10:15:35

# SQL joins using SQLAlchemy:-

- To perform a join with SQLalchemy, we use the `join()` method.
  By default, it performs **Left Inner Join.**

- `join()` method takes 2 parameters:-
  1. The model class name which represents the Right table.
  2. Condition of equality of columns based upon which the tables will be joined.

10:18:10

- To perform left outer join, we need to pass an additional keyword argument `isouter=True` to the `join()` method.

```
join_query = db.query(models.Post).join(models.Vote,
    models.Vote.post_id == models.Post.id, isouter=True)
```

left table class

right table class

column equality condition for join

parameter for performing outer join

# Group By using Sqlalchemy :-

- For counting no. of votes, we need to group the records first, based on posts.id column.

- For grouping records in Sqlalchemy, we use the `group_by()` method.
  As argument, it takes name of the static variable, which represents the column, based on which grouping is to be done.

Ex:
```
db.query(models.Post).group_by(models.Post.id)
```
(grouped based on id)

# Using Aggregate Functions like COUNT, in Sqlalchemy :-

- To use aggregate functions like COUNT, we need to import `func` as follow:-
```
from sqlalchemy import func
```

- To count, we have the `func.count()` method.
  Takes the static variable as argument, which represents the columns, on which counting operation is to be performed.

⇒ This function is to be used, in the place of 2nd argument of the db.query() method.

Ex:-

```
db.query(models.Post, func.count(models.Vote.post_id))
```

On this column, we are performing count operation.

⇒ To rename the count column (which shows the count of records), we use the label() method, and pass a string to it, which we want to be the alias of our column.

Ex:-

```
..., func.count(models.Vote.post_id).label("votes")...
```

we need to chain this method, with that column, which we want to rename.

---

⇒ Thus, to fetch the no. of votes alongwith the posts, the sqlalchemy query code would be as:-

```
results = db.query(models.Post, func.count(models.Vote.post_id).label("votes")).join(models.Vote, models.Vote.post_id == models.Post.id, isouter=True).group_by(models.Post.id).all()
```

we will use this query (by adding filters) in our get_posts() path operation now, and will return results

# Making changes in the Response schema for posts :-

As we are returning different set/type of objects, from the get_posts() path operations, we need to use a different response schema as follows :-

will inherit the default obtained pydantic class

```
class PostVotesResponse(BaseModel):
    Post : PostResponse
    votes : int
```

The prev. response model is used as type for this field.

we can check by removing response_model parameter from path operation function decorator, and find what things are getting returned, and based on that, we made our new schema model as above, and use it for the get_posts path operation response

---

We need to accordingly change the get_single post path operation, and change the filter() condition, & remove limit() & offset(). Rest query will be the same.

We need not return votes for update_post() & delete_post() path operation.