5:50:10

Creating User Functionality :-

- Creating user functionality means building elements such that users can create account, log in, and create posts, that will be associated with their own accounts.
Thus, it means handling user registrations.

- We first need a database table, to store all the registered user information. We are using ORM, so, we need to create an ORM model for the User's table.

```
class User(Base):
    __tablename__ = "users"
    id = Column(Integer, primary_key=True, nullable=False)
    email = Column(String, unique=True, nullable=False)
    password = Column(String, nullable=False)
    created_at = Column(TIMESTAMP(timezone=True), ......)
```

Upon running the project, due to this above model, a table named **users** will be created.

- Now, we need to define Path Operations. For that, we need schema (Pydantic models) to specify the format of data sent in request.

Schema for user request

for this, we need email-validator installed.

```
from pydantic import BaseModel, EmailStr

class UserCreate(BaseModel):
    email: EmailStr
    password: str
```

This EmailStr type will validate and ensure that the passed string is of proper email format, and not just some random piece of text.

⇒ With all these, we can now create the path operation for creating a new user, just like creating a new post. Take the data from user in path parameter, in schema format, and then, using it, create an ORM model object, and then add it to the database table using db.add(new_user_orm_obj) and db.commit() methods.

⇒ Now, the password of user's must not be stored as plain text, — it must be hashed first, and stored in hashed format.

To hash passwords, we need to install 2 python libraries:- passlib & Bcrypt 6:04:50

passlib handles the hashing. It can work with different algorithms, and we need to specify the algo.

Bcrypt is a hashing algorithm (most popular).
Installing the library with algorithm :-

```
pip install "passlib[bcrypt]"
```

Now, to use it, in our Python file for password hashing, we need following code in that file :-

```
from passlib.context import CryptContext

pwd_context = CryptContext(schemes=["bcrypt"],
                           ...deprecated="auto")
```

By the code above, we are specifying the default hashing algorithm for passlib.

=> Now, using the pwd_context object created above, we can hash and store the password in path operation as follow :-

```python
@app.post("/users", status_code=status.HTTP_201_CREATED, response_model=schemas.UserOut)
def create_user(user: schemas.UserCreate, db: Session = Depends(get_db)):

    #hash the password - user.password
    hashed_password = pwd_context.hash(user.password)
    user.password = hashed_password
    new_user = models.User(**user.dict())
    db.add(new_user)
    db.commit()
    db.refresh(new_user)

    return new_user
```

=> Using the hash() method of the pwd_context object, we can pass the raw text password, and get the

hashed password in return.

⇒ For organising our project, we will seperate all the hashing code & logic into a function in seperate file, say utils.py, and import the function and use it in main file to hash passwords.
Thus, code will be organised.

In utils.py :-

```
from passlib.context import CryptContext
pwd_context = CryptContext(schemes = "bcrypt",
                           deprecated = "auto")

def hash(password):
    return pwd_context.hash(password)
```

⇒ Now, we need to set up path operation, to fetch user details upon passing the user id.
This may be required when a logged in user wants to know details about his own account, or others account (such as we do in social media).

It will be simply taking user id as path parameter, & fetch user corresponding to id, and if user exists, we will simply return the user's data, in format of response_model (schemas.UserResponse in our case).