⇒ Now, we will code our path operation functions, where the database operations will be done with ORM models & methods (of Sqlalchemy)

⇒ This is an example path operation to fetch all the posts, for get request :-

```
@app.get("/sqlalchemy")
def test_posts(db: Session = Depends(get_db)):
    all_posts = db.query(models.Post).all()
    return {"data": all_posts}
```

dependency used for db connection and operations

model class corresponding to the database table.

⇒ As parameter, we pass db, that we already saw. This parameter will establish connection each time request is made to this Path operation.

Also, using this db object, the methods will be called, that will execute operations on database.

⇒ query() method of the db object takes the ORM model class as parameter, which represents the table on which we want to perform operation.

query() method actually returns a SELECT sql query. (can be seen by printing it). The query is simply :-

```
SELECT * FROM posts;
```

table name of class that is passed as parameter.

query().

Of this object, there is an all() method. This method returns all the records from the specified table (in the parameter of query()). These records are in usable form in Python code now.

Actually, using all() (and similar) method, we run the Sql query (that is being generated by query()). all() returns a list of ORM model class objects, where each object is a record in table.

5:07:55

Creating Post Path Operation :-

To create a post, we need to reference the models.Post class, and pass in the specific fields with which we want the new entry.

```
@app.post("/posts", status_code = 201)
def create_post (post:Post, db: Session = Depends(get
                                                  _db)):
    new_post = models.Post(...
        title=post.title, content=post.content, ...
        published = post.published)
    db.add (new_post)
    db.commit()
    db.refresh (new_post)
    return {"data": new_post}
```

- As parameter, we need to pass the pydantic model, as that is used to validate the format of data provided by user.
- Also, as we will perform operation on database using ORM, will be passing the db dependency object as parameter.
- Then, we are creating a model class object (which is the models.Post class here), by calling the model class named function, and in the parameter, passing the details passed by user, as keyword arguments, where the keywords are same as member variables of the model class.

  new_post

  Ex :- (title = ..., content = ..., ...)

Now, suppose the data provided by user contains large no. of fields, then assigning each value in the parameter is not possible. At that time, in the parameter of model class object function (i.e, models.Post() here), we will just unpack the pydantic class object (obtained as parameter, i.e, post : Post), as dictionary, as follows :-

```
new_post = models.Post(**post.dict())
```

It will assign all the member variables of model class, with the correct data of the user, implicitly.

⇒ Then, we have the model class object (new_post) with the required data. Thus, we can save this data to database, using the db.add() method (add() method of the database dependency object)
add() method actually incorporates the INSERT SQL query.

⇒ After performing the insertion, changes made to database must be committed. This is done by using the db.commit() method.

⇒ We cannot put RETURNING keyword to the backend SQL query. Thus, to get the changed/newly inserted row, we use the db.refresh() method, and in it, pass an object, which will have the new row, after execution of the function.

⇒ Getting a single Post Path operation :-

```
@app.get("/posts/{id}")
def get_post(id: int, db: Session = Depends(get_db)):
    req_post = db.query(models.Post).filter(models.
                Post.id == id).first()
    if req_post == None:
        raise HTTPException(status_code = 404,
                detail = f"Post with id={id} not found!"
    else:
        return {"data": req_post}
```

⇒ For getting post, we need SELECT query, thus we use the db.query() method primarily. It actually gives SQL query to return all the records.

⇒ From all records, we need to filter only specific record, using WHERE keyword, and obtained id. Thus, we apply the filter() method, which takes the condition as parameter (in Python class and member variable terms), and converts it to WHERE SQL clause condition.
In Python, the condition will be :-

models.Post.id = id

↳ the id member variable value of Post model class object should match the id provided by user as path parameter.

Till now, the statement generates SQL query.

⇒ To execute the SQL query generated, we must use the all() or first() method at the end. As we know there will be only 1 record with specified condition, we use the first() method here, as after finding 1 record, DBMS software will no more search till the end.
It will make the path operation efficient.
The record is returned as object, and is thus stored for usage (in req_post here).

⇒ At last the Exception Handling remains same. If req_post contains None., we return 404: Not Found. Else, we return the record.

5:20:00

⇒ Deleting Post : Path Operation :-

```
@app.delete("/posts/{id}", status_code=204)
async def delete_post(id: int, db: Session = Depends(get_db)):
    post_query = db.query(models.Post).filter(models.Post.id == id)
    if post_query.first() == None:
        raise HTTPException(status_code=404,
                            detail="Post does not exist".
    else:
        post_query.delete(synchronize_session=false)
        db.commit()
        return Response(status_code=204)
```

⇒ We use the filter() method to obtain the query that fetches the particular record.

⇒ We execute the query using first() method, and see if there are any records. If there are no records, it means post to be deleted do not exist.

» If records are returned, delete() method of the query object (post_query here) is used to delete the required record. To this method, this parameter is to be passed :-

synchronize_session = False

» Updating Post :-

```
@app.put("/posts/{id}")
async def update_post(id: int, updated_post=
    ...Post, db: Session=Depends(get_db)):
    post_query = db.query(models.Post).filter(
        ... models.Post.id == id)

    if post_query.first() == None:
        raise HTTPException(status_code=404,
            ... detail = "Post does not exist!")

    post_query.update(updated_post.dict(),
        ... synchronise_session = False)
    db.commit()
    return {"data": post_query.first()}
```

The updated post will be fetched by this statement now, and will be returned.

- As parameter, we take the post id, updated data, and the database dependency.

- We then form an sql query (using filter()) that will fetch the post having the specified id.
By executing the query, if no post is fetched, then it means post with that id do not exist, else...

- Of that query object (post_query here), we use the update() method, to change that post with new data. As the 1st parameter, we need to pass a dictionary containing new values. This dictionary can be obtained from pydantic models' dict() method

As second parameter, we pass synchronise_session=false for reliability.

- Then we commit the changes using db.commit(), and return the updated data, i.e., post.