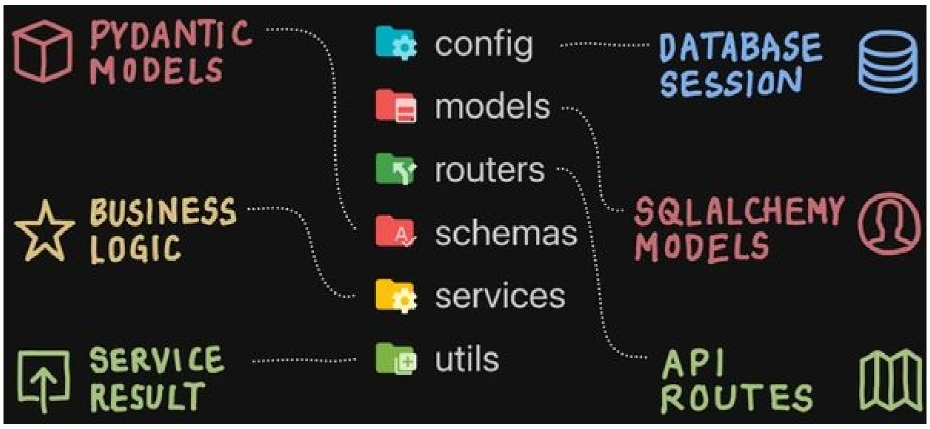

FastAPI Database Queries

Basic Queries:

1. query(): Create a query object to interact with the database.
2. filter(): Apply filtering complex conditions to the query results.
3. filter_by(): Apply filtering allows you to specify filtering conditions using keyword arguments.
4. all(): Return all records that match the query conditions.
5. first(): Return the first record that matches the query conditions.
6. one(): Return the first record that matches the query conditions or raise an exception if no record or multiple records are found.
7. get(): Retrieve a record by its primary key or return None if not found.

query()

```
query(): Create a query object to interact with the database.
```

code :

```
from sqlalchemy.orm import sessionmaker

Session = sessionmaker(bind=engine)
session = Session()

query = session.query(User)  # Creates a query object for the User model
```

filter()

```
Apply filtering conditions to the query results.
```

code :

```
from sqlalchemy import and_, or_

# Filtering with AND condition
query = session.query(User).filter(User.age >= 18, User.is_active == True)

# Filtering with OR condition
query = session.query(User).filter(or_(User.age >= 18, User.is_admin == True))
```

filter_by():

```
Apply filtering allows you to specify filtering conditions using keyword arguments.
```

code :

from sqlalchemy.orm import Session

Assuming you have a User model and a database session 'd :b

```
' query = db.query(User).filter_by(username="john_doe", is_active=True).al l()
```

all()

```
Return all records that match the query conditions.
```

code :

```
users = query.all()
```

first()

```
first(): Return the first record that matches the query conditions.
```

code :

```
users = query.first()
```

one()

```
one(): Return the first record that matches the query conditions or raise an exception if no
record or multiple records are found.
```

code :

```
user = query.one()
```

get()

```
get(): Retrieve a record by its primary key or return None if not found.
```

code :

```
user = session.query(User).get(1)  # Assuming 1 is the primary key value
```

Filtering and Conditionals:

1. and_(): Combine multiple filtering conditions with the logical AND operator.
2. or_(): Combine multiple filtering conditions with the logical OR operator.
3. not_(): Negate a filtering condition.

And()

```
and_(): Combine multiple filtering conditions with the logical AND operator.
```

code :

```
from sqlalchemy import and_

query = session.query(User).filter(and_(User.age >= 18, User.is_active == True))
```

OR()

```
or_(): Combine multiple filtering conditions with the logical OR operator.
```

code :

```
from sqlalchemy import or_

query = session.query(User).filter(or_(User.age >= 18, User.is_admin == True))
```

Not()

```
not_(): Negate a filtering condition.
```

code :

```
from sqlalchemy import not_

query = session.query(User).filter(not_(User.is_active == False))
```

(~)

```
In SQLAlchemy, the tilde (~) operator is used as a bitwise NOT operator when performing
queries. It is used to negate filtering conditions, making it convenient to express conditions
where a particular condition should not be true.
```

code :

```
from sqlalchemy import not_

# Assuming you have a User model with age and is_active attributes
# We want to retrieve users who are not active

query = session.query(User).filter(~User.is_active)
inactive_users = query.all()
```

```
In this example, the ~User.is_active expression negates the filtering condition. It will
return all users where the is_active attribute is not True or is False. Essentially, it
fetches users who are not active. ☝
```

Code :

```
# Assuming you have a User model with age attribute
# We want to retrieve users who are not 25 years old

query = session.query(User).filter(~(User.age == 25))
users_not_25_years_old = query.all()
```

```
Here, ~(User.age == 25) negates the condition, so it fetches users whose age is not equal to
25.
```

Sorting:

Order_by()

1. order_by(): Specify the sorting order of the query results based on one or more columns.

code :

```
from sqlalchemy import desc

query = session.query(User).order_by(desc(User.age))  # Sort by age in descending order
```

Limiting and Paging:

1. limit(): Limit the number of results returned by the query.
2. offset(): Skip a specified number of results from the beginning of the query result.

limit()

```
limit(): Limit the number of results returned by the query.
```

code :

```
query = session.query(User).limit(10)  # Return only the first 10 results
```

offset()

```
offset(): Skip a specified number of results from the beginning of the query result.
```

code :

```
query = session.query(User).offset(20)  # Skip the first 20 results
```

Aggregations and Grouping:

1. func.count(): Calculate the count of records.
2. func.sum(): Calculate the sum of a column.
3. func.avg(): Calculate the average of a column.
4. func.min(): Find the minimum value of a column.
5. func.max(): Find the maximum value of a column.
6. group_by(): Group the results based on one or more columns.

func.count()

```
func.count(): Calculate the count of records.
```

code :

```
from sqlalchemy import func

count = session.query(func.count(User.id)).scalar()
```

func.sum()

```
func.sum(): Calculate the sum of a column.
```

code :

```
    total_salary = session.query(func.sum(User.salary)).scalar()
```

func.avg()

```
func.avg(): Calculate the average of a column.
```

code :

```
  average_age = session.query(func.avg(User.age)).scalar()
```

func.min()

```
func.min(): Find the minimum value of a column.
```

code :

```
  min_age = session.query(func.min(User.age)).scalar()
```

func.max()

```
func.max(): Find the maximum value of a column.
```

code :

```
  max_salary = session.query(func.max(User.salary)).scalar()
```

func.group_by()

```
group_by(): Group the results based on one or more columns.
```

code :

```
from sqlalchemy import func

query = session.query(User.age, func.count(User.id)).group_by(User.age)
```

Joins:

1. join(): Perform an inner join with another table.
2. outerjoin(): Perform an outer join with another table.

join()

```
join(): Perform an inner join with another table.
```

code :

```
from sqlalchemy import join

query = session.query(User).join(Post, User.id == Post.user_id)
```

outerjoin()

```
outerjoin(): Perform an outer join with another table.
```

code :

```
from sqlalchemy import outerjoin
```

```
query = session.query(User).outerjoin(Post, User.id == Post.user_id)
```

Subqueries:

1. subquery(): Create a subquery to be used within another query. Aliases:

2. aliased(): Create an alias for a table or query.

Subqueries()

```
subquery(): Create a subquery to be used within another query.
```

code :

```
from sqlalchemy import subquery

subquery = session.query(User.id).filter(User.age >= 18).subquery()
query = session.query(Post).filter(Post.user_id.in_(subquery))
```

Aliases()

```
aliased(): Create an alias for a table or query.
```

code :

```
from sqlalchemy.orm import aliased

user_alias = aliased(User)
query = session.query(User, user_alias).join(user_alias, User.age == user_alias.age)
```

Transactions:

1. commit(): Commit changes made during a transaction.
2. rollback(): Roll back changes made during a transaction.

commit()

```
commit(): Commit changes made during a transaction.
```

code :

```
try:
    # Start a transaction
    with SessionLocal() as db:
    # Perform database operations
    # ...

    # Commit the changes
    db.commit()

except Exception as e:
    # Handle exceptions and rollback changes on error
    db.rollback()
    raise e
```

rollback()

```
rollback(): Roll back changes made during a transaction.
```

code :

```
try:
    # Start a transaction
    with SessionLocal() as db:
    # Perform database operations
    # ...

    # Rollback the changes (e.g., on error)
    db.rollback()

except Exception as e:
    # Handle exceptions
    raise e
```

> These are the most common querying operations that can be performed using SQLAlchemy in Python. SQLAlchemy provides a powerful and flexible ORM (Object-Relational Mapping) system that simplifies the process of interacting with databases, making it easier to work with your data in a Pythonic way.