김가현 포트폴리오

목차

신입 Node.js 백엔드 개발자 김가현입니다.

문제를 명확히 정의하고, 시간 안에 최선의 방법으로 해결하는 것을 가장 중요하게 생각합니다.

Main Skill

TypeScript Node.js Mocha AWS DynamoDB

Contact

✉️ kimgostring@naver.com

🔗 Github

✏️ Study (Notion, Obsidian)

‘캔랩코리아’ 인턴

23.09.01 ~ 23.12.31 | 4개월

백엔드팀 | 인턴

TypeScript Node.js Mocha

AWS DynamoDB

AWS Lambda

커뮤니티 기반 커머스 No-Code SaaS 개발

1. **[메인] 게시글 태그를 상품 및 유저에도 달 수 있도록 기능 확장**
    - 하나의 태그 그룹에만 속할 수 있던 태그를 여러 태그 그룹에 속할 수 있도록 구조 변경, 데이터 마이그레이션, API 버저닝
2. **기존 앱 링크를 받아 짧은 길이의 URL을 만들 수 있는 API 추가**
    - Lambda@Edge를 통해 짧은 길이의 URL로 접속 시 원본 앱 링크로 리다이렉트 되도록 수정
3. **주문서 및 주문서 목록 엑셀 다운로드 API 개발**
    - AWS S3에 업로드한 뒤 클라이언트에서 다른 API 호출을 통해 pre-signed URL을 응답받을 수 있도록 개발
4. **Open API 작업 및 Swagger 문서화**

한 태그가 여러 태그 그룹에 포함될 수 있도록 구조 변경

목적

게시글 태그를 상품 및 유저에도 달 수 있도록 기능을 확장하면서, 커뮤니티 관리자 입장에서
한 태그를 여러 태그 그룹에 담아 관리하고 태그를 통해 유저에게 관련 상품을 큐레이션 하고 싶은 니즈가 생김

Before

태그 그룹 = 계절, 태그 = 봄, 여름, …

1. 한 태그는 무조건 한 개의 태그 그룹에 속해야 함 (태그 : 태그 그룹 = 多 : 1)
2. 태그 이름은 같은 태그 그룹 내에서 중복 불가

After

1. 한 태그가 0..*개의 태그 그룹에 속할 수 있음 (태그 : 태그 그룹 = 多 : 多)
2. 태그 이름은 같은 커뮤니티 내에서 중복 불가

한 태그가 여러 태그 그룹에 포함될 수 있도록 구조 변경

결정 사항 TagSet 클래스 하나로 정의되어 있는 태그와 태그 그룹을 서로 다른 클래스로 분리

이유

1. **기존 구조는 단일 책임 원칙에 위배됨**
   TagSet이라는 클래스는 항상 태그 그룹 또는 태그 둘 중 하나의 역할만을 하게 되고, 맡은 역할에 따라 사용되는 프로퍼티도 달랐기 때문에 분리가 필요했음 (기존 백엔드는 태그가 태그 그룹의 역할도 겸할 수 있도록 이런 구조로 정했었지만, 클라이언트에서 이를 고려하지 않았고 PM 팀에서도 앞으로 이를 고려할 계획이 없었음)
2. **API URI를 리소스 별로 정의하기 위해 필요**
   기존 태그 리스트 GET API(GET /tag_sets)는 태그들을 넣은 전체 태그 그룹 목록을 응답으로 줌, 이 상황에서 태그 리스트만 주는 API를 새롭게 정의해야 했는데, 태그와 태그 그룹이 다 TagSet에 섞여 있으므로 URI만으로 의미를 명확하게 표현하기 어려웠음

한 태그가 여러 태그 그룹에 포함될 수 있도록 구조 변경

상황

태그 그룹과 태그의 多:多 관계를 어떤 방식으로 설계할지?

가능한 방법

✅ **다중 테이블 설계 (태그 그룹과 태그의 테이블을 분리하고 태그 그룹 테이블에 태그의 id 배열 저장)**

태그가 자신이 속한 태그 그룹을 알아야 하는 access pattern이 없었음, 추후에 이런 니즈가 생길 가능성도 적다고 생각함

⬜ 단일 테이블 설계

현재는 관리자만 태그 생성이 가능하지만 이후 사용자가 직접 태그를 생성할 수 있도록 개선할 예정, 그렇게 되면 태그 개수가 기하급수적으로 늘어나게 될 것이므로 그 전에 태그 그룹 데이터를 분리하는 게 성능에 더 좋을 것이라고 생각함

⬜ ~~매핑 테이블 생성~~

나쁜 성능 (DynamoDB에는 join 연산이 없음, 매핑 테이블을 두게 되면 한 태그 그룹에 속한 모든 태그를 가져오기 위해 무조건 3번의 쿼리 필요)

한 태그가 여러 태그 그룹에 포함될 수 있도록 구조 변경

상황 **태그 그룹과 태그의 多:多 관계를 어떤 방식으로 설계할지?**

선택한 방법 ✅ 다중 테이블 설계

아쉬운 점 **단일 테이블 설계**가 더 좋은 선택이었을 것 같음

1. 태그 그룹의 access pattern을 보면 태그 그룹을 읽을 때 항상 포함된 태그를 함께 가져와야 하는데, 다중 테이블 설계를 쓰면 쿼리가 2번 필요하지만 단일 테이블 설계였다면 1번만으로 가능했을 것
2. 사용자가 직접 태그를 생성할 수 있도록 개선할 때 태그 수정/삭제 API를 deprecated 시킬 예정이었음, 즉 태그 그룹 개수만큼 태그 데이터가 중복 저장되더라도 수정할 때의 부담이 없었을 것

한 태그가 여러 태그 그룹에 포함될 수 있도록 구조 변경

결정 사항

태그 테이블에 GSI 생성

- Partition Key = *커뮤니티 ID*, Sort Key = *태그 이름*

이유

1. 태그 이름 중복 validation의 범위가 같은 태그 그룹 내 → 같은 커뮤니티 내로 바뀌게 되면서, 특정 *커뮤니티 ID*에 생성된 *태그 이름*의 존재를 확인하는 access pattern이 추가됨
2. 추후 사용자가 직접 태그를 생성할 수 있게 되면 이름을 통해 존재하는 태그 데이터를 가져오거나 이름 중복을 체크해야 하는 빈도가 더 늘어날 것

‘막차 도우미’

23.04.03 ~ 23.06.09 | 2개월

캡스톤디자인 (2)

3인 팀 (BE 2, FE 1) | BE

🔗 Github (BE, FE)

🔗 Notion Page

JavaScript Node.js Express

MySQL

대중교통 끊긴 구간을 택시로 이동하는 길찾기 API 개발

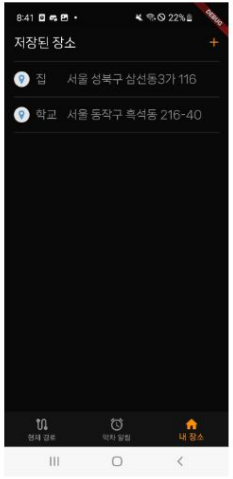
1. **대중교통 정류장 좌표 이상값/결측값 처리, 공공 데이터로 제공되지 않는 일부 지하철 노선 시간표 크롤링**
    - Pandas Python3 BeautifulSoup Selenium
2. **설문 결과를 토대로 문제 구체화, 대중교통 길찾기 알고리즘인 RAPTOR을 변형하여 ‘대중교통 끊긴 구간을 택시로 이동하는’ 알고리즘 설계 및 API 개발**
    - JavaScript Node.js Express MySQL
    - 환승 가능한 근처 정류장을 탐색할 때, 가까운 정류장끼리는 위경도 값이 유사하다는 특징에서 해시를 떠올리고 GeoHash 활용

막차 도우미

현재 위치를 기반으로 **목적지에 대한 막차 경로 및 출발 시간**을 자동으로 알려주고, **막차가 끊기더라도 목적지에 저렴하게 도달**할 수 있도록 대중교통 끊긴 구간을 택시로 이동하는 길찾기를 제공하는 애플리케이션

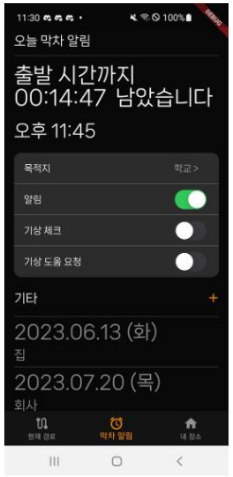
장소 등록

- + (장소 추가) 버튼
- 저장된 장소 목록



막차 알림 등록

- 간단 당일 알림 설정
- + (알림 추가) 버튼
- 생성된 알림 목록



막차 알림

- 출발까지 남은 시간
- 다음 세부 경로 안내
- 안내 종료, 안내 시작 버튼

세부 경로 안내 화면 - 지하철

- 남은 전체 경로
- 탑승 노선, 도착 시간 안내
- 지하철 경유역 정보
- 안내 종료, 다음 경로 버튼

세부 경로 안내 화면 - 도보

- 남은 전체 경로
- 다음 탑승 노선, 도착 시간 안내
- 도보 상세 경로
- 안내 종료, 다음 경로 버튼

세부 경로 안내 화면 - 택시

- 남은 전체 경로
- 택시 경로, 택시비 안내
- 안내 종료, 다음 경로 버튼

개발 동기

1. 일반적인 지도 애플리케이션에서는 **막차 시간을 찾기 불편**함 (출발 시간을 조금씩 조정해가며, 경로가 가장 마지막으로 나오는 시간을 확인해야 함)
2. **택시 요금의 인상**으로 인해, 막차가 끊긴 경우 택시를 타고 귀가하기 부담스러움

초기 문제 정의

서울에서 활동하며, 미리 정해 둔 서울 내 목적지로 대중교통을 막차를 타고 돌아가고 싶은 사람들에게,

1. 현 위치에서 목적지까지 가는 막차를 탑승하기 위한 예상 출발 시간 및 경로 제공
2. 막차를 놓쳤을 때, 대안으로써 대중교통 끊긴 구간을 택시로 이동하는 길찾기 경로 제공

설문 조사 진행

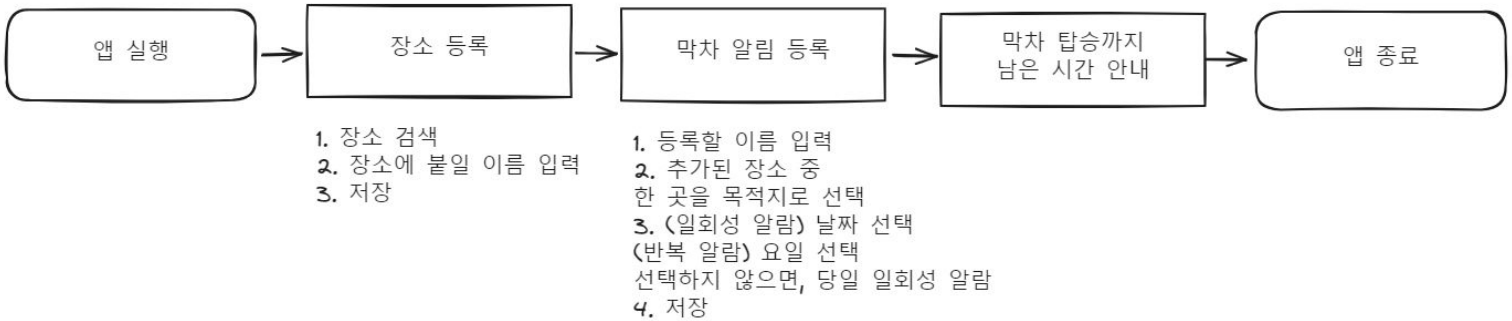
- **기간** : 23.04.01 15:00 ~ 23.04.07 23:59
- **참여 인원** : 54명
- 질문지 사본, 전체 개별 응답 (개인정보 제외)
- **목적**
    1. 유저 시나리오 검증 및 구체화 (유저 시나리오가 모호하다는 피드백 반영)
    2. 대중교통 끊긴 구간을 택시로 이동하는 알고리즘 결정 사항의 근거 마련
- 설문 결과, 문제 검증 및 구체화 과정

최종 문제 정의

서울에서 활동하며, 미리 정해 둔 서울 내 목적지로 대중교통을 막차를 타고 돌아가고 싶은 사람들에게,

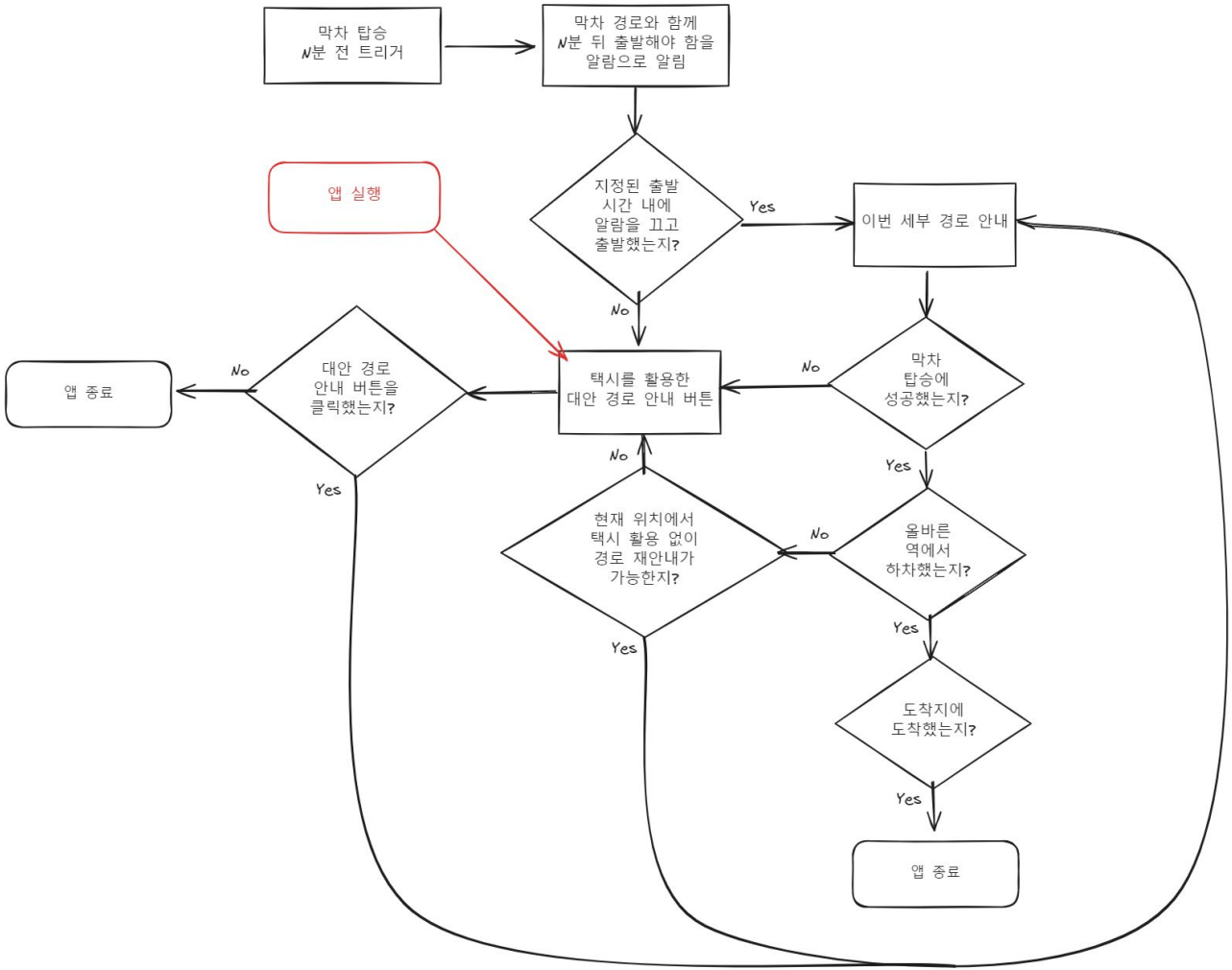
1. 현 위치에서 목적지까지 가는 막차를 탑승하기 위해, **가장 늦게 출발해도 되는 경로**를 기준으로 예상 출발 시간 및 경로 제공
2. 막차를 놓치거나, **막차 이후 일정이 끝나는 경우**, 대안으로 대중교통 끊긴 구간을 택시로 이동하는 경로 중 **'30분 이하는 도보로 이동 → 택시는 경로 후반에, 불가능하면 초반에 활용 → 택시비가 가장 적게 드는 (2만원 이하) → 가장 빠른'** 경로를 제공

최종 유저 시나리오 - 막차 알림 등록

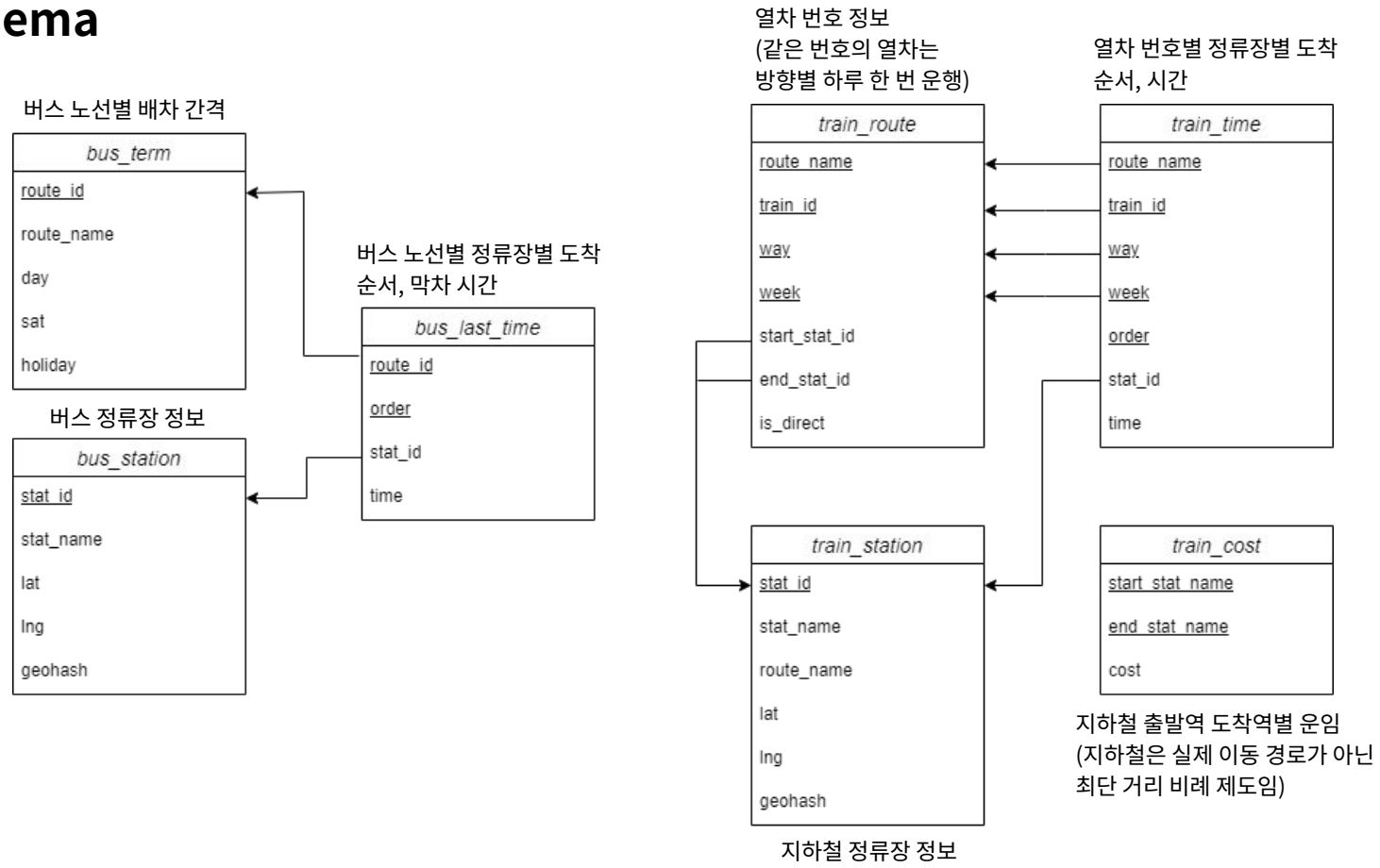


최종 유저 시나리오

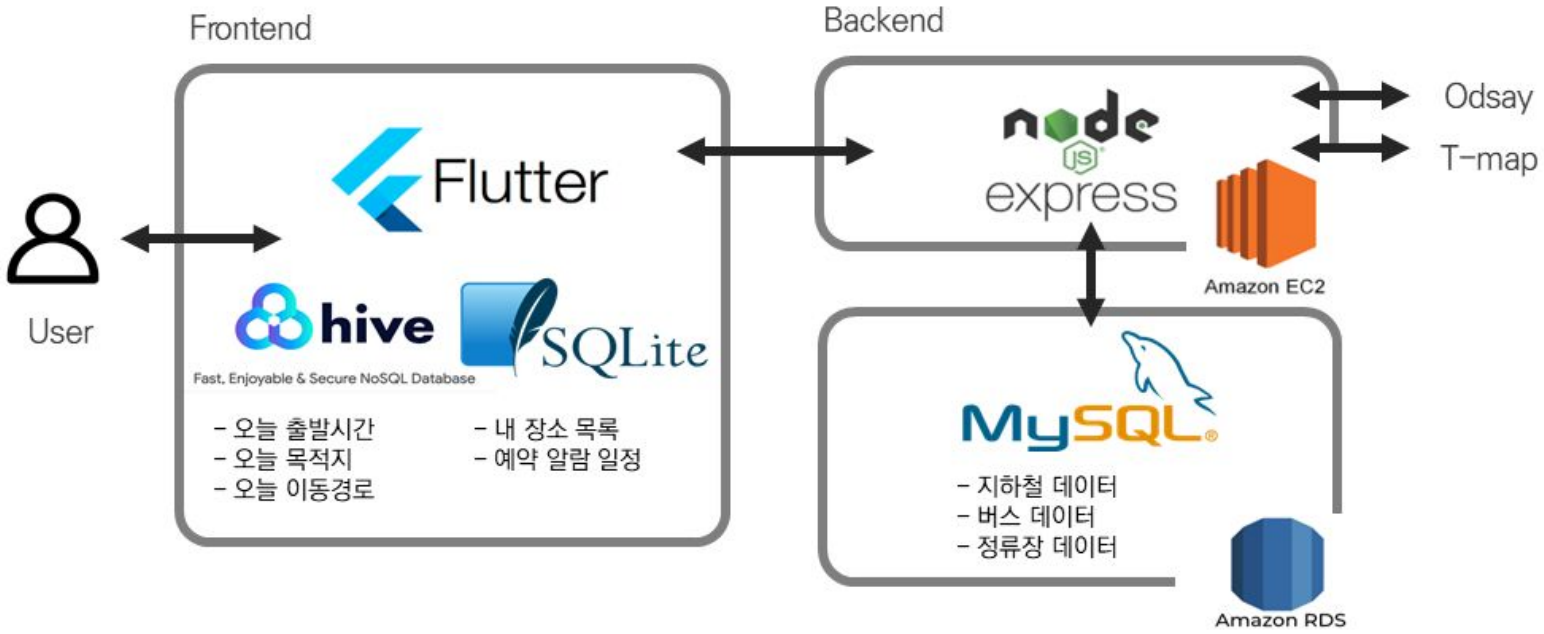
- 막차 알림 및 길안내



DB Schema


버스 노선별 배차 간격
bus_term
route_id
route_name
day
sat
holiday
버스 노선별 정류장별 도착 순서, 막차 시간
bus_last_time
route_id
order
stat_id
time
버스 정류장 정보
bus_station
stat_id
stat_name
lat
lng
geohash
열차 번호 정보 (같은 번호의 열차는 방향별 하루 한 번 운행)
train_route
route_name
train_id
way
week
start_stat_id
end_stat_id
is_direct
열차 번호별 정류장별 도착 순서, 시간
train_time
route_name
train_id
way
week
order
stat_id
time
train_station
stat_id
stat_name
route_name
lat
lng
geohash
지하철 정류장 정보
train_cost
start_stat_name
end_stat_name
cost
지하철 출발역 도착역별 운임 (지하철은 실제 이동 경로가 아닌 최단 거리 비례 제도임)

Software Architecture



GET /taxiRoute/findTaxiPath

- Query parameter

Name	Type	Required	Default	Explanation
startX	float	O		출발지의 X좌표 (경도)
startY	float	O		출발지의 Y좌표 (위도)
endX	float	O		도착지의 X좌표 (경도)
endY	float	O		도착지의 Y좌표 (위도)
time	string	X	현재 시간	출발지에서부터 출발하는 시간 (YYYY-MM-DDThh:mm:ss)
walkSpeed	int	X	50	도보 이동 속도 (m/분)
taxiSpeed	int	X	400	택시 이동 속도 (m/분)
maxTransfer	int	X	4	최대 환승 가능 횟수
maxCost	int	X	30000	최대 비용
maxTotalWalkTime	int	X	40	최대 총 도보 이동 시간
maxWalkTimePerStep	int	X	20	한 번의 도보 이동에 대한 최대 도보 이동 시간

GET /taxiRoute/findTaxiPath

- Response

```
{
  pathExistance: 경로 존재 여부,
  departureTime: 출발 시간,
  arrivalTime: 도착 시간,
  pathInfo: {
    info: {
      departureTime: 출발 시간,
      arrivalTime: 도착 시간,
      transferCount: 환승 횟수,
      firstStartStation: 첫 출발역,
      lastEndStation: 최종 도착역,
      totalTime: 총 소요시간,
      totalWalkTime: 총 도보 이동 시간,
      totalTaxiTime: 총 택시 이동 시간,
      payment: 총 요금,
      taxiPayment: 택시 요금,
      transportPayment: 대중교통 총 요금,
    },
    subPath: [
      {
        trafficType: 이동 수단 종류 (1-지하철, 2-버스, 3-도보, 4-환승 도보, 5-택시),
        sectionTime: 이동 소요 시간,
        taxiPayment: 택시 비용,
        departureTime: 도보/택시 출발 시간,
        arrivalTime: 도보/택시 도착 시간,
        stationCount: 버스/지하철 이동하여 정차하는 정거장 수,
        lane: [
          {
            busNo: 버스명,
            type: 버스 타입 코드,
            busLocalBlID: 버스 노선 코드,
            name: 지하철 노선명,
            subwayCode: 지하철 노선 코드,
            departureTime: 버스/지하철 출발 시간,
            arrivalTime: 버스/지하철 도착시간
          }
        ],
        startName: 승차 정류장명,
        startX,
        startY,
        startLocalStationID: 승차 정류장 버스역 코드,
        startStationID: 승차 정류장 지하철 역코드,
        endName: 하차 정류장명,
        endX,
        endY,
        endLocalStationID: 하차 정류장 버스역 코드,
        endStationID: 하차 정류장 지하철 역코드,
        way: 지하철 방면 정보,
        wayCode: 지하철 방면 정보 코드 (1-상행, 2-하행),
        passStopList: {
          stations: [
            {
              index: 순서,
              stationName: 정류장 이름,
              arrivalTime: 도착 시간,
              x,
              y,
              localStationID: 버스 정류장 코드,
              stationID: 지하철 정류장 코드
            }
          ]
        },
        steps: [
          {
            type: 종류 문자열,
            geometry: [
              type: 종류 문자열,
              coordinates: [lat, lng]
            ]
          }
        ]
      }
    ]
  }
}
```

택시를 포함한 막차 경로 탐색 알고리즘

1. 출발 좌표에서, 도보로 이동 가능한 모든 역들을 계산하고 출발 역으로 설정
2. 대중교통 길찾기 알고리즘 RAPTOR 수행
3. 도착 좌표에서 도보로 이동 가능한 모든 역들을 계산하고 도착 역으로 설정
    a. 모든 도착 역 중 RAPTOR 알고리즘을 통해 도달한 역이 있는지 확인
    b. 도달한 역이 없다면, 도착 좌표에서 택시로 이동 가능한 모든 역을 계산하고 도착 역으로 설정,
    범위가 넓어진 도착 역 중에서 RAPTOR로 도달한 역이 있는지 확인
    c. 도달한 역이 없다면,
        i. 출발 좌표에서 택시로 이동 가능한 모든 역들을 계산하고 출발 가능 역으로 설정
        ii. 모든 출발 역에 대해, RAPTOR 알고리즘 수행
        iii. 도착 좌표에서 도보로 이동 가능한 도착 가능 역 중, RAPTOR 알고리즘을 통해 도달한 역이 있는지 확인
4. 도달한 역이 없다면, 경로가 없음을 알리고 알고리즘 종료
5. 도달한 역이 하나라도 있다면, 해당하는 모든 역에 대해 세부 정보 및 경로 계산
    a. 도착 역부터 시작해서, 첫 역까지 역순으로 거슬러 올라가며 정보 추가
        i. 탑승 노선 번호, 구간 이동 시간, 추산 비용, 세부 경로에 대한 정류장 이름 및 좌표
    b. 출발 좌표 → 첫 역, 마지막 역 → 도착 좌표 정보 추가
6. 계산된 모든 경로에 대해, 택시비 적은 순 → 도착 시간 빠른 순 → 도보 이동 시간 짧은 순 → 환승 횟수 적은 순으로
정렬한 뒤 가장 맨 앞의 경로 선택
7. 선택된 경로 및 세부 정보 리턴

알고리즘 속도 개선

k = 최대 환승 횟수 = 4
p = 정류장 개수 = 10,025

원인

두 정류장 간 직선 거리를 계산하여 도보(택시) 이동 시간을 추산하는 횟수가 최대 $O(kp^2)$ = 402,002,500

해결 방법

1. 인접 정류장 간 좌표값이 유사하다는 점에서 해시를 떠올렸고, GeoHash를 이용하기로 함
2. 같은 GeoHash 간에는 도보 이동이 가능하고, 같은 GeoHash 내의 모든 정류장 간 도보 이동 시간이 동일한 것으로 취급

중간 결과

계산 횟수가 $O(kp)$ = 40,100으로 줄었지만, 이동 시간 계산 결과가 너무 부정확함

해결 방법

같은 GeoHash의 모든 정류장에 대해 이동 시간을 추산해 보고, 이동 시간을 만족하는 정류장이 하나라도 있는 경우 재귀적으로 인접 8방향 정류장들에 대해서도 이동 시간 확인

결과

GeoHash level 6 기준 한 해시값에 평균 12개의 정류장이 매핑되고, 400m/s 속력으로 최대 20분 도보 이동 기준 GeoHash 약 169개가 필요하므로, 호출 횟수는 약 $O(kp)$ * 12 * 169 = 81,322,800

감사합니다.