Here is the line-by-line breakdown of `schemas.py`.

Think of this file as the **"Security Guard"** or the **"Bouncer"** of your API.

- **Models (`models.py`)** = What the **Database** looks like.
- **Schemas (`schemas.py`)** = What the **JSON** (API request/response) looks like.

This file decides:

1. **Validation:** "Is the user sending the right data?" (Input)
2. **Filtering:** "What data should we show the user?" (Output)

---

1. The Setup (Tools)

```
from pydantic import BaseModel, EmailStr
from typing import Optional
```

- `pydantic`: This is the library that does the heavy lifting. It validates data types automatically.
- `BaseModel`: The parent class for all schemas. Just like `models.Base`, you inherit from this to get superpowers.
- `EmailStr`: A special type that automatically checks if a string is a valid email (has `@` and `.`). If a user sends "hello", Pydantic rejects it immediately.

---

2. The Post Schemas

Input Schema (Creating a Post)

```
class Post(BaseModel):
    title: str
    content: str
```

- **Purpose:** This is what the user **MUST send** when creating a post.
- **Missing `id`?** Yes. Users don't pick their ID; the database assigns it.
- **Missing `owner_id`?** Yes. The code assigns this automatically based on who is logged in.

Output Schema (Viewing a Post)

```
class PostResponse(BaseModel):
    id: int
    title: str
    content: str
    owner_id: int
```

```
    class Config:
        orm_mode = True
```

- **Purpose:** This is what the user **GETS back**.
- **Added Fields:** We add `id` and `owner_id` because now they exist in the database and we want to show them.
- `class Config: orm_mode = True`: **(Very Important)**
- **The Problem:** Pydantic expects a dictionary (`{"title": "..."}`). But SQLAlchemy returns a Python Object (`post.title`).
- **The Fix:** `orm_mode = True` tells Pydantic: *"It's okay if the data isn't a dictionary. Just use `object.title` to read the data."*
- **Without this:** Your API would crash with `value is not a valid dict` error.

---

3. The User Schemas

Input Schema (Registration)

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

- Standard stuff. Notice `password` is here. We need it to create the account.

Output Schema (Profile View)

```
class UserResponse(BaseModel):
    id: int
    username: str
    email: str
    is_active: bool

    class Config:
        orm_mode = True
```

- **Filtering (The "Security" Feature):**
- Notice that **`password` is MISSING** here.
- This is intentional! When you send a user profile back to the frontend, Pydantic will strip out the password field automatically.
- This prevents you from accidentally leaking user passwords.

Login Schema

```
class UserLogin(BaseModel):
    email : EmailStr
    password:str
    class Config:
        orm_mode = True
```

- **Purpose:** Defines what the `/login` endpoint expects.
- **Note:** You actually don't strictly *need* `orm_mode = True` here because login data usually comes as pure JSON (a dictionary), not a database object, but keeping it doesn't hurt.

---

4. The Token Schemas

```
class Token(BaseModel):
    access_token : str
    token_type : str
```

- This is exactly what your login endpoint returns: `{"access_token": "ey...", "token_type": "bearer"}`

```
class TokenData(BaseModel):
    id : Optional[str] = None
```

- **Purpose:** This is used internally during validation.
- When we decode the JWT token, we extract the `id` and stick it into this schema to ensure the token actually contained an ID.

Summary of the Flow

1. **User Sends JSON Pydantic (`Post`)** checks it.
2. **Code** saves it to Database (`models.Post`).
3. **Database** returns an Object (`post_object`).
4. **Pydantic (`PostResponse`)** converts that Object back to JSON (and hides secrets).

Ready for the next file? (Probably `routers/user.py` or `utils.py` is good now).