In our project, we are using both Pydantic Models and Sqlalchemy ORM models.

Pydantic Models are used to define the format and fields in which user should send the data in response. It provides easy validation of data (in terms of data-type & whether data for a particular field is mandatory) that is received by the API (i.e the path operations here).

On the other hand, Sqlalchemy ORM models act as the schema for the database tables. In these models, each static variable corresponds to each column in the database table. Using this class, we perform operations on the database table, using Python code, and not SQL queries.

In FastAPI projects, using both ORM and pydantic models, the classes are kept into different files as follows :-

Pydantic models —made in→ schemas.py

ORM (Sqlalchemy) models —made in→ models.py

Seperate Pydantic Models for Request & Response :-

As we are till now using a specific Pydantic model for structuring of data sent in request, now, we will use another Pydantic model, for structuring the data sent in response, in a specified standard structure.

Till now, we just returned the entire new/updated data (i.e post in our case), as a value of dictionary but that's not the standard way. This is because, at times, we don't want to send the user all the fields, as some fields are not intended to be shown or returned, such as the unique id's, and passwords.

Creating a pydantic model for response, in the schemas.py file :- 5:43:30

```
class PostResponse(BaseModel):
    title: str
    content: str
    published: bool
    class Config:
        orm_mode = True
```

we can also add:
created_at: datetime

Pydantic's orm_mode (of subclass Config) will tell this pydantic model (class) to read the data even if it is not a Python dictionary, ie, if the object returned as response (from Path operation) is not a dictionary, it will be read & converted to dictionary, if orm_mode is set to True.

[but for that, the object being read must have similar attributes, as in dict, such as an ORM model object]

But, for that, this Pydantic model should be set as response_model in the path operation as follows:-

Pydantic model name

```
@app.post("/posts",..., response_model=schemas.PostResponse)
def create_post(.....):
    ...
    new_post = models.Post(**post.dict())
    ...
    db.refresh(new_post)
    return new_post
```

this ORM model will now be converted to specified Pydantic model and returned as response (thus response data will only contain fields specified in the Pydantic model)

NOTE:-

With API's, we should explicitly define the exact format of data we want to receive, and we want to send.

Pydantic model for request's data format specification.

Pydantic model for specifying format of response data

```python
class PostBase(BaseModel):
    title: str
    content: str
    published: bool = True


class PostCreate(PostBase):
    pass


class Post(PostBase):
    id: int
    created_at: datetime

    class Config:
        orm_mode = True
```

Like this, for better code quality, we can use inheritance for making multiple similar models. It is preferable.

5:48:00

- Like this, we need to set the response_model for all the path operations that sends post data. The response model should be same for all the path operations regarding a single API set (convention)

- In case of getting all the post, we cannot simply set the Pydantic model as the response_model, as it must be a list of all the posts (pydantic model objects). Thus, to make the path operation return a list of pydantic model objects, following is the code :-

```python
from typing import List

@app.get("/posts", response_model=List[schemas.PostResponse])
def get_posts(db: Session = Depends(get_db)):
    all_posts = db.query(models.Post).all()
```

return all_posts → now, it will be converted to a list of Pydantic model objects and then returned as response.