서버구조

통신방법

- HTTP 프로토콜
- HTTP(하이퍼텍스트 전송 프로토콜)는 웹에서 데이터를 전송하는 데 사용되는 프로토콜
- 요청과 응답: HTTP는 클라이언트에서 웹 서버로의 요청(Request)과 웹 서버에서 클라이언트로의 응답
- 상태 코드(Status Code): HTTP 응답은 상태 코드를 포함하며, 이 코드는 요청의 결과

상태코드

1xx (Informational) - 요청이 수신되었으며 처리 중

- 100 Continue -서버가 클라이언트의 일부 요청을 받아들였으며, 클라이언트는 나머지 요청을 계속할 수 있음

2xx (Successful) - 요청이 성공적으로 처리

- 200 OK -요청이 성공적으로 처리되었음
- 201 Created -요청이 성공적으로 처리되었고, 새로운 리소스가 생성되었음
- 204 No Content - 요청은 성공적으로 처리되었지만, 응답 본문에 내용이 없음

3xx (Redirection) - 요청을 완료하려면 추가 조치가 필요

- 301 Moved Permanently -요청한 리소스가 새로운 URL로 영구적으로 이동되었음
- 302 Found -요청한 리소스가 일시적으로 다른 URL로 이동되었음
- 304 Not Modified - 리소스가 변경되지 않았으므로 클라이언트는 캐시된 버전을 사용할 수 있음
-

4xx (Client Error) - 클라이언트 요청에 오류가 있거나 요청을 처리할 수 없음

- 400 Bad Request: 클라이언트 요청이 잘못되었거나 서버가 이해할 수 없음
- 401 Unauthorized: 클라이언트가 인증되지 않았거나 인증 정보가 유효하지 않음
- 403 Forbidden: 클라이언트가 요청한 리소스에 액세스할 권한이 없음
- 404 Not Found: 요청한 리소스를 서버에서 찾을 수 없음

5xx (Server Error) - 서버에서 요청을 처리하는 중에 오류가 발생했음

- 500 Internal Server Error: 서버에서 처리 중에 오류가 발생했음

- 502 Bad Gateway: 서버가 게이트웨이 또는 프록시 역할을 하고 있으며, 업스트림 서버로부터 잘못된 응답을 수신
- 503 Service Unavailable: 서버가 일시적으로 서비스를 이용할 수 없음

HTTP Request verbs

- HTTP 프로토콜을 이용해 통신하는 방법에서 데이터를 전달하고 받는데 사용하는 기능

GET -> 조회

PUT -> 업데이트

POST -> 삽입

DELETE -> 삭제

개발 환경

Flask

- Python으로 웹 애플리케이션을 개발하기 위한 경량 마이크로 웹 프레임워크
- 뛰어난 확장성과 RESTful 서비스의 제공

서버 코드 구성

서버를 만들기 위한 코드의 구성은 크게 아래와 같이 분류가 가능하다

> 구성
>
> - 프레임워크 및 라이브러리 사용을 위한 import section
> - Flask를 이용해 웹 애플리케이션을 생성하는 section(기본적인 데이터 소스가 저장되어 있을 static 폴더에 대한 절대 경로를 지정하는 과정을 포함한다)
> - mysql 데이터베이스, 데이블과 연동하기 위해 필요한 사항을 변수에 저장하는 section
> - 앱에 대한 기능 설정으로 Flask 어플리케이션 설정과 SQLAlchemy를 사용해 데이터베이스에 접근하기 위한 요소들을 저장한 변수를 불러와 url에 넣어 데이터베이스를 서버와 연결하는 section
> - 연동된 데이터베이스의 테이블 구성요소(스키마 등)에 대한 처리를 위해 테이블 구조에 맞는 클래스를 생성하는 section(데이터베이스 내부의 테이블의 이름을 넣어 각 테이블과 생성한 클래스를 연결)
> - 서버 운용을 위해 필요한 함수들과 API서버의 url을 통해 PUT, GET, POST, DELETE 작업이 이루어 질 수 있도록 Python Decorator 작성한 section
> - 현재 서버에 대한 속성을 지정하는 section(포트번호, 디버그모드, host ip에 대한 접근 권한 지정 등)

프레임워크 및 라이브러리 사용을 위한 import section

import 목록

```
import os
from urllib.parse import quote_plus
from flask import Flask, request, jsonify, url_for
from flask_sqlalchemy import SQLAlchemy
from datetime import datetime
from werkzeug.exceptions import HTTPException
```

- import os
  - 서버를 작동시키는 PC 내부 Static 폴더 경로에 존재하는 사진 소스를 이용하기 위해 OS를 통한 접근 방식이 필요해 사용
- from urllib.parse import quote_plus
  - url 쿼리 문자열에 대한 인코딩을 처리하기 위해 사용
- from flask import Flask, request, jsonify, url_for
  - 기본적인 서버의 구성과 API 서버로서의 통신, JSON 형태로이 파싱, 이미지를 처리하는데 이미지의 절대 경로를 기반으로 url을 생성해 다른 디바이스(ios)등에서 표기할 수 있게 하기 위해 사용
- from flask_sqlalchemy import SQLAlchemy

    - ORM(Object-Relational Mapping) 라이브러리로 MySQL 데이터베이스와 연동과 쿼리를 진행하기 위해 사용
    - 직접적인 쿼리문의 작성 없이 함수의 사용을 통해 간편하게 쿼리를 진행할 수 있다
- from datetime import datetime
    - 데이터베이스 테이블 내부에 date와 time 타입의 속성이 존재하는데 String 타입의 데이터를 데이터베이스 테이블에 저장하고, 테이블로 받은 데이터를 수정하기 위해 String으로 변환하는 작업에 필요해 사용
- from werkzeug.exceptions import HTTPException
    - HTTP 오류를 처리하기 위한 예외 클래스로 클라이언트가 서버에 요청을 한 경우 상태, 권한에 따른 응답코드와 함께 예외가 발생한 경우 이를 안전하게 처리하기 위해 사용

- Flask를 이용해 웹 애플리케이션을 생성하는 section(기본적인 데이터 소스가 저장되어 있을 static 폴더에 대한 절대 경로를 지정하는 과정을 포함한다)

- 아래에 작성한 내용의 경우 실제 서버에서 다루는 데이터들이 존재하는 디렉토리의 위치를 삽입한 부분으로 생략을 하였다
- 기본적인 구성은 아래와 같다

```
app = Flask(__name__, static_folder='절대 경로 문자열 삽입')
```

- app은 Flask 애플리케이션을 사용하기 위한 인스턴스를 생성하기 위한 변수다
- name은 현재 서버에서 다루는 Flask 애플리케이션을 의미한다
- static_folder는 Flask 애플리케이션이 다루게 될 데이터를 저장한 디렉토리의 기본적인 절대경로를 지정한다

---

mysql 데이터베이스, 데이블과 연동하기 위해 필요한 사항을 변수에 저장하는 section

- 데이터베이스에 접근하기 위한 내용들을 저장하는 변수는 아래와 같이 선언하였다
- "" 내부에 존재하는 내용의 경우 실제 해당 시스템의 비밀번호, 호스트이름, 유저이름, 데이터베이스 이름을 포함하기 때문에 임의의 예시로 작성하여 넣어 두었다.

```
password = "password" #mysql 데이터베이스에 접근하기 위한 root(권한자)의 비밀번호
HOST = 'host' # 호스트
USERNAME = "root" # 유저이름
PASSWORD = quote_plus(password) #비밀번호를 저장한 변수를 비밀번호에 입력
DATABASE_NAME = "Database 이름" # 데이터베이스 이름
```

각 변수에 대한 설명

- password - mysql 데이터베이스에 접근하기 위한 root(권한자)의 비밀번호
- HOST - 데이터 베이스 서버 호스트
- USERNAME - 유저이름
- PASSWORD - 비밀번호를 저장한 변수를 비밀번호에 입력
- DATABASE_NAME - 데이터베이스 이름

앱에 대한 기능 설정으로 Flask 어플리케이션 설정과 SQLAlchemy를 사용해 데이터베이스에 접근하기 위한 요소들을 저장한 변수를 불러와 url에 넣어 데이터베이스를 서버와 연결하는 section

```
app.config['SQLALCHEMY_DATABASE_URI'] = f'mysql+pymysql://{USERNAME}:
{PASSWORD}@{HOST}:3306/{DATABASE_NAME}?charset=utf8mb4'
db = SQLAlchemy(app)
```

- 기본적으로 데이터 베이스에 접근하기 위해 사용하는 코드

app.config

```
app.config['SQLALCHEMY_DATABASE_URI'] = URL String
```

- SQLAlchemy가 데이터베이스에 연결하는 데 사용하는 URI(Uniform Resource Identifier)를 설정
- 여기서 app은 맨처음 Flask 애플리케이션 인스턴스를 생성한 변수 app이 된다

db = SQLAlchemy(app)

```
db = SQLAlchemy(app)
```

- Flask 애플리케이션인 app을 SQLAlchemy를 통해 데이터베이스와 연동하도록 한다
- 앞으로 다루는 모든 테이블에 대해서는 db라는 변수에 존재하는 데이터베이스 테이블에 접근하여 사용하게 된다

- 연동된 데이터베이스의 테이블 구성요소(스키마 등)에 대한 처리를 위해 테이블 구조에 맞는 클래스를 생성하는 section(데이터베이스 내부의 테이블의 이름을 넣어 각 테이블과 생성한 클래스를 연결)

CheckPoint table

- 아래 테이블 병은 실제로 이번 프로젝트에서 사용한 테이블 명이다

- 아래 클래스의 구성을 통해 데이터베이스 테이블과 맵핑을 하게 된다
- 테이블의 구성요소로는 단속 위치 주소명, 위도, 경도. 제한속도에 대한 속성이 있다
- 또한 이후 언급할 TrafficViolations 테이블은 CheckPoint 테이블과의 join 작업을 필요로 하므로 관계에 대한 정의가 필요하다
- join을 수행하는 외래키는 각 테이블에 존재하는 location 속성이 된다

```
# 모델 클래스를 지정
# (차량 번호, 위반 날짜, 위반 시간)이 하나의 키가 된다
# 외래키는 CheckPoint 테이블의 location을 참조한다
class CheckPoint(db.Model):
    __tablename__ = 'CheckPoint'

    location = db.Column(db.String(255), primary_key=True)
    lat = db.Column(db.Float, nullable=False)
    lon = db.Column(db.Float, nullable=False)
    speed_limit = db.Column(db.Integer, nullable=False)

    # TrafficViolation과의 관계 설정
    violations = db.relationship('TrafficViolation', backref='Checkpoint')

    @property
    def serialize(self):
        return {
            'location': self.location,
            'lat': self.lat,
            'lon': self.lon,
            'speed_limit': self.speed_limit
        }
```

테이블명 선언

```
__tablename__ = "테이블명"
```

- 위에서 선언한 db 변수로부터 테이블 이름을 통해 접근을 할 수 있도록 한다

속성 맵핑

```
location = db.Column(db.String(255), primary_key=True)
lat = db.Column(db.Float, nullable=False)
lon = db.Column(db.Float, nullable=False)
speed_limit = db.Column(db.Integer, nullable=False)
```

- 각 변수는 테이블이 가진 속성에 맞게 맵핑된다

JSON 표기를 위한 프로퍼티 선언

```
@property
    def serialize(self):
        return {
            'location': self.location,
            'lat': self.lat,
            'lon': self.lon,
            'speed_limit': self.speed_limit
        }
```

- 실제 get 명령어를 실행한 경우 변환된 JSON 데이터에 대한 표기를 지정하는 부분

TrafficViolations table

아래 테이블 병은 실제로 이번 프로젝트에서 사용한 테이블 명이다

- 아래 클래스의 구성을 통해 데이터베이스 테이블과 맵핑을 하게 된다
- 테이블의 구성요소로는 단속 위치 주소명, 위도, 경도. 제한속도에 대한 속성이 있다
- 또한 이후 언급할 TrafficViolations 테이블은 CheckPoint 테이블과의 join 작업을 필요로 하므로 관계에 대한 정의가 필요하다
- join을 수행하는 외래키는 각 테이블에 존재하는 location 속성이 된다
- TrafficViolations 테이블은 외래키로 location 속성을 이용하며 CheckPoint 테이블의 location 을 참조한다
- 그렇게 되면 CheckPoint의 location 속성중 하나가 삭제된다면 TrafficViolation에서 동일한 location 속성을 가진 레코드가 모두 삭제됨을 의미한다

```
class TrafficViolation(db.Model):
    __tablename__ = 'TrafficViolations'

    car_number = db.Column(db.String(255), primary_key=True)
    overspeed = db.Column(db.Integer, nullable=False)
    location = db.Column(db.String(255),
db.ForeignKey('CheckPoint.location'), nullable=False)
    violation_time = db.Column(db.Time, primary_key=True)
    violation_date = db.Column(db.Date, primary_key=True)
    image_path = db.Column(db.String(255), nullable=False)
    @property
    def serialize(self):
        image_url = url_for('static',
filename=f'images/testImage/{os.path.basename(self.image_path)}',
_external=True)
        return {
            'car_number': self.car_number,
            'overspeed': self.overspeed,
            'location': self.location,
```

```
            'violation_time': self.violation_time.isoformat() if
self.violation_time else None,
            'violation_date': self.violation_date.isoformat() if
self.violation_date else None,
            'image_path': image_url  # 이미지 파일의 상대 경로를 반환
        }
```

테이블명 선언

```
__tablename__ = "테이블명"
```

- 위에서 선언한 db 변수로부터 테이블 이름을 통해 접근을 할 수 있도록 한다

속성 맵핑

```
car_number = db.Column(db.String(255), primary_key=True)
overspeed = db.Column(db.Integer, nullable=False)
location = db.Column(db.String(255), db.ForeignKey('CheckPoint.location'),
nullable=False)
violation_time = db.Column(db.Time, primary_key=True)
violation_date = db.Column(db.Date, primary_key=True)
image_path = db.Column(db.String(255), nullable=False)
```

- 각 변수는 테이블이 가진 속성에 맞게 맵핑된다

JSON 표기를 위한 프로퍼티 선언

```
@property
    def serialize(self):
        image_url = url_for('static',
filename=f'images/testImage/{os.path.basename(self.image_path)}',
_external=True)
        return {
            'car_number': self.car_number,
            'overspeed': self.overspeed,
            'location': self.location,
            'violation_time': self.violation_time.isoformat() if
self.violation_time else None,
            'violation_date': self.violation_date.isoformat() if
self.violation_date else None,
            'image_path': image_url  # 이미지 파일의 상대 경로를 반환
        }
```

- 실제 get 명령어를 실행한 경우 변환된 JSON 데이터에 대한 표기를 지정하는 부분

- 위 CheckPoint와 다르게 image_path에 대한 부분이 image_url으로 지정되어 있는데 서버를 운용하는 PC에 저장되어있는 데이터를 데이터가 저장되어있는 절대경로 + 파일명의 형태로 가져와 url을 생성해 서버에서 해당 이미지를 실제로 표기할 수 있도록 하기 위한 작업이다
- 위 코드에서 설명은 상대경로라고 되어 있지만 실제 Flask 애플리케이션을 선언할때 사용한 static폴더를 기준으로 작성되는 상대경로이므로 JSON으로 변환한 데이터에 대해 실제로 GET 명령어를 실행한 경우 절대경로로 표시된다

- 서버 운용을 위해 필요한 함수 sectcion

서버운용에 필요한 함수

```
# API 키 검증 함수
# 키를 저장하는 배열을 생성하고 매개변수로 받은 키가 정당한 권한을 가진 키인지 확인하는 함수
def check_api_key(api_key):
    valid_api_keys = ['API_KEY1', 'API_KEY2']
    return api_key in valid_api_keys
```

- 데이터의 교환이 이루어지는 엔드포인트에서 접근에 대한 정당한 권한을 확인하기 위해 api_key를 사용하게 된다
- 해당 키에 대한 목록을 배열에 넣고 매개변수로 받은 api_key에 대한 정당한 권한이 존재하는지 여부를 판단하는 함수이다
- 위 코드 예시는 실제 서버에서 사용하는 api_key를 작성해 두어 예시로 대체해 작성하였다

API서버의 url을 통해 PUT, GET, POST, DELETE 작업이 이루어 질 수 있도록 Python Decorator 작성한 section

구성

- TrafficViolation 테이블과 CheckPoint 테이블을 join한 결과에 대해 GET 명령을 수행하는 함수
- CheckPoint 테이블에 대한 모든 레코드에 대해 GET 명령어 수행하는 함수
- TrafficViolation 테이블과 CheckPoint 테이블을 join한 테이블에 대해 POST를 통해 새로운 레코드를 삽입하는 명령을 수행하는 함수
- 실제 PC에 저장된 이미지 파일의 절대 경로를 기반으로 이미지 URL을 생성하여 서버에 전송하고 저장하기 위한 함수
- TrafficViolation 테이블과 CheckPoint 테이블을 join한 테이블 내에 존재하는 레코드에 대해 DELETE 명령을 수행하는 함수
- TrafficViolation 테이블과 CheckPoint 테이블을 join한 테이블 내에 존재하는 레코드에 대해 PUT을 통해 수정 명령을 수행하는 함수

각 함수들에 대한 공통적인 논리 구조는 아래와 같다

- 정당한 권한을 가진자의 접근인지 확인하기 위해 아래 함수를 호출해 api_key에 대한 권한을 확인

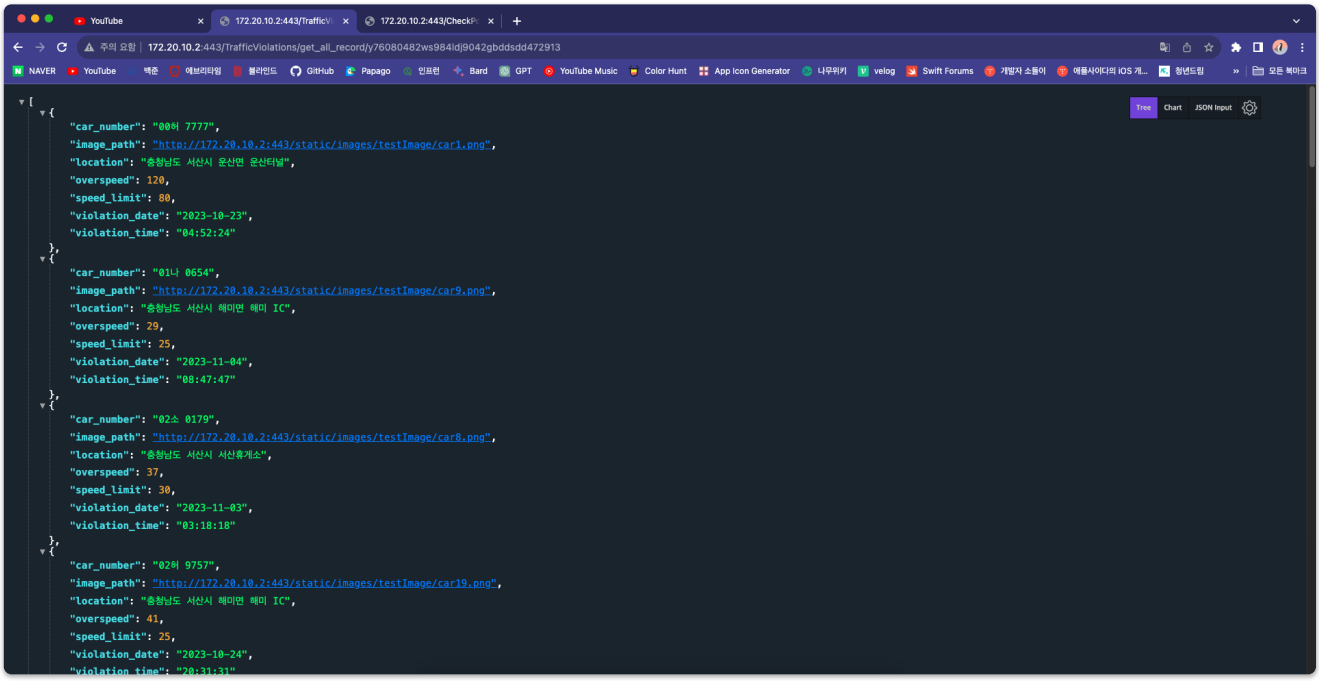
```
def check_api_key(api_key)
```

- try - except 를 통해 서버가 갑작스럽게 다운되는 현상을 방지하고 성공과 실패에 따라 상태코드를 반환하도록 지정
- try 내부에서는 GET, POST, PUT, DELETE에 따른 필요한 로직을 구현

TrafficViolation 테이블과 CheckPoint 테이블을 join한 결과에 대해 GET 명령을 수행하는 함수

```
@app.route('/TrafficViolations/get_all_record/<api_key>', methods=["GET"])
def get_all_records(api_key):
```

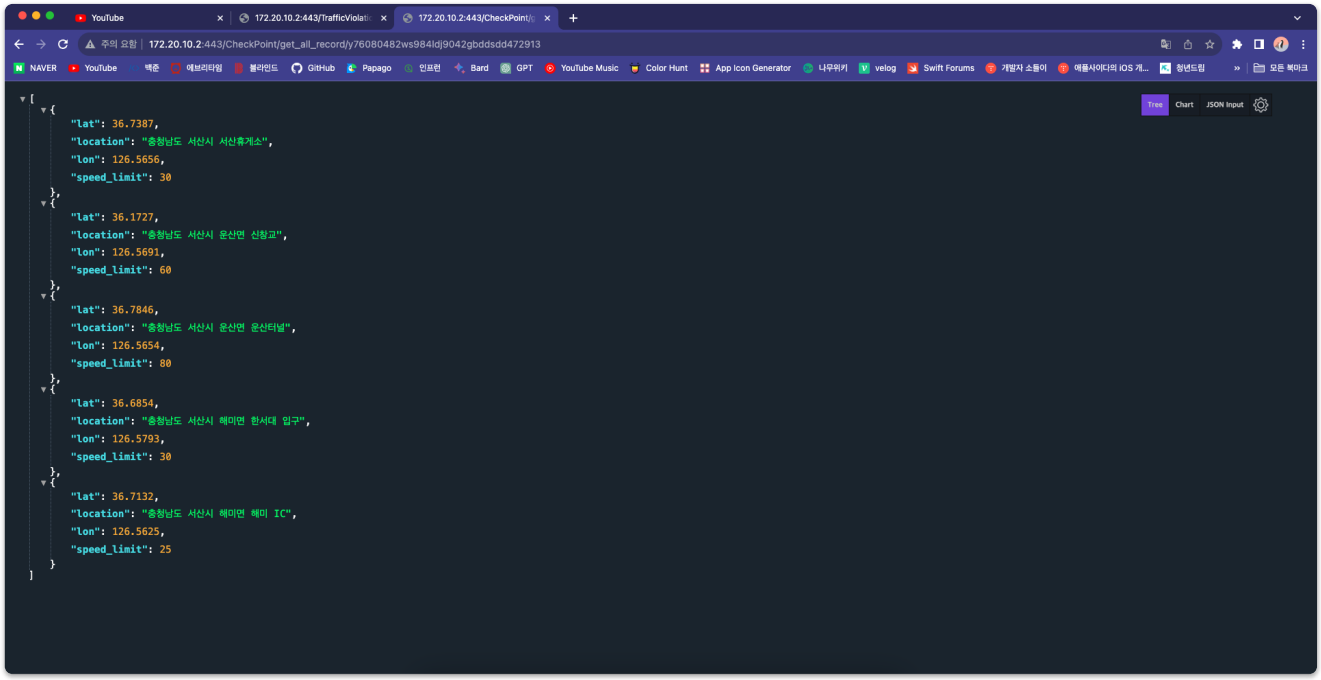
- api_key를 url에 입력되는 인자로 지정하고 해당 인자를 `def get_all_records(api_key)` 함수의 매개변수로 사용한다. 해당 매개변수를 바탕으로 `def check_api_key(api_key)` 를 통해 키에 대한 접근 권한을 검사하고 정당하지 않은 권한이라면 403 상태코드를 반환하여 JSON 데이터에 표시한다
- 우선 TrafficViolation 테이블과 CheckPoint 테이블을 join 명령을 수행한 후 JSON 데이터로 변환하기 위한 별도의 맵핑을 하지 않았으므로 해당 함수 내부에서 JSON 맵핑 작업을 진행한다
- 이미지 url의 경우 별도의 static 폴더를 기준으로 한 상대 경로의 지정, url 생성 과정을 필요로 한다
- 이후 GET 명령이 정상적으로 작동된 경우 데이터를 확인할 수 있다
- 실행결과는 아래와 같다



CheckPoint 테이블에 대한 모든 레코드에 대해 GET 명령어 수행하는 함수

```
@app.route('/CheckPoint/get_all_record/<api_key>', methods=["GET"])
def get_all_records_CheckPoint(api_key):
```

- api_key를 url에 입력되는 인자로 지정하고 해당 인자를 `def get_all_records_CheckPoint(api_key)` 함수의 매개변수로 사용한다. 해당 매개변수를 바탕으로 `def check_api_key(api_key)`를 통해 키에 대한 접근 권한을 검사하고 정당하지 않은 권한이라면 403 상태코드를 반환하여 JSON 데이터에 표시한다
- 단순히 CheckPoint 테이블의 모든 레코드를 불러오는 작업이므로 그대로 모든 레코드를 가져와 JSON으로 변환한다
- 실행 결과는 아래와 같다



TrafficViolation 테이블과 CheckPoint 테이블을 join한 테이블에 대해 POST를 통해 새로운 레코드를 삽입하는 명령을 수행하는 함수

```
@app.route('/TrafficViolations/insert_record/<api_key>', methods=['POST'])
def insert_record(api_key):
```

- api_key를 url에 입력되는 인자로 지정하고 해당 인자를 `def insert_record(api_key))` 함수의 매개변수로 사용한다. 해당 매개변수를 바탕으로 `def check_api_key(api_key)`를 통해 키에 대한 접근 권한을 검사하고 정당하지 않은 권한이라면 403 상태코드를 반환하여 JSON 데이터에 표시한다
- 우선 ios 애플리케이션은 String 타입의 데이터를 통해 수정, 삽입이 이루어지므로 TrafficViolations 테이블 내에서 상대적으로 특이한 타입인 date타입과 time에 대한 핸들링이 필요하다. 그래서 ios 어플 및 드론으로부터 데이터를 전달하는 응용프로그램 내부에서는 String 타입으로 수정과 삽입 연산을 진행하도록 하고, 서버에서 해당 String 타입의 데이터를 받아 date와 time 형태로 변환하는 과정이 필요하다. 그래서 아래 코드를 통해 String 타입의 데이터를 날짜와 시간으로 변환한다

```
formatted_violation_date = datetime.strptime(violation_date, '%Y-%m-
%d').date()
formatted_violation_time = datetime.strptime(violation_time,
'%H:%M:%S').time()
```

- 이후 기본키 쌍인 (차량번호, 위반 날짜, 위반 시간)을 기준으로 데이터를 구성한다
- 데이터의 기초적인 구성이 끝난 경우 나머지 요소인 초과속도, 위치, 이미지 경로에 대한 정보를 갱신한다
- 기본적으로 유저가 GET 요청을 하는 경우 테이블 구조가 insert 를 진행할때와 다른데 location과 제한 속도에 관한 사항은 이미지처리 응용프로그램, ios 애플리케이션 내부에서 제약을 걸어 CheckPoint 테이블에 존재하는 데이터 중 하나로만 보내는 작업을 응용프로그램 내부에서 진행하도록 한다
- 실행결과는 ios 및 이미지처리 파트에서 언급된다

실제 PC에 저장된 이미지 파일의 절대 경로를 기반으로 이미지 URL을 생성하여 서버에 전송하고 저장하기 위한 함수

```
@app.route('/TrafficViolations/upload_image/<api_key>', methods=['POST'])
def upload_image(api_key):
```

- api_key를 url에 입력되는 인자로 지정하고 해당 인자를 `def upload_image(api_key)` 함수의 매개변수로 사용한다. 해당 매개변수를 바탕으로 `def check_api_key(api_key)`를 통해 키에 대한 접근 권한을 검사하고 정당하지 않은 권한이라면 403 상태코드를 반환하여 JSON 데이터에 표시한다
- 이미지를 보낼때는 OS 라이브러리를 통해 실제 PC 혹은 이미지 처리 응용프로그램을 작동시키는 PC에서 부터 절대 경로를 파악하여 파일에 대해 접근한다
- 접근한 파일을 서버를 통해 서버 PC에 저장하고 이미지 파일이 저장되는 경로는 맨 처음 Flask 애플리케이션 선언시 작성한 static 폴더 내부가 된다

TrafficViolation 테이블과 CheckPoint 테이블을 join한 테이블 내에 존재하는 레코드에 대해 DELETE 명령을 수행하는 함수

```
@app.route('/TrafficViolations/delete/<car_number>/<violation_date>/<viola
tion_time>/<api_key>', methods=["DELETE"])
def delete_record(api_key, car_number, violation_date, violation_time):
```

- 우선 삭제를 하는데 필요한 것은 키를 통한 단일 레코드의 제거이므로 기본키로 작용하는 (차량번호, 위반 날짜, 위반 시간)을 url 주소의 인자로 받는다
- 파이썬 데코레이터를 통한 함수 또한 기본키에 대한 데이터들을 매개변수로 받는다
- api_key를 url에 입력되는 인자로 지정하고 해당 인자를 `def delete_record(api_key, car_number, violation_date, violation_time)` 함수의 매개변수로 사용한다. 해당 매개변수를 바탕으로 `def check_api_key(api_key)`를 통해 키에 대한 접근 권한을 검사하고 정당하지 않은 권한이라면 403 상태코드를 반환하여 JSON 데이터에 표시한다

- 우선 기본키로 작용되는 요소중 데이터베이스의 접근을 위해 방해가 발생하는 부분으로 위반 날짜와 위반 시간이 존재한다. 위에서 언급했듯 응용프로그램들 내에서는 날자와 시간을 String 타입의 데이터를 이용해 수정과 삽입을 이루고 삽입이 발생할 경우 date와 time에 맞는 타입 변환이 필요하다. 그에 따라 데이터베이스의 접근을 위해 아래코드를 이용하여 타입을 변화시키고 해당 내용을 기반으로 기본 키 쌍을 생성해 키본 키에 부합하는 레코드를 지정한다

```
formatted_violation_date = datetime.strptime(violation_date, '%Y-%m-
%d').date()
formatted_violation_time = datetime.strptime(violation_time,
'%H:%M:%S').time()
```

- 기본키를 통해 레코드가 지정된 경우 키를 통해 중복되는 데이터가 없음을 증명하기 때문에 그대로 삭제 명령을 내려주면 된다
- 실행결과는 ios 파트에서 언급된다

TrafficViolation 테이블과 CheckPoint 테이블을 join한 테이블 내에 존재하는 레코드에 대해 PUT을 통해 수정 명령을 수행하는 함수

```
@app.route('/TrafficViolations/update_record/<car_number>/<violation_date>
/<violation_time>/<api_key>', methods=['PUT'])
def update_record(api_key, car_number, violation_date, violation_time):
```

- 우선 수정을 진행 하는데 필요한 것은 키를 통한 단일 레코드의 특정이므로 기본키로 작용하는 (차량 번호, 위반 날짜, 위반 시간)을 url 주소의 인자로 받는다
- 파이썬 데코레이터를 통한 함수 또한 기본키에 대한 데이터들을 매개변수로 받는다
- api_key를 url에 입력되는 인자로 지정하고 해당 인자를 `def update_record(api_key, car_number, violation_date, violation_time)` 함수의 매개변수로 사용한다. 해당 매개변수를 바탕으로 `def check_api_key(api_key)` 를 통해 키에 대한 접근 권한을 검사하고 정당하지 않은 권한이라면 403 상태코드를 반환하여 JSON 데이터에 표시한다
- 삭제와 마찬가지로 특정 레코드에 대한 접근을 위해서는 위반 날짜와 위반 시간에 대한 타입변환을 통해 정확한 레코드의 특정이 필요하다 그러므로 아래 코드를 이용해 String 타입의 데이터를 date, time 타입으로 변환한다

```
formatted_violation_date = datetime.strptime(violation_date, '%Y-%m-
%d').date()
formatted_violation_time = datetime.strptime(violation_time,
'%H:%M:%S').time()
```

- 그럼 기본키의 쌍을 이용해 특정 레코드에 대한 접근을 실시한다
- 이후 ios 앱 내부에서 수정된 데이터들을 서버로 보내고 commit을 진행한다
- 실행 결과는 ios 앱 내부에서 진행한다

현재 서버에 대한 속성을 지정하는 section(포트번호, 디버그모드, host ip에 대한 접근 권한 지정 등)

```
if __name__ == '__main__':
    app.run(debug=False, host='0.0.0.0', port=443)
```

- 서버 접근에 대한 모든 ip의 접근을 허용하도록 host를 지정한다
- 서버와의 통신을 위한 port 번호를 443 포트로 지정한다

결과

- 정상적으로, 기본적인 GET, PUT, POST, DELETE에 대한 명령을 정상적으로 수행한다
- 데이터에 대한 삽입, 삭제, 수정, 조회, 검색에 대한 기능은 이미지 처리 애플리케이션과 ios 애플리케이션 내부에서 수행하므로 이를 참조하면 된다

아쉬운 점

- 보안성에 대한 취약
- 도메인과 SSL 인증이 존재하지 않음
- 표시 결과가 TrafficViolation 과 CheckPoint 테이블의 join을 통해 산출되므로 삽입, 수정의 경우 약간의 제약사항이 존재하여 해당 부분은 서버와 통신하는 애플리케이션 내부에서 제약사항을 걸도록 만들 수 밖에 없는 구조다