# Project Structure & FastAPI Introduction

## Outline

- **Project Structure**
- Type Hinting
- Introduction to FastAPI

# Typical Python Project Structure

```
project/
│
├── main.py
├── requirements.txt
├── README.md
│
├── models/
│   ├── __init__.py
│   └── user.py
│
└── utils/
    ├── __init__.py
    └── helper.py
```

main.py: Entry point of the application

utils.py: Contains helper functions

models/: Directory to organize model-related code

__init__.py: Marks a directory as a Python package

Importing Across Modules

```python
from utils.helper import greet_user
from models.user import User
```

# Python Project File

**models/user.py**

```python
class User:
    def __init__(self, name, email):
        self.name = name
        self.email = email

    def get_email_domain(self):
        return self.email.split('@')[-1]
```

**utils/helper.py**

```python
def greet_user(user):
    return f"Hello, {user.name}! Your email domain is {user.get_email_domain()}."
```

**main.py**

```python
from models.user import User
from utils.helper import greet_user

def main():
    user = User(name="Eka", email="eka@example.com")
    print(greet_user(user))

if __name__ == "__main__":
    main()
```

**requirements.txt**

```
pydantic>=2.6.4
pandas>=2.2.2
numpy>=1.26.4
```

# Virtual Environment (venv)

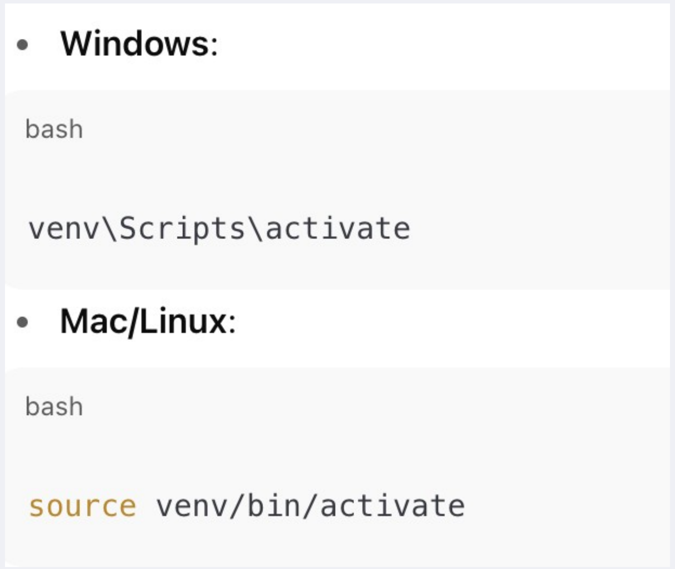
**Create Virtual Environment**

```
python -m venv venv
```

**Deactivate**

```
deactivate
```

**Activate Virtual Environment**



- **Windows**:

bash

```
venv\Scripts\activate
```

- **Mac/Linux**:

bash

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

**Install Library**

```
pip install -r requirements.txt
```

## Exercise

- Create a models/ and utils/ folder inside your project directory.
- Move the Customer class into a new file:
  - models/customer_model.py
- Move the welcome_message function into a new file:
  - utils/message_utils.py
- In your main.py:
  - Import and use the Customer class from models/customer_model.py
  - Import and use the welcome_message function from utils/message_utils.py
- Create and activate a virtual environment
- Install FastAPI using pip

## Outline

- Project Structure
- **Type Hinting**
- Introduction to FastAPI

# Type Hinting

Python has support for optional "type hints" (also called "type annotations").

These "type hints" or annotations are a special syntax that allow declaring the type of a variable. By declaring types for your variables, editors and tools can give you better support.

Example:

```
def get_full_name(first_name, last_name):
    full_name = first_name.title() + " " + last_name.title()
    return full_name


print(get_full_name("john", "doe"))
```

Output

```
John Doe
```

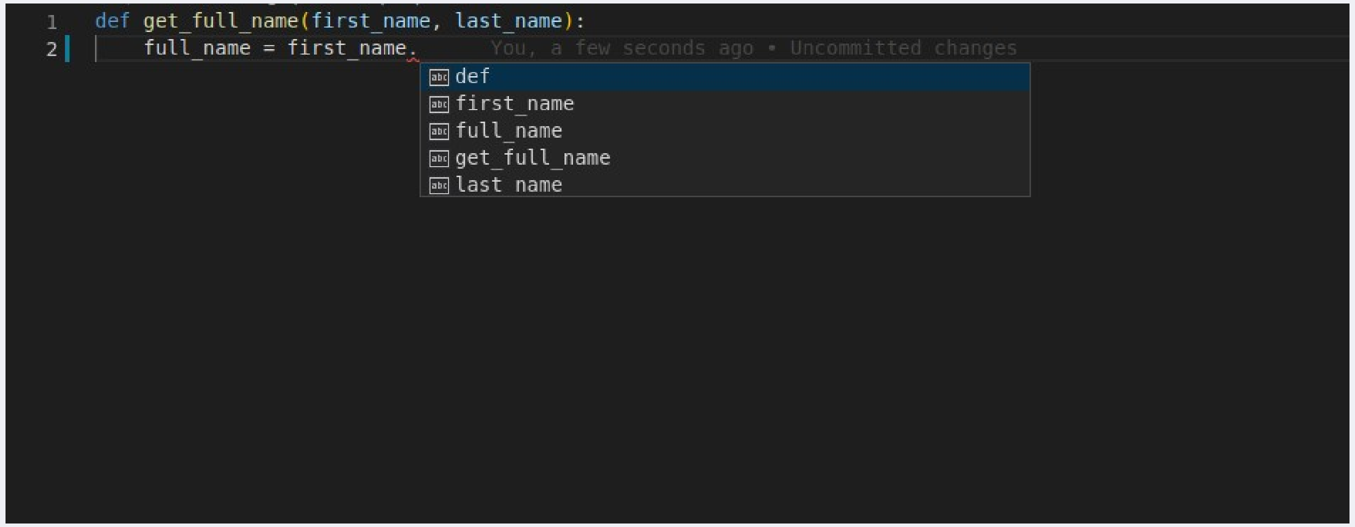
It's a very simple program.But now imagine that you were writing it from scratch. At some point you would have started the definition of the function, you had the parameters ready...

But then you have to call "that method that converts the first letter to upper case".

Was it upper? Was it uppercase? first_uppercase? capitalize?

Then, you try with the old programmer's friend, editor autocompletion. You type the first parameter of the function, first_name, then a dot (.) and then hit Ctrl+Space to trigger the completion.

But, sadly, you get nothing useful:



```
def get_full_name(first_name, last_name):
    full_name = first_name.        You, a few seconds ago • Uncommitted changes
```

# Type Hinting (Cont)

Let's modify a single line from the previous version.

We will change exactly this fragment, the parameters of the function, from:

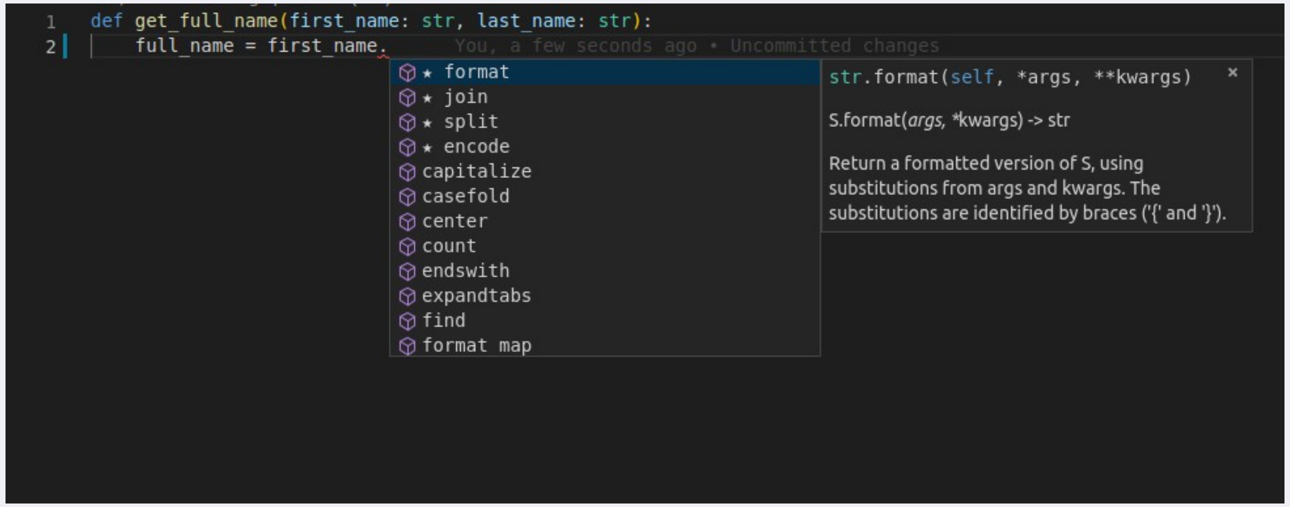
```
  first_name, last_name
```

to

```
first_name: str, last_name: str
```

```
def full_name = first_name.title() + " " + last_name.title()
   return full_name


print(get_full_name("john", "doe"))
```

At the same point, you try to trigger the autocomplete with Ctrl+Space and you see:



```
def get_full_name(first_name: str, last_name: str):
    full_name = first_name.      You, a few seconds ago • Uncommitted changes
```

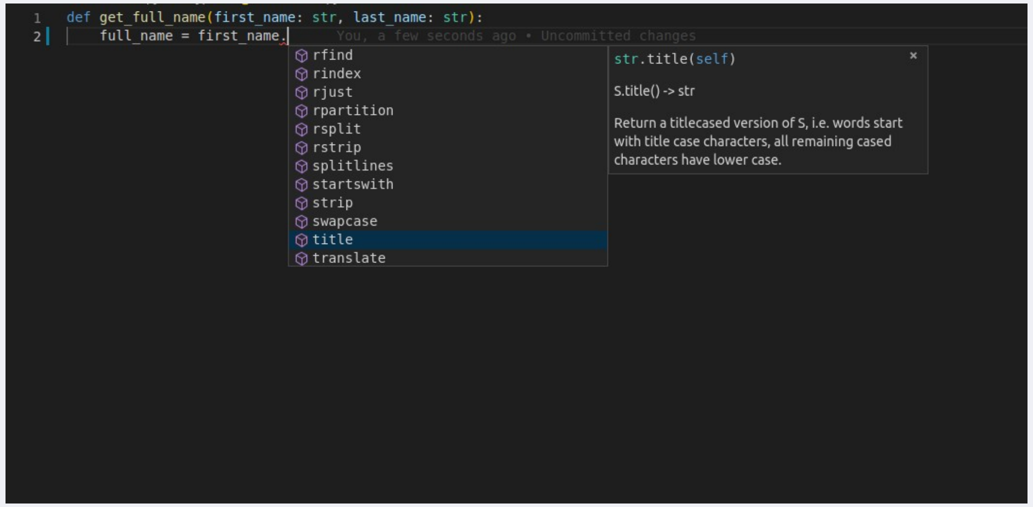
```
★ format
★ join
★ split
★ encode
capitalize
casefold
center
count
endswith
expandtabs
find
format_map
```

str.format(self, *args, **kwargs)

S.format(*args*, *kwargs) -> str

Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

With that, you can scroll, seeing the options, until you find the one that "rings a bell":



```
def get_full_name(first_name: str, last_name: str):
    full_name = first_name.      You, a few seconds ago • Uncommitted changes
```

```
rfind
rindex
rjust
rpartition
rsplit
rstrip
splitlines
startswith
strip
swapcase
title
translate
```

str.title(self)

S.title() -> str

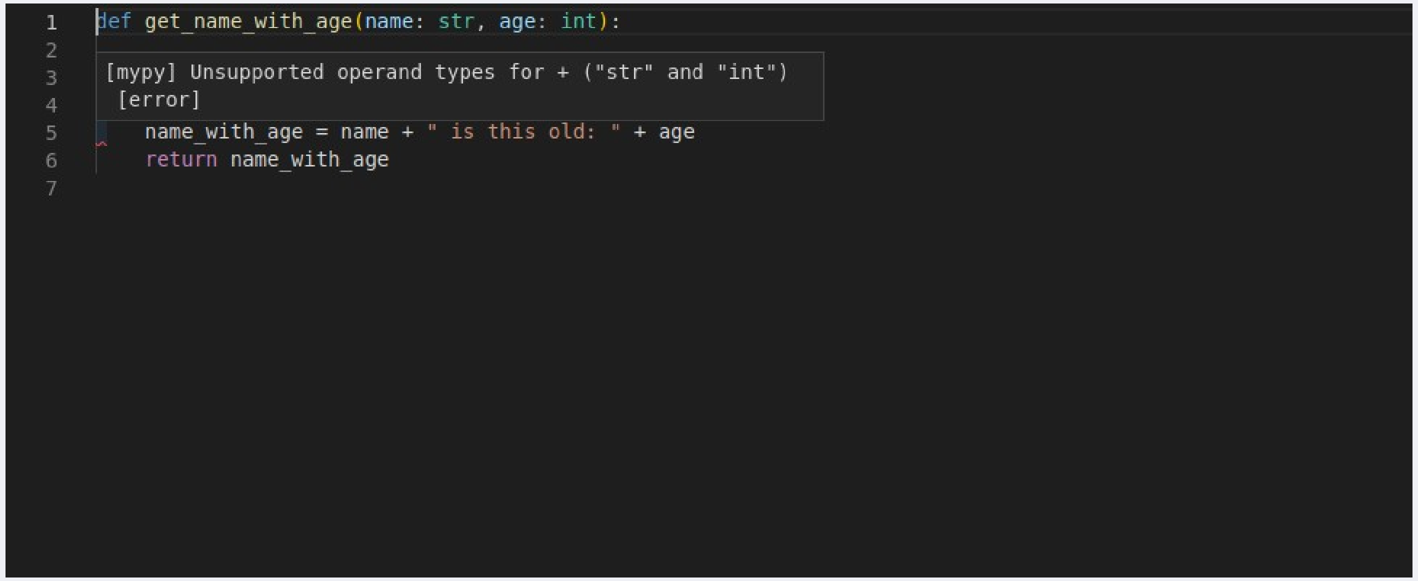
Return a titlecased version of S, i.e. words start with title case characters, all remaining cased characters have lower case.

# Type Hinting (Cont)

Check this function, it already has type hints:

```python
def get_name_with_age(name: str, age: int):

    name_with_age = name + " is this old: " + age
    return name_with_age
```

Because the editor knows the types of the variables, you don't only get completion, you also get error checks:



```
def get_name_with_age(name: str, age: int):

[mypy] Unsupported operand types for + ("str" and "int")
 [error]
    name_with_age = name + " is this old: " + age
    return name_with_age
```

Now you know that you have to fix it, convert age to a string with str(age):

```python
def get_name_with_age(name: str, age: int):

    name_with_age = name + " is this old: " + str(age)

    return name_with_age
```

## Exercise

Please redeclared this function using type hinting:

```
def fillerup(items_names,items_weights,capacity):
    output = []
     total_weight = 0
    i = 0
    while i < len(items_names) and total_weight + items_weights[i] <= capacity:
        output.append(items_names[i])
        total_weight += items_weights[i]
        i+=1
    return output
```

## Outline

- Project Structure
- Type Hinting
- **Introduction to FastAPI**

# Quick Start

## Key Points:

Simple app creation with minimal code Use

`uvicorn main:app --reload` to run the app

## Code Snippet:

```
from fastapi import FastAPI
app = FastAPI()
@app.get("/") def
read_root():
    return {"message": "Hello, FastAPI!"}
```

## Path Parameters

You can declare path parameters in your endpoint. They’re passed as arguments:

```
@app.get("/items/{item_id}") def
read_item(item_id:  int):
   return {"item_id":  item_id}
```

Here, `item_id` is a path parameter and is automatically converted to `int`.

## Query Parameters

Query parameters are automatically parsed:

```
@app.get("/items/")
def read_item(skip:    int =  0,    limit: int =  10):
   return {"skip":     skip,  "limit":      limit}
```

You can access them via `?skip=0&limit=10` in the URL.

## Data Validation with Pydantic

FastAPI uses **Pydantic** models to validate request data and define schemas:

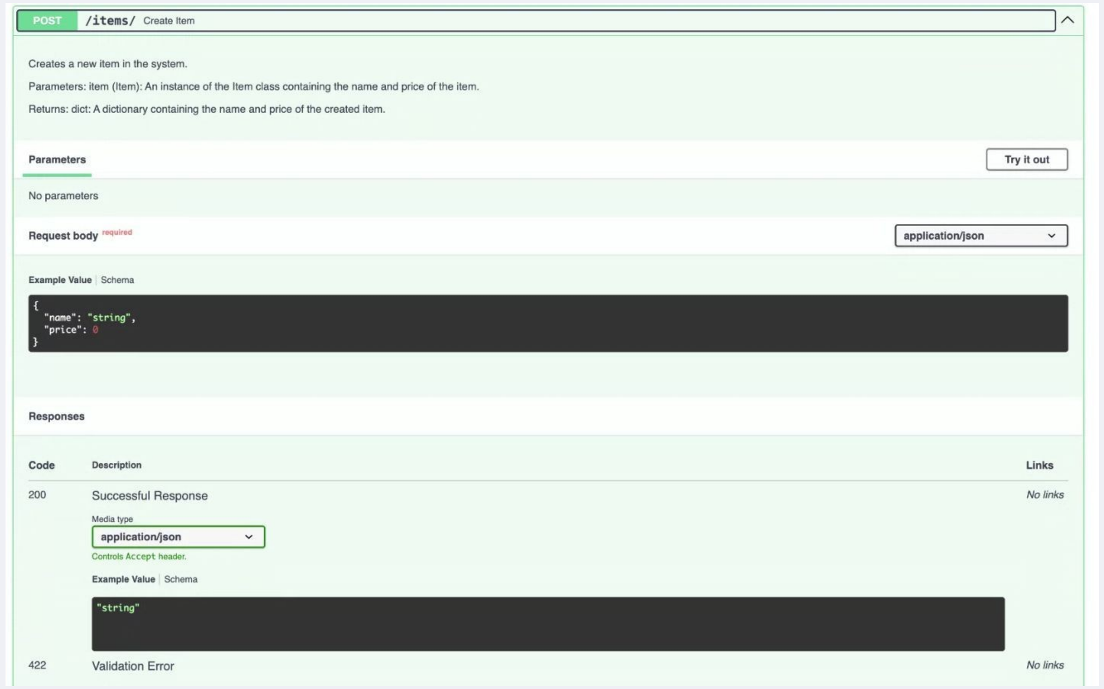
```
from pydantic import BaseModel

class Item(BaseModel): name: str
   price: float

@app.post("/items/")
def create_item(item: Item):
   return {"name": item.name, "price": item.price}
```

Pydantic automatically validates the input and converts types.

# Automatic API Documentation

FastAPI automatically generates interactive API documentation using **Swagger UI** and **ReDoc**.

## Swagger UI: `/docs`



POST /items/ Create Item

Creates a new item in the system.

Parameters: item (Item): An instance of the Item class containing the name and price of the item.

Returns: dict: A dictionary containing the name and price of the created item.

Parameters | Try it out

No parameters

Request body required | application/json

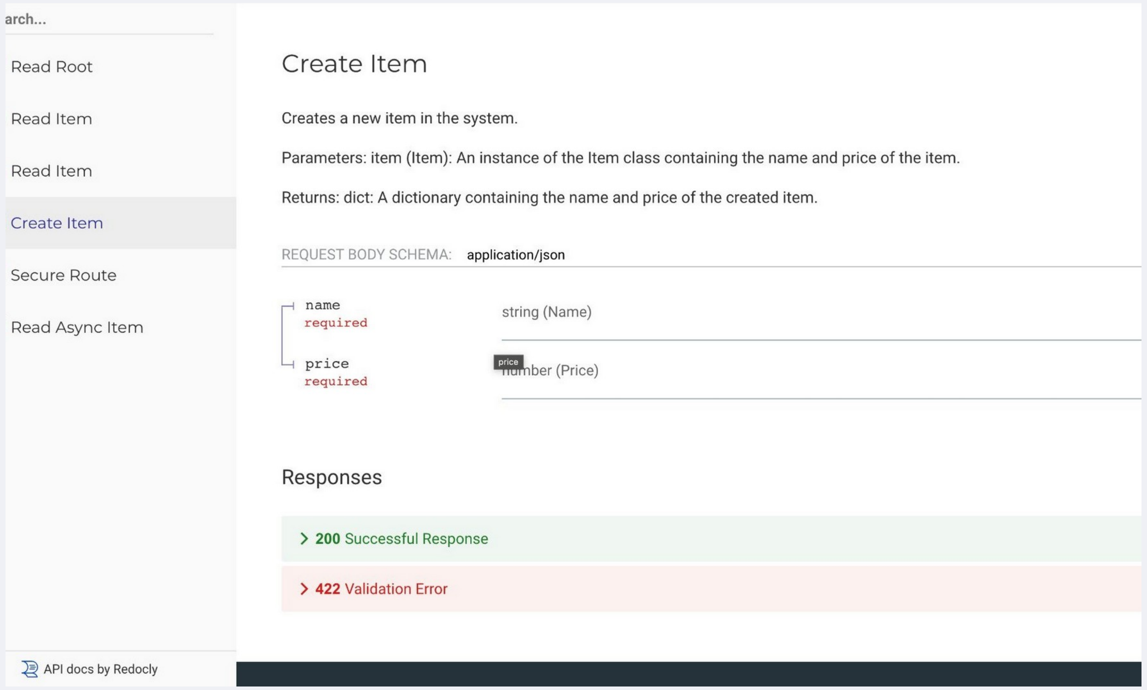
Example Value | Schema

```
{
  "name": "string",
  "price": 0
}
```

Responses

| Code | Description | Links |
|---|---|---|
| 200 | Successful Response<br>Media type: application/json<br>Controls Accept header.<br>Example Value \| Schema<br>"string" | No links |
| 422 | Validation Error | No links |

## ReDoc: /redoc



arch...

Read Root
Read Item
Read Item
Create Item
Secure Route
Read Async Item

API docs by Redocly

Create Item

Creates a new item in the system.

Parameters: item (Item): An instance of the Item class containing the name and price of the item.

Returns: dict: A dictionary containing the name and price of the created item.

REQUEST BODY SCHEMA: application/json

| | |
|---|---|
| name required | string (Name) |
| price required | number (Price) |

Responses

> 200 Successful Response

> 422 Validation Error

SwaggerUI - http://127.0.0.1:8000/docs
ReDoc - http://127.0.0.1:8000/redoc

## Asynchronous Support

FastAPI is asynchronous by nature, which means you can define async functions to handle requests efficiently:

```
@app.get("/async_items/") async def
read_async_item():
    return {"message":  "This is asynchronous"}
```

## Exercise

- Create a User model using Pydantic
- Implement:
    - POST /users to add user to an in-memory list
    - GET /users to return the full list of users

## Outline

- Project Structure
- Type Hinting
- Introduction to FastAPI
- **Modularisasi FastAPI dan Error Handling**

# Request Body dan Response Model

**Menggunakan Pydantic Model untuk Input/Output**

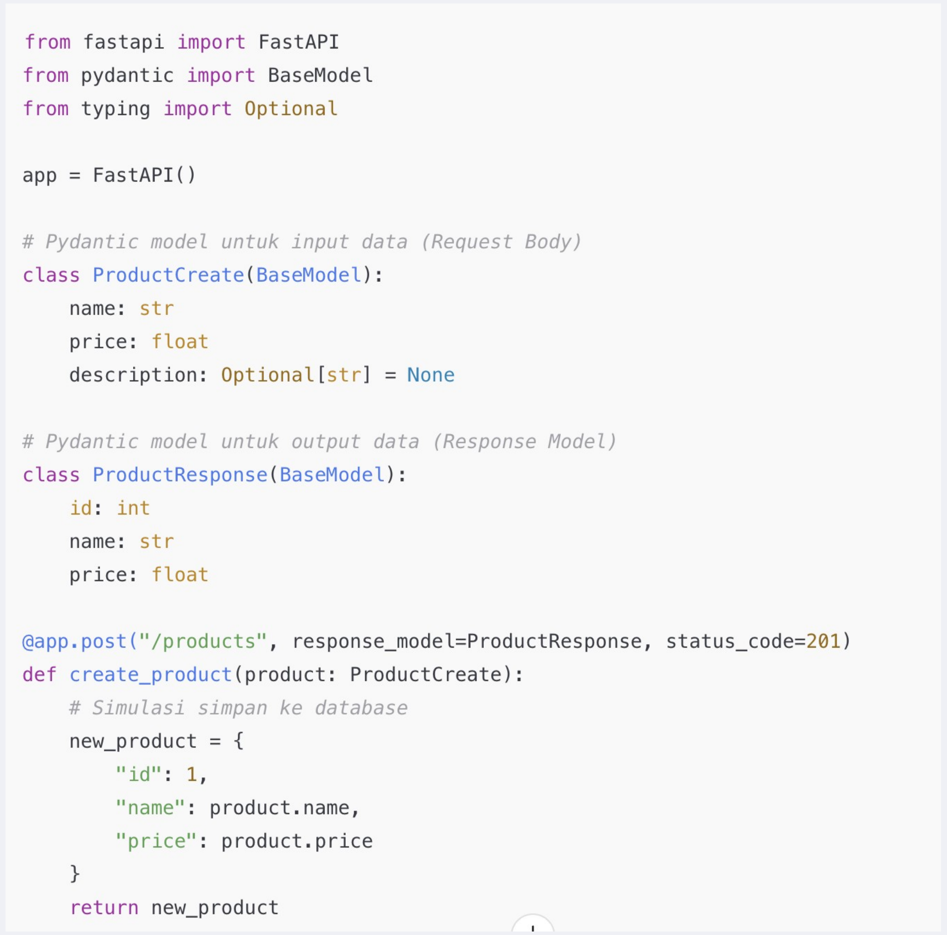
- Pydantic digunakan untuk membuat **model data** yang memvalidasi input secara otomatis.
- Model ini akan digunakan sebagai:
    - **Request Body**: format data yang diterima API.
    - **Response Model**: format data yang dikirim kembali ke client.

**Validasi Otomatis FastAPI**

- FastAPI akan otomatis memeriksa tipe data sesuai Pydantic model.
- Jika data tidak sesuai, FastAPI akan mengembalikan **error 422** dengan pesan validasi.

**Response Model dan Status Code**

- response_model digunakan untuk mendefinisikan format keluaran API.
- status_code digunakan untuk menentukan kode HTTP, misalnya 201 Created untuk data baru.



```python
from fastapi import FastAPI
from pydantic import BaseModel
from typing import Optional

app = FastAPI()

# Pydantic model untuk input data (Request Body)
class ProductCreate(BaseModel):
    name: str
    price: float
    description: Optional[str] = None

# Pydantic model untuk output data (Response Model)
class ProductResponse(BaseModel):
    id: int
    name: str
    price: float

@app.post("/products", response_model=ProductResponse, status_code=201)
def create_product(product: ProductCreate):
    # Simulasi simpan ke database
    new_product = {
        "id": 1,
        "name": product.name,
        "price": product.price
    }
    return new_product
```

**ProductCreate** → model untuk data input.

**ProductResponse** → model untuk data output.

Jika input salah tipe (misalnya harga string), FastAPI otomatis balas error validasi.

# FastAPI Project Structure

Struktur proyek yang modular memudahkan pengelolaan kode:

```
fastapi_app/
│
├── main.py
│
├── routers/
│   ├── __init__.py
│   └── user_router.py
│
├── schemas/
│   ├── __init__.py
│   └── user_schema.py
│
├── services/
│   ├── __init__.py
│   └── user_service.py
│
└── requirements.txt
```

1. **main.py**

- File **entry point** aplikasi FastAPI.
- Di sini kita **mendaftarkan semua router** dari folder routers/.
- **Fungsi:** Menginisialisasi aplikasi dan menggabungkan semua router.

Contoh isi:

```python
from fastapi import FastAPI
from routers import user_router

app = FastAPI(title="Modular FastAPI App")

app.include_router(user_router.router)
```

# FastAPI Project Structure

Struktur proyek yang modular memudahkan pengelolaan kode:

```
fastapi_app/
│
├── main.py
│
├── routers/
│   ├── __init__.py
│   └── user_router.py
│
├── schemas/
│   ├── __init__.py
│   └── user_schema.py
│
├── services/
│   ├── __init__.py
│   └── user_service.py
│
└── requirements.txt
```

**2. Folder routers/**

Tempat menyimpan semua **route handler** (endpoint API).

- Setiap file .py di sini biasanya untuk satu fitur atau resource (misal: user_router.py, product_router.py).
- **__init__.py** → membuat folder ini dianggap **Python package**, sehingga bisa diimport.
- **user_router.py** → berisi definisi endpoint untuk resource User.
- **Fungsi:** Mengatur URL path dan HTTP method, lalu memanggil fungsi dari services/

```python
from fastapi import APIRouter
from schemas.user_schema import UserCreate, UserResponse
from services.user_service import create_user

router = APIRouter()

@router.post("/users", response_model=UserResponse)
def add_user(user: UserCreate):
    return create_user(user)
```

# FastAPI Project Structure

Struktur proyek yang modular memudahkan pengelolaan kode:

```
fastapi_app/
│
├── main.py
│
├── routers/
│   ├── __init__.py
│   └── user_router.py
│
├── schemas/
│   ├── __init__.py
│   └── user_schema.py
│
├── services/
│   ├── __init__.py
│   └── user_service.py
│
└── requirements.txt
```

### 3. Folder schemas/

Tempat menyimpan **Pydantic models** untuk validasi request dan format response.

- **user_schema.py** → mendefinisikan struktur data UserCreate (input) dan UserResponse (output).
- **__init__.py** → menjadikan folder ini package Python.

**Fungsi:** Memastikan data masuk/keluar sesuai format dan tervalidasi otomatis.

```python
from pydantic import BaseModel, EmailStr

class UserCreate(BaseModel):
    name: str
    email: EmailStr

class UserResponse(BaseModel):
    id: int
    name: str
    email: EmailStr
```

# FastAPI Project Structure

Struktur proyek yang modular memudahkan pengelolaan kode:

```
fastapi_app/
│
├── main.py
│
├── routers/
│   ├── __init__.py
│   └── user_router.py
│
├── schemas/
│   ├── __init__.py
│   └── user_schema.py
│
├── services/
│   ├── __init__.py
│   └── user_service.py
│
└── requirements.txt
```

### 4. Folder services/

Tempat menyimpan **logika bisnis** (business logic) dan operasi data.

- **user_service.py** → berisi fungsi membuat user baru, cek email unik, dll.

- **__init__.py** → menjadikan folder ini package
- **Fungsi:** Memisahkan logika pemrosesan data dari route agar lebih rapi dan mudah diuji..

```python
from fastapi import HTTPException

users_db = []

def create_user(user_data):
    # Cek email duplikat
    for u in users_db:
        if u["email"] == user_data.email:
            raise HTTPException(status_code=400, detail="Email already registered")
    new_user = {
        "id": len(users_db) + 1,
        "name": user_data.name,
        "email": user_data.email
    }
    users_db.append(new_user)
    return new_user
```

# FastAPI Project Structure

Struktur proyek yang modular memudahkan pengelolaan kode:

```
fastapi_app/
│
├── main.py
│
├── routers/
│   ├── __init__.py
│   └── user_router.py
│
├── schemas/
│   ├── __init__.py
│   └── user_schema.py
│
├── services/
│   ├── __init__.py
│   └── user_service.py
│
└── requirements.txt
```

**5. requirements.txt**

- Berisi daftar library yang diperlukan untuk menjalankan proyek.
- **Fungsi:** Memudahkan instalasi dependency (pip install -r requirements.txt).

```
fastapi
uvicorn
pydantic
pandas
numpy
```

# Exercise

1. **Refactor API user ke struktur modular** seperti contoh di atas.

2. Tambahkan validasi:

- Email unik

- Nama minimal 3 karakter

1. Gunakan HTTPException untuk mengembalikan pesan error yang sesuai.

2. Uji coba API menggunakan **FastAPI docs** di http://127.0.0.1:8000/docs.

## Outline

- Project Structure
- Type Hinting
- Introduction to FastAPI
- Modularisasi FastAPI dan Error Handling
- **Middleware**

# Middleware

Middleware adalah **lapisan perantara** yang **dijalankan sebelum dan sesudah request** diproses oleh endpoint.

**Tujuan utama:**

- Logging request/response
- Authentication global
- Modifikasi request atau response
- Menangani CORS, rate limiting, dll.

Alurnya:

```
Request masuk → Middleware → Endpoint → Middleware → Response keluar
```

```python
import time
from fastapi import FastAPI, Request

app = FastAPI()

@app.middleware("http")
async def log_request_time(request: Request, call_next):
    start_time = time.time()

    # Jalankan endpoint
    response = await call_next(request)

    process_time = time.time() - start_time
    print(f"Request: {request.url.path} took {process_time:.4f} seconds")

    return response

@app.get("/")
def home():
    return {"message": "Hello Middleware"}
```

`@app.middleware("http")` akan membungkus semua request HTTP.

`call_next(request)` adalah cara memanggil proses berikutnya (termasuk endpoint).

Middleware bisa:

- Menambah log
- Mengubah request/response
- Menolak request tertentu

## Exercise

1. **Middleware Log Waktu Request**

   Buat middleware yang mencatat waktu eksekusi setiap request.

   Tampilkan log di terminal dalam format:

   bash
   CopyEdit

```
[TIME] Request ke /endpoint_name selesai dalam X.XXXX detik
```

## Outline

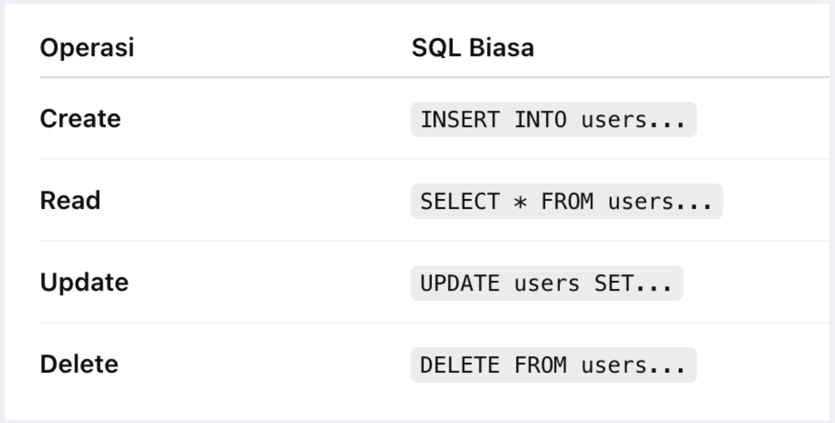
- Project Structure
- Type Hinting
- Introduction to FastAPI
- Modularisasi FastAPI dan Error Handling
- Middleware
- **Integrasi Database**

# Integrasi Database

Menghubungkan aplikasi FastAPI dengan database SQLite secara asynchronous menggunakan **SQLModel** (built di atas SQLAlchemy).

## Konsep Dasar

1. **SQLite**
   - Database ringan, berbasis file (`.db`), cocok untuk prototyping.
   - Tidak memerlukan server terpisah.
2. **Async Database Access**
   - Dengan `aiosqlite`, kita bisa menjalankan query SQL secara asynchronous.
   - Tidak perlu ORM, kita langsung kirim perintah SQL.
3. **CRUD**
   CRUD adalah singkatan dari:
   - **C**reate → Menambah data ke tabel
   - **R**ead → Mengambil data dari tabel
   - **U**pdate → Mengubah data yang sudah ada
   - **D**elete → Menghapus data



| Operasi | SQL Biasa |
|---|---|
| **Create** | `INSERT INTO users...` |
| **Read** | `SELECT * FROM users...` |
| **Update** | `UPDATE users SET...` |
| **Delete** | `DELETE FROM users...` |

# Integrasi Database

## 1. Install Library

Kita pakai:

- **FastAPI** → framework API
- **aiosqlite** → koneksi SQLite async
- **uvicorn** → server untuk menjalankan FastAPI

```
pip install fastapi uvicorn aiosqlite
```

## 2. Buat Koneksi Database

Kita menggunakan **event handler** di FastAPI:

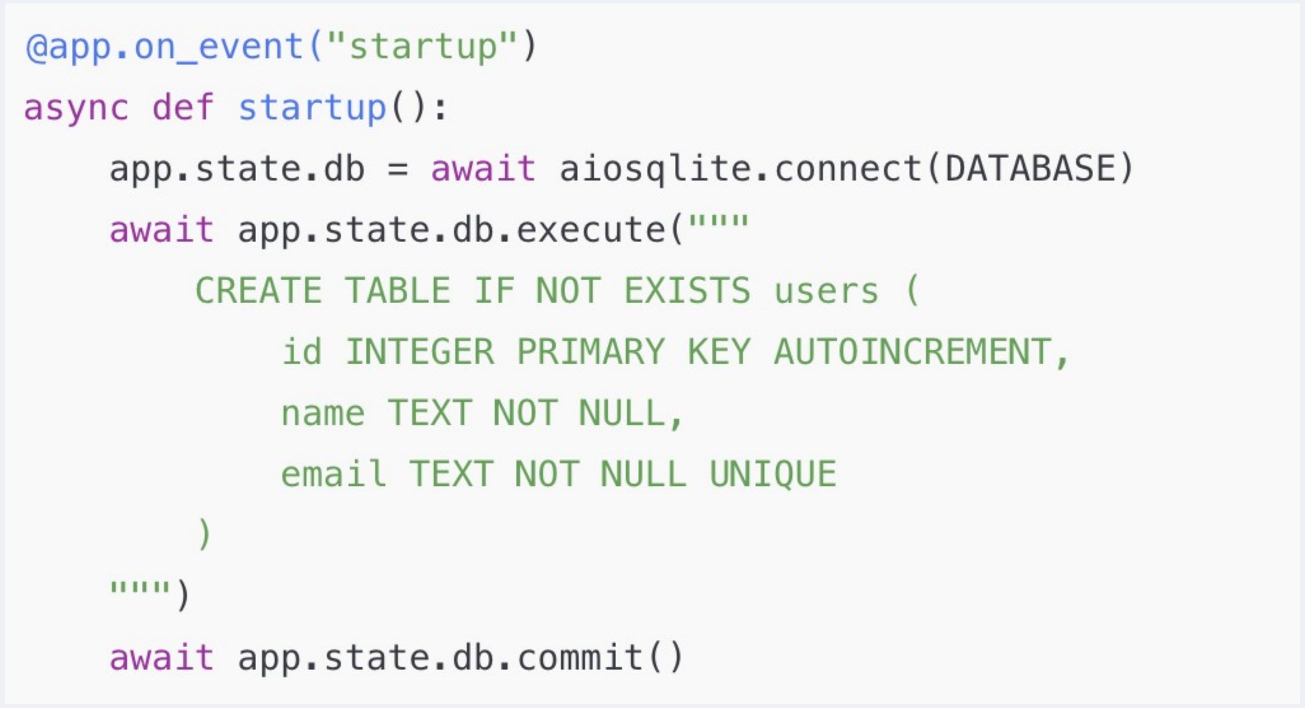
- startup → koneksi ke database saat aplikasi mulai
- shutdown → menutup koneksi saat aplikasi berhenti

```python
import aiosqlite
from fastapi import FastAPI

app = FastAPI()
DATABASE = "users.db"

@app.on_event("startup")
async def startup():
    app.state.db = await aiosqlite.connect(DATABASE)

@app.on_event("shutdown")
async def shutdown():
    await app.state.db.close()
```

## Integrasi Database

### 3. Buat Tabel (Schema)

Tabel akan dibuat saat aplikasi dijalankan pertama kali.



```python
@app.on_event("startup")
async def startup():
    app.state.db = await aiosqlite.connect(DATABASE)
    await app.state.db.execute("""
        CREATE TABLE IF NOT EXISTS users (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            name TEXT NOT NULL,
            email TEXT NOT NULL UNIQUE
        )
    """)
    await app.state.db.commit()
```

# Integrasi Database

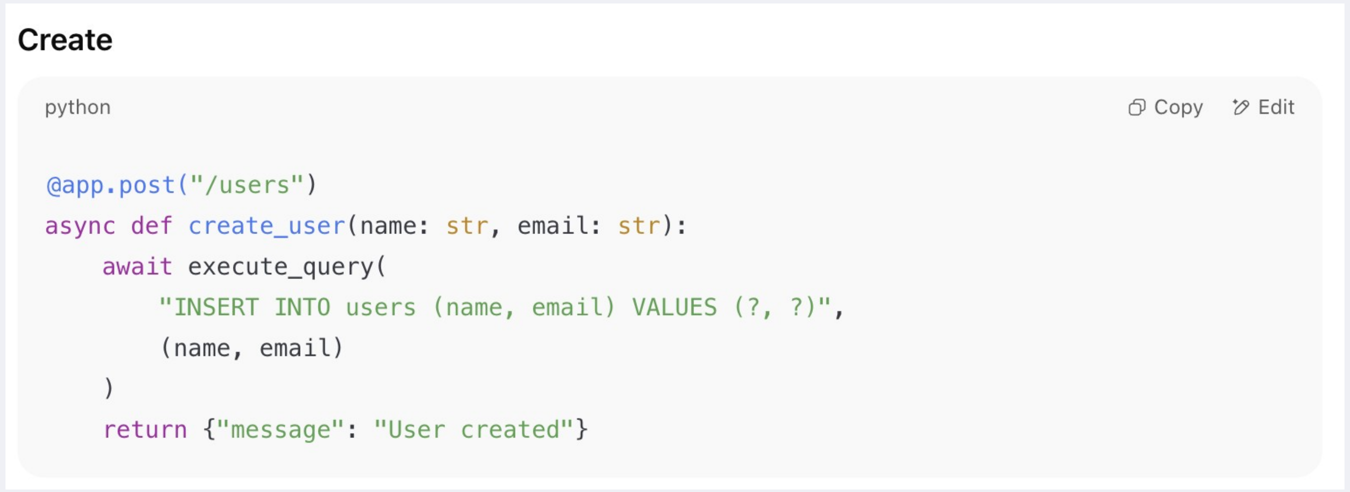
## 4. Fungsi Helper untuk Query

Supaya tidak copy-paste query, kita buat fungsi:

```python
async def execute_query(query, params=()):
    await app.state.db.execute(query, params)
    await app.state.db.commit()

async def fetch_query(query, params=()):
    cursor = await app.state.db.execute(query, params)
    rows = await cursor.fetchall()
    return rows
```

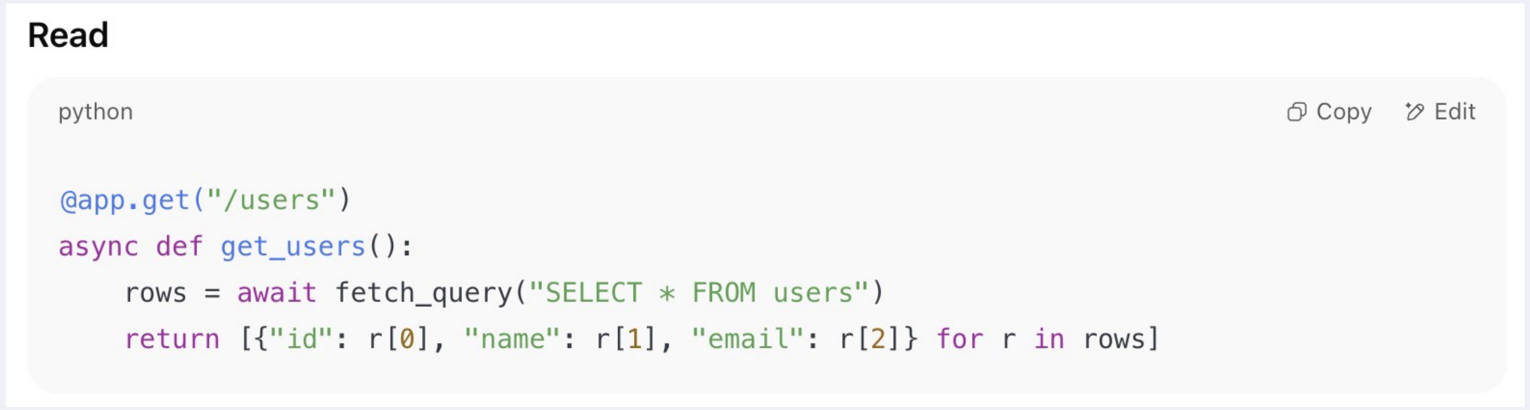
# Integrasi Database

5. **CRUD Endpoint**



**Create**

```python
@app.post("/users")
async def create_user(name: str, email: str):
    await execute_query(
        "INSERT INTO users (name, email) VALUES (?, ?)",
        (name, email)
    )
    return {"message": "User created"}
```

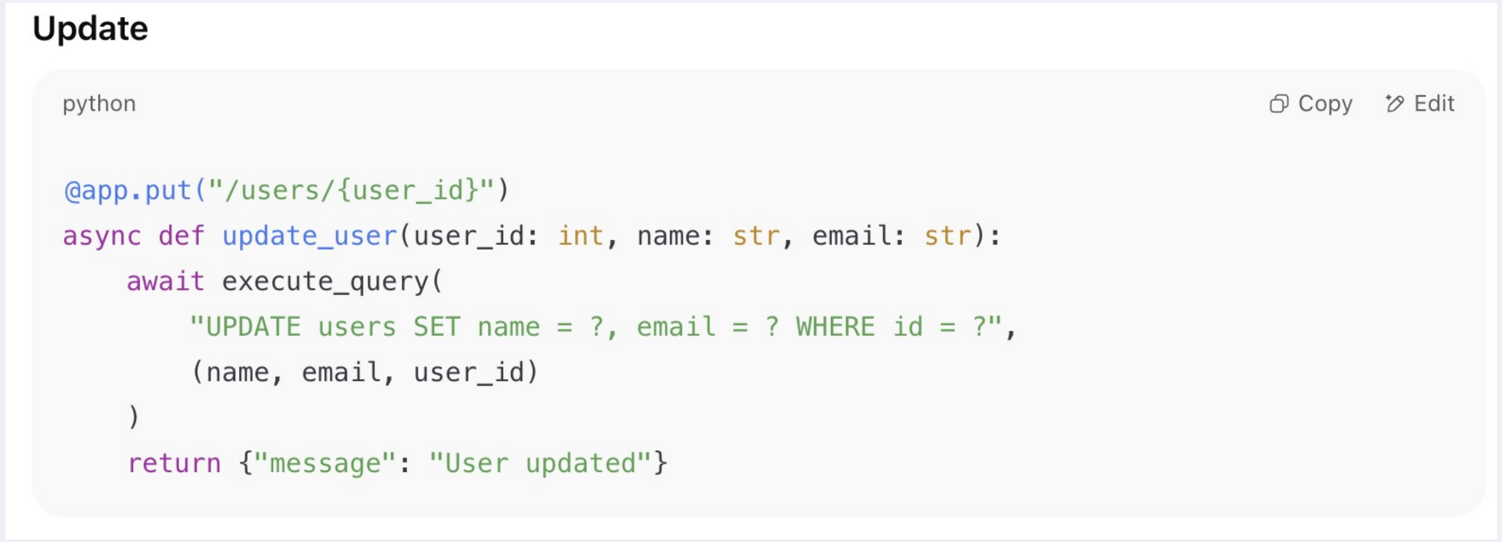


**Read**

```python
@app.get("/users")
async def get_users():
    rows = await fetch_query("SELECT * FROM users")
    return [{"id": r[0], "name": r[1], "email": r[2]} for r in rows]
```

# Integrasi Database

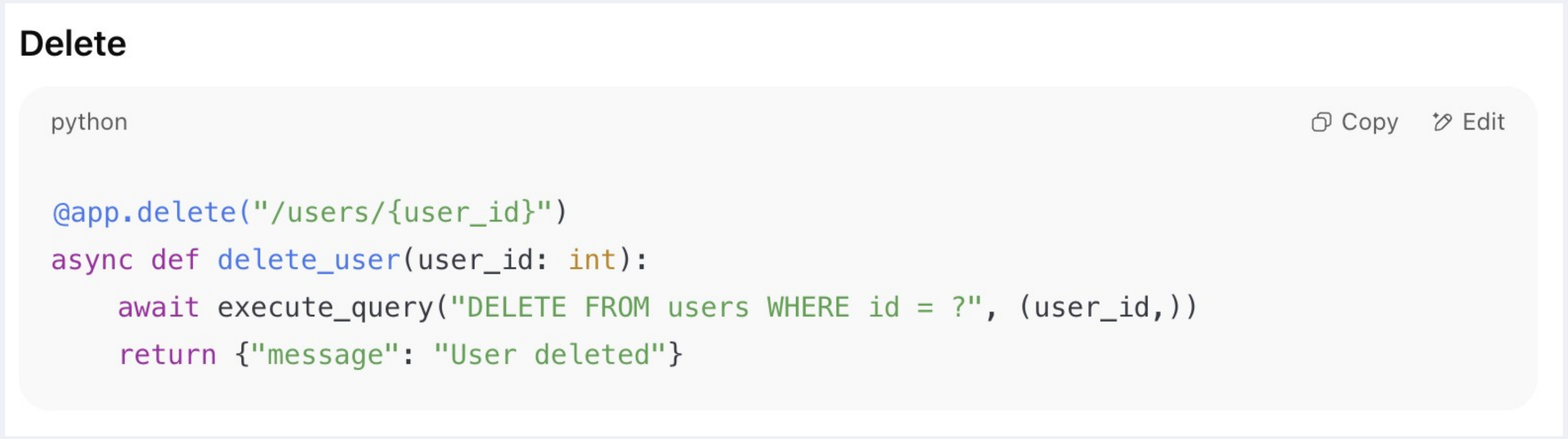
## 5. CRUD Endpoint


Update

python

Copy Edit

```python
@app.put("/users/{user_id}")
async def update_user(user_id: int, name: str, email: str):
    await execute_query(
        "UPDATE users SET name = ?, email = ? WHERE id = ?",
        (name, email, user_id)
    )
    return {"message": "User updated"}
```


Delete

python

Copy Edit

```python
@app.delete("/users/{user_id}")
async def delete_user(user_id: int):
    await execute_query("DELETE FROM users WHERE id = ?", (user_id,))
    return {"message": "User deleted"}
```

# Exercise

1. Jalankan aplikasi di local:
2. Gunakan **Swagger UI** di http://127.0.0.1:8000/docs.
3. Coba:

- Tambah 3 user.
- Ambil semua user.
- Update salah satu user.
- Hapus salah satu user.

1. Modifikasi kode agar ada pencarian user berdasarkan email.

## Outline

- Project Structure
- Type Hinting
- Introduction to FastAPI
- Modularisasi FastAPI dan Error Handling
- Middleware
- Integrasi Databse
- **Mini Project**

# Mini Project

**Tujuan:**

Menggabungkan seluruh materi sebelumnya dalam sebuah proyek FastAPI yang memiliki endpoint CRUD sederhana untuk catatan keuangan.

## 1. Spesifikasi API

- **Tambah catatan keuangan**
  **Method:** POST /catatan
  **Body:** JSON (judul, jumlah, tanggal, kategori)
  **Validasi:** Pydantic
  **Autentikasi:** Header Token

- **Lihat semua catatan**
  **Method:** GET /catatan
  **Autentikasi:** Header Token

- **Lihat catatan berdasarkan ID**
  **Method:** GET /catatan/{id}
  **Autentikasi:** Header Token

- **Hapus catatan**
  **Method:** DELETE /catatan/{id}
  **Autentikasi:** Header Token

## Mini Project

### 2. Fitur yang Digunakan

- **Pydantic** → Validasi input request
- **Autentikasi Token Sederhana** → Menggunakan X-Token di header
- **SQLite (tanpa ORM)** → Menggunakan sqlite3 dan query SQL langsung
- **CRUD** (Create, Read, Delete)

### 3. Alur Integrasi

1. **Setup Database**
   - Buat file database.py untuk koneksi dan fungsi eksekusi query
   - Pastikan tabel dibuat jika belum ada
2. **Model Pydantic**
   - Buat schemas.py untuk mendefinisikan struktur request & response
3. **Autentikasi**
   - Middleware / Dependency untuk memeriksa token dari header
4. **CRUD**
   - POST → Insert data ke SQLite
   - GET → Select data dari SQLite
   - DELETE → Hapus data dari SQLite

Terimakasih

Danke

Thank you

Gracias

Grazie

Merci

Arigatou

Obrigado

Syukron

Gamsa Hamnida

Xie-Xie

Kheilil Mamnun