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

Think of this file as the **Architect**. It draws the blueprints for your database tables. It defines *what* data you are storing and *how* it connects.

1. The Setup (Imports)

```
from sqlalchemy import Column, Integer, String, Boolean, ForeignKey
from sqlalchemy.orm import relationship
from .database import Base
```

- `sqlalchemy` **imports**: These are the building blocks. You need specific types for your columns (`Integer` for IDs, `String` for text, `Boolean` for True/False).
- `relationship`: This is the "Magic" tool we discussed earlier that lets you access data like objects (`user.posts`) instead of writing raw SQL.
- `from .database import Base`: This imports the "Registry". By inheriting from `Base`, your classes tell SQLAlchemy: "I am a database table, please create me."

---

2. The User Table ("The Parent")

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

- `class User(Base)`: Defines a Python class. Because it inherits from `Base`, it becomes a SQL table.
- `__tablename__ = "users"`: This is the actual name of the table inside Postgres. (In SQL, you will query `SELECT * FROM users`).

```
    id = Column(Integer, primary_key=True, index=True)
```

- `primary_key=True`: This makes the ID the unique identifier. It automatically increments (1, 2, 3...) every time you add a user.
- `index=True`: This creates a "search index". It makes looking up a user by ID *extremely fast*.

```
    username = Column(String, unique=True, index=True, nullable=False)
    email = Column(String, unique=True, index=True, nullable=False)
```

- `unique=True`: Ensures no two users can have the same username or email. If you try, the database will throw an error.
- `nullable=False`: This means this field **cannot be empty**. You must provide a username/email.

```
    password = Column(String, nullable=False)
```

- Stores the **Hashed** password (the gibberish string), not the real password.

```
    is_active = Column(Boolean, server_default='true')
```

- `server_default='true'`: If you create a user but forget to say if they are active, the *Database itself* will set this to True automatically.

```
    posts = relationship("Post", back_populates="owner")
```

- **The Python Link:** This creates the "virtual list" of posts.
- It tells Python: "Find all items in the `Post` table that belong to this user and give them to me as a list `[]`."

---

3. The Post Table ("The Child")

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

    id = Column(Integer, primary_key=True, index=True)
    title = Column(String, index=True)
    content = Column(String)
```

- Standard columns. Note that `content` does not have `index=True` because searching through huge paragraphs of text is slow and heavy, so we usually don't index the main body content unless necessary.

```
    owner_id = Column(Integer, ForeignKey("users.id"))
```

- **The Anchor (Crucial):** This is the physical link in the database.

- `ForeignKey("users.id")`: This enforces a rule: "The number saved in this column **MUST** exist in the `id` column of the `users` table."
- If you try to create a post with `owner_id=999` and User 999 doesn't exist, the database will reject it.

```
    owner = relationship("User", back_populates="posts")
```

- **The Python Link:** This lets you say `my_post.owner` to get the User object who wrote the post.
- `back_populates="posts"`: This connects the two relationship lines together so they stay in sync.

Summary

- `User` **class** = `users` table.
- `Post` **class** = `posts` table.
- `ForeignKey` = The rule that ties a post to a user ID.
- `relationship` = The Python convenience tool to fetch linked data easily.

Which file would you like to do next? `schemas.py` is usually the best next step, as it handles the data *before* it gets to these models.