据2021年2月深圳地铁官网显示，前海湾站共有7个出入口（其中5个出入口缓开），车站总面积51428万平方米，建筑面积为7.82万平方米；项目总投资11.82亿元人民币 。前海湾站（Qianhaiwan Station），位于中国广东省深圳市南山区听海大道下方，是深圳市地铁集团有限公司管理运营的地铁车站，也是深圳地铁1号线、深圳地铁5号线和深圳地铁11号线的换乘站，也是深圳市前海交通枢纽的重要组成部分 。前海湾站于2011年6月15日通车运营并通行深圳地铁1号线；于2011年6月22日通行深圳地铁5号线，于2016年6月28日通行深圳地铁11号线 。据2021年2月深圳地铁官网显示，前海湾站共有7个出入口（其中5个出入口缓开），车站总面积51428万平方米，建筑面积为7.82万平方米；项目总投资11.82亿元人民币 。

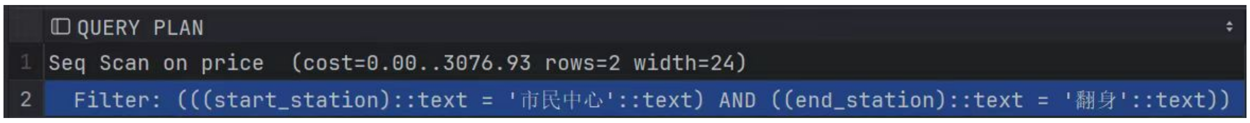
3.2.4 Utilization of Index and View

- **Index**

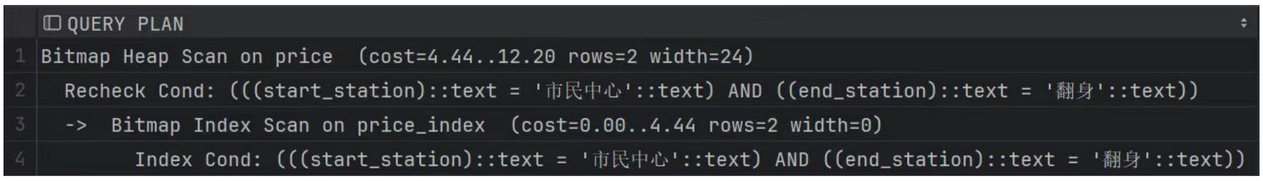
Considering the amounts of data are huge (>100000) in tables `business_price`, `price`, `passenger_ride`, and `card_ride`, to increase the efficiency of queries, we add indexes to these tables:

```
create index price_index on price(start_station, end_station);
create index business_price_index on business_price(start_station, end_station);
create index card_ride_index on card_ride(card_code, start_station, end_station, price, start_time, end_time);
create index passenger_ride_index on passenger_ride(passenger_id, start_station, end_station, price, start_time, end_time);
```

The following are plans of the query `select * from price where start_station = '市民中心' and end_station = '翻身';` before and after creating the index:



```
QUERY PLAN
Seq Scan on price  (cost=0.00..3076.93 rows=2 width=24)
  Filter: (((start_station)::text = '市民中心'::text) AND ((end_station)::text = '翻身'::text))
```



```
QUERY PLAN
Bitmap Heap Scan on price  (cost=4.44..12.20 rows=2 width=24)
  Recheck Cond: (((start_station)::text = '市民中心'::text) AND ((end_station)::text = '翻身'::text))
  ->  Bitmap Index Scan on price_index  (cost=0.00..4.44 rows=2 width=0)
        Index Cond: (((start_station)::text = '市民中心'::text) AND ((end_station)::text = '翻身'::text))
```

- **View**

In order to see the modified data instantly, and to avoid plenty of selected data taking up the system memory, we use view to optimize our select queries.

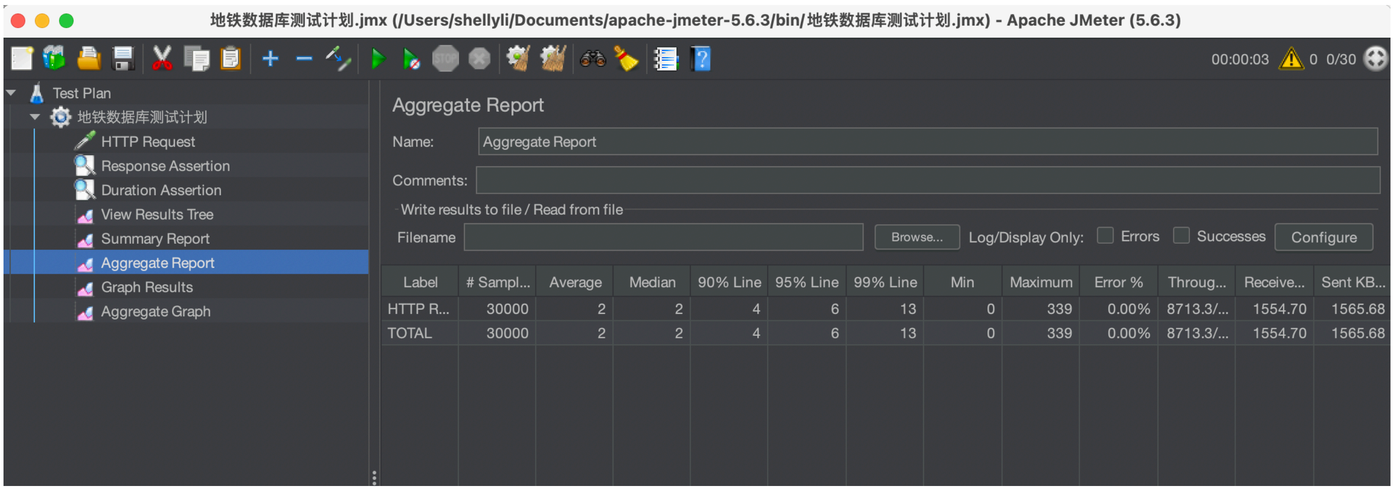
```
create view transportation_view as
select sub1.English_name station_name, sub1.number exit, buses, buildings
from (select distinct sub.English_name, number, array_agg(bln) over (partition by (sub.English_name, number)) buses
      from (select distinct English_name, number, bus_line.name bln
            from exit
                     left join station s on exit.station_id = s.id
                     left join bus_stop bs on exit.id = bs.exit_id
                     left join bus_line on bs.id = bus_line.bus_stop_id
                     left join buildings b on exit.id = b.exit_id) sub) sub1
         join (select distinct sub.English_name,
                               number,
                               array_agg(bn) over (partition by (sub.English_name, number)) buildings
               from (select distinct English_name, number, b.name bn
                     from exit
                              left join station s on exit.station_id = s.id
                              left join bus_stop bs on exit.id = bs.exit_id
                              left join bus_line on bs.id = bus_line.bus_stop_id
                              left join buildings b on exit.id = b.exit_id) sub)
sub2
               on sub1.English_name = sub2.English_name and sub1.number =
sub2.number;
```

3.2.5 Pressure tests conduction

We use **Apache Jmeter** to conduct pressure tests on our API interface.

We set the Thread Group parameters as: `User=30, Loop Count=3000`, so the final amount of samples is 30000. We add a `HTTP request` with the method `Get` and the relevant Respond Assertion, and set the assertion duration to be 3000. Moreover, we add some listeners, including View Result Trees, Aggregate Report and Summary Report to analyse the result.

Finally, the result in the Aggregate Report is in the picture below:



Aggregate Report

Name: Aggregate Report

Comments:

Filename

Label	# Sampl...	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throug...	Receive...	Sent KB...
HTTP R...	30000	2	2	4	6	13	0	339	0.00%	8713.3/...	1554.70	1565.68
TOTAL	30000	2	2	4	6	13	0	339	0.00%	8713.3/...	1554.70	1565.68

Analysis

- **Stable response time**

We can notice that `Average=2ms, 90%Line=4ms`, which means the average response time of a single request is 2ms, while 90% users can get the response in 4ms. A low average response time and a small response time variance are generally considered good indicators, and our result indicates that **the response time of the application is within acceptable limits and be stable during testing**.

- **High throughput**

Under the setting of `# Sample=30000`, We can notice that `Throughtput=8713.3 req/s`, which means our application can be able to **handle high concurrency requests** as well as **handle a large number of requests** per unit of time.

- **Low error rate**

We can notice that `Error %=0`, which shows that our application can **handle requests correctly under pressure and return a low error rate.**

In summary, our project can support high-concurrency under proper pressure test.