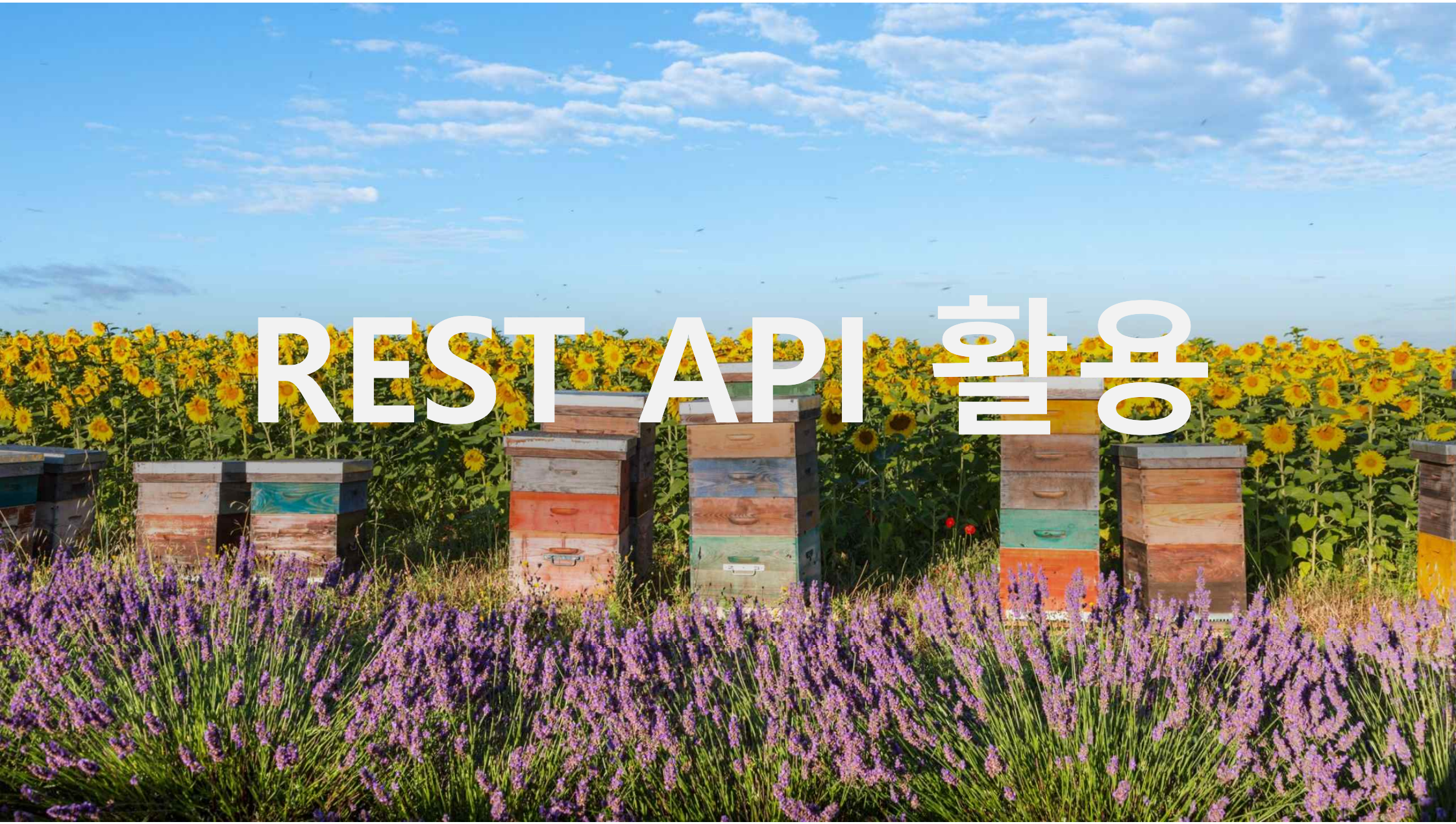

REST API 활용

Flask 웹 프레임워크

1. Flask 개요

- Flask는 Python 기반의 마이크로 웹 프레임워크이다.
- Django보다 가볍고, 필요한 기능을 직접 추가하여 사용할 수 있다.
- 확장성이 뛰어나며, REST API 개발에 많이 사용된다.
- 단순한 구조로 배우기 쉽고 빠르게 개발 가능하다.
- Jinja2 템플릿 엔진을 기본적으로 지원하여 HTML 렌더링이 용이하다.
- Werkzeug WSGI 툴킷을 기반으로 하며, Pythonic한 코드 스타일을 지향한다.

2. Flask 특징

1) 경량 프레임워크

- 최소한의 기능만 제공하여 가볍고 빠르게 동작한다.
- 필요한 라이브러리를 선택적으로 추가 가능하다.
- 단순한 아키텍처로 초보자도 쉽게 익힐 수 있다.

2) 유연성과 확장성

- ORM, 폼 처리, 인증 기능 등 필요에 따라 추가 가능하다.
- 플러그인을 활용하여 쉽게 확장 가능하다.
- 다양한 플러그인(Flask-SQLAlchemy, Flask-WTF, Flask-Login 등)과 호환된다.

3) 쉬운 라우팅과 요청 처리

- URL 매핑을 쉽게 정의할 수 있다.
- HTTP 메서드(GET, POST 등) 처리가 간편하다.
- Blueprint를 활용하여 대규모 애플리케이션 개발 시 라우트를 효율적으로 관리할 수 있다.

4) 템플릿 엔진 Jinja2 지원

- 동적 HTML 페이지를 쉽게 렌더링할 수 있다.
- 필터, 제어문, 템플릿 상속 등의 기능 제공
- 템플릿 내에서 Python 코드 사용 가능

5) 다양한 확장 라이브러리 지원

- Flask-SQLAlchemy (ORM 지원)
- Flask-WTF (폼 처리)
- Flask-Login (사용자 인증)
- Flask-RESTful (REST API 개발)
- Flask-Mail (이메일 전송)
- Flask-Caching (캐싱 기능)

3. Flask 설치 및 기본 사용법

1) Flask 설치

```
pip install flask
```

2) 기본적인 Flask 코드

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def home():
    return "Hello, Flask!"

if __name__ == '__main__':
    app.run(debug=True)
```

3) 라우팅 및 동적 처리

```
@app.route('/user/<name>')
def user(name):
    return f"Hello, {name}!"
```

4) 템플릿 사용 (Jinja2)

템플릿 파일 (templates/index.html)

```
<!DOCTYPE html>
<html>
<head>
    <title>Flask Example</title>
</head>
<body>
    <h1>Hello, {{ name }}!</h1>
</body>
</html>
```

Flask 코드:

```
from flask import render_template

@app.route('/hello/<name>')
def hello(name):
    return render_template('index.html', name=name)
```

4. Flask 주요 기능

1) 폼 데이터 처리

```python
from flask import request

@app.route('/submit', methods=['POST'])
def submit():
    username = request.form['username']
    return f"Submitted: {username}"
```

2) JSON 데이터 반환

```python
from flask import jsonify

@app.route('/api/data')
def api_data():
    data = {"message": "Flask JSON Response"}
    return jsonify(data)
```

📩 GET & POST 요청 처리

Flask에서는 HTTP 요청(GET, POST 등)을 처리할 수 있습니다.

python 복사 편집

```python
from flask import request

@app.route("/login", methods=["GET", "POST"])
def login():
    if request.method == "POST":
        username = request.form["username"]
        return f"Welcome, {username}!"
    return '<form method="post"><input name="username"><input type="submit"></form>'
```

3) Flask-SQLAlchemy 사용 (데이터베이스 연동)

```
from flask_sqlalchemy import SQLAlchemy

app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///database.db'
db = SQLAlchemy(app)

class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(80), nullable=False)

db.create_all()
```

4) 세션(Session) 및 쿠키 관리

```python
from flask import session

app.secret_key = 'supersecretkey'

@app.route('/set_session')
def set_session():
    session['user'] = 'Jane Doe'
    return 'Session set'

@app.route('/get_session')
def get_session():
    return session.get('user', 'No session set')
```

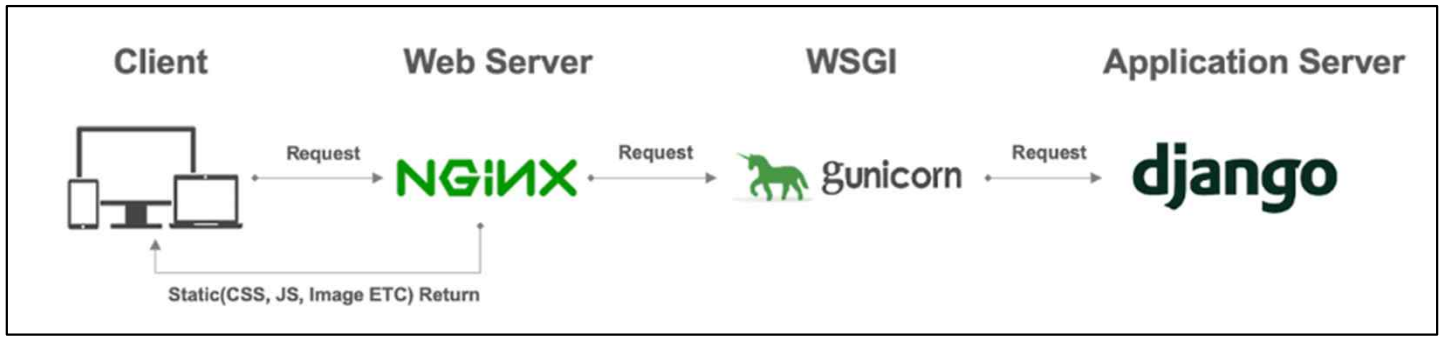
5. Flask 실행 및 배포

1) 개발 서버 실행

```
flask run
```

2) Gunicorn을 활용한 배포

```
pip install gunicorn
gunicorn -w 4 -b 0.0.0.0:5000 app:app
```



출처 : https://leffept.tistory.com/345

3) Docker를 활용한 배포

Dockerfile

```
FROM python:3.9
WORKDIR /app
COPY . /app
RUN pip install flask gunicorn
CMD ["gunicorn", "-w", "4", "-b", "0.0.0.0:5000", "app:app"]
```

Flask 확장 기능

Flask는 확장 기능을 통해 다양한 기능을 추가할 수 있습니다.

확장 기능	설명
Flask-SQLAlchemy	데이터베이스 연동
Flask-WTF	폼 처리
Flask-Login	사용자 인증
Flask-Mail	이메일 전송
Flask-RESTful	REST API 개발

6. Flask 활용 사례

- REST API 서버 구축
- 간단한 웹 애플리케이션 개발
- IoT 및 데이터 시각화 대시보드
- 챗봇 및 머신러닝 모델 서빙
- 인증 시스템 구축 (OAuth, JWT 등 활용)
- 대규모 애플리케이션을 위한 Blueprint 구조 적용

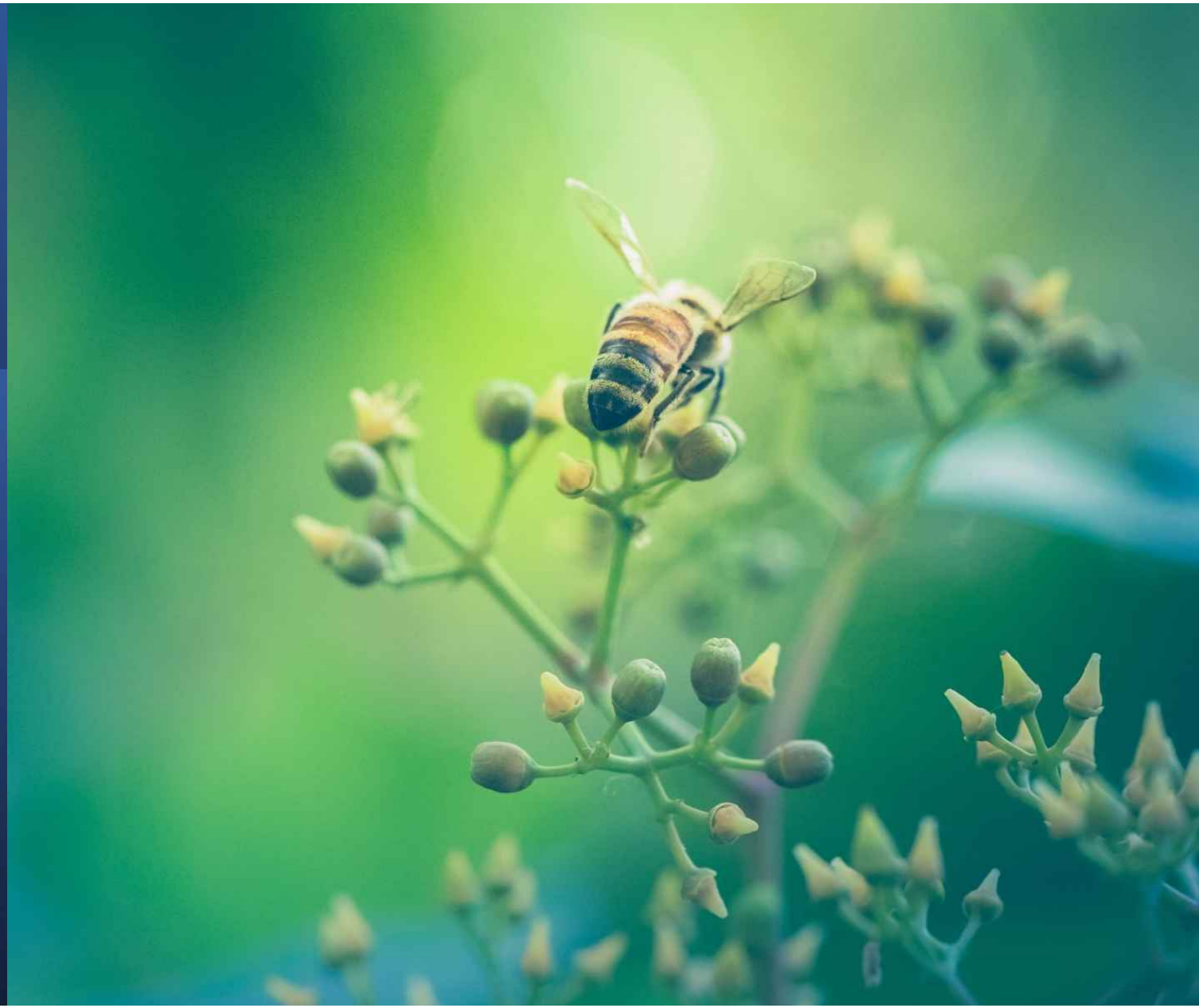
7. Flask vs Django 비교

비교 항목	Flask	Django
구조	가볍고 유연함	풀스택 프레임워크
학습 난이도	쉬움	상대적으로 어려움
ORM	선택 사항	기본 내장
확장성	필요에 따라 추가	기본 기능 제공
API 개발	REST API 개발에 적합	웹 애플리케이션 개발에 적합
보안 기능	직접 구현 필요	내장된 보안 기능 지원

8. 결론

- Flask는 가벼우면서도 강력한 웹 프레임워크로, 빠르게 웹 애플리케이션을 개발할 수 있다.
- 확장성이 뛰어나며, 다양한 플러그인과 라이브러리를 활용할 수 있다.
- REST API 및 데이터 처리 중심의 프로젝트에 적합하다.
- 대규모 프로젝트의 경우 Django가 더 적합할 수 있지만, Flask는 유연한 확장성을 갖추고 있다.

API 인증 및 키 관리



API 인증 개요

API 인증(Authentication)은 클라이언트가 API를 사용할 수 있도록 신원을 확인하는 과정입니다. API 키(API Key), OAuth, JWT(JSON Web Token) 등의 방법이 있으며, 가장 일반적인 방식 중 하나가 API 키를 사용하는 것입니다.

API 인증 및 키 관리

1. API 키 관리

API 키는 특정 애플리케이션이 API에 접근할 수 있도록 허용하는 고유한 문자열입니다. API 제공자는 보안 강화를 위해 키를 적절히 관리해야 합니다.

◆ API 키 발급 및 사용

- API 제공자가 API 키를 생성하여 개발자에게 제공합니다.
- 개발자는 API 요청 시 이 키를 포함하여 인증을 수행합니다.

◆ 보안 유지 방법

- **환경 변수에 저장**: API 키를 코드에 직접 포함하지 않고 환경 변수에서 불러와 사용합니다.
- **IP 제한**: 특정 IP 주소에서만 API 키를 사용할 수 있도록 설정합니다.
- **사용량 제한**: 요청 횟수(rate limit)를 설정하여 과도한 API 호출을 방지합니다.
- **만료 및 갱신**: 정기적으로 API 키를 교체하고, 필요 시 즉시 폐기할 수 있도록 합니다.

2. API 요청 시 헤더와 파라미터 사용

API 키를 포함하는 방법은 API 제공자의 정책에 따라 다를 수 있으며, 보통 **헤더(Header) 또는 쿼리 파라미터(Query Parameter)** 방식이 사용됩니다.

✅ 헤더(Header) 방식 (권장)

API 키를 요청 헤더에 포함하는 방식으로, 보안성이 높고 URL에 노출되지 않기 때문에 가장 많이 사용됩니다.

📌 예제 (cURL)

sh 복사 편집

```sh
curl -H "Authorization: Bearer YOUR_API_KEY" \
     -H "Content-Type: application/json" \
     https://api.example.com/data
```

API 인증 및 키 관리

📌 예제 (Python - requests)

```python
import requests

url = "https://api.example.com/data"
headers = {
    "Authorization": "Bearer YOUR_API_KEY",
    "Content-Type": "application/json"
}

response = requests.get(url, headers=headers)
print(response.json())
```

📌 헤더에 포함되는 일반적인 정보

- `Authorization` : 인증 방식(Bearer, Basic 등)과 API 키를 포함
- `Content-Type` : 요청 본문의 데이터 형식 (예: `application/json`)
- `Accept` : 응답 데이터의 형식 지정 (예: `application/json`)

API 인증 및 키 관리

✅ 쿼리 파라미터(Query Parameter) 방식

URL에 API 키를 직접 포함하는 방식으로, 상대적으로 보안이 취약하지만 간단한 테스트나 공개 API에 사용됩니다.

📌 예제

sh 복사 편집

```sh
curl "https://api.example.com/data?api_key=YOUR_API_KEY"
```

📌 예제 (Python - requests)

python 복사 편집

```python
import requests

url = "https://api.example.com/data"
params = {
    "api_key": "YOUR_API_KEY"
}

response = requests.get(url, params=params)
print(response.json())
```

API 인증 및 키 관리

📌 쿼리 파라미터 방식의 단점

- URL에 API 키가 그대로 노출되어 보안에 취약
- 로그에 남을 가능성이 높아 악용 위험 존재

✅ **보안이 중요한 API의 경우, 쿼리 파라미터 방식보다 헤더 방식 사용을 권장합니다.**

3. 추가적인 보안 강화 방법

- **OAuth 2.0** 사용: API 키보다 보안성이 뛰어나며, 사용자 인증 및 권한 부여 기능을 포함
- **JWT (JSON Web Token)** 활용: API 인증 및 사용자 정보를 포함한 토큰 사용
- **HMAC (Hash-based Message Authentication Code)** 적용: API 키를 해싱하여 전송, 데이터 변조 방지

Flask를 사용한 API 인증 예제

Flask에서 API 키를 이용한 인증을 구현하는 방법을 알아보겠습니다.
Flask는 Python 기반의 가벼운 웹 프레임워크로, REST API를 쉽게 개발할 수 있습니다.

1️⃣ Flask API 기본 구조

우선, Flask를 설치하고 기본 API 엔드포인트를 설정해 봅니다.

📌 설치

```sh
pip install flask
```

API 인증 및 키 관리

📌 기본 API 코드 (`app.py`)

python

```python
from flask import Flask, request, jsonify

app = Flask(__name__)

@app.route('/api/data', methods=['GET'])
def get_data():
    return jsonify({"message": "Hello, API!"})

if __name__ == '__main__':
    app.run(debug=True)
```

🚀 실행

sh

```sh
python app.py
```

이제 `http://127.0.0.1:5000/api/data`로 GET 요청을 보내면 JSON 응답을 받습니다.

API 인증 및 키 관리

2️⃣ API 키를 사용한 인증 추가

이제 API 키를 사용하여 인증을 추가해보겠습니다.

✅ 1. API 키 관리

서버에서 사용할 API 키를 설정합니다.

```python
API_KEY = "my_secret_api_key"
```

API 인증 및 키 관리

✅ 2. API 키를 헤더에서 확인하는 함수

클라이언트가 보낸 `Authorization` 헤더에서 API 키를 확인하는 기능을 추가합니다

python

```python
from flask import Flask, request, jsonify

app = Flask(__name__)

API_KEY = "my_secret_api_key"  # API 키 설정

def verify_api_key(request):
    """ 요청의 API 키를 검증하는 함수 """
    api_key = request.headers.get("Authorization")
    if api_key and api_key == f"Bearer {API_KEY}":
        return True
    return False
```

API 인증 및 키 관리

```python
@app.route('/api/data', methods=['GET'])
def get_data():
    if not verify_api_key(request):
        return jsonify({"error": "Unauthorized"}), 401  # 인증 실패

    return jsonify({"message": "Authenticated!", "data": [1, 2, 3, 4, 5]})

if __name__ == '__main__':
    app.run(debug=True)
```

API 인증 및 키 관리

📌 실행 후 테스트

sh 복사

```sh
curl -H "Authorization: Bearer my_secret_api_key" http://127.0.0.1:5000/api/data
```

📌 응답 (성공)

json 복사

```json
{
    "message": "Authenticated!",
    "data": [1, 2, 3, 4, 5]
}
```

API 인증 및 키 관리

📌 API 키 없이 요청하면 인증 실패

```sh
curl http://127.0.0.1:5000/api/data
```

📌 응답 (실패)

```json
{
    "error": "Unauthorized"
}
```

API 인증 및 키 관리

3 쿼리 파라미터로 API 키 인증

API 키를 헤더가 아니라 URL 파라미터로 전달하는 방식도 있습니다.

python

```python
@app.route('/api/data', methods=['GET'])
def get_data():
    api_key = request.args.get("api_key")
    if not api_key or api_key != API_KEY:
        return jsonify({"error": "Unauthorized"}), 401

    return jsonify({"message": "Authenticated!", "data": [1, 2, 3, 4, 5]})
```

API 인증 및 키 관리

📌 요청 (쿼리 파라미터 방식)

```sh
curl "http://127.0.0.1:5000/api/data?api_key=my_secret_api_key"
```

📌 응답 (성공)

```json
{
    "message": "Authenticated!",
    "data": [1, 2, 3, 4, 5]
}
```

API 인증 및 키 관리

4 보안 강화

1. **API 키를 환경 변수에 저장** (코드에 직접 포함하지 않음)

```python
import os
API_KEY = os.getenv("API_KEY", "default_key")
```

`.env` 파일에 `API_KEY=my_secret_api_key`를 저장하고 `python-dotenv` 패키지로 불러올 수도 있음.

2. **HTTPS 사용** (API 키 노출 방지)

✅ 1. HTTP vs HTTPS 차이점

프로토콜	보안 방식	API 키 보호
HTTP	데이터가 **암호화되지 않음**	API 키가 네트워크에서 **평문(Plain Text)으로 노출됨**
HTTPS	**TLS/SSL 암호화** 사용	API 키가 **암호화되어 안전하게 전송됨**

```
curl -H "Authorization: Bearer my_secret_api_key" https://api.example.com/data
```

API 인증 및 키 관리

3. **Rate Limiting 적용** (Flask-Limiter 사용)

sh 복사

```
pip install flask-limiter
```

python 복사

```
from flask_limiter import Limiter
from flask_limiter.util import get_remote_address

limiter = Limiter(get_remote_address, app=app, default_limits=["10 per minute"])
```

✅ 정리

- Flask API에서 **헤더 방식**으로 API 키를 전달하는 것이 보안적으로 더 안전함.
- 쿼리 파라미터 방식도 가능하지만 보안이 취약할 수 있음.
- **환경 변수 사용, HTTPS 적용, Rate Limiting** 등 보안 강화를 추천.

실전 REST API 활용

실전 REST API 활용

1. REST API 기본 개념

REST API(Representational State Transfer API)는 클라이언트와 서버 간의 통신을 HTTP 프로토콜을 사용하여 수행하는 방식이다. API 요청(Request)과 응답(Response)은 일반적으로 JSON 또는 XML 형식으로 이루어진다.

REST API의 주요 특징

- **Stateless(무상태성):** 각 요청은 독립적이며 서버는 이전 요청을 기억하지 않는다.
- **CRUD 메서드 활용:**
  - `GET` → 데이터 조회
  - `POST` → 데이터 생성
  - `PUT/PATCH` → 데이터 수정
  - `DELETE` → 데이터 삭제
- **엔드포인트(Endpoint) 구성:** URL 구조를 기반으로 리소스를 식별함
  예) `https://api.example.com/users/123`

2. 여러 API 엔드포인트 활용

REST API에서는 하나의 서비스에서 여러 개의 엔드포인트를 제공하여 다양한 기능을 수행할 수 있다.

(1) 다중 API 호출 패턴

여러 개의 엔드포인트를 활용하여 데이터를 조합할 때는 다음과 같은 패턴이 사용된다.

① 순차적 API 호출

한 API 호출의 결과를 기반으로 다음 API를 호출하는 방식.
예제: 사용자 정보를 조회한 후, 해당 사용자의 주문 정보를 가져오기

실전 REST API 활용

```python
import requests

# 1. 사용자 정보 조회
user_response = requests.get("https://api.example.com/users/123")
user_data = user_response.json()

# 2. 사용자의 주문 정보 조회
orders_response = requests.get(f"https://api.example.com/orders?user_id={user_data['id']}"
orders_data = orders_response.json()

print(orders_data)
```

실전 REST API 활용

② 병렬 API 호출

여러 개의 API를 동시에 호출하여 성능을 최적화하는 방식.

예제: `asyncio`와 `aiohttp`를 사용한 비동기 API 호출

python

```python
import asyncio
import aiohttp

async def fetch_data(url):
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as response:
            return await response.json()
```

실전 REST API 활용

```python
async def main():
    user_url = "https://api.example.com/users/123"
    orders_url = "https://api.example.com/orders?user_id=123"

    # 두 개의 API를 동시에 호출
    user_data, orders_data = await asyncio.gather(
        fetch_data(user_url),
        fetch_data(orders_url)
    )

    print(user_data, orders_data)

asyncio.run(main())
```

3. 데이터 가공

API로 가져온 데이터는 대부분 정제 및 가공 과정이 필요하다.

(1) JSON 데이터 가공

JSON 데이터에서 특정 필드만 추출하거나 변환하는 방법.

python

```python
import json

# 예제 JSON 데이터
response_data = '''
{
    "user": {
        "id": 123,
        "name": "Jane Doe",
        "email": "jane.doe@example.com"
    },
    "orders": [
        {"id": 1, "item": "Laptop", "price": 1200},
        {"id": 2, "item": "Mouse", "price": 50}
    ]
}
'''
```

실전 REST API 활용

```
# JSON 파싱
data = json.loads(response_data)

# 필요한 데이터만 가공
user_info = {
    "id": data["user"]["id"],
    "name": data["user"]["name"],
    "total_orders": len(data["orders"])
}

print(user_info)
```

출력 결과:

```
json

{"id": 123, "name": "Jane Doe", "total_orders": 2}
```

(2) Pandas를 활용한 데이터 정리

REST API 응답 데이터를 Pandas 데이터프레임으로 변환하여 분석.

python

```python
import pandas as pd

# API 응답 데이터
orders = [
    {"id": 1, "item": "Laptop", "price": 1200},
    {"id": 2, "item": "Mouse", "price": 50},
    {"id": 3, "item": "Keyboard", "price": 100}
]

# 데이터프레임 변환
df = pd.DataFrame(orders)

# 총 가격 계산
df["total_price"] = df["price"] * 1.1  # 부가세 10% 적용

import ace_tools as tools
tools.display_dataframe_to_user(name="주문 내역", dataframe=df)
```

(3) API 데이터 필터링 및 정렬

특정 조건을 만족하는 데이터만 필터링하고 정렬할 수도 있다.

python 복사

```python
# 100달러 이상인 주문만 필터링 후 가격 기준으로 정렬
filtered_df = df[df["price"] > 100].sort_values(by="price", ascending=False)

tools.display_dataframe_to_user(name="100달러 이상 주문 내역", dataframe=filtered_df)
```

4. REST API 데이터 활용 예제

(1) 외부 API 활용 예제 - OpenWeather API

날씨 데이터를 가져와서 현재 온도를 출력하는 예제.

```
import requests

API_KEY = "your_api_key"
city = "Seoul"
url = f"http://api.openweathermap.org/data/2.5/weather?q={city}&appid={API_KEY}&units=metric"

response = requests.get(url)
weather_data = response.json()

# 필요한 데이터만 출력
current_temp = weather_data["main"]["temp"]
print(f"{city}의 현재 온도: {current_temp}°C")
```

5. API 통합 및 자동화

REST API를 활용할 때 여러 개의 API를 조합하여 자동화할 수 있다.

(1) 웹훅(Webhook) 활용

- 특정 이벤트 발생 시 API 호출을 자동화
- 예: 사용자가 가입하면 이메일 API를 호출하여 환영 이메일 발송

```python
import requests

# 이메일 API 엔드포인트 (예제)
email_api_url = "https://api.emailservice.com/send"

# 이메일 전송 데이터
payload = {
    "to": "jane.doe@example.com",
    "subject": "Welcome!",
    "body": "Jane, 환영합니다!"
}

# POST 요청 보내기
response = requests.post(email_api_url, json=payload)
print(response.status_code, response.json())
```

6. 결론

REST API를 활용하여 여러 엔드포인트에서 데이터를 가져오고, 이를 가공하여 활용할 수 있다.

- 순차적 API 호출과 병렬 API 호출을 적절히 활용
- JSON 및 Pandas를 사용한 데이터 정제 및 가공
- 실시간 API 데이터를 활용하여 자동화 가능

The End