Software Engineering For Data Science (SEDS)

Class: 2nd Year 2nd Cycle
Branch: AIDS

Dr. Belkacem KHALDI| ESI-SBA

Lecture 10:

Web Development for Data Science: Building Restful API with FastAPI – Part II

Web Development for Data Science: Building Restful API with FastAPI – Part II

Asynchronous ORMs

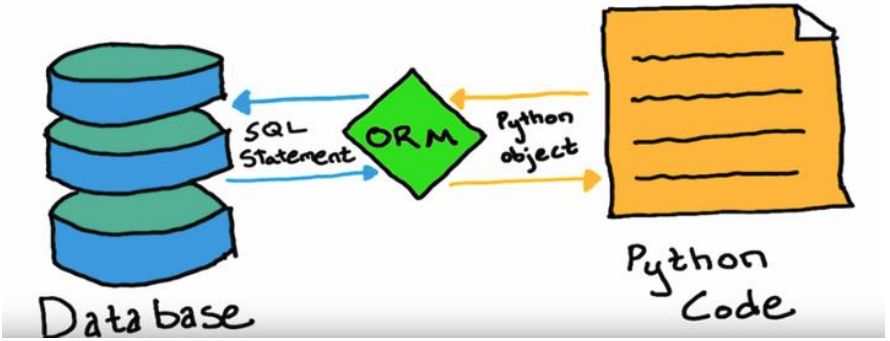
Dr. Belkacem KHALDI
e-mail: b.khaldi@esi-sba.dz

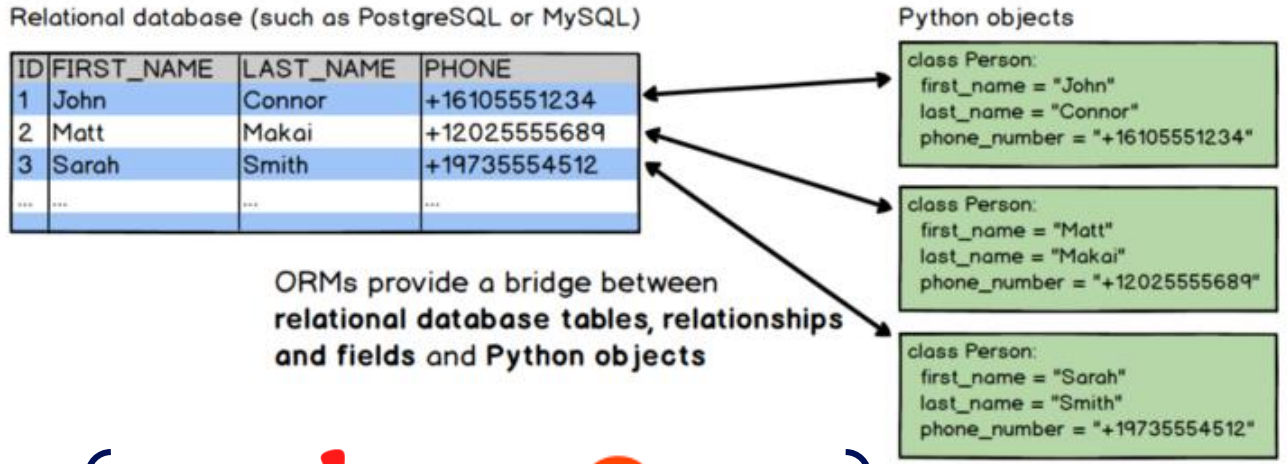
Building Restful API with FastAPI - Part II

Asynchronous ORMs

ORMs

- **FastAPI** works with any database and any style of library to talk to the database.
- A common pattern is to use an "**ORM**": an "**O**bject-**R**elational **M**apping" library.
- An ORM has tools to convert ("**map**") between **objects** in code and **database tables** ("**relations**").Allows communications between application components
- With an **ORM**, you normally create a class that represents a table in a SQL database, each attribute of the class represents a column, with a name and a type.





SQLAlchemy

- **The** Python SQL toolkit and Object Relational Mapper that gives application developers the full power and flexibility of SQL.

```
conda install sqlalchemy
pip install sqlalchemy
```

Few Companies using SQLAlchemy

Building Restful API with FastAPI - Part II

Asynchronous ORMs

- **Example: A User Posts Application**

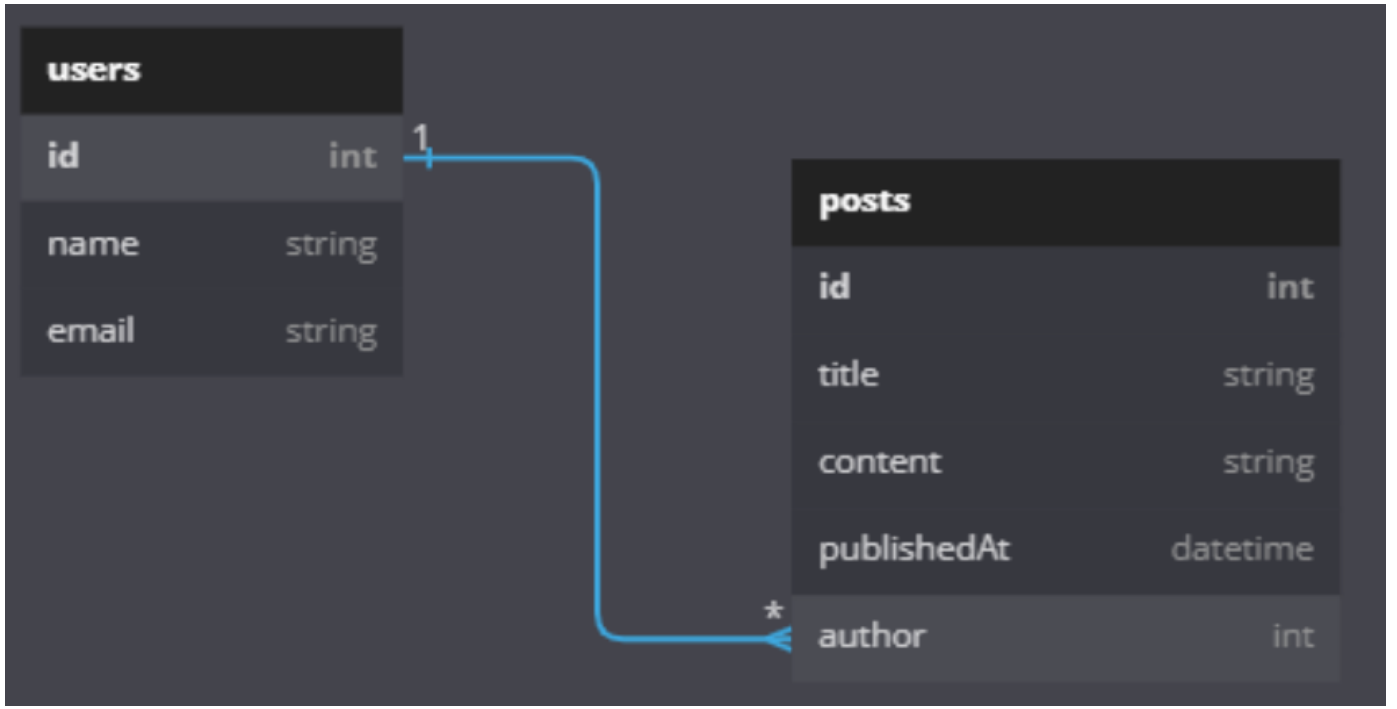
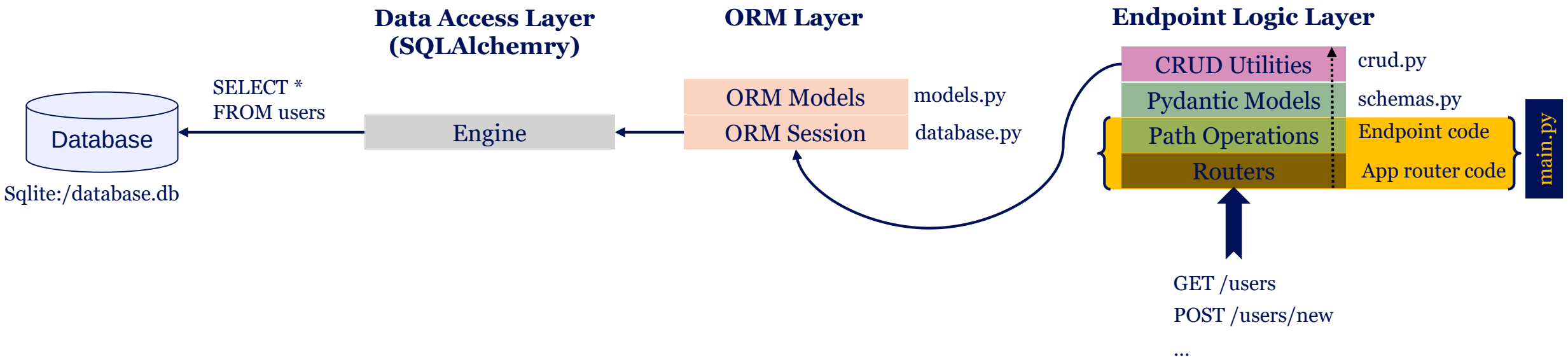


Fig. A relational database schema example for a blog application

Building Restful API with FastAPI - Part II

Asynchronous ORMs

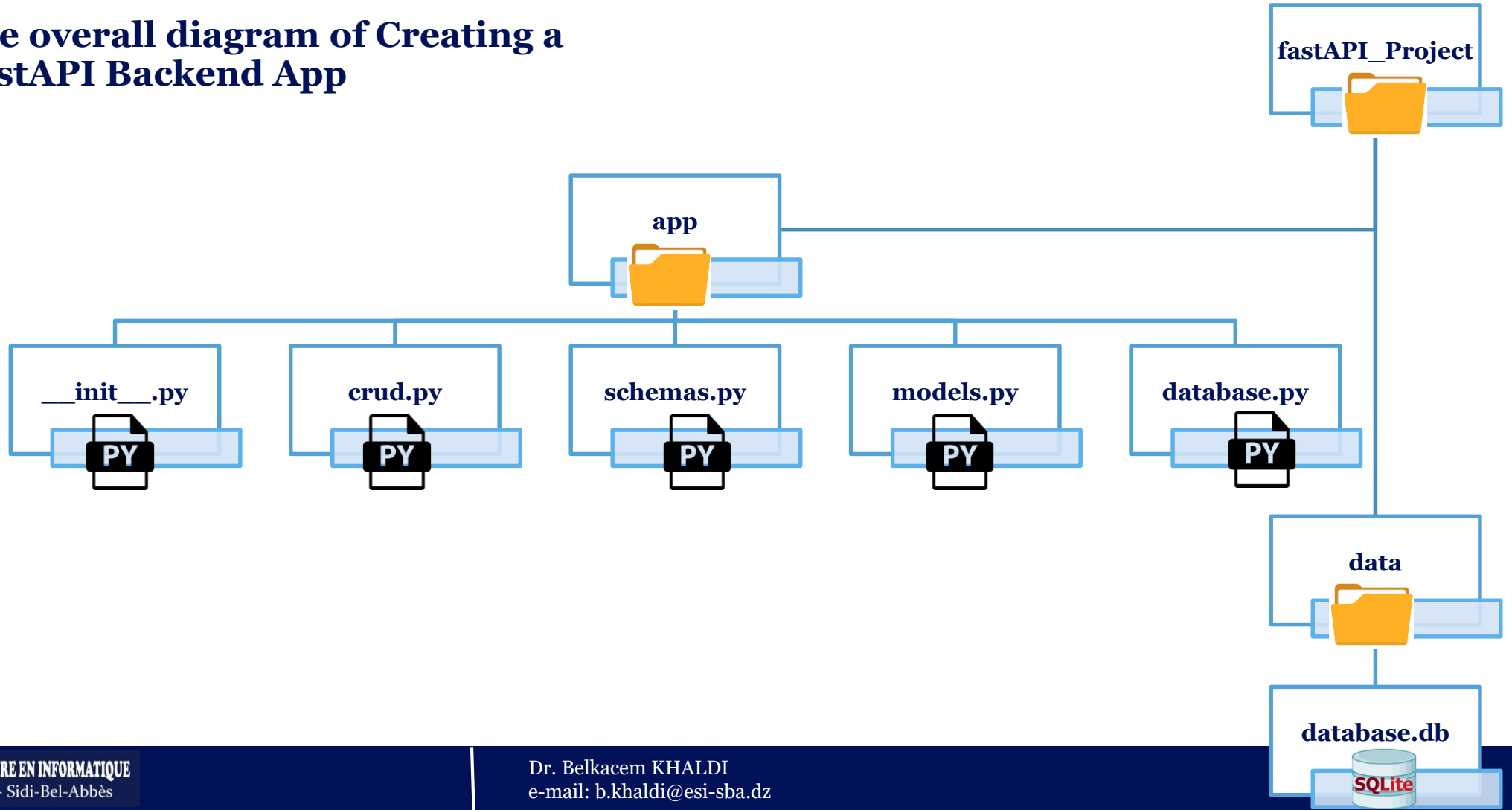
- **The overall diagram of Creating a FastAPI Backend App**



Building Restful API with FastAPI - Part II

Asynchronous ORMs

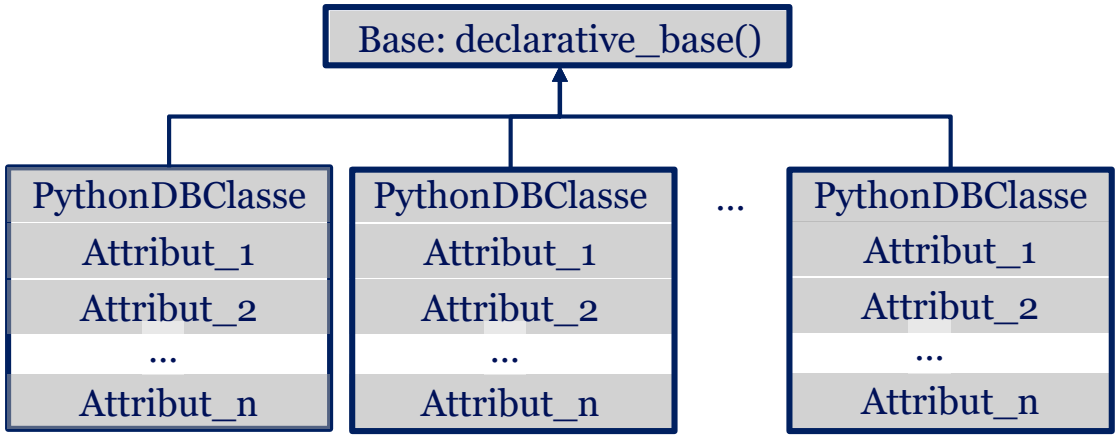
- **The overall diagram of Creating a FastAPI Backend App**

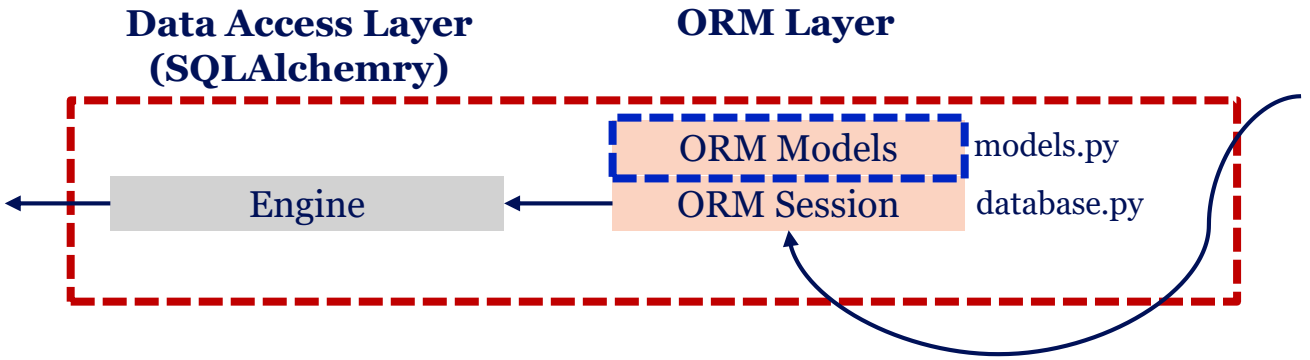


Building Restful API with FastAPI - Part II

Asynchronous ORMs

- **ORM and Data Access Layers:**
  - We want to define tables and columns from Python **Classes** using the **ORM**
  - In **SQLAlchemy**, this is enabled through a ***declarative mapping*** using:
    - **SQLALchemy declarative_base (**See **Base** object in module **models.py)**





models.py

```
from sqlalchemy.ext.declarative import declarative_base

from sqlalchemy import Boolean, Column, ForeignKey,
Integer, String, DateTime, Text
from sqlalchemy.orm import relationship
from sqlalchemy.sql import func

#Create a Base class using declarative_base(). Used  to
create the ORM models
Base = declarative_base()
```

Building Restful API with FastAPI - Part II

Asynchronous ORMs

- **ORM and Data Access Layers:**
  - All **DB** model classes are then **inherit** from the **Base** class **(**See **User** class in module **models.py)**:
  - Database **users table** defined with a **Python class**, which inherits from the **Base class** ➔ Allows **SQLAlchemy** to automatically detect and map the class to a **database table.**
  - Each model (class) **attribute** ➔ represents a **column** in its corresponding **database table**.
  - **Relationships** can be created using **relationship built-in function** provided by **SQLAlchemy ORM**.
  - Specifying a **foreign key** ➔ implies defining a **Parent** table (**users table**).

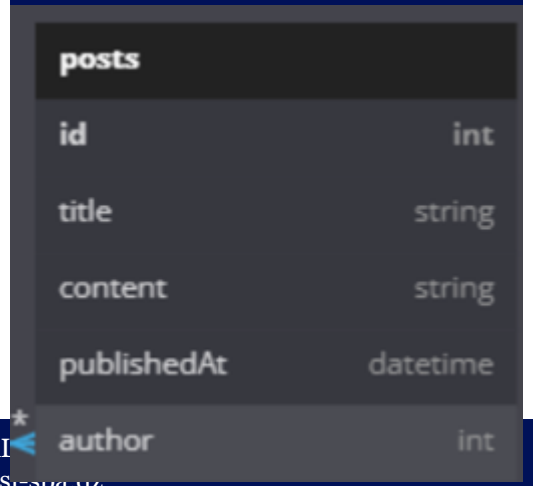
https://docs.sqlalchemy.org/en/14/orm/basic_relationships.html

models.py

ORM Models models.py

```
class Post(Base):
    __tablename__ = "posts"

    id = Column(Integer, primary_key=True, index=True,
autoincrement=True)
    publishedAt = Column(DateTime(timezone=True),
nullable=True, server_default=func.now())
    title = Column(String, index=True, nullable=False)
    content = Column(Text, nullable=False)
    author = Column(Integer, ForeignKey("users.id"))
    #Establish a bidirectional One-To-Mant relationship
    owner = relationship("User", back_populates="posts")
```

Database Table



To establish a **bidirectional** relationship in **one-to-many**, where the "reverse" side is a **many to one** ➔ Specify an additional relationship() and connect the two using the **relationship.back_populates** parameter:

Building Restful API with FastAPI - Part II

Asynchronous ORMs

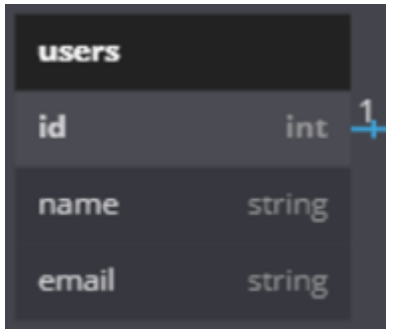
- **ORM and Data Access Layers:**
  - All **DB** model classes are then **inherit** from the **Base** class **(**See **User** class in module **models.py)**:
  - Database **users table** defined with a **Python class**, which inherits from the **Base class** ➔ Allows **SQLAlchemy** to automatically detect and map the class to a **database table.**
  - Each model (class) **attribute** ➔ represents a **column** in its corresponding **database table**.
  - **Relationships** can be created using **relationship built-in function** provided by **SQLAlchemy ORM**.

models.py

ORM Models models.py

```
class User(Base):
    __tablename__ = "users"

    id = Column(Integer, primary_key=True, index=True,
autoincrement=True)
    name = Column(String, nullable=False)
    email = Column(String, unique=True, index=True)
    #Establish a bidirectional One-To-Many relationship
    posts = relationship("Post", back_populates="owner")
```

Database Table



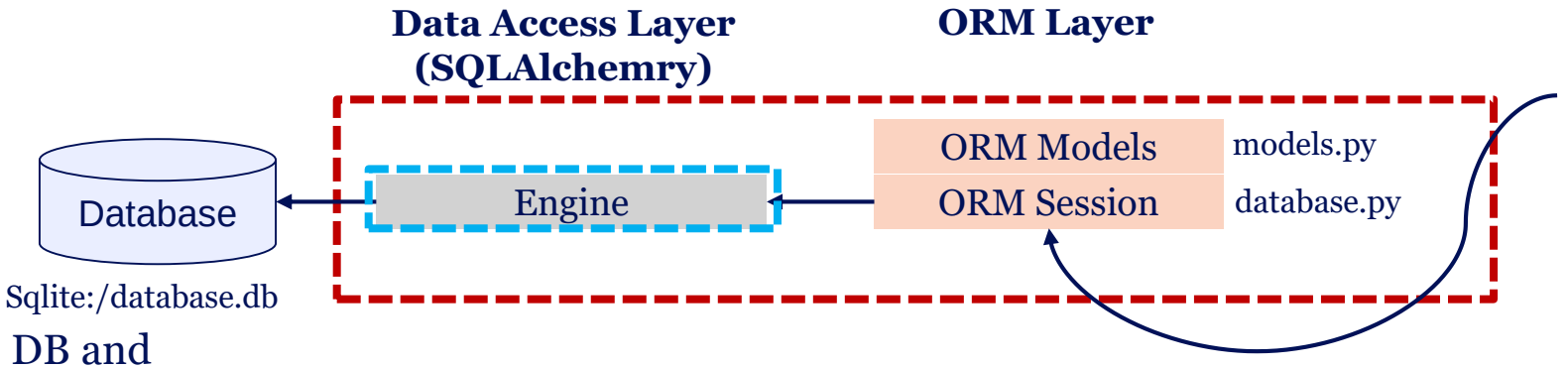
To establish a **bidirectional** relationship in **one-to-many**, where the "reverse" side is a **many to one** ➔ Specify an additional relationship() and connect the two using the **relationship.back_populates** parameter:

https://docs.sqlalchemy.org/en/14/orm/basic_relationships.html

Building Restful API with FastAPI - Part II

Asynchronous ORMs

- **ORM and Data Access Layers:**
  - The **SQLALchemy Engine**:
    - handles how to connect to the DB and maps the SQL automatically.
    - An instance **engine** object is instantiated in the **database.py** **module**:
    - The SQLALCHEMY_DATABASE_URI defines the file where **SQLite** will persist data.
    - The **SQLAlchemy** create_engine() ➔ instantiates the **engine** (Much more complex **Connection String** may include drivers, dialects, database server locations, users, passwords and ports)



database.py

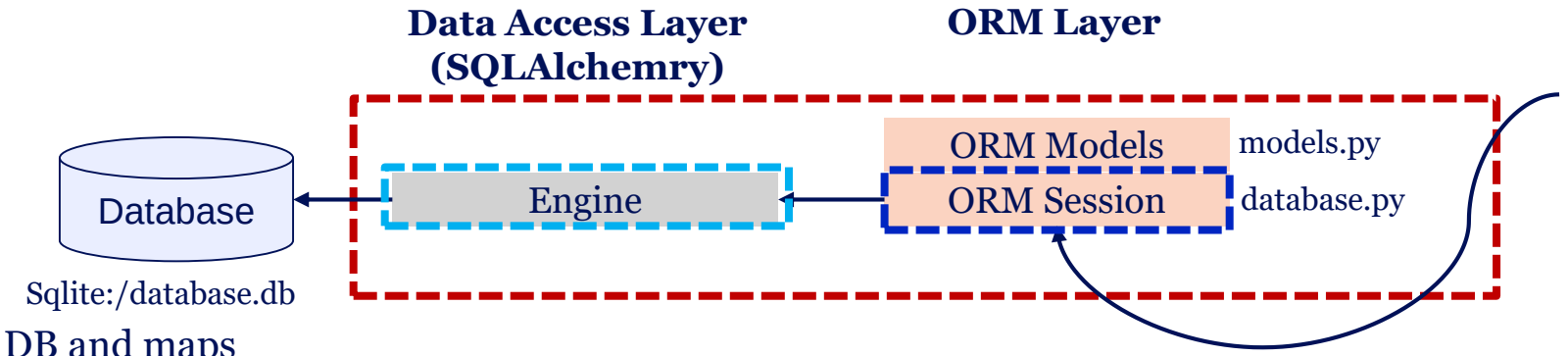
```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker

#Create a database URL for SQLAlchemy
SQLALCHEMY_DATABASE_URL = "sqlite:///./data/database.db"
#Create the SQLAlchemy engine
engine = create_engine(SQLALCHEMY_DATABASE_URL,
                       connect_args={"check_same_thread": False}
                       )
```

Required only when using SQLite ➔ FastAPI can access the database with **multiple threads** during a single request, so SQLite needs to be configured to allow that

Building Restful API with FastAPI - Part II

Asynchronous ORMs

- **ORM and Data Access Layers:**
  - The **SQLALchemy Engine**:
    - Handles how to connect to the DB and maps the **SQL** automatically.
    - An **ORM Session**, which (unlike the engine) is ORM-specific. When working with the **ORM**, the session object is our main access point to the database.
    - The **ORM Session** (SessionLocal) ➔ Establishes all conversations with the **database.**
      - It represents a "**holding zone**" for all the **objects** loaded or associated with it during its **lifespan**.
      - All **SQL operations** will be performed by this **session**.



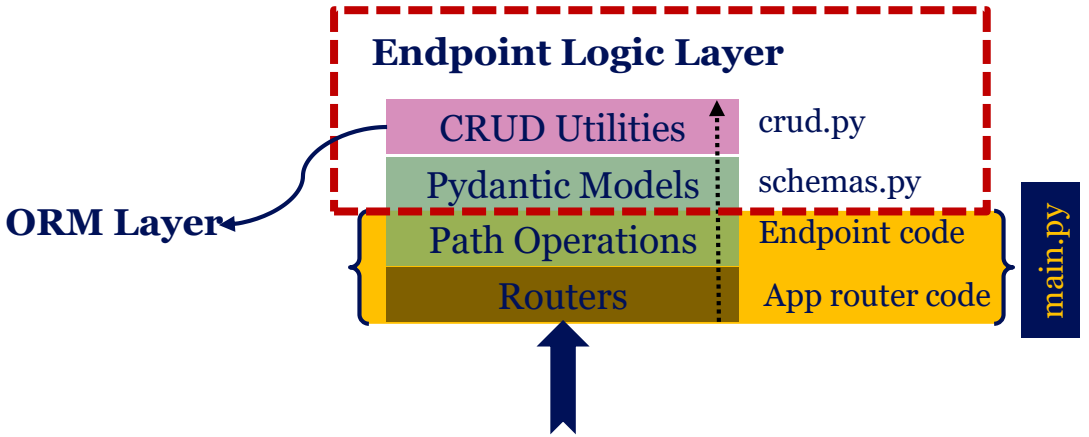
database.py

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker

#Create a database URL for SQLAlchemy
SQLALCHEMY_DATABASE_URL = "sqlite:///./data/database.db"
#Create the SQLAlchemy engine
engine = create_engine(SQLALCHEMY_DATABASE_URL,
                       connect_args={"check_same_thread": False}
                       )
#Initializing a new Session object.
SessionLocal = sessionmaker(bind=engine)
```

Building Restful API with FastAPI - Part II

Asynchronous ORMs

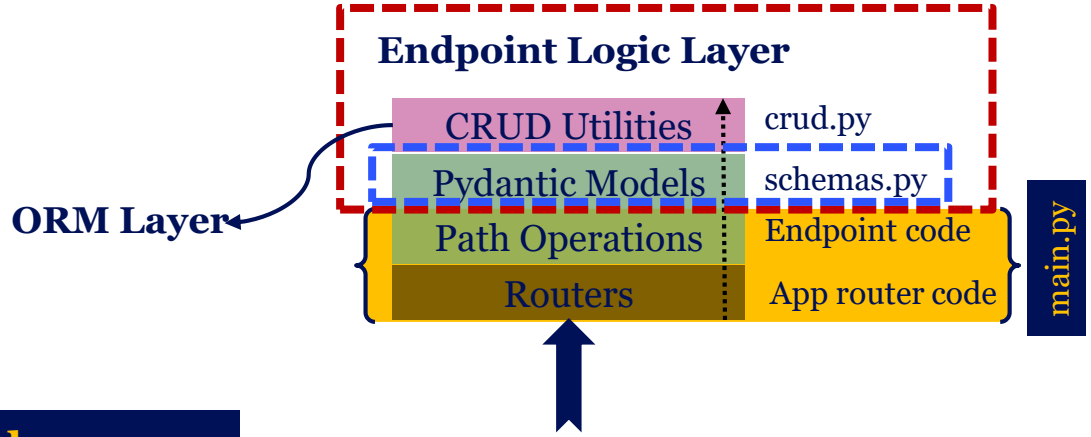
- **Endpoint logic Layer: Pydantic DB Schemas and CRUD Utilities:**
  - **Pydantic describes itself as:** Data validation and settings management using python type annotations.
    - Very useful for **reading/writing** data from various **APIs**.
    - **Pydantic classes** are recommended to be placed in a separate module ➔ **shemas.py**
  - **CRUD Utilities module (crud.py) help us to do things like:**
    - Reading from a table by ID or a particular attribute (e.g. by user email)
    - Read multiple entries from a table (defining filters and limits)
    - Doing CRUD operations (Insert, Update, or Delete a row in a table)



Building Restful API with FastAPI - Part II

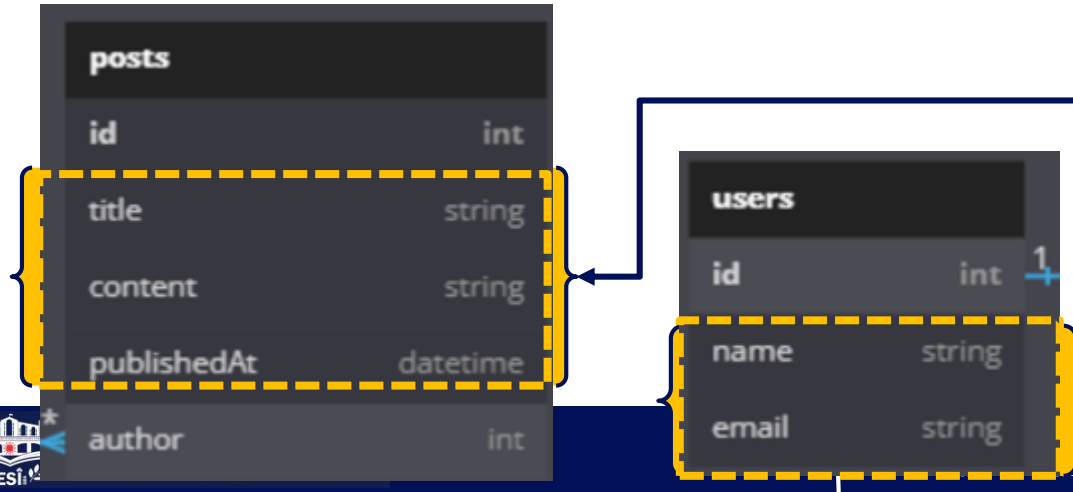
Asynchronous ORMs

- **Endpoint logic Layer: Pydantic DB Schemas and CRUD Utilities:**
  - **Pydentic Base Models** with common attributes are very often recommended to be created while creating or reading data (e,g. **PostBase** and **UserBase**).
  - **Pydantic models** declare the types using **:**, the new type annotation syntax/type hints:
    - ***Pydantic*** supports many **common types** from the **Python standard library**



schemas.py

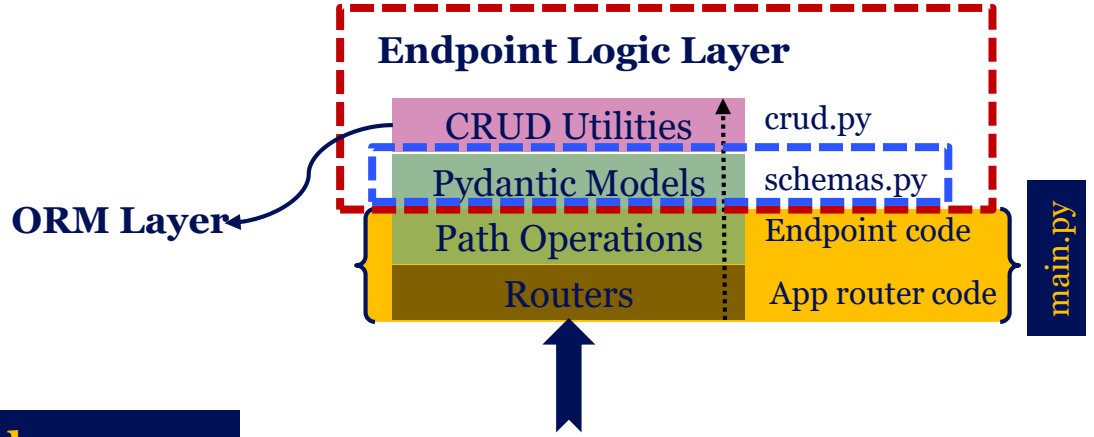
```python
from pydantic import BaseModel
import datetime

#Create Pydantic Base models whith common attributes while creating or reading data.
class PostBase(BaseModel):
    publishedAt: datetime.datetime
    title: str
    content: str

class UserBase(BaseModel):
    name: str
    email: str
```



Building Restful API with FastAPI - Part II

Asynchronous ORMs

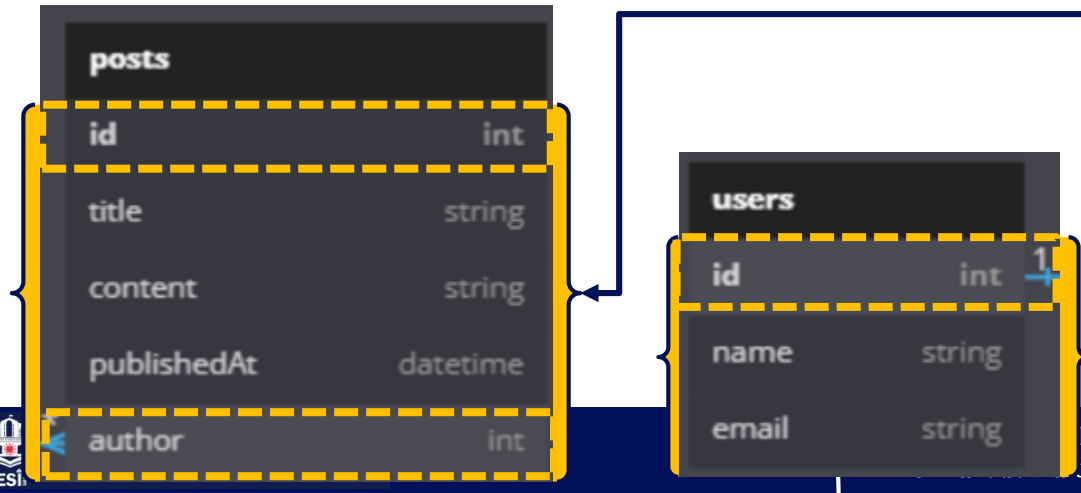
- **Endpoint logic Layer: Pydantic DB Schemas and CRUD Utilities:**
  - Extend **Pydantic** models to be used when reading data returned to Client.
    - Behaviour of **pydantic** can be controlled via the **Config class** ➔ Used to provide configurations to **Pydantic**.
    - **Pydantic**'s **orm_mode** will tell the **Pydantic model** to read the data as an ORM model.



schemas.py

```
#create Pydantic models that will be used when reading data
(returning it from the API)
class Post(PostBase):
    id: int
    author: int

    class Config:
        orm_mode = True
```

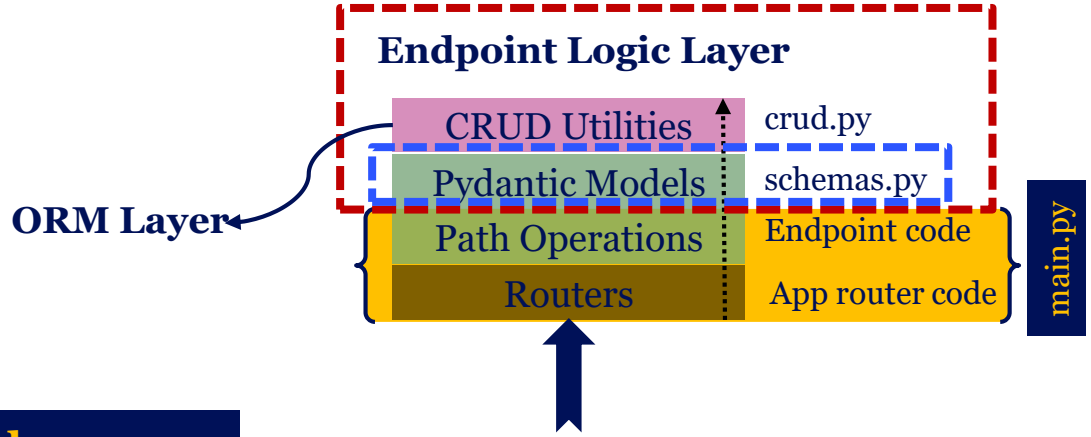
```
class User(UserBase):
    id: int

    class Config:
        orm_mode = True
```



Building Restful API with FastAPI - Part II

Asynchronous ORMs

- **Endpoint logic Layer: Pydantic DB Schemas and CRUD Utilities:**
  - Extend **Pydantic** models to be used when creating data.
    - Allows us to separate fields which are only relevant for the **DB**, or which we don't want to return to the client (such as a password field)



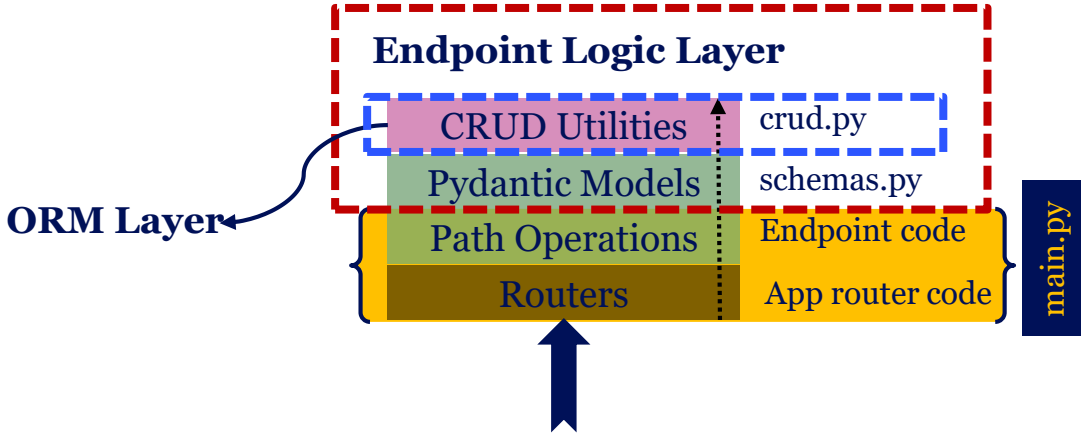
schemas.py

```
#create Pydantic models needed for creation purposes
class PostCreate(PostBase):
    pass

class UserCreate(UserBase):
    posts: list[Post] = []
    pass
```

Building Restful API with FastAPI - Part II

Asynchronous ORMs

- **Endpoint logic Layer: Pydantic DB Schemas and CRUD Utilities:**
  - **CRUD Utilities** ➔ Contain reusable **functions** to interact with the data in the database.



crud.py

```
from sqlalchemy.orm import Session
from . import models, schemas
```

ORM-level SQL construction functions

Fetch First Row using first() ORM Query function

SELECT * FROM users where id=:user_id

```
def get_user(db: Session, user_id: int):
    return db.query(models.User).filter(models.User.id == user_id).first()
```

Fetch First Row using first() ORM Query function

SELECT * FROM users where email=:email

```
def get_user_by_email(db: Session, email: str):
    return db.query(models.User).filter(models.User.email == email).first()
```

Fetch All Rows using all() ORM Query function

SELECT * FROM users

```
def get_users(db: Session, skip: int = 0, limit: int = 100):
    return db.query(models.User).offset(skip).limit(limit).all()
```

Fetch All Rows using all() ORM Query function

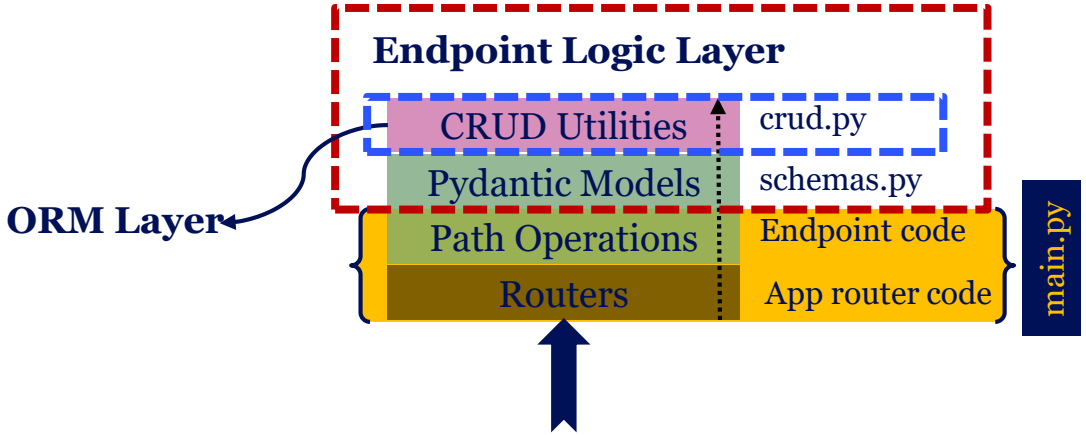
SELECT * FROM posts

```
def get_posts(db: Session, skip: int = 0, limit: int = 100):
    return db.query(models.Post).offset(skip).limit(limit).all()
```

Building Restful API with FastAPI - Part II

Asynchronous ORMs

- **Endpoint logic Layer: Pydantic DB Schemas and CRUD Utilities:**
  - **CRUD Utilities** ➔ Contain reusable **functions** to interact with the data in the database.



crud.py

ORM-level SQL construction functions

```
def create_user(db: Session, user: schemas.UserCreate):
    db_user = models.User(email=user.email,
                          name=user.name)
    # instance object added to database session
    db.add(db_user)
    # changes are saved to the database
    db.commit()
    # instance is refreshed to contain any new data from the database, like
the generated ID
    db.refresh(db_user)
    return db_user
```

Add a new row using add() ORM function

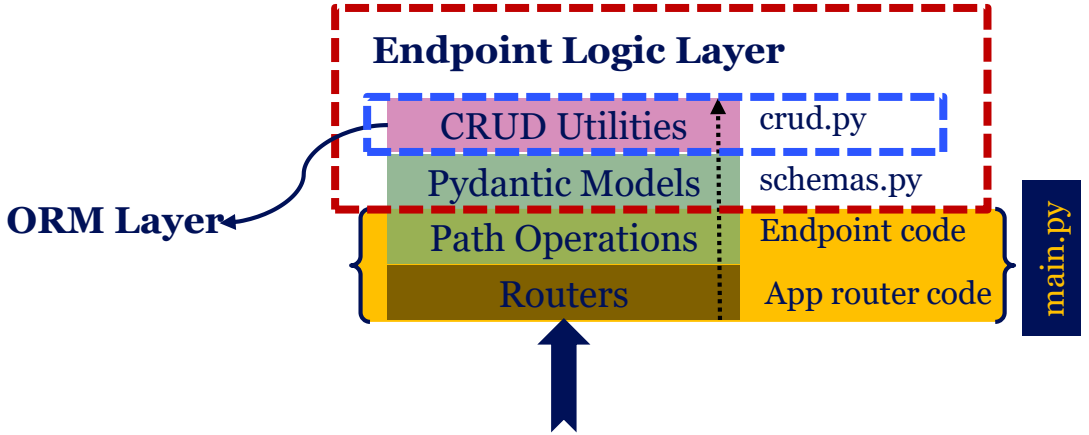
```
INSERT INTO posts
VALUES (:user.id,
        :user.email,
        :user.name)
```

:user.id will be automatically populated using the **autoincrement option** defined while creating ORM models in **models.py**

Building Restful API with FastAPI - Part II

Asynchronous ORMs

- **Endpoint logic Layer: Pydantic DB Schemas and CRUD Utilities:**
  - **CRUD Utilities** ➔ Contain reusable **functions** to interact with the data in the database.



crud.py

ORM-level SQL construction functions

```
def create_user_post(db: Session, post: schemas.PostCreate, user_id: int):
    db_post = models.Post(**post.dict(), author=user_id)
    db.add(db_post)
    db.commit()
    db.refresh(db_post)
    return db_post
```

Add a new row using add() ORM function

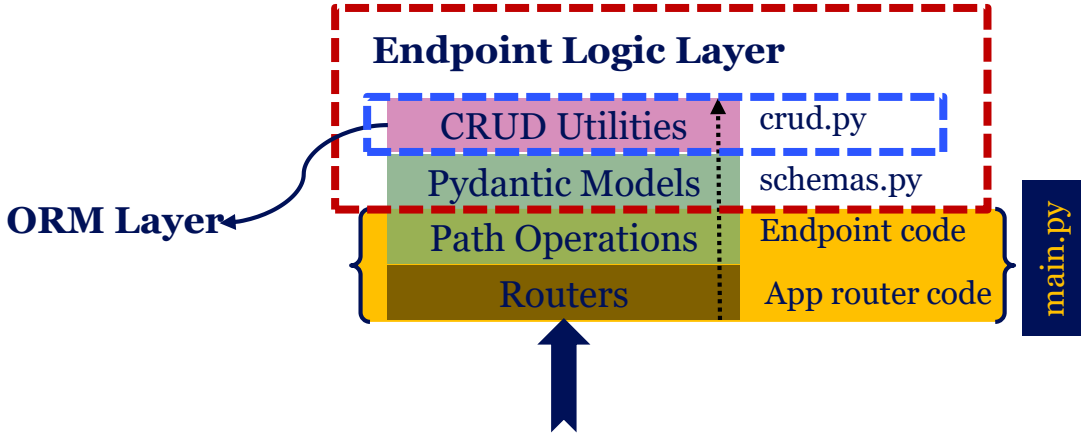
```
INSERT INTO users
VALUES (:post.id,
        :post.title,
        :post.content,
        :publishedAt,
        :post.author
)
```

- **:post.id** will be automatically populated using the **autoincrement option** defined while creating ORM models in **models.py**
- **:post.author** will be automatically popuplated from the **author=user_id** parameter in the **create_user_post()** function

Building Restful API with FastAPI - Part II

Asynchronous ORMs

- **Endpoint logic Layer: Pydantic DB Schemas and CRUD Utilities:**
  - **CRUD Utilities** ➔ Contain reusable **functions** to interact with the data in the database.



crud.py

ORM-level SQL construction functions

```
def delete_user_post(db: Session,
                     post: schemas.Post
                      ):
    db.delete(post)
    db.commit()
```

Delete a row using delete() ORM function

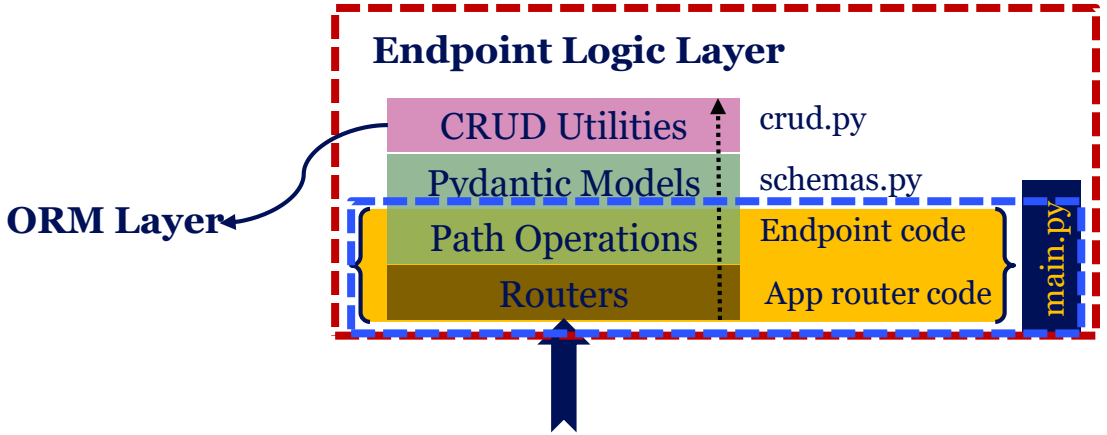
```
DELETE FROM posts
WHERE id=:post.id
```

- Deleting the current post in the session **(:post.id)**

Building Restful API with FastAPI - Part II

Asynchronous ORMs

- **Endpoint logic Layer: Routers and Path Operations:**
  - All requests are routed to the correct path operations (i.e. the function for handling it, such as our root function in **main py** file)



main.py

```
from fastapi import Depends, FastAPI, HTTPException, Response, status
from sqlalchemy.orm import Session

from . import crud, models, schemas
from .database import SessionLocal, engine
```

In a very simplistic way create the **database tables**

```
models.Base.metadata.create_all(bind=engine)
```

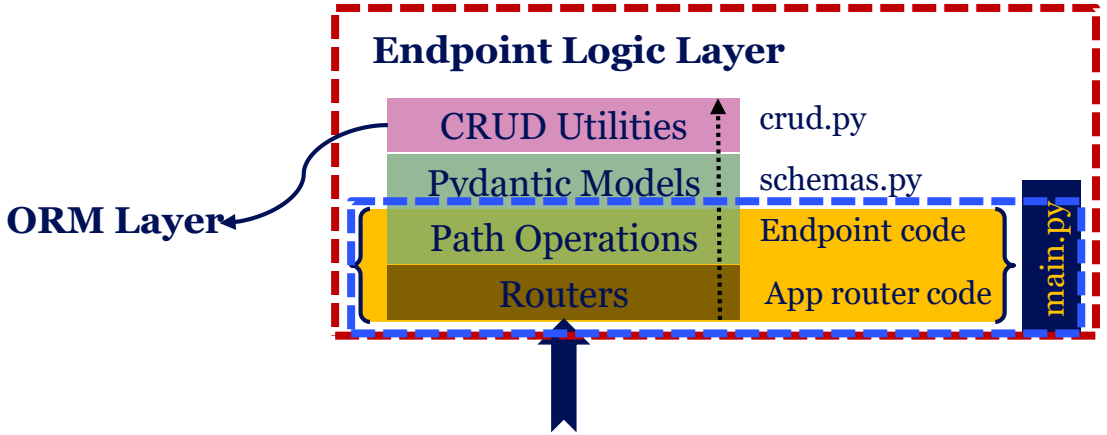
Instantiating a **FastAPI** object handling all **API routes**

```
app = FastAPI()
```

Building Restful API with FastAPI - Part II

Asynchronous ORMs

- **Endpoint logic Layer: Routers and Path Operations:**
  - All requests are routed to the correct path operations (i.e. the function for handling it, such as our root function in **main py** file)



main.py

```
# Dependency
def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()
```

Dependency ➔ Create a new **SQLAlchemy SessionLocal** to be used in a single request, and then to be closed once the request is finished.

- Only the code prior to and including the **yield** statement is executed before sending a response.
- The **yielded** value is what is injected into path operations.
- The code following the **yield** statement is executed after the response has been delivered.

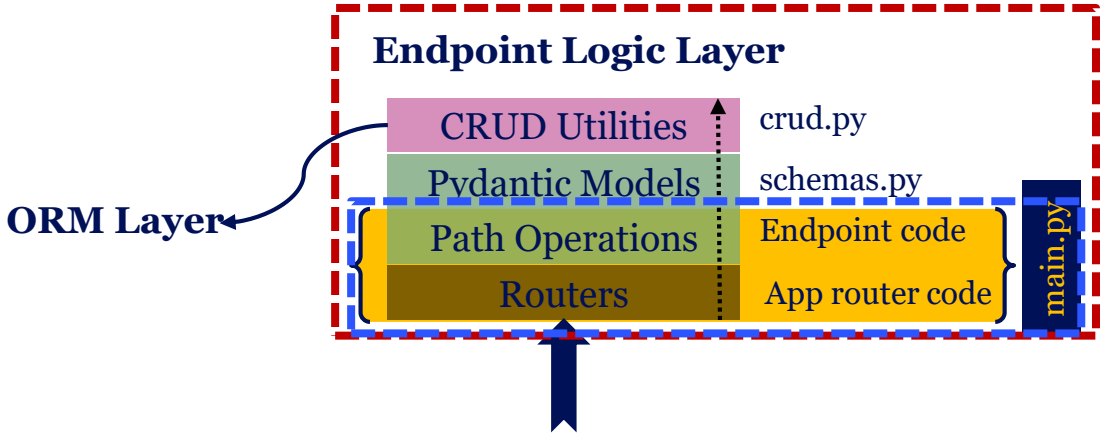
Define a **router (/users/)** associated with its **Path operation function (get_users)**

```
@app.get("/users/", response_model=list[schemas.User])
async def get_users(skip: int = 0, limit: int = 100, db: Session =
Depends(get_db)):
    users = crud.get_users(db, skip=skip, limit=limit)
    return users
```

Building Restful API with FastAPI - Part II

Asynchronous ORMs

- **Endpoint logic Layer: Routers and Path Operations:**
  - All requests are routed to the correct path operations (i.e. the function for handling it, such as our root function in **main py** file)



main.py

Define a **router (/users/{user_id})** associated with its **Path operation function (get_user_by_id)**

```
@app.get("/users/{user_id}", response_model=schemas.User)
async  def get_user_by_id(user_id: int, db: Session = Depends(get_db)):
    db_user = crud.get_user(db, user_id=user_id)
    if db_user is None:
        raise HTTPException(status_code=404, detail="User not found")
    return db_user
```

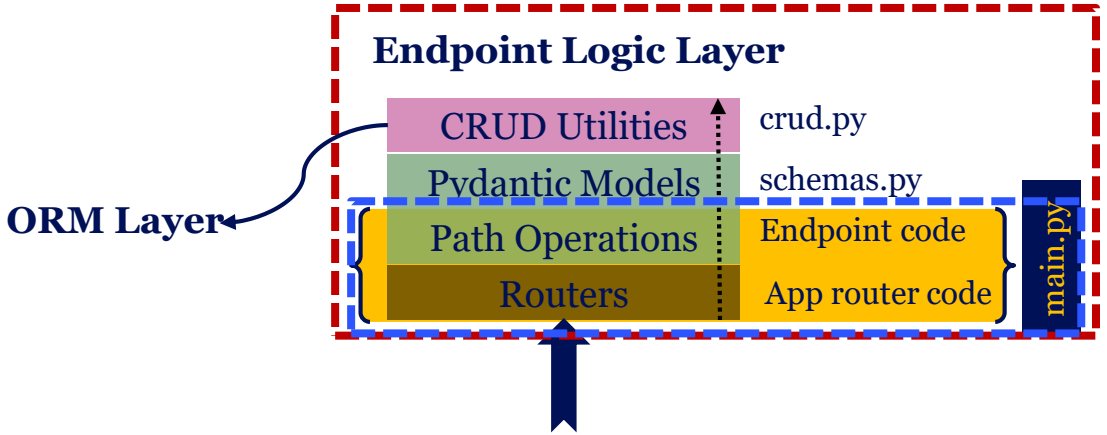
Define a **router (/users/{user_id}/posts/)** associated with its **Path operation function (get_user_posts)**

```
@app.get("/users/{user_id}/posts/", response_model=list[schemas.Post])
async  def get_user_posts(user_id: int, skip: int = 0, limit: int = 100, db:
Session = Depends(get_db)):
    db_user = crud.get_user(db, user_id=user_id)
    if db_user is None:
        raise HTTPException(status_code=404, detail="User not found")
    posts = crud.get_user_posts(db,user_id,skip, limit)
    return posts
```

Building Restful API with FastAPI - Part II

Asynchronous ORMs

- **Endpoint logic Layer: Routers and Path Operations:**
  - All requests are routed to the correct path operations (i.e. the function for handling it, such as our root function in **main py** file)



main.py

Define a **router (/users/new)** associated with its **Path operation function (create_user)**

```
@app.post("/users/new", response_model=schemas.User)
async  def create_user(user: schemas.UserCreate, db: Session =
Depends(get_db)):
    db_user = crud.get_user_by_email(db, email=user.email)
    if db_user:
        raise HTTPException(status_code=400, detail="Email already
registered")
    return crud.create_user(db=db, user=user)
```

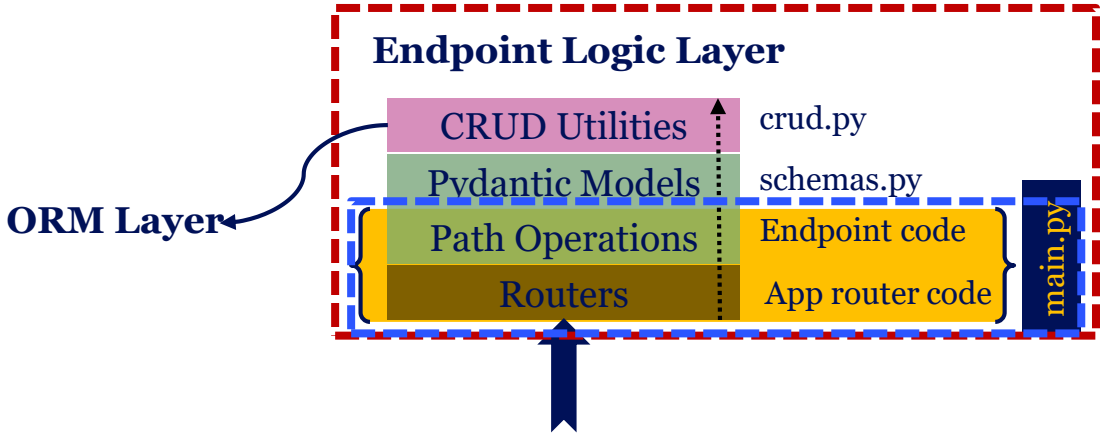
Define a **router (/users/{user_id}/posts/new)** associated with its **Path operation function (create_post_for_user)**

```
@app.post("/users/{user_id}/posts/new", response_model=schemas.Post)
async  def create_post_for_user(user_id: int,
                                post: schemas.PostCreate,
                                db: Session = Depends(get_db)
                               ):
    return crud.create_user_post(db=db, post=post, user_id=user_id)
```

Building Restful API with FastAPI - Part II

Asynchronous ORMs

- **Endpoint logic Layer: Routers and Path Operations:**
  - All requests are routed to the correct path operations (i.e. the function for handling it, such as our root function in **main py** file)



Define a **router (/users/{user_id}/delete_post/{post_id})** associated with its **Path operation function (delete_post_for_user)**

main.py

```
@app.delete("/users/{user_id}/delete_post/{post_id}")
async  def delete_post_for_user(user_id: int,
                                post_id: int,
                                db: Session = Depends(get_db)):
    db_post =  db.query(models.Post).filter(models.Post.author == user_id,
models.Post.id == post_id).first()
    if db_post is None:
        raise HTTPException(status_code=404, detail="User or Post not
found")

    crud.delete_user_post(db=db, post=db_post)
    return  {"msg": "Successfully Deleted"}
```

Thanks for your Listening

Dr. Belkacem KHALDI
e-mail: b.khaldi@esi-sba.dz