FastAPI Sections 5, 6 & 7 - Complete Guide with Examples

Section 5: Python + Raw SQL

1. Setup App Database

Install Required Dependencies:

```
bash

pip install psycopg2-binary
```

Database Configuration:

```
python

# config.py
DATABASE_HOSTNAME = "localhost"
DATABASE_PORT = "5432"
DATABASE_PASSWORD = "your_password"
DATABASE_NAME = "fastapi"
DATABASE_USERNAME = "postgres"
```

Create Database Table:

```
sql

-- In pgAdmin or psql
CREATE TABLE posts (
    id SERIAL PRIMARY KEY,
    title VARCHAR NOT NULL,
    content VARCHAR NOT NULL,
    published BOOLEAN NOT NULL DEFAULT TRUE,
    created_at TIMESTAMP WITH TIME ZONE NOT NULL DEFAULT NOW()
);
```

2. Connecting to Database with Python

Connection Setup:

python

```python
import psycopg2
from psycopg2.extras import RealDictCursor
import time

def connect_to_db():
    while True:
        try:
            conn = psycopg2.connect(
                host='localhost',
                database='fastapi',
                user='postgres',
                password='your_password',
                cursor_factory=RealDictCursor
            )
            cursor = conn.cursor()
            print("Database connection successful!")
            return conn, cursor
        except Exception as error:
            print(f"Connection failed: {error}")
            time.sleep(2)

# Global connection
conn, cursor = connect_to_db()
```

Complete main.py Setup:

python

```python
from fastapi import FastAPI, HTTPException, status, Response
from pydantic import BaseModel
import psycopg2
from psycopg2.extras import RealDictCursor
import time

app = FastAPI()

# Database connection
while True:
    try:
        conn = psycopg2.connect(
            host='localhost',
            database='fastapi',
            user='postgres',
            password='your_password',
            cursor_factory=RealDictCursor
        )
        cursor = conn.cursor()
        print("Database connection successful!")
        break
    except Exception as error:
        print(f"Connection failed: {error}")
        time.sleep(2)

class Post(BaseModel):
    title: str
    content: str
    published: bool = True
```

3. Retrieving Posts

python

```python
@app.get("/posts")
def get_posts():
    cursor.execute("SELECT * FROM posts")
    posts = cursor.fetchall()
    return {"data": posts}
```

Postman Test:

- Method: GET
- URL: `http://localhost:8000/posts`
- Expected Response: Array of post objects

4. Creating Post

```python
@app.post("/posts", status_code=status.HTTP_201_CREATED)
def create_post(post: Post):
    cursor.execute(
        """INSERT INTO posts (title, content, published)
           VALUES (%s, %s, %s) RETURNING *""",
        (post.title, post.content, post.published)
    )
    new_post = cursor.fetchone()
    conn.commit()
    return {"data": new_post}
```

Postman Test:

- Method: POST
- URL: `http://localhost:8000/posts`
- Headers: `Content-Type: application/json`
- Body (JSON):

```json
{
    "title": "My First Post",
    "content": "This is the content of my first post",
    "published": true
}
```

5. Get One Post

```python
@app.get("/posts/{id}")
def get_post(id: int):
    cursor.execute("SELECT * FROM posts WHERE id = %s", (id,))
    post = cursor.fetchone()

    if not post:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail=f"Post with id {id} was not found"
        )

    return {"data": post}
```

Postman Test:

- Method: GET
- URL: `http://localhost:8000/posts/1`
- Expected Response: Single post object or 404 error

6. Delete Post

```python
@app.delete("/posts/{id}", status_code=status.HTTP_204_NO_CONTENT)
def delete_post(id: int):
    cursor.execute("DELETE FROM posts WHERE id = %s RETURNING *", (id,))
    deleted_post = cursor.fetchone()
    conn.commit()

    if not deleted_post:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail=f"Post with id {id} was not found"
        )

    return Response(status_code=status.HTTP_204_NO_CONTENT)
```

Postman Test:

- Method: DELETE
- URL: `http://localhost:8000/posts/1`

- Expected Response: 204 No Content or 404 error

7. Update Post

```python
@app.put("/posts/{id}")
def update_post(id: int, post: Post):
    cursor.execute(
        """UPDATE posts
           SET title = %s, content = %s, published = %s
           WHERE id = %s RETURNING *""",
        (post.title, post.content, post.published, id)
    )
    updated_post = cursor.fetchone()
    conn.commit()

    if not updated_post:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail=f"Post with id {id} was not found"
        )

    return {"data": updated_post}
```

Postman Test:

- Method: PUT
- URL: `http://localhost:8000/posts/1`
- Headers: `Content-Type: application/json`
- Body (JSON):

```json
{
    "title": "Updated Post Title",
    "content": "Updated content",
    "published": false
}
```

Section 6: ORMs (Object-Relational Mapping)

1. ORM Introduction

What is an ORM?

- Object-Relational Mapping
- Translates between database and Python objects
- Eliminates need to write raw SQL
- Provides database abstraction layer

Benefits:

- Type safety
- Query building
- Database migration handling
- Less boilerplate code

2. SQLAlchemy Setup

Install SQLAlchemy:

```bash
pip install sqlalchemy
```

Database Configuration:

python

```python
# database.py
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

SQLALCHEMY_DATABASE_URL = "postgresql://postgres:your_password@localhost/fastapi"

engine = create_engine(SQLALCHEMY_DATABASE_URL)
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
Base = declarative_base()

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

Models Definition:

python

```python
# models.py
from sqlalchemy import Column, Integer, String, Boolean, text
from sqlalchemy.sql.sqltypes import TIMESTAMP
from .database import Base

class Post(Base):
    __tablename__ = "posts"

    id = Column(Integer, primary_key=True, nullable=False)
    title = Column(String, nullable=False)
    content = Column(String, nullable=False)
    published = Column(Boolean, server_default='TRUE', nullable=False)
    created_at = Column(TIMESTAMP(timezone=True),
                        nullable=False,
                        server_default=text('now()'))
```

Updated main.py:

python

```python
from fastapi import FastAPI, HTTPException, status, Depends
from sqlalchemy.orm import Session
from . import models, schemas
from .database import engine, get_db

# Create tables
models.Base.metadata.create_all(bind=engine)

app = FastAPI()
```

3. Adding CreatedAt Column

Migration (if needed):

sql

```sql
-- Add created_at column if not exists
ALTER TABLE posts ADD COLUMN created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW();
```

Model Update:

python

```python
# Already included in models.py above
created_at = Column(TIMESTAMP(timezone=True),
                    nullable=False,
                    server_default=text('now()'))
```

4. Get All Posts

python

```python
@app.get("/posts")
def get_posts(db: Session = Depends(get_db)):
    posts = db.query(models.Post).all()
    return {"data": posts}
```

Postman Test:

- Method: GET
- URL: `http://localhost:8000/posts`

- Expected Response: Array of post objects with created_at

5. Create Posts

python

```python
@app.post("/posts", status_code=status.HTTP_201_CREATED)
def create_post(post: schemas.PostCreate, db: Session = Depends(get_db)):
    new_post = models.Post(**post.dict())
    db.add(new_post)
    db.commit()
    db.refresh(new_post)
    return {"data": new_post}
```

Postman Test:

- Method: POST
- URL: `http://localhost:8000/posts`
- Body (JSON):

json

```json
{
    "title": "ORM Post",
    "content": "Created using SQLAlchemy ORM",
    "published": true
}
```

6. Get Post by ID

python

```python
@app.get("/posts/{id}")
def get_post(id: int, db: Session = Depends(get_db)):
    post = db.query(models.Post).filter(models.Post.id == id).first()

    if not post:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail=f"Post with id {id} was not found"
        )

    return {"data": post}
```

Postman Test:

- Method: GET
- URL: `http://localhost:8000/posts/1`

7. Delete Post

```python
@app.delete("/posts/{id}", status_code=status.HTTP_204_NO_CONTENT)
def delete_post(id: int, db: Session = Depends(get_db)):
    post_query = db.query(models.Post).filter(models.Post.id == id)
    post = post_query.first()

    if not post:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail=f"Post with id {id} was not found"
        )

    post_query.delete(synchronize_session=False)
    db.commit()

    return Response(status_code=status.HTTP_204_NO_CONTENT)
```

Postman Test:

- Method: DELETE
- URL: `http://localhost:8000/posts/1`

8. Update Post

```python
@app.put("/posts/{id}")
def update_post(id: int, post: schemas.PostCreate, db: Session = Depends(get_db)):
    post_query = db.query(models.Post).filter(models.Post.id == id)
    existing_post = post_query.first()

    if not existing_post:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail=f"Post with id {id} was not found"
        )

    post_query.update(post.dict(), synchronize_session=False)
    db.commit()

    return {"data": post_query.first()}
```

Postman Test:

- Method: PUT
- URL: `http://localhost:8000/posts/1`
- Body (JSON):

```json
{
    "title": "Updated ORM Post",
    "content": "Updated using SQLAlchemy ORM",
    "published": false
}
```

Section 7: Pydantic Models

1. Pydantic vs ORM Models

Pydantic Models (schemas.py):

- Data validation
- Serialization/Deserialization
- API request/response formatting

- Type conversion

ORM Models (models.py):

- Database table structure
- Database operations
- Relationships
- Constraints

2. Pydantic Models Deep Dive

Complete schemas.py:

python

```python
# schemas.py
from pydantic import BaseModel
from datetime import datetime
from typing import Optional

# Base Post schema
class PostBase(BaseModel):
    title: str
    content: str
    published: bool = True

# For creating posts (request body)
class PostCreate(PostBase):
    pass

# For updating posts (request body)
class PostUpdate(PostBase):
    pass

# For responses (includes all fields)
class Post(PostBase):
    id: int
    created_at: datetime

    class Config:
        orm_mode = True

# Alternative response model with optional fields
class PostResponse(BaseModel):
    id: int
    title: str
    content: str
    published: bool
    created_at: datetime

    class Config:
        orm_mode = True
```

Why orm_mode = True?

- Allows Pydantic to read data from ORM objects

- Enables automatic conversion from SQLAlchemy models
- Required for returning ORM objects in responses

3. Response Model

Using Response Models in Routes:

python

```
from typing import List

@app.get("/posts", response_model=List[schemas.Post])
def get_posts(db: Session = Depends(get_db)):
    posts = db.query(models.Post).all()
    return posts

@app.post("/posts", status_code=status.HTTP_201_CREATED, response_model=schemas.Post)
def create_post(post: schemas.PostCreate, db: Session = Depends(get_db)):
    new_post = models.Post(**post.dict())
    db.add(new_post)
    db.commit()
    db.refresh(new_post)
    return new_post

@app.get("/posts/{id}", response_model=schemas.Post)
def get_post(id: int, db: Session = Depends(get_db)):
    post = db.query(models.Post).filter(models.Post.id == id).first()

    if not post:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail=f"Post with id {id} was not found"
        )

    return post

@app.put("/posts/{id}", response_model=schemas.Post)
def update_post(id: int, post: schemas.PostCreate, db: Session = Depends(get_db)):
    post_query = db.query(models.Post).filter(models.Post.id == id)
    existing_post = post_query.first()

    if not existing_post:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail=f"Post with id {id} was not found"
        )

    post_query.update(post.dict(), synchronize_session=False)
    db.commit()

    return post_query.first()
```

Benefits of Response Models:

- Automatic data validation
- Consistent response format
- Documentation generation
- Type safety

Advanced Pydantic Features

Custom Validators:

```python
from pydantic import BaseModel, validator

class PostCreate(BaseModel):
    title: str
    content: str
    published: bool = True

    @validator('title')
    def title_must_not_be_empty(cls, v):
        if not v.strip():
            raise ValueError('Title cannot be empty')
        return v

    @validator('content')
    def content_must_be_long_enough(cls, v):
        if len(v) < 10:
            raise ValueError('Content must be at least 10 characters')
        return v
```

Field Validation:

```python
from pydantic import BaseModel, Field

class PostCreate(BaseModel):
    title: str = Field(..., min_length=1, max_length=100)
    content: str = Field(..., min_length=10)
    published: bool = True
```

Complete Project Structure

```
app/
├── __init__.py
├── main.py
├── database.py
├── models.py
├── schemas.py
└── requirements.txt
```

requirements.txt:

```
fastapi==0.104.1
uvicorn[standard]==0.24.0
sqlalchemy==2.0.23
psycopg2-binary==2.9.8
```

Postman Collection Testing

1. Create Collection

- Open Postman
- Click "New" > "Collection"
- Name: "FastAPI Posts API"

2. Add Environment

- Click "Environments"
- Add new environment: "FastAPI Local"
- Variables:
  - `base_url`: `http://localhost:8000`

3. Test Requests

1. Get All Posts

```
GET {{base_url}}/posts
```

2. Create Post

```
POST {{base_url}}/posts
Content-Type: application/json

{
    "title": "Test Post",
    "content": "This is a test post created via Postman",
    "published": true
}
```

3. Get Single Post

```
GET {{base_url}}/posts/1
```

4. Update Post

```
PUT {{base_url}}/posts/1
Content-Type: application/json

{
    "title": "Updated Test Post",
    "content": "This post has been updated",
    "published": false
}
```

5. Delete Post

```
DELETE {{base_url}}/posts/1
```

4. Test Scenarios

Test Cases to Verify:

1. ✅ Create post with valid data
2. ✅ Get all posts returns array
3. ✅ Get single post returns correct post
4. ✅ Update post modifies data
5. ✅ Delete post removes from database
6. ✅ Get non-existent post returns 404
7. ✅ Update non-existent post returns 404

8. ✅ Delete non-existent post returns 404
9. ✅ Create post with invalid data returns 422
10. ✅ Response models format data correctly

5. Automation Scripts

Pre-request Script (for authentication later):

```javascript
// Set common headers
pm.request.headers.add({
    key: 'Content-Type',
    value: 'application/json'
});
```

Test Script:

```javascript
// Test successful response
pm.test("Status code is 200", function () {
    pm.response.to.have.status(200);
});

// Test response structure
pm.test("Response has data field", function () {
    const responseJson = pm.response.json();
    pm.expect(responseJson).to.have.property('data');
});
```

This comprehensive guide covers all three sections with practical examples, complete code snippets, and detailed Postman testing instructions. Each section builds upon the previous one, showing the progression from raw SQL to ORM to proper response models.