**Hour 1:**

**🧩 SQLAlchemy + FastAPI — Revision Notes**

**🔹 1. Model (in models.py)**

**✅ What it is:**

A **Python class** that represents a **database table**.

**✅ How to define:**

from sqlalchemy import Column, Integer, String

from sqlalchemy.ext.declarative import declarative\_base

Base = declarative\_base()

class User(Base):

\_\_tablename\_\_ = "users" # Table name in DB

id = Column(Integer, primary\_key=True, index=True)

name = Column(String)

email = Column(String, unique=True, index=True)

**✅ Key Points:**

* Base → parent class for all models.
* Each Column() = one column in the DB table.
* \_\_tablename\_\_ → the actual table name in the database.
* This approach = **Code First** (SQLAlchemy creates DB tables from code).
* Importing the model is **required** before calling Base.metadata.create\_all().

**⚙️ Column Arguments:**

| **Argument** | **Meaning** |
| --- | --- |
| primary\_key=True | Marks it as the main unique ID |
| unique=True | No duplicate values allowed |
| index=True | Speeds up searching |
| nullable=False | Field cannot be empty |

**🔹 2. Database Connection (in database.py)**

**✅ Purpose:**

To connect Python to the actual database and create a **session** to interact with it.

**✅ Example:**

from sqlalchemy import create\_engine

from sqlalchemy.orm import sessionmaker

SQLALCHEMY\_DATABASE\_URL = "sqlite:///./test.db"

engine = create\_engine(

SQLALCHEMY\_DATABASE\_URL,

connect\_args={"check\_same\_thread": False} # Only for SQLite

)

SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

**✅ Key Points:**

* sqlite:///./test.db → SQLite file named test.db in your current folder.
* You **don’t need to manually create** this file — SQLAlchemy creates it when you call Base.metadata.create\_all(bind=engine).
* connect\_args={"check\_same\_thread": False} → allows FastAPI to use the same SQLite DB in multiple threads.
* sessionmaker() → makes sessions (connections) to talk to the DB.

**⚙️ sessionmaker Parameters:**

| **Parameter** | **Meaning** |
| --- | --- |
| autocommit=False | You must call .commit() manually to save changes |
| autoflush=False | Prevents auto sync; gives you more control |
| bind=engine | Tells session which DB to use |

**🔹 3. Creating Tables (in main.py)**

**✅ Code:**

from models import Base, User

from database import engine

Base.metadata.create\_all(bind=engine)

**✅ Key Points:**

* Base.metadata.create\_all() scans all models that inherit from Base.
* You **must import** your model (e.g., User) before calling this — otherwise it won’t be registered.
* It automatically creates all the tables defined in your models.

**🔹 4. Using Session to Add & Query Data**

**✅ Example:**

from models import User

from database import SessionLocal

db = SessionLocal() # Create session

# ➕ Add data

new\_user = User(name="Alice", email="alice@mail.com")

db.add(new\_user)

db.commit() # Save changes

db.refresh(new\_user) # Get the new ID from DB

# 🔍 Query data

users = db.query(User).all()

for user in users:

print(user.id, user.name, user.email)

db.close() # Close connection

**✅ Flow Summary:**

1. Open session
2. Add data using model
3. Commit to save
4. Query to read
5. Close session

**🔹 5. Schema (in schemas.py)**

**✅ Purpose:**

To define **how data is received from and sent to the user** in FastAPI routes.

**✅ Example:**

from pydantic import BaseModel

class UserCreate(BaseModel): # Input Schema

name: str

email: str

class UserResponse(BaseModel): # Output Schema

id: int

name: str

email: str

class Config:

orm\_mode = True

**✅ Key Points:**

* Schemas are **Pydantic models** used for data validation.
* UserCreate → used when creating a new user (input).
* UserResponse → used when returning data to user (output).
* orm\_mode = True → allows conversion from SQLAlchemy model objects to Pydantic objects.

**🔹 6. How Everything Connects (Full Flow)**

| **Step** | **File** | **Description** |
| --- | --- | --- |
| 1 | models.py | Defines your database tables |
| 2 | database.py | Creates connection & session to DB |
| 3 | schemas.py | Defines how data enters & leaves API |
| 4 | main.py | Combines all: creates tables, runs app, handles CRUD |

**🧠 Key Concepts to Remember**

* **Model = Database Table Structure**
* **Schema = API Request/Response Format**
* **Session = Bridge between Python & Database**
* **Base.metadata.create\_all()** → Creates tables based on models
* **Import Models Before Create** → Otherwise no table is made
* **commit()** → Saves changes permanently
* **orm\_mode=True** → Converts ORM objects → JSON responses

**🧩 Bonus Tip**

If your FastAPI app uses SQLite, you’ll see the file:

test.db

in your project folder after the first run.  
You can open it using tools like **DB Browser for SQLite** to view your tables.

Would you like me to turn these notes into a **PDF file** (easy to download and review anytime)?

**Hour 2**

**🕐 Hour 2 — Creating First Model, Schema & Testing API**

**🧱 Step 1 — Create Your First Model (SQLAlchemy)**

**Purpose:** Define what your database table looks like.  
➡️ Models represent **database structure** (tables, columns).

Example:

# models.py

from sqlalchemy import Column, Integer, String

from database import Base

class User(Base):

\_\_tablename\_\_ = "users"

id = Column(Integer, primary\_key=True, index=True)

name = Column(String)

email = Column(String)

✅ This creates a **users table** with 3 columns: id, name, and email.

**🧩 Step 2 — Create Matching Pydantic Schema**

**Purpose:** Define how data will be **validated & returned** through the API.  
➡️ Schemas are used for **input and output**.

Example:

# schemas.py

from pydantic import BaseModel

from pydantic import BaseModel

class userCreate(BaseModel):

    name: str

    email: str

class userResponse(BaseModel):

    id: int

    name: str

    email: str

    class Config:

        orm\_mode = True

✅ UserSchema (with orm\_mode=True) allows conversion from SQLAlchemy objects → JSON.  
✅ UserCreate is used when **creating a new user** (input).  
✅ UserSchema is used when **returning data** (output).

**⚙️ Step 3 — Create the Table in Database**

Add this line in main.py:

Base.metadata.create\_all(bind=engine)

✅ It ensures that your users table is created automatically if it doesn’t exist.

**🚀 Step 4 — Create Your First Route**

Example route to **read all users**:

@app.get("/users/", response\_model=list[UserSchema])

def read\_users():

users = db.query(User).all()

return users

✅ response\_model=list[UserSchema]  
→ Tells FastAPI to automatically convert each SQLAlchemy object to JSON using Pydantic.

✅ No manual conversion needed — FastAPI + Pydantic handle it automatically.

**🧠 Concept Summary**

| **Concept** | **Description** |
| --- | --- |
| **Model** | Defines structure of table in database |
| **Schema** | Defines structure of data in API (input/output) |
| **orm\_mode=True** | Allows Pydantic to read SQLAlchemy ORM objects |
| **response\_model** | Converts ORM object → JSON automatically |
| **Base.metadata.create\_all()** | Creates database tables from models |

**✅ End Result**

When you visit /users/, you’ll get a JSON response like:

[

{"id": 1, "name": "Ahmed", "email": "Ahmed1@gmail.com"},

{"id": 2, "name": "Ali", "email": "ali@gmail.com"}

]

Even though you returned **SQLAlchemy objects**,  
FastAPI + Pydantic automatically converted them to **JSON** 🎯

Would you like me to write **Hour 3 notes next** (where we add a POST route to create users)?

**Hour 3**

**🧩 Hour 3 — CRUD Operations (POST, GET, DELETE)**

**1️⃣ Concepts Learned**

| **Concept** | **Explanation** |
| --- | --- |
| **CRUD** | Create, Read, Update, Delete operations on database objects |
| **Dependency Injection** | FastAPI feature that injects resources (like DB session) into endpoint using Depends(get\_db) |
| **Pydantic Models as Response** | Automatically validates input and converts SQLAlchemy objects to JSON using response\_model |
| **SQLAlchemy ORM** | Query and manipulate database using Python classes instead of raw SQL |

**2️⃣ Database Session with Dependency Injection**

* get\_db() function creates a **new session per request**:

def get\_db():

db = SessionLocal() # create a new DB session

try:

yield db # pause function, provide session to route

finally:

db.close() # close session automatically after route finishes

**Key Points:**

* yield allows FastAPI to **pause and resume** for cleanup (db.close())
* Each API call gets its **own session**
* Without Depends(get\_db), you’d need to manually create/close sessions in every route

**3️⃣ CRUD Routes**

**Create User (POST)**

@app.post("/users", response\_model=schemas.UserResponse)

def create\_user(user: schemas.UserCreate, db: Session = Depends(get\_db)):

existing\_user = db.query(models.User).filter(models.User.email == user.email).first()

if existing\_user:

raise HTTPException(status\_code=400, detail="Email already registered")

new\_user = models.User(name=user.name, email=user.email)

db.add(new\_user)

db.commit()

db.refresh(new\_user) # load generated ID

return new\_user

✅ **Key Points**

* db.add() → stage object for insertion
* db.commit() → persist in DB
* db.refresh() → reload object with generated fields (like id)
* response\_model converts SQLAlchemy object → JSON using Pydantic

**Get All Users (GET)**

@app.get("/users", response\_model=list[schemas.UserResponse])

def get\_users(db: Session = Depends(get\_db)):

return db.query(models.User).all()

* Returns **all users**
* SQLAlchemy .all() → returns list of objects
* FastAPI + Pydantic → automatically converts to JSON

**Get Single User by ID (GET)**

@app.get("/users/{id}", response\_model=schemas.UserResponse)

def get\_user(id: int, db: Session = Depends(get\_db)):

user = db.query(models.User).filter(models.User.id == id).first()

if not user:

raise HTTPException(status\_code=404, detail="User not found")

return user

* .filter() → add conditions like WHERE id=...
* .first() → returns first match or None

**Delete User by ID (DELETE)**

@app.delete("/users/{id}")

def delete\_user(id: int, db: Session = Depends(get\_db)):

user = db.query(models.User).filter(models.User.id == id).first()

if not user:

raise HTTPException(status\_code=404, detail="User not found")

db.delete(user)

db.commit()

return {"message": "User deleted successfully"}

* db.delete() → marks object for deletion
* db.commit() → persist changes

**4️⃣ Execution Flow for CRUD**

1. FastAPI receives request
2. Input validated by Pydantic schema (UserCreate)
3. db: Session = Depends(get\_db) → injects DB session
4. SQLAlchemy ORM executes query (add, query, delete)
5. commit → write changes
6. db.refresh → update object with generated fields (optional)
7. FastAPI returns **response in JSON** (using response\_model)

**5️⃣ Tips / Gotchas**

* **Always use db.refresh(obj) after commit** to get updated object fields
* **Depends(get\_db) ensures automatic session cleanup** — prevents connection leaks
* **Pydantic response\_model converts ORM objects to JSON**, no manual dict conversion needed
* **Check for existing objects** (like email) to avoid duplicates
* **StopIteration concept** for yield in get\_db() ensures db.close() runs after request

This set of notes covers **all main concepts from Hour 3**: dependency injection, session usage, CRUD operations, and the integration with Pydantic schemas.

If you want, I can make a **combined, super-condensed cheat sheet** for **Hours 1–3** so you can revise everything in **1 page**.

Do you want me to do that?