🚀 FastAPI – Full Notes

📘 Introduction

FastAPI is a modern, fast (high-performance), web framework for building APIs with Python 3.7+ based on:

- **Starlette** for the web parts
- **Pydantic** for data validation

✅ Features

- Automatic interactive API docs (Swagger & ReDoc)
- Fast performance (comparable to NodeJS & Go)
- Type hints for validation and documentation
- Dependency injection system
- Asynchronous support (async/await)

🛠️ Setup

🧰 Installation

To run the program we need to install the server which is uvicorn

```
pip install fastapi uvicorn
```

To install required dependencies for running the application we can save all these in a requirement.txt file and run this file to install all these.

requirements_for_fast_API.txt

▶️ Run the server

To run the server first you need to create virtual environment user below command to create the storage:

```
python3 -m venv venv
```

To activate virtual environment in the project directory use below command:

```
source venv/bin/activate
```

To run the main server we use command:

```
uvicorn main:app --reload
```

If another process is running on the same port number i.e. 8000 then use below command:

```
uvicorn app.main:app --reload --port 8003
```

- `main`: filename (main.py)
- `app`: FastAPI instance
- `-reload`: auto-reload during development

🏗️ Project Structure (Basic)

```
project/
├── app/
│   ├── main.py
│   ├── models/
│   ├── routes/
│   ├── schemas/
│   ├── services/
│   └── utils/
└── requirements.txt
```

🌐 Creating Routes

✳️ Basic Example

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
def read_root():
    return {"Hello": "World"}
```

📥 Path and Query Parameters

```
@app.get("/items/{item_id}")
def read_item(item_id: int, q: str = None):
    return {"item_id": item_id, "q": q}
```

- `item_id`: path param (required)
- `q`: query param (optional)

📦 Request Body with Pydantic

```
from pydantic import BaseModel

class Item(BaseModel):
    name: str
        price: float
    is_offer: bool = None

@app.post("/items/")
def create_item(item: Item):
    return item
```

📄 Response Models

Control output format using Pydantic:

```
class ItemResponse(BaseModel):
    name: str
    price: float

@app.get("/items/{item_id}", response_model=ItemResponse)
def get_item(item_id: int):
    return {"name": "Book", "price": 10.5, "is_offer": False}
```

⚙️ Dependency Injection

`Depends` is a **dependency injection tool**. It's used to **declare a dependency** that FastAPI will automatically resolve and inject into your function.

💡 Why use `Depends`?

It helps you:

- Avoid code duplication.
- Extract reusable logic (like DB sessions, authentication).
- Keep route handlers clean and modular.

```
from fastapi import Depends

def common_parameters(q: str = None):
    return {"q": q}

@app.get("/items/")
def read_items(commons: dict = Depends(common_parameters)):
    return commons
```

🔐 Security (Basics)

```
from fastapi.security import OAuth2PasswordBearer
from fastapi import Depends

oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")

@app.get("/users/me")
def read_users_me(token: str = Depends(oauth2_scheme)):
    return {"token": token}
```

⚠️ Error Handling

1. HTTPException

```
from fastapi import HTTPException

@app.get("/items/{item_id}")
def get_item(item_id: int):
    if item_id == 0:
        raise HTTPException(status_code=404, detail="Item not found")
```

2. Validation Error

Handled automatically by FastAPI using Pydantic. Returns 422.

3. Custom Exception Handler

```
class CustomException(Exception):
    def __init__(self, name: str):
        self.name = name

@app.exception_handler(CustomException)
async def custom_exception_handler(request, exc: CustomException):
    return JSONResponse(
        status_code=418,
```

```
        content={"message": f"Oops! {exc.name} caused an error."},
    )
```

📊 Interactive Documentation

- **Swagger UI:** `http://127.0.0.1:8000/docs`
- **ReDoc:** `http://127.0.0.1:8000/redoc`

🔄 Async Support

```
@app.get("/async-task")
async def async_task():
    await some_async_function()
    return {"status": "done"}
```

🧪 Testing

Using `pytest` and `TestClient`:

```
from fastapi.testclient import TestClient

client = TestClient(app)

def test_read_main():
    response = client.get("/")
    assert response.status_code == 200
```

📂 Middleware

```
from fastapi import Request

@app.middleware("http")
```

```
async def log_requests(request: Request, call_next):
    response = await call_next(request)
    return response
```

📌 Extra Features

🔄 CORS

```
from fastapi.middleware.cors import CORSMiddleware

app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_methods=["*"],
    allow_headers=["*"],
)
```

🧾 Background Tasks

```
from fastapi import BackgroundTasks

def write_log(msg: str):
    with open("log.txt", "a") as f:
        f.write(msg)

@app.post("/send/")
def send_notification(background_tasks: BackgroundTasks):
    background_tasks.add_task(write_log, "Notification sent")
```

🧠 Pro Tips

- Use `response_model` to hide sensitive data.
- Use `Depends()` to keep routes clean.

- Use `tags=["category"]` in route decorators for Swagger grouping.
- Use `status_code=201` for POST responses.

🧳 FastAPI Code Flow — Step by Step

🔷 What it does:

- Connects to your database (SQLite, PostgreSQL, MySQL, etc.).
- Creates a **session maker** so you can talk to the DB.
- Defines the **Base class** to build your models on.

🧠 Why it's required:

- You need a way to interact with your database.
- SQLAlchemy needs a consistent connection and session management.

Database Connection (e.g., SQLAlchemy engine/session)

```
# db.py
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker, declarative_base

DATABASE_URL = "sqlite:///./test.db"  # or PostgreSQL/MySQL/etc
#postgresql://<username>:<password>@<host>:<port>/<database>


engine = create_engine(DATABASE_URL, connect_args={"check_same_threa
d": False})
SessionLocal = sessionmaker(bind=engine, autoflush=False, autocommit=Fal
se)
Base = declarative_base()
```

2. Create the Models (Pydantic)

🔷 What it does:

- Defines your **database schema** using SQLAlchemy ORM.
- Each class maps to a **database table**.

🧠 Why it's required:

- You need a way to interact with your database.
- SQLAlchemy needs a consistent connection and session management.

```
# models.py
from sqlalchemy import Column, Integer, String
from .db import Base

class Item(Base):
    __tablename__ = "items"
    id = Column(Integer, primary_key=True, index=True)
    name = Column(String, index=True)
    description = Column(String)
```

3. Create Pydantic Schemas (For Validation & Serialization)

🔷 What it does:

- Defines the structure of **incoming requests** and **outgoing responses**.
- Validates and serializes data.

🧠 Why it's required:

- Ensures data is correct before it hits your logic layer.
- Separates database schema from API interface.
- Automatically generates docs using OpenAPI.

```
# schemas.py
from pydantic import BaseModel

class ItemBase(BaseModel):
```

```
    name: str
    description: str

class ItemCreate(ItemBase):
    pass

class ItemRead(ItemBase):
    id: int
    class Config:
        orm_mode = True  # Needed to work with SQLAlchemy models
```

4. CRUD Operations

🔷 What it does:

- Handles **actual database operations** like fetching, inserting, updating, deleting.
- Uses SQLAlchemy session to perform actions.

🧠 Why it's required:

- Keeps business logic separate from routing.
- Makes the code modular, clean, and testable.

```
# crud.py  → contains business logic
from sqlalchemy.orm import Session
from . import models, schemas

def get_item(db: Session, item_id: int):
    return db.query(models.Item).filter(models.Item.id == item_id).first()

def create_item(db: Session, item: schemas.ItemCreate):
    db_item = models.Item(**item.dict())
    db.add(db_item)
    db.commit()
```

```
    db.refresh(db_item)
    return db_item
```

5. Dependency for DB Session

🔷 What it does:

- Provides a **reusable way to get a DB session**.
- Uses `yield` and `try/finally` to manage resource cleanup.

🧠 Why it's required:

- Ensures the DB session is **created per request** and **closed properly**.
- FastAPI uses dependency injection to pass this into routes automatically.

```
# deps.py
from .db import SessionLocal
from fastapi import Depends

def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()
```

6. Create FastAPI Routes

🔷 What it does:

- Handles **HTTP endpoints** like `POST /items/` and `GET /items/{id}`.
- Accepts input, calls the business logic, and returns the response.

🧠 Why it's required:

- This is where **requests enter** the system.

- Defines how the API behaves and what logic runs.

```
# main.py
from fastapi import FastAPI, Depends, HTTPException
from sqlalchemy.orm import Session
from . import crud, models, schemas, db, deps

app = FastAPI()

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

@app.post("/items/", response_model=schemas.ItemRead)
def create_item(item: schemas.ItemCreate, db: Session = Depends(deps.get_
db)):
    return crud.create_item(db, item)

@app.get("/items/{item_id}", response_model=schemas.ItemRead)
def read_item(item_id: int, db: Session = Depends(deps.get_db)):
    db_item = crud.get_item(db, item_id)
    if db_item is None:
        raise HTTPException(status_code=404, detail="Item not found")
    return db_item
```

🔁 Full Flow Recap

1. **Request**: `POST /items/` with JSON body
2. **Routing**: Finds `create_item` route
3. **Validation**: Pydantic checks request body
4. **Dependency**: Injects DB session
5. **CRUD Logic**: SQLAlchemy creates object, commits to DB
6. **Response**: Pydantic serializes to JSON and returns

✅ Summary Table

Step	File	Purpose
1. Database Setup	`db.py`	Connect to DB & setup session
2. SQLAlchemy Models	`models.py`	Define DB table structure
3. Pydantic Schemas	`schemas.py`	Validate input/output
4. CRUD Logic	`crud.py`	DB operations & business logic
5. Dependency Injection	`deps.py`	Provide DB session to routes
6. Routes	`main.py`	Handle API requests
7. Table Creation	`main.py`	Initialize DB schema
8. Docs	Auto	Test API, generate docs