Flask 특강

3일차: DB 연동 & 프로젝트 구조화

목 차

- 2일차 복습
- 데이터베이스 기초
- SQLAlchemy ORM
- DB 세팅 및 기초 실습
- API + DB 연결: To-Do API 2차 버전

2일차 복습

[실습 2] 실습 1에 Jinja2 적용하기

레이아웃 상속: extends / block

```html
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>{% block title %}Flask 설문{% endblock %}</title>
</head>
<body>
  <header>
    <h1>Flask 설문 시스템</h1>
    <hr>
  </header>

  <!-- 개별 페이지 내용이 들어가는 영역 -->
  {% block content %}{% endblock %}

  <footer>
    <hr>
    <p>© BE14 캠프</p>
  </footer>
</body>
</html>
```

templates/base.html

[실습 2] 실습 1에 Jinja2 적용하기

레이아웃 상속: extends / block

```
{% extends "base.html" %}

{% block title %}메인 페이지{% endblock %}

{% block content %}
  <h2>메인 페이지</h2>
  <a href="/survey">설문조사 하러 가기</a>
{% endblock %}
```

templates/index.html

[실습 2] 실습 1에 Jinja2 적용하기

레이아웃 상속: extends / block

```
{% extends "base.html" %}

{% block title %}설문조사{% endblock %}

{% block content %}
  <h2>설문조사</h2>
  <form action="/result" method="get">
    {% for q in questions %}
      <p>
        {{ loop.index }}. {{ q }}
        <input type="text" name="answer">
      </p>
    {% endfor %}
    <button type="submit">제출</button>
  </form>
{% endblock %}
```

templates/survey.html

[실습 2] 실습 1에 Jinja2 적용하기

레이아웃 상속: extends / block

```
{% extends "base.html" %}

{% block title %}결과{% endblock %}

{% block content %}
  <h2>설문 결과</h2>
  <ul>
    {% for a in answers %}
      <li>{{ a }}</li>
    {% endfor %}
  </ul>
{% endblock %}
```

templates/result.html

API 문서화: 좋은 API 문서화 요소

- API 설계 원칙: **일관성**, 엔드포인트 설계, HTTP 상태 코드
  - API 사용자가 엔드포인트와 데이터 구조를 예측 가능하게 사용하도록 구성

```
GET /users              // 사용자 목록 조회
GET /users/123          // 특정 사용자 조회
POST /users             // 사용자 생성
DELETE /users/123       // 사용자 삭제
```



```
GET /getUsers           // 사용자를 조회
GET /fetchUser/123      // 특정 사용자 조회
POST /newUser           // 사용자 생성
DELETE /deleteUser/123 // 사용자 삭제
```

API 문서화: 좋은 API 문서화 요소

- API 설계 원칙: 일관성, **엔드포인트 설계**, HTTP 상태 코드
  - 리소스를 명확히 나타내야 하며, 동작이 아닌 데이터를 중심으로 설계하도록 구성
  - 설계 원칙
    - 복수형 사용: /users vs /user
    - 계층적 구조: 관계가 있는 리소스를 논리적으로 표현

```
// 사용자 123의 주문 목록 조회
GET /users/123/orders

GET /ordersByUserId?userId=123
```

API 문서화: 좋은 API 문서화 요소

- API 설계 원칙: 일관성, **엔드포인트 설계**, HTTP 상태 코드
  - 리소스를 명확히 나타내야 하며, 동작이 아닌 데이터를 중심으로 설계하도록 구성
  - 설계 원칙
    - 필터링, 정렬, 페이징 지원 (= 성능 최적화)

```
GET /products?category=shirts          // 필터링
GET /products?sort=price&order=asc     // 정렬
GET /products?page=2&limit=20          // 페이징
```

API 문서화: 좋은 API 문서화 요소

- API 설계 원칙: 일관성, 엔드포인트 설계, **HTTP 상태 코드**
  - 상태 코드를 통해 요청의 결과를 명확히 전달

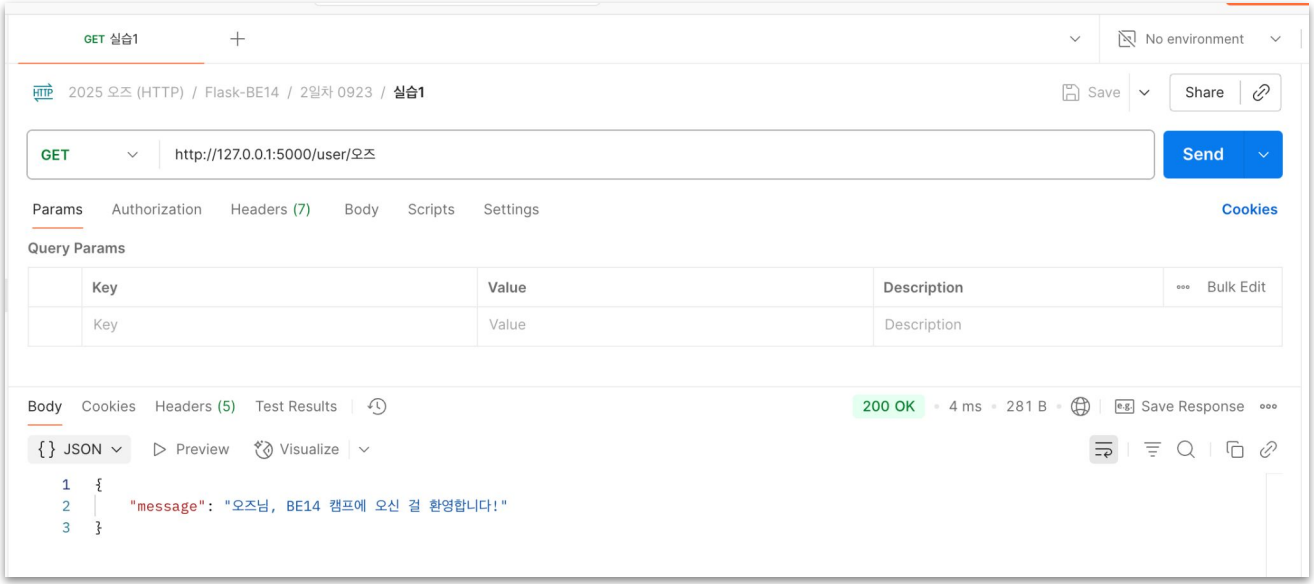
https://developer.mozilla.org/ko/docs/Web/HTTP/Status

클래스	설명	대표 상태 코드
2xx	요청이 성공적으로 처리됨	200 OK 201 Created 204 No Content
3xx	Redirection. 요청 완료를 위해 추가 작업이 필요함	301 Moved Permanently 302 Found 304 Not Modified
4xx	Client Error 클라이언트의 잘못된 요청으로 서버가 처리할 수 없음	400 Bad Request 401 Unauthorized 403 Forbidden 404 Not Found
5xx	Server Error 서버가 요청을 처리하던 도중 오류 발생	500 Internal Server Error 502 Bad Gateway 503 Service Unavailable 504 Gateway Timeout

API 문서화: 문서화 도구 추천

[실습 2] API 문서 세팅하기 (Postman, Swagger)

- Postman



API 문서화: 문서화 도구 추천

[실습 2] API 문서 세팅하기 (Postman, Swagger)

- Swagger

```python
from flask import Flask, jsonify
from flasgger import Swagger

app = Flask(__name__)
swagger = Swagger(app)

@app.route('/hello')
def hello():
    """
    Hello API
    ---
    responses:
      200:
        description: 성공 응답
        schema:
          type: object
          properties:
            message:
              type: string
              example: "Hello, OZ BE14!"
    """
    return jsonify({"message": "Hello, OZ BE14!"})

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

CRUD <-> HTTP 메서드

동작	메서드	예시 엔드포인트
Create	POST	/todos
Read (전체)	GET	/todos
Read (특정)	GET	/todos/<id>
Update	PUT (또는 PATCH)	/todos/<id>
Delete	DELETE	/todos/<id>

Flask CRUD API 만들기

[실습 3] CRUD <-> HTTP 메서드 표에서 아래 API 구현하기

- [GET] /todos

```
@app.route("/todos", methods=["GET"])
def get_todos():
    return jsonify(todos)
```

```
@app.route("/todos/<int:todo_id>", methods=["GET"])
def get_todo(todo_id):
    task = todos.get(todo_id)
    if not task:
        return jsonify({"error": "Todo not found"}), 404
    return jsonify({todo_id: task})
```

Flask CRUD API 만들기

[실습 3] CRUD <-> HTTP 메서드 표에서 아래 API 구현하기

- [POST] /todos (Body JSON)

```python
@app.route("/todos", methods=["POST"])
def create_todo():
    data = request.get_json()
    new_id = max(todos.keys()) + 1 if todos else 1
    todos[new_id] = data["task"]
    return jsonify({new_id: todos[new_id]}), 201
```

Flask CRUD API 만들기

[실습 3] CRUD <-> HTTP 메서드 표에서 아래 API 구현하기

- [PUT] /todos/<int:todo_id> (Body JSON)

```
@app.route("/todos/<int:todo_id>", methods=["PUT"])
def update_todo(todo_id):
    if todo_id not in todos:
        return jsonify({"error": "Todo not found"}), 404
    data = request.get_json()
    todos[todo_id] = data["task"]
    return jsonify({todo_id: todos[todo_id]})
```

Flask CRUD API 만들기

[실습 3] CRUD <-> HTTP 메서드 표에서 아래 API 구현하기

- [DELETE] /todos/<int:todo_id>

```
@app.route("/todos/<int:todo_id>", methods=["DELETE"])
def delete_todo(todo_id):
    if todo_id not in todos:
        return jsonify({"error": "Todo not found"}), 404
    deleted = todos.pop(todo_id)
    return jsonify({"deleted": deleted})
```

데이터베이스 기초

데이터베이스 저장 방식 비교

- 파일 저장: txt, csv
- 메모리 (dict, list): 휘발성, 서버 꺼지면 다 사라짐
- 데이터베이스 (DB): 영구저장 + 동시성 + 무결성 보장

👉 **따라서 대부분의 서비스는 DB를 사용!**

데이터베이스의 중요성

- 데이터 저장:
  - 사용자 정보, 게시글, 주문 내역 등 다양한 데이터를 파일 대신 체계적으로 저장
  - 데이터를 **영구적으로 보존**하며, 필요 시 언제든지 접근 가능

데이터베이스의 중요성

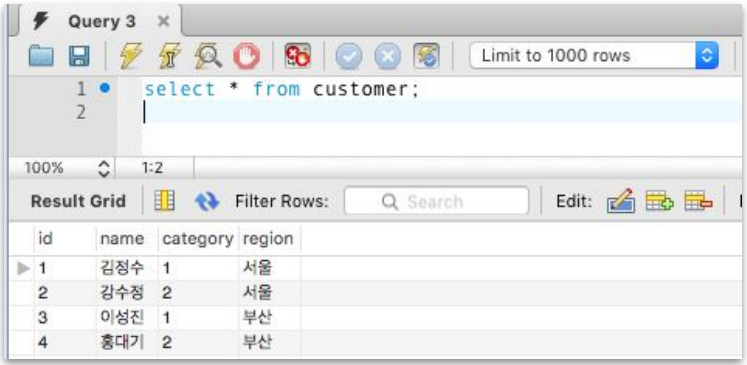
- 데이터 관리:
  - 구조화된 형태: DB는 데이터를 테이블 형태로 저장하여 관리
  - 검색 및 정렬: SQL 쿼리를 통해 데이터를 검색하거나 정렬 가능
  - 효율성: 인덱스(Index) 및 쿼리 최적화를 통해 대량 데이터도 빠르게 검색 가능

데이터베이스의 중요성

- 데이터 무결성:
  - 정확성 보장: 제약조건(Constraints)을 통해 잘못된 데이터 입력 방지
    - UNIQUE: 이메일 필드가 중복되지 않도록 설정
    - NOT NULL: 특정 필드는 비워둘 수 없도록 설정
    - FOREIGN KEY: 사용자와 게시글의 관계 유지
  - 트랜잭션(Transaction)
    - 데이터 일관성을 보장하기 위해 작업 단위를 묶어서 처리
    - ex) 주문이 완료되었을 때만 재고를 줄이고 & 결제 정보 저장

데이터베이스의 종류

- 관계형 데이터베이스 (**R**elational **D**ata**B**ase)
  - 표(table) 기반 구조
  - SQL 사용
  - 예: MySQL, PostgreSQL, SQLite 등
  - 장점: 구조적, 무결성 강함



데이터베이스의 종류

- NoSQL 데이터베이스
  - Key-value, document, graph 등 다양한 형태
  - 예시: MongoDB, Redis
  - 장점: 유연하고 대규모 분산에 강함

데이터베이스의 종류

- In-memory 데이터베이스
  - RAM 기반, 매우 빠름
  - 예시: Redis, Memcached, 변수로 핸들링하는 데이터 (2일차 to-do)
  - 단점: 전원 끄면 사라짐

수업에서 사용할 데이터베이스

- **SQLite**
  - 경량 SQL DBMS
  - 별도의 서버 없이 단일 파일 (.db)로 저장되는 내장형 데이터베이스
  - DB 서버를 설치할 필요 없이 바로 사용 가능

  - 동시성 & 대규모 데이터 처리에는 약함: 이 때는 PostgreSQL, MySQL 등의 전문 DB 서버를 사용할 것을 추천
    - 스키마 변경(마이그레이션) 쪽으로 alembic을 많이 사용함

수업에서 사용할 데이터베이스

SQLite 사용법

- 상대경로 (프로젝트 폴더 안에서 생성)
  - 현재 실행한 python 파일이 있는 위치에 db 생성
  - 가장 흔하게 쓰는 방식 - 간단하기 때문
  - 단점: 실행 위치가 바뀌면 db가 안보일 수 있음

```
engine = create_engine("sqlite:///users.db")
```

수업에서 사용할 데이터베이스

SQLite 사용법

- 하위 디렉토리 지정
  - instance/ 폴더 안에 db 파일 생성
  - **Flask 권장 패턴**
  - 장점: 코드와 데이터 파일이 분리되어 깔끔함

```
engine = create_engine("sqlite:///instance/users.db")
```

수업에서 사용할 데이터베이스

SQLite 사용법

```
engine = create_engine("sqlite:////Users/oz/workspace/db/users.db")
```

- 절대 경로 지정
  - 특정 경로에 db 파일 저장
  - 슬래시 4개 //// 필요
  - 장점: 경로가 확실히 고정됨
  - 단점: 다른 환경 (ex. Windows <-> Mac / Linux)에서 경로가 달라져서 공유하기 어려움

수업에서 사용할 데이터베이스

SQLite 사용법

- 메모리 db
  - 메모리에만 저장: 프로그램 종료 시 데이터 사라짐
    - 때문에 테스트용으로 많이 사용
  - 장점: 빠르고 테스트에 적합
  - 단점: 영구 저장 불가

```
engine = create_engine("sqlite:///:memory:")
```

SQLAlchemy ORM

SQL과 ORM (Object-Relational Mapping)

SQL (Structured Query Language)

- SQL이란?
  - 관계형 데이터베이스와 상호작용하기 위한 언어
  - 데이터를 삽입(INSERT), 조회(SELECT), 수정(UPDATE), 삭제(DELETE)하는 명령 제공

SQL과 ORM (Object-Relational Mapping)

SQL (Structured Query Language)

- SQL 사용의 장점
  - 데이터베이스 제어가 명확하고 강력함
  - 복잡한 데이터 조작 및 분석을 직접 수행 가능
- SQL의 단점
  - 애플리케이션 코드와 분리된 언어로 작성 필요
  - 데이터베이스별 SQL 문법에 약간의 차이 존재
  - SQL 코드가 많아질수록 유지보수가 어려워짐

SQL과 ORM (Object-Relational Mapping)

ORM (Object-Relational Mapping)

- ORM이란?
  - 관계형 데이터베이스의 테이블을 객체지향 프로그래밍의 객체로 매핑하는 기술
  - SQL 없이도 데이터 조작 가능
  - Python에서는 SQLAlchemy, Django ORM, Tortoise ORM 등이 널리 사용됨

SQL과 ORM (Object-Relational Mapping)

ORM (Object-Relational Mapping)

- ORM의 장점
  - 개발자 친화적
    - SQL 대신 Python 코드로 데이터 처리 가능
    - 테이블 변경 시 코드를 수정하면 쿼리와 매핑이 자동으로 처리됨
  - 데이터베이스 독립성: 코드 수정 없이 SQL 전환 가능 (ex. MySQL -> PostgreSQL)
  - 안정성: SQL Injection 같은 보안 취약점을 자동으로 방지

SQL과 ORM (Object-Relational Mapping)

ORM (Object-Relational Mapping)

- ORM의 단점
  - 복잡한 쿼리는 직접 SQL 작성 필요
  - ORM 레이어로 인해 성능이 약간 저하될 수 있음

SQL과 ORM (Object-Relational Mapping)

SQL과 ORM 비교

SQL	ORM
SQL 쿼리 직접 작성 필요	Python 객체로 데이터 조작
데이터베이스에 최적화된 성능	유지보수 및 확장성이 높음
데이터베이스별 SQL 문법 차이 고려 필요	데이터베이스 독립적 코드 작성 가능

SQLAlchemy 소개

- Python에서 가장 널리 사용되는 ORM 및 데이터베이스 툴킷
- 관계형 데이터베이스와 상호작용하기 위한 강력하고 유연한 도구 제공
- 단순 데이터베이스 연동, 복잡한 쿼리 작성 등 다양한 요구를 충족할 수 있도록 설계됨

SQLAlchemy 소개

SQLAlchemy의 특징

- 유연성과 강력함
  - 단순한 데이터 조작부터 복잡한 쿼리 작성까지 다양한 기능 제공
  - 관계형 데이터베이스의 강점을 최대한 활용 가능
- 데이터베이스 독립성
  - SQLite, PostgreSQL, MySQL, Oracle 등 대부분의 데이터베이스를 지원
  - 데이터베이스를 변경해도 코드 수정이 최소화됨

SQLAlchemy 소개

SQLAlchemy의 특징

- 고성능과 확장성
  - SQLAlchemy Core를 사용하면 SQL 쿼리를 직접 작성 가능
  - 대규모 프로젝트에서도 안정적으로 동작
- 사용자 친화적: Pythonic한 문법으로 직관적이고 간결한 코드 작성 가능

SQLAlchemy 소개

SQLAlchemy의 구성 요소

- **Engine: DB 연결 객체**
  - 데이터베이스와의 연결을 관리하는 핵심 요소
  - 데이터베이스 URL을 통해 연결 설정

```python
from sqlalchemy import create_engine

# SQLite 로컬 DB 연결
engine = create_engine("sqlite:///todos.db", echo=True)
```

SQLAlchemy 소개

SQLAlchemy의 구성 요소

- **Session: DB 작업을 실행하는 통로**
  - DB 연결을 통해 실제 CRUD 실행

```python
from sqlalchemy.orm import sessionmaker

# 세션 팩토리 생성
SessionLocal = sessionmaker(bind=engine)

# 세션 열기
db = SessionLocal()

# 세션으로 작업 → 나중에 닫기
todos = db.query(Todo).all()
db.close()
```

SQLAlchemy 소개

SQLAlchemy의 구성 요소

- **Base**
  - 테이블 정의를 위한 클래스 기반
  - 모든 테이블 클래스는 Base를 상속받아 정의

```
from sqlalchemy.ext.declarative import declarative_base
Base = declarative_base()
```

SQLAlchemy 소개

SQLAlchemy의 구성 요소

- **Model**
  - Python 클래스 형태로 테이블 구조를 정의
  - 테이블 컬럼은 클래스의 속성으로 매핑

```python
from sqlalchemy import Column, Integer, String

class User(Base):
    __tablename__ = "users"
    id = Column(Integer, primary_key=True)
    name = Column(String)
```

column	역할	데이터 형식	PK
id	고유 id	INT	True
name	이름	STRING	False

SQLAlchemy 소개

- SQLAlchemy 주요 데이터 타입

타입	Integer	String	Float	Boolean	Date, DateTime
설명	정수형 데이터	문자열 데이터	부동 소수점 데이터	참, 거짓	날짜 및 시간

```python
from sqlalchemy import Column, Integer, String, Float, Boolean, DateTime
from datetime import datetime

class Product(Base):
    __tablename__ = "products"
    id = Column(Integer, primary_key=True, index=True)
    name = Column(String, index=True)
    price = Column(Float, nullable=False)
    is_active = Column(Boolean, default=True)
    created_at = Column(DateTime, default=datetime.utcnow)
```

SQLAlchemy 소개

- **SQLAlchemy 기본 문법**
  - 데이터베이스 생성

```python
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

DATABASE_URL = "sqlite:///./test.db"
engine = create_engine(DATABASE_URL, echo=True)
Base = declarative_base()
SessionLocal = sessionmaker(bind=engine)

# 테이블 생성
Base.metadata.create_all(bind=engine)
```

SQLAlchemy 소개

- SQLAlchemy 기본 문법
  - 테이블 생성

```
from sqlalchemy import Column, Integer, String

class User(Base):
    __tablename__ = "users"
    id = Column(Integer, primary_key=True, index=True)
    name = Column(String, index=True)
    email = Column(String, unique=True, index=True)
```

```
Base.metadata.create_all(bind=engine)
```

SQLAlchemy 소개

- SQLAlchemy 기본 문법
  - 데이터 삽입

```python
from sqlalchemy import Column, Integer, String

class User(Base):
    __tablename__ = "users"
    id = Column(Integer, primary_key=True, index=True)
    name = Column(String, index=True)
    email = Column(String, unique=True, index=True)

# 세션 시작
session = SessionLocal()

# 데이터 삽입
new_user = User(name="John Doe", email="john.doe@example.com")
session.add(new_user)
session.commit()
session.close()
```

SQLAlchemy 소개

- **SQLAlchemy 기본 문법**
  - 데이터베이스 조회

전체 데이터 조회

```
users = session.query(User).all()
for user in users:
    print(user.id, user.name, user.email)
```

조건에 따른 데이터 조회

```
user = session.query(User).filter(User.name == "John Doe").first()
print(user.id, user.name, user.email)
```

SQLAlchemy 소개

- **SQLAlchemy 기본 문법**
  - 데이터베이스 수정

```
user = session.query(User).filter(User.name == "John Doe").first()
if user:
    user.email = "new.email@example.com"
    session.commit()
```

SQLAlchemy 소개

- SQLAlchemy 기본 문법
  - 데이터베이스 삭제

```
user = session.query(User).filter(User.name == "John Doe").first()
if user:
    session.delete(user)
    session.commit()
```

점심 & 쉬는 시간

DB 세팅 및 기초 실습

DB 세팅 및 기초 실습

[실습 1] SQLite + SQLAlchemy 설정 및 연습해보기

실시간으로 진행합니다.

API + DB 연결:
To-Do API 2차 버전

Flask와 DB 연동

[실습 2] 2일차 실습 5 코드에 SQLite 세팅하기

실시간으로 진행합니다.

Flask와 DB 연동

[실습 2] 2일차 실습 5 코드에 SQLite 세팅하기 + α

```
db = SessionLocal()
# ... 쿼리 실행 ...
db.close()
```

- 장점: 직관적이고 간단함
- 단점
  - 라우트마다 반복됨
  - return전에 예외가 나면 db.close() 실행이 안될 수 있음 -> 세션 누수 위험

Flask와 DB 연동

[실습 2] 2일차 실습 5 코드에 SQLite 세팅하기 + α

- 개선방법 1: Context Manager (with 구문)

```python
from contextlib import contextmanager

@contextmanager
def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()

@app.route("/todos", methods=["GET"])
def get_todos():
    with get_db() as db:
        todos = db.query(Todo).all()
        return jsonify([{"id": t.id, "task": t.task} for t in todos])
```

-> 예외가 발생해도 finally에서 무조건 db.close() 실행됨

Flask와 DB 연동

[실습 2] 2일차 실습 5 코드에 SQLite 세팅하기 + α

- 개선방법 2: Flask의 before_request / teardown_appcontext 활용

```python
from flask import g

@app.before_request
def before_request():
    g.db = SessionLocal()

@app.teardown_appcontext
def shutdown_session(exception=None):
    db = g.pop("db", None)
    if db is not None:
        db.close()

@app.route("/todos", methods=["GET"])
def get_todos():
    todos = g.db.query(Todo).all()
    return jsonify([{"id": t.id, "task": t.task} for t in todos])
```

-> 각 request마다 자동으로 세션 열고,
요청 끝날 때 자동으로 닫음

Flask와 DB 연동

구조화의 필요성

- 지금까지 모든 코드를 하나의 python파일에 몰아넣었음
- 작은 앱은 문제가 없지만, 기능이 특히 API가 많아지면?
- **관련 있는 기능들끼리 묶어 모듈화(그룹핑) 하자**

모듈화 소개

- Blueprint: 라우트 묶음

```python
from flask import Flask, jsonify

app = Flask(__name__)

@app.route("/todos")
def get_todos():
    return jsonify({"message": "Get Todos"})

@app.route("/users")
def get_users():
    return jsonify({"message": "Get Users"})

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

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

# 1. Blueprint 객체 생성
todo_bp = Blueprint("todo", __name__)
user_bp = Blueprint("user", __name__)

# 2. Blueprint에 라우트 등록
@todo_bp.route("/todos")
def get_todos():
    return jsonify({"message": "Get Todos"})

@user_bp.route("/users")
def get_users():
    return jsonify({"message": "Get Users"})

# 3. Flask 앱에 등록
app = Flask(__name__)
app.register_blueprint(todo_bp)   # /todos
app.register_blueprint(user_bp)   # /users

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

모듈화 소개

- **프로젝트 구조 예시**
  - **app/ 폴더**: 실제 애플리케이션 코드
    - routes, models
  - **instance/ 폴더**: DB, 설정 파일 등 실행환경별로 다른 것
  - (**templates/ 폴더**: HTML 템플릿)

```
project/
| app.py
|
├── app/
|   ├── __init__.py
|   ├── models.py
|   └── routes.py
|
└── instance/
    └── todos.db
```

모듈화 소개

[실습 3] 오른쪽의 구조에 맞춰서 실습 2번 코드 구조화(모듈화) 하기

실시간으로 진행합니다.

QnA