

실전! FastAPI 입문 강의 정리

섹션 1. 오리엔테이션

1. 강좌 소개 및 강사 소개

(프론트엔드 Jinja: Jinja는 Python 프로그래밍 언어를 위한 웹 템플릿 엔진입니다.)

(Django(장고): Django는 Python의 오픈 소스 웹 프레임워크이자 풀 스택 프레임워크이다.)

비동기 처리는 다루지 않는다.

www.qu3vipon.com

github.com/qu3vipon

python 3.10 기준, Docker: 기본적인 사용법 익혀야 함, PyCharm(VSCode도 괜찮음)

3. Python 가상 환경 구축

py -3.10 -m venv 가상환경

가상환경\Scripts\activate (가상 환경 활성화)

deactivate (가상 환경 비활성화)

(fastapi) C:\backend>which python

'which'은(는) 내부 또는 외부 명령, 실행할 수 있는 프로그램, 또는 배치 파일이 아닙니다.

4. Docker 설치

5. Pycharm 설치

가상환경 디렉토리를 open(프로젝트 디렉토리가 가상환경 폴더이다.)

섹션 2. FastAPI 알아보기

1. FastAPI 소개

섹션 1. FastAPI 알아보기

FastAPI란?

[GitHub Star History](#)

최근(2018년 말)에 만들어진 파이썬 웹 프레임워크로 단기간 내에 많은 인기
<https://star-history.com/#tiangolo/fastapi&Date>

사용하는 기업

MicroSoft, Uber, Netflix와 같은 글로벌 테크 기업에서 FastAPI 사용
<https://fastapi.tiangolo.com/#opinions>

비교

	<Django+DRF>	<FastAPI>
기능	Battery Included	경량 Framework
확장성	제한적	자유로움
성능		Django 보다 우수

실전! FastAPI 입문

Django: DRF(Django REST Framework)와 함께 사용.

MVC 패턴? 내장된 시리얼라이저? data validation?

프레임워크에 강하게 결합 -> 다른 프레임워크로 변환 or 다른 디자인 패턴을 적용하기에 제한적

Admin과 같이 기본적으로 제공되는 기능 활용 -> 빠르게 프로토타이핑하는 프로젝트에 적합

vs

FastAPI: 개발자 요구에 맞는 디자인 패턴 적용 용이, 비동기 처리 프레임워크 자체적 지원(장고보다 성능 우수), 프레임워크 자체가 가벼움(장고보다 성능 우수)

기능 단순한 가벼운 웹서버, 직접 디자인 패턴 적용해서 확장 가능한 어플리케이션 만들 때 사용하기 좋음

2. FastAPI 장점

섹션 1. FastAPI 알아보기

FastAPI 장점

성능

직관적인 디자인

path, query, body, response, dependency

Type Hints

Pydantic

Data validation

IDE Support

비동기처리

AsyncIO, Background Task

자동 API 문서 생성

OpenAPI, SwaggerUI

실전! FastAPI 입문

3. API

섹션 1. FastAPI 알아보기

API

API

Application Programming Interface

Interface 예시

리모컨

기능을 외부에서 사용할 수 있게 제공
사용자 -> 버튼(물리 신호) -> 전기 신호 -> 장비

에어컨

<내부>

_실내온도 감지

_실외기 조작

<외부>

전원 On/Off

온도 조절

바람 세기 조절

실전! FastAPI 입문

서버의 기능을 클라이언트가 사용할 수 있도록 제공하는 인터페이스.

컴퓨터 프로그래밍에서 **인터페이스**는 두 시스템 간의 신호를 주고받는 접점을 의미함.(리모컨: 내부 기능은 굳이 사용자에게 노출하지 않음, 사용자에게 필요한 기능은 사용하기 편한 형태로 제공)

4. REST API

섹션 1. FastAPI 알아보기

RESTful API

REST

Representational State Transfer

Resource

URL를 통해 고유한 리소스 표현

예시 /api/v1/posts

/api/v1/posts/123/comments

Method

HTTP Method 통해 API의 동작 표현

예시 GET /api/v1/posts

-> posts 조회

POST /api/v1/posts

-> post 생성

PATCH /api/v1/posts/123

-> post 수정

DELETE /api/v1/posts/123

-> post 삭제

실전! FastAPI 입문

API 설계하는 스타일 가이드 중 하나.

서버에서 관리하는 데이터의 상태가 표현되는 디자인 이라는 의미

REST 아키텍처에서 리소스(예: 사용자, 게시물 등)는 고유한 URI(보통 URL 형태)로 식별됩니다. 클라이언트는 이 URL을 통해 특정 리소스를 지정하고 HTTP 메서드로 원하는 작업을 수행합니다.

5. ToDo 서비스 만들기

섹션 1. FastAPI 알아보기

ToDo 서비스 만들기

GET

전체 ToDo 조회: /api/v1/todos

단일 ToDo 조회: /api/v1/todos/<id>

POST

ToDo 생성: /api/v1/todos

PATCH

ToDo 수정: /api/v1/todos/<id>

DELETE

ToDo 삭제: /api/v1/todos/<id>

실전! FastAPI 입문

할 일을 적고 체크하는 기능

5가지 API 만들기

pip install fastapi == 0.97.0

6. (실습) FastAPI 프로젝트 세팅

src = source (src code)

main.py에 FastAPI 관련 코드들을 적을 것임

src > Mark Directory as > Sources Root : 파이썬 파일들을 임포트하는 경로에서 이 소스 디렉토리를 루트 패스로 잡는다.

app = FastAPI() : 앱이라는 이름으로 FastAPI를 instantiate 해준다.

app이라는 객체에 api를 연결해서 서버를 띄우게 되면 http 요청을 보낼 수 있게 된다.

```
@app.get("/")
```

```
def health_check_handler():
```

```
    return {"ping": "pong"}
```

: FastAPI서버에 루트 패스로 get 요청을 보내면, 아래의 함수가 실행을 해서 그 api 요청을 처리한다

```
cd src
```

```
uvicorn main:app
```

main.py의 app이라는 이 객체를 이용할 거다. 식의 맵핑을 해주는 것

.../docs: swagger fastapi가 자동으로 문서 생성

7. (실습) GET API - 전체 조회

일반적으로 REST API에서는 자원을 복수형으로 적는다

<http://127.0.0.1:8000/docs>

```
curl -X 'GET' \
  'http://127.0.0.1:8000/todos' \
  -H 'accept: application/json'
```

query parameter: get 요청 뒤에 path 가장 뒤에 물음표 형태로, 예를 들어 order 이런 식으로 붙여서 desc 이렇게 써주게 되면은 ...

```
@app.get("/todos?order=DESC")
```

조회 시에 추가적인 정렬을 한다거나, (DESC: 디센딩)

```
@app.get("/todos?query=apple")
```

apple이 들어간 단어, apple이 들어간 아이템을 찾는거나 ...

추가적인 작업 가능

but FastAPI에서는 직접 적어주지는 않고 함수에 인자로 전달 해줄 수 있다.

list indexing: 리스트변수[시작인덱스:종료인덱스:step], step은 생략 가능

```
s = 'show how to index into sequences'.split() >>> s ['show', 'how', 'to', 'index', 'into', 'sequences']
s[1:-1] ['how', 'to', 'index', 'into']
```

8. (실습) GET API - 단일 조회

```
@app.get("/todos/{todo_id}")
```

서브path에 {todo_id}로 적어주면(중괄호), path로 이용 가능.

```
def get_todo_handler(todo_id: int):
```

이렇게 path를 인자로 전달받아서 사용할 수가 있다.

```
dictionary.get(todo_id, {})
```

, {}은 기본값

--reload 옵션: 코드 변화가 감지되면, 자동으로 서버를 FastAPI가 재시작 한다. 새로고침

9. (실습) POST API - 생성

todo를 생성하기 위해서는 사용자로부터 데이터를 전달받아야 한다.

request body에 대한 처리가 필요.

FastAPI에서는 Pydantic의 BaseModel을 이용해서 쉽게 request body를 처리할 수 있다.

```
class CreateToDoRequest(BaseModel):
```

BaseModel을 상속받는 클래스를 하나 만들어줌, 이 클래스에 이 request body가 어떤 형태인지를 표현해 줘야 함.

Todo 아이템들이 갖는 형태와 동일한 스키마로 이 리퀘스트를 만들어 줄 것임.

```
id: int
```

contents: str

is_done: bool

지금은 일단 유저가 입력하는 아이디를 그대로 그냥 사용하는 구조

이제 이 리퀘스트를 사용하는 방법이 함수에 request라는 인자를 하나 만들어서.

dict(): BaseModel에 구현이 되어 있는 메소드

response에 대한 type hint 제공 해야...(뒤에 다룸)

10. (실습) PATCH API - 수정

reload 반영 안될 시 일단 재부팅으로 해결했음

안녕하세요, 인프런 AI 인턴입니다.

이야기하신 uvicorn 서버 종료 문제는 다양한 이유로 발생할 수 있습니다. 일반적으로 `Ctrl + C` 를 눌러 서버를 종료할 수 있지만, 반응이 없는 경우도 있습니다. 몇 가지 해결책을 제안드립니다:

1. **강제 종료:** 만일 `Ctrl + C` 가 작동하지 않으면 터미널 창에서 `Ctrl + Break` 를 시도해보세요. 또는, `Ctrl + Z` 로 프로세스를 백그라운드로 보내고 `kill` 명령어를 사용하여 프로세스를 종료할 수도 있습니다. 예를 들어, `jobs` 명령어로 실행 중인 작업을 확인한 다음, `kill %1` 과 같이 적어 해당 작업을 종료할 수 있습니다.
2. **Task Manager/Activity Monitor 사용:** Windows에서는 작업 관리자를, macOS에서는 활동 모니터를 통해 uvicorn 프로세스를 직접 종료할 수 있습니다.
3. **업데이트 문제가 있을 수 있음:** uvicorn과 관련 패키지가 최신인지 여부를 확인하시고, 필요시 업데이트 해보세요.
4. **서버의 로그인 세션과 관련된 문제:** 특정한 개발 환경 설정에 의해서 발생할 수도 있습니다. 해당 환경을 점검해 보시는 것도 좋습니다.
DB 처리 안하면 그냥 새로 코드 실행 시 마다 항상 코드에 있는 `todo_data`만 남는다.

11. (실습) DELETE API - 삭제

12. HTTP Status Code

섹션 1. FastAPI 알아보기

Status Code

2xx

200 OK	요청 성공, 범용적, GET/POST/PUT/PATCH
201 Created	요청 성공, 새로운 자원 생성, POST
204 No Content	요청 성공, 응답할 자원 없음, DELETE

4xx

400 Bad Request	요청 실패, 요청이 잘못된 경우(query param, body)
401 Unauthorized	인증 실패
403 Forbidden	권한 문제 또는 잘못된 method
404 Not Found	자원이 없는 경우 또는 잘못된 endpoint

5xx

500 Internal Server Error	범용적인 서버 에러
502 Bad Gateway	Reverse Proxy에서 서버의 응답을 처리할 수 없는 경우
503 Service Unavailable	서버가 요청을 처리할 수 없는 경우(e.g. 일시적 부하, 서버 다운)

실전! FastAPI 입문

필요에 따라 검색해보며 사용, but 위 내용은 알아두자.

13. (실습) Error 처리

명시적으로 적어주기

```
@app.get("/todos", status_code=200)
```

예외처리

```
& raise HTTPException(status_code=404, detail="ToDo Not Found")
```

섹션 3. 데이터베이스

1. 데이터베이스란?

섹션 2. 데이터베이스

데이터베이스란?

데이터베이스

대량의 데이터를 영구적으로 저장/관리하기 위한 시스템

분류

- 관계형 데이터베이스(Relational database, RDB)
관계형 모델에 기반해서 데이터를 테이블, 행, 열 구조로 관리(=Schemaful)
Oracle, MySQL, PostgreSQL, Sqlite, ...
- 비관계형 데이터베이스(NoSQL)
 - Key-value Redis, etcd
 - Document MongoDB
 - Wide-column Cassandra, ScyllaDB
 - Timeseries Apache Druid, InfluxDB
 - Graph Neo4j

실전! FastAPI 입문

RDB: 엑셀 생각해라(데이터 일반성 유지 쉬워서 일반적으로 많이 사용됨)

NoSQL: 특정 목적에 맞게 골라서, 관계형DB와 조합해 사용

우리는 MySQL 사용 (오픈소스)

2. sqlalchemy 소개

FastAPI에서 DB 사용하려면, SQLAlchemy 라는 Python 라이브러리 필요

섹션 2. 데이터베이스

sqlalchemy 소개

sqlalchemy

관계형 데이터베이스를 사용하기 위한 High-level 인터페이스를 제공하는 Python 라이브러리
ORM, Query, Transaction, Connection Pooling

ORM(Object-Relational Mapping)

관계형 데이터베이스를 객체 지향 프로그래밍(OOP)에 대응하여 사용하는 프로그래밍 기술

하나의 테이블 = 하나의 클래스
하나의 행(레코드) = 하나의 객체

<데이터베이스>	<Python>
id username	user = User(id=1, username='qu3vipon')
1 qu3vipon	

실전! FastAPI 입문

ORM-> 파이썬으로 SQL 언어 사용하지 않고 작업 가능

Object-Relational Mapping: SQL과 같은 관계형 DB를 OOP에 대응하여 사용한다.

3. (실습) MySQL 컨테이너 실행

Docker 명령어

- `docker run -p 3306:3306 -e MYSQL_ROOT_PASSWORD=todos -e MYSQL_DATABASE=todos -d -v todos:/db --name todos mysql:8.0`
- `docker ps`: 동작하는 컨테이너 정보 확인
- `docker logs todos`: todos 컨테이너의 로그 확인
- `docker volume ls`: 도커 볼륨 생성한 것 확인

MySQL 접속

- `docker exec -it todos bash`
- `mysql -u root -p`

SQL

- `SHOW databases;`
- `USE todos;`
- `CREATE TABLE todo(`
`id INT NOT NULL AUTO_INCREMENT,`
`contents VARCHAR(256) NOT NULL,`
`is_done BOOLEAN NOT NULL,`
`PRIMARY KEY (id)`
`);`
- `INSERT INTO todo (contents, is_done) VALUES ("FastAPI Section 0", true);`
- `SELECT * FROM todo;`

`docker run -p 3306:3306 -e MYSQL_ROOT_PASSWORD=todos -e MYSQL_DATABASE=todos -d -v todos:/db --name todos mysql:8.0`

so many options

- `run`: 도커 컨테이너를 동작시키는 명령어
- `-p 3306:3306`: 포트 매핑, MySQL은 기본적으로 3306번 포트 이용, 로컬 호스트 3306번 - 도커 컨테이너 안의 3306번 포트가 연결됨
- `-e MYSQL_ROOT_PASSWORD=todos`: 환경변수 옵션, 기본 루트 계정을 이용할 것이고 그 것의 패스워드 지정
- `-e MYSQL_DATABASE=todos`: MySQL DATABASE의 환경변수를 지정해달라는 옵션, 도커 컨테이너 동작 시 todos라는 DB를 기본적으로 하나 생성해달라는 명령어
- `-d`: detach 옵션, 컨테이너가 백그라운드에서 동작하게 하도록 하는 옵션
- `-v todos:/db`: volume 옵션, 기본적으로 도커 컨테이너는 종료 시 데이터가 모두 삭제됨, 실습 시 컨테이너를 내렸다 올렸다 할 수 있기 때문에 todos 이름의 volume을 생성해서 로컬 디바이스에 데이터 남겨서 실습 원활히 진행할 수 있도록
- `--name todos`: 동작시킬 도커 컨테이너의 이름을 지정해주는 옵션
- `mysql:8.0`: 컨테이너 동작시킬 때 사용할 도커 이미지 지정

4. (실습) DDL - Create Table

Data Definition Language

```
C:\backend>docker exec -it todos bash
bash-5.1# mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 8
Server version: 8.0.42 MySQL Community Server - GPL

Copyright (c) 2000, 2025, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show databases;
+-----+
| Database |
+-----+
| information_schema |
| mysql |
| performance_schema |
| sys |
| todos |
+-----+
5 rows in set (0.01 sec)
```

```
mysql> use todos;
Database changed
mysql> CREATE TABLE todo(
  ->     id INT NOT NULL AUTO_INCREMENT,
  ->     contents VARCHAR(256) NOT NULL,
  ->     is_done BOOLEAN NOT NULL,
  ->     PRIMARY KEY (id)
  -> );
Query OK, 0 rows affected (0.05 sec)
```

```
mysql> insert into todo (contents, is_done) values ("FastAPI Section 0", true);
Query OK, 1 row affected (0.00 sec)
```

```
mysql> insert into todo (contents, is_done) values ("FastAPI Section 1", true);
Query OK, 1 row affected (0.00 sec)
```

```
mysql> insert into todo (contents, is_done) values ("FastAPI Section 2", false)
;
Query OK, 1 row affected (0.01 sec)
```

```
mysql> select * from todo;
+----+-----+-----+
| id | contents          | is_done |
+----+-----+-----+
| 1  | FastAPI Section 0 | 1       |
| 2  | FastAPI Section 1 | 1       |
| 3  | FastAPI Section 2 | 0       |
+----+-----+-----+
3 rows in set (0.00 sec)
```

5. (실습) 데이터베이스 연결

sqlalchemy를 이용해 생성한 DB와 python 프로젝트를 연결하는 실습

pip install sqlalchemy, pip install pymysql, pip install cryptography

sqlalchemy:

pymysql: python과 mysql을 연동할 때 사용되는 드라이버

cryptography: pymysql 통해 mysql에 접속 시 인증이나 암호 관련 처리해주는 라이브러리

mysql+pymysql://root:todos@127.0.0.1:3306/todos

root:todos 비번

```
engine = create_engine(DATABASE_URL, echo=True)
```

SQLAlchemy로 DB 접속 위해서는 먼저 엔진이라는 개체를 만들어줘야 한다

echo=True: sqlalchemy에 의해 대신 처리되는 쿼리에 어떤 sql이 사용됐는지 사용되는 시점에 사용되는 sql을 출력해주는 옵션(실제 프로덕션 X, 개발 환경 디버깅 시)

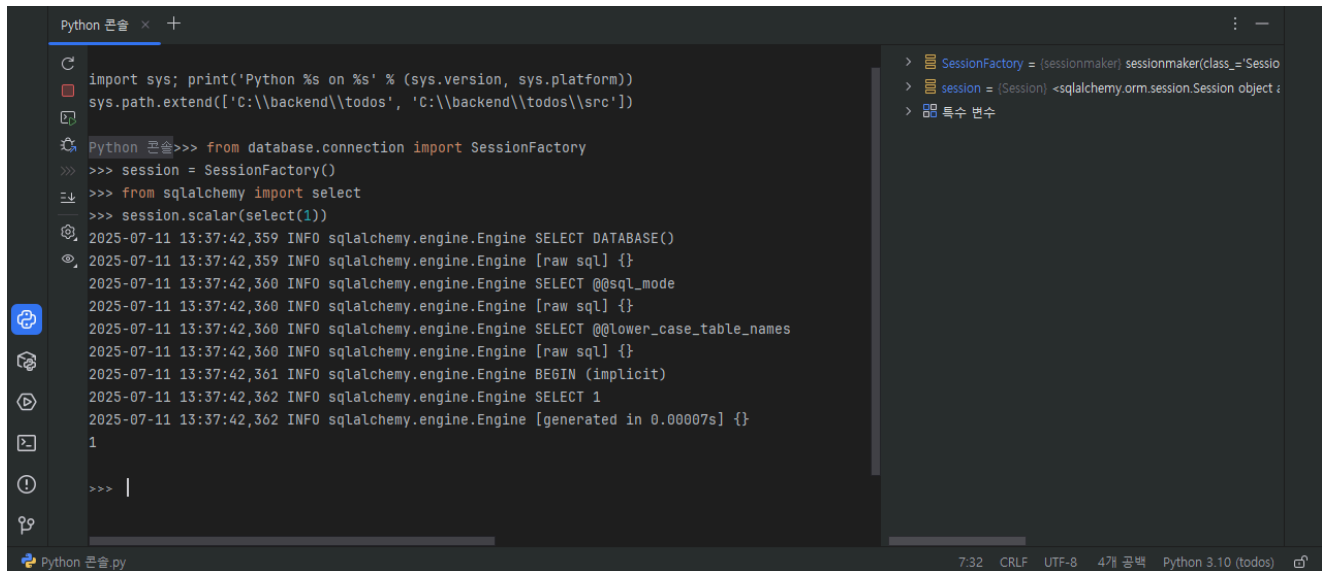
```
sessionFactory = sessionmaker(autocommit=False, autoflush=False, bind=engine)
```

session을 생성해서 인스턴스를 통해 데이터베이스와 통신

커밋, 플러쉬 동작(연산)을 자동으로 수행시키지 않고 명시적으로 하겠다.

bind로 앞서 생성한 engine 연결, 전달한 DB URL을 통해 엔진이라는 객체가 생성됨.

그 엔진을 사용해 실제로 session을 만들 수 있게 된다.



```
Python 콘솔 x +
import sys; print('Python %s on %s' % (sys.version, sys.platform))
sys.path.append('C:\\backend\\todos', 'C:\\backend\\todos\\src')

Python 콘솔>>> from database.connection import SessionFactory
>>> session = SessionFactory()
>>> from sqlalchemy import select
>>> session.scalar(select(1))

2025-07-11 13:37:42,359 INFO sqlalchemy.engine.Engine SELECT DATABASE()
2025-07-11 13:37:42,359 INFO sqlalchemy.engine.Engine [raw sql] {}
2025-07-11 13:37:42,360 INFO sqlalchemy.engine.Engine SELECT @@sql_mode
2025-07-11 13:37:42,360 INFO sqlalchemy.engine.Engine [raw sql] {}
2025-07-11 13:37:42,360 INFO sqlalchemy.engine.Engine SELECT @@lower_case_table_names
2025-07-11 13:37:42,360 INFO sqlalchemy.engine.Engine [raw sql] {}
2025-07-11 13:37:42,361 INFO sqlalchemy.engine.Engine BEGIN (implicit)
2025-07-11 13:37:42,362 INFO sqlalchemy.engine.Engine SELECT 1
2025-07-11 13:37:42,362 INFO sqlalchemy.engine.Engine [generated in 0.00007s] {}
1

>>> |
```

Python 콘솔.py 7:32 CRLF UTF-8 47% 공백 Python 3.10 (todos)

6. (실습) ORM 모델링

```
프로젝트 > todos library 루트, C:\backend\wtod
> Include
> Lib
> Scripts
> src
  > database
    connection.py
    orm.py
  main.py
  pyenv.cfg
> 외부 라이브러리
스크래치 및 콘솔

main.py connection.py orm.py x
1 from sqlalchemy import Boolean, Column, Integer, String
2 from sqlalchemy.orm import declarative_base
3
4 Base = declarative_base()
5
6 class Todo(Base):
7     __tablename__ = "todo"
8
9     id = Column(Integer, primary_key=True, index=True)
10    contents = Column(String(256), nullable=False)
11    is_done = Column(Boolean, nullable=False)
12
13    def __repr__(self):
14        return f'Todo(id={self.id}, contents={self.contents}, is_done={self.is_done})'
```

```
Python 콘솔>>> from database.connection import SessionFactory
>>> session = SessionFactory()
>>> from sqlalchemy import select
>>> from database.orm import Todo
>>> session.scalars(select(Todo))
```

```
>>> todos = list(session.scalars(select(Todo)))
2025-07-11 18:20:12,313 INFO sqlalchemy.engine.Engine SELECT todo.id, todo.contents, todo.is_done
FROM todo
2025-07-11 18:20:12,313 INFO sqlalchemy.engine.Engine [cached since 142.2s ago] {}
>>> todos
[Todo(id=1, contents=FastAPI Section 0, is_done=True), Todo(id=2, contents=FastAPI Section 1, is_done=True), Todo(id=3, conte
>>> for todo in todos:
...     print(todo)
...
Todo(id=1, contents=FastAPI Section 0, is_done=True)
Todo(id=2, contents=FastAPI Section 1, is_done=True)
Todo(id=3, contents=FastAPI Section 2, is_done=False)
>>>
```

ORM: 데이터베이스 테이블과 어떤 클래스를 연동시켜서 사용하는 기술

ORM을 이용해서 DB에서 테이블을(실제 데이터를) 조회하였음.

실습을 진행하는 동안 객체를 출력하는 경우가 많음. 그때마다 데이터(어떤 todo 객체가 출력되는지)를 보기 쉽게 하기 위해서 repr이란 매직 메소드를 오버라이드해서 사용.

session.scalars(select(Todo)): todo 객체(테이블)에 있는 모든 레코드를 출력할 수 있게 된다.

(= SELECT from Todo)

그런데 ScalarResult 라는 오브젝트로 반환이 돼서 보기가 힘들다.

->list()로 묶어서 보기 좋게 출력

7. (실습) ORM 적용 - GET 전체 조회 API

: main.py에 생성했던 Todos API 들에 ORM을 적용해서 실제 데이터베이스와 연동되도록 코드를 리팩토링 해보는 실습 (여기서부터 쪽 리팩토링)

```

todo_data = {
    1: {
        "id" : 1,
        "contents": "실전! FastAPI 섹션 0 수강",
        "is_done": True,
    },
    2: {
        "id": 2,
        "contents": "실전! FastAPI 섹션 1 수강",
        "is_done": False,
    },
    3: {
        "id": 3,
        "contents": "실전! FastAPI 섹션 2 수강",
        "is_done": False,
    },
}

```

↳ 이전 구조(백엔드 과제 1번처럼 딕셔너리 형태로 직접 코드 내에 DB를 작성해봤다.)
딕셔너리를 사용하지 않고, 실제 DB와 연결해서 데이터 조회, 변경, 삭제 해보기

1. database/connection.py에서 get_db()라는 제너레이터를 하나 만든다.

python 제너레이터 (with yield)

<https://www.daleseo.com/python-yield/>

파이썬에서 제너레이터는 쉽게 말해서 여러 개의 데이터를 미리 만들어 놓지 않고 필요할 때마다
즉석해서 하나씩 만들어낼 수 있는 객체

일단 FastAPI를 사용해서 DB에 접근 가능하다는 것만 알아두기

```

main.py  connection.py  orm.py
1  from sqlalchemy import create_engine
2  from sqlalchemy.orm import sessionmaker
3
4  DATABASE_URL = "mysql+pymysql://root:todos@127.0.0.1:3306/todos"
5
6  engine = create_engine(DATABASE_URL, echo=True)
7  SessionFactory = sessionmaker(autocommit=False, autoflush=False, bind=engine)
8
9  def get_db(): 2개의 사용 위치
10     session = SessionFactory()
11     try:
12         yield session
13     finally:
14         session.close()

```

2. get_db라는 제너레이터를 API에서 이용한다.

FastAPI에서 리퀘스트가 들어왔을 때 get_db()(제너레이터)에서 session이 생성이 돼서 yield

문으로 리턴이 된 다음에 사용이 되다가 응답을 한 이후에 이 session을 클로즈해서 삭제하는 식으로 FastAPI가 session을 관리, 처리해주게 된다.

이때, main.py의 get_todos_handler의 ret = list(todo_data.values())로 얻어오는 방식에서 Session을 이용해서 데이터를 조회하게 수정.

3. database/repository.py 생성

레포지토리 패턴(레포지토리라는 파일로 코드를 분리하는 것을 말한다.)(섹션 4에서 추후 설명) 함수(get_todos)를 통해서 DB 조회하는 부분을 생성을 하고, 그 함수를 main.py에서 사용.

```
main.py repository.py × connection.py orm.py
1 from typing import List
2
3 from sqlalchemy import select
4 from sqlalchemy.orm import Session
5 from database.orm import ToDo
6
7 def get_todos(session: Session) -> List[ToDo]:
8     return list(session.scalars(select(ToDo)))
9
```

4. 다시 main.py로 와서 get_todos_handler 수정

수정 전

```
@app.get(path: "/todos", status_code=200)
def get_todos_handler(
    order: str | None = None,
    # session: Session = Depends(get_db),
):
    # todos: List[ToDo] = get_todos(session=session)

    ret = list(todo_data.values())
    if order and order == "DESC":
        return ret[::-1]
    return ret
```

수정 후

```
@app.get(path: "/todos", status_code=200)
def get_todos_handler(
    order: str | None = None,
    session: Session = Depends(get_db),
):
    todos: List[ToDo] = get_todos(session=session)

    if order and order == "DESC":
        return todos[::-1]

    return todos
```

다만, 정렬 부분도 orm을 이용해서 정렬하는데, 그 부분은 생략.(현재는 기존 구현한 파이썬 정렬)

Curl

```
curl -X 'GET' \
  'http://127.0.0.1:8080/todos' \
  -H 'accept: application/json'
```

Request URL

```
http://127.0.0.1:8080/todos
```

Server response

Code	Details
200	<p>Response body</p> <pre>[{ "contents": "FastAPI Section 0", "is_done": true, "id": 1 }, { "contents": "FastAPI Section 1", "is_done": true, "id": 2 }, { "contents": "FastAPI Section 2", "is_done": false, "id": 3 }]</pre> <p>Response headers</p> <pre>content-length: 167 content-type: application/json date: Mon, 14 Jul 2025 03:09:40 GMT server: uvicorn</pre>

DB에 의한 정상 호출 확인!

8. (실습) ORM 적용 - HTTP Response 처리

<https://www.daleseo.com/python-type-annotations/> (type hint)

```
class CreateToDoRequest(BaseModel): 1개의 사용 위치
    id: int
    contents: str
    is_done: bool
# pydantic을 이용한 리퀘스트 바디 정의
```

이렇게 리퀘스트 바디를 정의했던 것처럼,

```
if order and order == "DESC":
    return todos[::-1]
return todos
```

여기서 `get_todos_handler`의 리턴 부분도 리스폰스를 반환하는 형태로 api를 디자인하게 된다.

1. schema 라는 디렉토리 생성, `schema/response.py` 생성
`get_todos` API가 어떤 식으로 응답해야 하는지 schema를 정의하기

`ToDoSchema`가 `SQLAlchemy`의 ORM 객체를 전달받아서 내가 정의한 형태에 맞게 데이터를 변환을 하고 최종적으로 리턴을 해주게 된다.

그래서 왜 굳이 이렇게 리스폰스를 한 번 더 분리하고 정의해야 되냐라고 한다면은 지금은 저희가 컬럼이 되게 단순하고 이 구조가 단순하지만 이 응답의 구조가 단순하지만 만약에 어떤 컬럼 간의 연산이 있거나 혹은 이 객체를 뭔가 중첩된 구조로 반환을 한다거나 아니면은 뭐 컨텐츠 값을 제외하고 리턴을 하고 싶다 그러니까 아이디와 이즈던 값만 리턴을 하고 싶다 뭐 이런 식의 저희가 다양한 유즈 케이스가 있을 수 있기 때문에 미리 이런 식으로 리스폰스 객체를 분리해놓게 되면은 그런 부분에 대해서 저희가 훨씬 유연하게 코드 변경을 할 수가 있어서 저희 같은 경우에 먼저 리스폰스를 이런 식으로 분리하는 작업을 진행하도록 하겠습니다. 근데 저희가 이제 `Pydantic`에서 `SQLAlchemy`를 좀 바로 읽어줄 수 있도록 하려면은 뭔가 하나 옵션을 좀 추가를 해줘야 되는데요. 여기에 `class configuration`을 정의를 해준 다음에 여기에 ORM 모드는 `true` 이런 식으로 적어 주게 되면 파이덴틱에 정의한 ORM모드라는 걸 사용할 수 있게 됩니다. 그래서 이 to do 스키마에 `SQLAlchemy` ORM 객체를 던져주게 되면 이 파이덴틱이 그 ORM 객체를 잘

해석을 해서 여기에 맞게 변경을 시켜주게 됩니다.

```
main.py response.py repository.py connection.py orm.py
1 from typing import List
2
3 from pydantic import BaseModel
4
5
6 class TodoSchema(BaseModel): 1개의 사용 위치
7     id: int
8     contents: str
9     is_done: bool
10
11     class Config:
12         orm_mode = True
13
14
15 class ListToDoResponse(BaseModel):
16     todos: List[TodoSchema]
```

2. 여기까지 실습

```
Python 콘솔>>> from schema.response import TodoSchema
>>> from database.orm import ToDo
>>> todo = ToDo(id=100, contents="test", is_done=True)
```

일단 객체 생성. 근데 여기까지는 데이터베이스에 저장되거나 이런 건 아니고 말 그대로 Python 객체.

```
>>> TodoSchema.from_orm(todo)
TodoSchema(id=100, contents='test', is_done=True)
```

그래서 저희가 뭘 할 거냐면은 저희가 to do 스키마에서 from orm이라는 메소드를 이용을 해서 이 to do를 여기 안에 전달을 해주게 되면은 이런식으로 Todo Schema 객체가 생성이 되게 됩니다. 그래서 저희가 별도로 작업을 해주지 않아도 앞서서 ORM 모드를 톨으로 켜기 때문에 이 From ORM을 이용을 하면은 저희 ToDo ORM 객체를 이런식으로 Pydantic Schema로 쉽게 변환을 할 수가 있게 됩니다.

3. 이제 main.py의 Response API에 from_orm적용

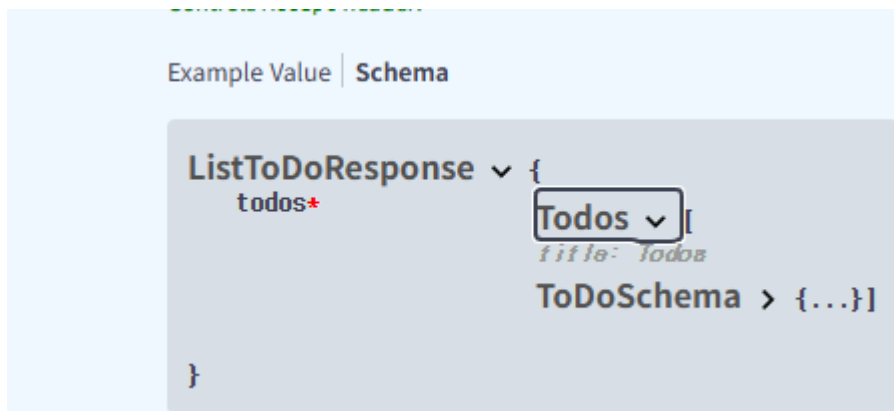
그래서 저희가 이제 이 From ORM을 API에서 사용하도록 한번 코드를 바꿔 보겠습니다.

```

@/todos
@app.get(path: "/todos", status_code=200)
def get_todos_handler(
    order: str | None = None,
    session: Session = Depends(get_db),
) -> ListToDoResponse:
    todos: List[ToDo] = get_todos(session=session)

    if order and order == "DESC":
        return ListToDoResponse(
            todos=[ToDoSchema.from_orm(todo) for todo in todos[::-1]]
        )
    return ListToDoResponse(
        todos=[ToDoSchema.from_orm(todo) for todo in todos]
    )

```



예시 스키마(Response Type을 지정해서 Swagger UI에서 볼 수 있다.)

(실습) ORM 적용 - GET 단일 조회 API

: 단일 ToDo API에도 ORM을 적용하는 실습

1. database/repository.py에 단일 todo 조회하는 함수(get_todo_by_todo_id) 만들기
(Optional[ToDo] 보다 ToDo | None이 최신 문법)

```

def get_todo_by_todo_id(session: Session, todo_id: int) -> ToDo | None:
    return session.scalar(select(ToDo).where(ToDo.id == todo_id))

```

2. main.py
수정 전

```

@/todos/{todo_id}
@app.get(path: "/todos/{todo_id}", status_code=200)
def get_todo_handler(todo_id: int):
    todo = todo_data.get(todo_id)
    if todo:
        return todo
    raise HTTPException(status_code=404, detail="ToDo Not Found")

```

수정 후

```
🔗 /todos/{todo_id}
@app.get(path: "/todos/{todo_id}", status_code=200)
def get_todo_handler(
    todo_id: int,
    ⚠ session: Session = Depends(get_db),
) -> TodoSchema:
    todo: Todo | None = get_todo_by_todo_id(session=session, todo_id=todo_id)
    if todo:
        return TodoSchema.from_orm(todo)
    raise HTTPException(status_code=404, detail="ToDo Not Found")
```