RUBY ABDULLAH

PYTHON ZERO TO HERO

FROM NOTHING TO PYTHON DEVELOPER

Catatan Penulis

Halo pembaca yang budiman,

Saya, Ruby Abdullah, merasa sangat terhormat dapat berbagi pengetahuan dan pengalaman saya dalam dunia pemrograman Python melalui buku ini. Sebagai seorang AI Software Engineer dan CEO dari Ruby Thalib AI and Data Consultant, saya memiliki kesempatan untuk bekerja dengan berbagai teknologi canggih dan menghadapi beragam tantangan menarik. Buku ini, "Python Zero to Hero", adalah hasil dari perjalanan panjang saya dalam memahami dan menerapkan Python dalam berbagai aspek pengembangan perangkat lunak.

Bab 1: Pengenalan Python
Bab pertama ini akan membawa Anda mengenal Python dari dasar. Kita akan membahas sejarah singkat Python, mengapa Python menjadi pilihan utama banyak pengembang, serta dasar-dasar sintaksis dan struktur bahasa Python. Bab ini dirancang untuk pemula yang belum memiliki pengalaman dengan Python maupun pemrograman pada umumnya.

Bab 2: Pengolahan Data Menggunakan Python
Pengolahan data adalah salah satu aplikasi paling kuat dari Python. Di bab ini, kita akan mengeksplorasi berbagai pustaka Python seperti Pandas, NumPy, dan Matplotlib untuk memanipulasi, menganalisis, dan memvisualisasikan data. Anda akan belajar bagaimana membersihkan data, melakukan analisis statistik, dan membuat visualisasi yang menarik dan informatif.

Bab 3: Pembuatan REST API dengan FastAPI
Membangun REST API adalah keterampilan penting dalam pengembangan web modern. Bab ini akan membimbing Anda melalui proses pembuatan REST API menggunakan FastAPI, salah satu framework Python terbaru yang cepat dan efisien. Anda akan mempelajari cara mendefinisikan endpoint, mengelola permintaan dan respons, serta melakukan validasi data.

Saya berharap buku ini dapat menjadi panduan yang bermanfaat bagi Anda yang ingin memulai atau memperdalam pengetahuan tentang Python. Terima kasih telah memilih buku ini sebagai referensi Anda. Semoga sukses dalam perjalanan belajar Anda!

Salam hangat,

Ruby Abdullah
AI Software Engineer & CEO
Ruby Thalib AI and Data Consultant

Bab 1 Pengenalan Python

Instalasi Python pada Windows, Linux, dan Mac

Instalasi Python pada Windows

1. **Unduh Installer Python**
   - Buka situs resmi Python.
   - Unduh installer untuk Windows (contoh: `python-3.x.x.exe`).
2. **Jalankan Installer**
   - Buka file installer yang sudah diunduh.
   - Pada jendela installer, pastikan Anda mencentang opsi "Add Python to PATH".
   - Pilih "Install Now" untuk instalasi standar atau "Customize installation" untuk opsi kustom.
3. **Verifikasi Instalasi**
   - Buka Command Prompt (CMD).
   - Ketik `python --version` atau `python -V` untuk memverifikasi instalasi Python.

Instalasi Python pada Linux

1. **Perbarui Paket Manajer**
   - Buka terminal.
   - Jalankan perintah berikut untuk memperbarui manajer paket:

```
sudo apt update
sudo apt upgrade
```

2. **Instal Python**
   - Untuk distribusi berbasis Debian/Ubuntu, gunakan:

```
sudo apt install python3
```

- Untuk distribusi berbasis Fedora, gunakan:

```
sudo dnf install python3
```

3. **Verifikasi Instalasi**
   - Ketik `python3 --version` atau `python3 -V` di terminal untuk memverifikasi instalasi.

Instalasi Python pada Mac

1. **Gunakan Homebrew**
   - Jika belum memiliki Homebrew, instal terlebih dahulu dengan perintah berikut di terminal:

```
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

2. **Instal Python**
   - Setelah Homebrew terinstal, jalankan perintah berikut untuk menginstal Python:

```
brew install python
```

3. **Verifikasi Instalasi**
   - Ketik `python3 --version` atau `python3 -V` di terminal untuk memverifikasi instalasi.

Pengaturan Lingkungan Pengembangan

1. **Mengatur Lingkungan Virtual**
   - Disarankan untuk menggunakan lingkungan virtual untuk setiap proyek Python agar paket dan dependensi tetap terisolasi.
   - Instal `virtualenv` dengan perintah berikut:

```
pip install virtualenv
  • Buat dan aktifkan lingkungan virtual:
```

```
virtualenv myenv
source myenv/bin/activate  # Linux/Mac
.\myenv\Scripts\activate  # Windows
```

2. **Menginstal Editor Teks atau IDE**
   - Gunakan editor teks atau IDE yang mendukung Python seperti Visual Studio Code, PyCharm, atau Jupyter Notebook.

Dengan mengikuti langkah-langkah di atas, Anda sudah siap untuk memulai perjalanan belajar Python dengan lingkungan yang sudah terkonfigurasi dengan baik di Windows, Linux, atau Mac. Selanjutnya, kita akan membahas tipe data dasar dalam Python.

Tipe Data

Dalam Python, terdapat beberapa tipe data dasar yang sering digunakan dalam pemrograman. Berikut adalah penjelasan tentang tipe data dasar tersebut:

1. Integer

Integer adalah tipe data yang digunakan untuk menyimpan bilangan bulat (tanpa desimal). Contohnya, 5, -3, 100.

Contoh:

```
x = 5
y = -3
z = 100
print(type(x))  # Output: <class 'int'>
print(type(y))  # Output: <class 'int'>
print(type(z))  # Output: <class 'int'>
```

2. Float

Float adalah tipe data yang digunakan untuk menyimpan bilangan desimal. Contohnya, 5.0, -3.14, 100.25.

Contoh:

```
a = 5.0
b = -3.14
c = 100.25
print(type(a))  # Output: <class 'float'>
print(type(b))  # Output: <class 'float'>
print(type(c))  # Output: <class 'float'>
```

3. String

String adalah tipe data yang digunakan untuk menyimpan teks. String dapat didefinisikan menggunakan tanda kutip tunggal (' ') atau tanda kutip ganda (" ").

Contoh:

```
s1 = 'Hello'
s2 = "Python"
print(type(s1))  # Output: <class 'str'>
print(type(s2))  # Output: <class 'str'>
```

4. Boolean

Boolean adalah tipe data yang hanya memiliki dua nilai: True atau False. Tipe data ini sering digunakan dalam operasi logika dan kondisi.

Contoh:

```
is_active = True
is_closed = False
print(type(is_active))  # Output: <class 'bool'>
print(type(is_closed))  # Output: <class 'bool'>
```

Operasi Dasar dengan Tipe Data

Operasi Aritmatika dengan Integer dan Float:

```
x = 10
y = 3
print(x + y)  # Output: 13
print(x - y)  # Output: 7
print(x * y)  # Output: 30
print(x / y)  # Output: 3.3333333333333335
print(x // y)  # Output: 3 (pembagian bulat)
print(x % y)  # Output: 1 (sisa bagi)
print(x ** y)  # Output: 1000 (pangkat)
```

Operasi dengan String:

```
s1 = 'Hello'
s2 = 'World'
```

```
print(s1 + ' ' + s2)  # Output: 'Hello World' (penggabungan string)
print(s1 * 3)  # Output: 'HelloHelloHello' (pengulangan string)
```

Operasi dengan Boolean:

```
a = True
b = False
print(a and b)  # Output: False
print(a or b)  # Output: True
print(not a)  # Output: False
```

Dengan memahami tipe data dasar ini, Anda dapat mulai memanipulasi data dalam program Python Anda. Tipe data ini merupakan fondasi penting yang akan sering digunakan dalam berbagai operasi pemrograman di Python. Selanjutnya, kita akan membahas tentang kondisi jika-maka dalam Python.

Kondisi Jika-Maka (Conditional Statements)

Kondisi jika-maka dalam Python memungkinkan Anda untuk membuat keputusan di dalam program berdasarkan kondisi tertentu. Struktur ini sering digunakan untuk mengontrol alur program berdasarkan perbandingan atau pengujian logika.

1. Pernyataan `if`

Pernyataan `if` digunakan untuk mengeksekusi blok kode tertentu jika suatu kondisi terpenuhi (benar).

Contoh:

```
x = 10
if x > 5:
    print("x lebih besar dari 5")  # Output: x lebih besar dari 5
```

2. Pernyataan `if-else`

Pernyataan `if-else` digunakan ketika Anda ingin mengeksekusi blok kode alternatif jika kondisi `if` tidak terpenuhi (salah).

Contoh:

```
x = 3
if x > 5:
    print("x lebih besar dari 5")
else:
    print("x tidak lebih besar dari 5")  # Output: x tidak lebih besar dari 5
```

3. Pernyataan `if-elif-else`

Pernyataan `if-elif-else` digunakan untuk menguji beberapa kondisi. Jika kondisi `if` pertama tidak terpenuhi, program akan menguji kondisi `elif` berikutnya, dan seterusnya. Jika tidak ada kondisi yang terpenuhi, blok `else` akan dieksekusi.

Contoh:

```
x = 7
if x > 10:
    print("x lebih besar dari 10")
elif x > 5:
    print("x lebih besar dari 5 tetapi tidak lebih besar dari 10")  # Output: x lebih
besar dari 5 tetapi tidak lebih besar dari 10
else:
    print("x tidak lebih besar dari 5")
```

4. Pernyataan Bersarang (Nested `if`)

Anda dapat menggunakan pernyataan `if` di dalam pernyataan `if` lainnya untuk membuat keputusan yang lebih kompleks.

Contoh:

```
x = 8
if x > 5:
    print("x lebih besar dari 5")  # Output: x lebih besar dari 5
    if x % 2 == 0:
        print("x adalah bilangan genap")  # Output: x adalah bilangan genap
    else:
        print("x adalah bilangan ganjil")
else:
    print("x tidak lebih besar dari 5")
```

Contoh Aplikasi Praktis

Contoh 1: Mengecek Kelulusan Berdasarkan Nilai

```
nilai = 75
if nilai >= 85:
    print("Nilai A")
elif nilai >= 70:
    print("Nilai B")  # Output: Nilai B
elif nilai >= 60:
    print("Nilai C")
else:
    print("Nilai D")
```

Contoh 2: Mengecek Apakah Angka Positif, Negatif, atau Nol

```
angka = -3
if angka > 0:
    print("Angka positif")
elif angka == 0:
    print("Angka nol")
else:
    print("Angka negatif")  # Output: Angka negatif
```

Dengan memahami kondisi jika-maka, Anda dapat membuat program yang dapat mengambil keputusan berdasarkan kondisi tertentu. Ini adalah salah satu konsep dasar yang sangat penting dalam pemrograman, yang memungkinkan Anda untuk mengontrol alur eksekusi program Anda. Selanjutnya, kita akan membahas tentang looping dalam Python.

Looping

Looping atau perulangan memungkinkan Anda untuk mengeksekusi blok kode berulang kali selama kondisi tertentu terpenuhi. Python mendukung dua jenis loop utama: `while` loop dan `for` loop.

1. `while` Loop

`while` loop akan terus mengeksekusi blok kode selama kondisi yang diberikan bernilai benar (True). Jika kondisi menjadi salah (False), eksekusi loop akan berhenti.

Struktur `while` loop:

```
while kondisi:
    # Blok kode yang akan diulang
```

Contoh:

```
i = 1
while i <= 5:
    print(i)  # Output: 1 2 3 4 5
    i += 1
```

2. `for` Loop

`for` loop digunakan untuk mengulangi elemen-elemen dalam urutan (seperti list, tuple, string, atau range).

Struktur `for` loop:

```
for elemen in urutan:
    # Blok kode yang akan diulang
```

Contoh:

```
for i in range(1, 6):
    print(i)  # Output: 1 2 3 4 5
```

Contoh dengan list:

```
buah = ["apel", "jeruk", "pisang"]
for item in buah:
    print(item)
    # Output:
    # apel
    # jeruk
    # pisang
```

Pernyataan `break` dan `continue`

`break`

`break` digunakan untuk menghentikan loop sebelum kondisi loop terpenuhi.

Contoh:

```
i = 1
while i <= 10:
    if i == 5:
        break
    print(i)  # Output: 1 2 3 4
    i += 1
```

`continue`

`continue` digunakan untuk melewati iterasi saat ini dan melanjutkan ke iterasi berikutnya.

Contoh:

```
for i in range(1, 6):
    if i == 3:
        continue
    print(i)  # Output: 1 2 4 5
```

Loop Bersarang (Nested Loops)

Loop bersarang adalah loop di dalam loop lainnya. Ini digunakan ketika Anda perlu melakukan perulangan dalam perulangan.

Contoh:

```
for i in range(1, 4):  # Loop luar
    for j in range(1, 4):  # Loop dalam
        print(f'i: {i}, j: {j}')
    # Output:
    # i: 1, j: 1
    # i: 1, j: 2
    # i: 1, j: 3
    # i: 2, j: 1
    # i: 2, j: 2
    # i: 2, j: 3
    # i: 3, j: 1
```

```
        # i: 3, j: 2
        # i: 3, j: 3
```

Contoh Aplikasi Praktis

Contoh 1: Menghitung Jumlah Bilangan dalam Rentang

```
total = 0
for i in range(1, 11):
    total += i
print("Total:", total)  # Output: Total: 55
```

Contoh 2: Menampilkan Bilangan Genap dalam Rentang

```
for i in range(1, 11):
    if i % 2 == 0:
        print(i)  # Output: 2 4 6 8 10
```

Dengan memahami konsep looping, Anda dapat mengeksekusi tugas berulang dengan efisien dalam program Python Anda. Looping adalah alat yang sangat kuat dalam pemrograman, memungkinkan Anda untuk mengelola dan memproses data dalam jumlah besar dengan mudah. Selanjutnya, kita akan membahas tentang struktur data dalam Python.

Struktur Data

Struktur data adalah cara untuk mengorganisir dan menyimpan data agar bisa diakses dan dimodifikasi dengan efisien. Python memiliki beberapa struktur data built-in yang sangat berguna: `list`, `tuple`, `dictionary`, dan `set`.

1. List

`List` adalah struktur data yang dapat menyimpan berbagai jenis data dalam urutan tertentu. List bersifat mutable, artinya elemen-elemen di dalamnya dapat diubah setelah list dibuat.

Contoh:

```
# Membuat list
buah = ["apel", "jeruk", "pisang"]

# Mengakses elemen dalam list
```

```
print(buah[0])  # Output: apel
print(buah[1])  # Output: jeruk

# Menambahkan elemen ke dalam list
buah.append("mangga")
print(buah)  # Output: ['apel', 'jeruk', 'pisang', 'mangga']

# Menghapus elemen dari list
buah.remove("jeruk")
print(buah)  # Output: ['apel', 'pisang', 'mangga']

# Mengubah elemen dalam list
buah[1] = "anggur"
print(buah)  # Output: ['apel', 'anggur', 'mangga']
```

2. Tuple

`Tuple` mirip dengan `list`, namun bersifat immutable, artinya elemen-elemen di dalamnya tidak dapat diubah setelah tuple dibuat.

Contoh:

```
# Membuat tuple
angka = (1, 2, 3, 4, 5)

# Mengakses elemen dalam tuple
print(angka[0])  # Output: 1
print(angka[1])  # Output: 2

# Tuple bersifat immutable
# angka[1] = 10  # Akan menghasilkan error

# Tuple juga dapat menyimpan berbagai jenis data
data = (1, "hello", 3.14)
print(data)  # Output: (1, 'hello', 3.14)
```

3. Dictionary

`Dictionary` adalah struktur data yang menyimpan data dalam pasangan kunci-nilai. Kunci harus unik dan bersifat immutable, sedangkan nilai dapat diubah.

Contoh:

```
# Membuat dictionary
mahasiswa = {
    "nama": "Budi",
    "umur": 21,
    "jurusan": "Informatika"
}

# Mengakses nilai dalam dictionary
print(mahasiswa["nama"])  # Output: Budi
print(mahasiswa["umur"])  # Output: 21

# Menambahkan pasangan kunci-nilai
mahasiswa["universitas"] = "Universitas XYZ"
print(mahasiswa)
# Output: {'nama': 'Budi', 'umur': 21, 'jurusan': 'Informatika', 'universitas':
'Universitas XYZ'}

# Mengubah nilai dalam dictionary
mahasiswa["umur"] = 22
print(mahasiswa)  # Output: {'nama': 'Budi', 'umur': 22, 'jurusan': 'Informatika',
'universitas': 'Universitas XYZ'}

# Menghapus pasangan kunci-nilai
del mahasiswa["jurusan"]
print(mahasiswa)  # Output: {'nama': 'Budi', 'umur': 22, 'universitas': 'Universitas
XYZ'}
```

4. Set

`Set` adalah struktur data yang menyimpan koleksi elemen unik dan tidak berurutan. Set digunakan untuk operasi matematika seperti union, intersection, dan difference.

Contoh:

```
# Membuat set
angka = {1, 2, 3, 4, 5}

# Menambahkan elemen ke dalam set
angka.add(6)
print(angka)  # Output: {1, 2, 3, 4, 5, 6}

# Menghapus elemen dari set
angka.remove(3)
print(angka)  # Output: {1, 2, 4, 5, 6}

# Operasi set: union, intersection, difference
```

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
print(set1 | set2)  # Union: {1, 2, 3, 4, 5}
print(set1 & set2)  # Intersection: {3}
print(set1 - set2)  # Difference: {1, 2}
```

Contoh Aplikasi Praktis

Contoh 1: Mengelola Daftar Belanja dengan List

```
belanja = ["susu", "roti", "telur"]
belanja.append("gula")
print(belanja)  # Output: ['susu', 'roti', 'telur', 'gula']
```

Contoh 2: Menggunakan Dictionary untuk Informasi Kontak

```
kontak = {
    "Andi": "081234567890",
    "Budi": "089876543210"
}
print(kontak["Andi"])  # Output: 081234567890
kontak["Cici"] = "082112345678"
print(kontak)  # Output: {'Andi': '081234567890', 'Budi': '089876543210', 'Cici':
'082112345678'}
```

Contoh 3: Menghitung Frekuensi Elemen dalam List menggunakan Dictionary

```
data = ["apel", "jeruk", "apel", "mangga", "jeruk", "apel"]
frekuensi = {}
for item in data:
    if item in frekuensi:
        frekuensi[item] += 1
    else:
        frekuensi[item] = 1
print(frekuensi)  # Output: {'apel': 3, 'jeruk': 2, 'mangga': 1}
```

Contoh 4: Operasi Set untuk Menghilangkan Duplikat dalam List

```
data = [1, 2, 3, 1, 2, 4, 5]
data_set = set(data)
print(data_set)  # Output: {1, 2, 3, 4, 5}
```

Dengan memahami struktur data ini, Anda dapat mengorganisir dan memanipulasi data dalam program Python Anda dengan cara yang lebih efisien dan efektif. Struktur data ini sangat penting dalam pemrograman karena memungkinkan Anda untuk mengelola data dalam berbagai bentuk dan ukuran. Selanjutnya, kita akan membahas tentang cara membuat fungsi dalam Python.

Function (Fungsi)

Fungsi adalah blok kode yang dapat digunakan kembali yang melakukan tugas tertentu. Fungsi membantu membuat kode lebih modular, mudah dibaca, dan mudah dipelihara. Python memungkinkan Anda untuk mendefinisikan fungsi Anda sendiri atau menggunakan fungsi bawaan.

1. Definisi Fungsi

Fungsi didefinisikan menggunakan kata kunci `def` diikuti oleh nama fungsi, tanda kurung, dan titik dua. Blok kode fungsi harus diindentasi.

Struktur dasar:

```
def nama_fungsi(parameter1, parameter2, ...):
    # Blok kode fungsi
    return nilai
```

Contoh:

```
def salam(nama):
    return f"Halo, {nama}!"

print(salam("Ruby"))  # Output: Halo, Ruby!
```

2. Memanggil Fungsi

Untuk menggunakan fungsi, Anda harus memanggilnya dengan menyebutkan nama fungsi diikuti oleh tanda kurung yang berisi argumen (jika ada).

Contoh:

```
def tambah(a, b):
    return a + b
```

```
hasil = tambah(5, 3)
print(hasil)  # Output: 8
```

3. Parameter dan Argumen

Fungsi dapat memiliki parameter, yang merupakan variabel yang digunakan untuk menerima nilai saat fungsi dipanggil. Nilai yang Anda berikan kepada parameter saat memanggil fungsi disebut argumen.

Contoh:

```
def cetak_info(nama, umur):
    print(f"Nama: {nama}, Umur: {umur}")

cetak_info("Budi", 25)  # Output: Nama: Budi, Umur: 25
```

4. Parameter Default

Anda dapat memberikan nilai default kepada parameter. Jika argumen tidak diberikan saat fungsi dipanggil, nilai default akan digunakan.

Contoh:

```
def salam(nama, pesan="Selamat datang!"):
    return f"Halo, {nama}! {pesan}"

print(salam("Budi"))  # Output: Halo, Budi! Selamat datang!
print(salam("Budi", "Selamat pagi!"))  # Output: Halo, Budi! Selamat pagi!
```

5. Fungsi dengan Jumlah Argumen yang Tidak Terbatas

Anda dapat menggunakan `*args` dan `**kwargs` untuk menangani jumlah argumen yang tidak terbatas dalam fungsi.

Contoh `*args`:

```
def jumlahkan(*args):
    total = 0
```

```
    for angka in args:
        total += angka
    return total

print(jumlahkan(1, 2, 3, 4))  # Output: 10
```

Contoh `kwargs`:**

```
def cetak_info(**kwargs):
    for kunci, nilai in kwargs.items():
        print(f"{kunci}: {nilai}")

cetak_info(nama="Budi", umur=25, kota="Jakarta")
# Output:
# nama: Budi
# umur: 25
# kota: Jakarta
```

6. Fungsi Lambda

Fungsi lambda adalah fungsi anonim kecil yang dapat memiliki sejumlah argumen tetapi hanya satu ekspresi. Fungsi lambda sering digunakan untuk tugas-tugas sederhana yang membutuhkan fungsi singkat.

Contoh:

```
tambah = lambda a, b: a + b
print(tambah(5, 3))  # Output: 8

genap = lambda x: x % 2 == 0
print(genap(4))  # Output: True
print(genap(5))  # Output: False
```

Contoh Aplikasi Praktis

Contoh 1: Fungsi untuk Menghitung Faktorial

```
def faktorial(n):
    if n == 0:
        return 1
    else:
        return n * faktorial(n - 1)
```

```
print(faktorial(5))  # Output: 120
```

Contoh 2: Fungsi untuk Mengecek Bilangan Prima

```
def adalah_prima(n):
    if n <= 1:
        return False
    for i in range(2, n):
        if n % i == 0:
            return False
    return True

print(adalah_prima(11))  # Output: True
print(adalah_prima(4))  # Output: False
```

Dengan memahami konsep fungsi dalam Python, Anda dapat menulis kode yang lebih modular, rapi, dan mudah dipelihara. Fungsi memungkinkan Anda untuk membagi program menjadi bagian-bagian kecil yang dapat dikelola dan diuji secara terpisah. Selanjutnya, kita akan membahas tentang cara mengimpor dan menginstal library serta package dalam Python.

Import Library, Install Library, dan Import Package

Python memiliki ekosistem library dan package yang sangat kaya, yang memungkinkan Anda untuk menambah fungsionalitas ke dalam program Anda tanpa harus menulis kode dari awal. Library dan package ini bisa diimpor ke dalam program Python Anda untuk memanfaatkan fungsionalitas tambahan.

1. Mengimpor Library dan Package

Mengimpor library atau package dalam Python dilakukan menggunakan pernyataan `import`. Anda dapat mengimpor seluruh library atau hanya bagian tertentu dari library tersebut.

Mengimpor seluruh library:

```
import math

# Menggunakan fungsi dalam library math
print(math.sqrt(16))  # Output: 4.0
```

Mengimpor bagian tertentu dari library:

```
from math import sqrt

# Menggunakan fungsi sqrt langsung
print(sqrt(16))  # Output: 4.0
```

Memberi alias pada library:

```
import numpy as np

# Menggunakan alias np untuk mengakses fungsi dalam numpy
array = np.array([1, 2, 3])
print(array)  # Output: [1 2 3]
```

2. Menginstal Library dengan pip

`pip` adalah manajer paket untuk Python yang digunakan untuk menginstal library dan package dari Python Package Index (PyPI).

Menginstal library:

```
pip install nama_library
```

Contoh: Menginstal library numpy:

```
pip install numpy
```

Menginstal versi tertentu dari library:

```
pip install nama_library==versi
```

Contoh: Menginstal versi tertentu dari numpy:

```
pip install numpy==1.21.0
```

Mengupgrade library:

```
pip install --upgrade nama_library
```

Contoh: Mengupgrade numpy:

```
pip install --upgrade numpy
```

Menghapus library:

```
pip uninstall nama_library
```

Contoh: Menghapus numpy:

```
pip uninstall numpy
```

3. Mengelola Dependensi dengan Requirements File

Anda dapat mengelola dependensi proyek Anda dengan menggunakan file `requirements.txt`. File ini berisi daftar library yang dibutuhkan oleh proyek Anda.

Contoh isi file `requirements.txt`:

```
numpy==1.21.0
pandas==1.3.3
matplotlib==3.4.3
```

Menginstal library dari file `requirements.txt`:

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

4. Contoh Penggunaan Library dan Package

Contoh 1: Menggunakan numpy untuk operasi array:

```
import numpy as np

array = np.array([1, 2, 3, 4])
```

```
print(array)  # Output: [1 2 3 4]
print(np.mean(array))  # Output: 2.5
```

Contoh 2: Menggunakan pandas untuk manipulasi data:

```
import pandas as pd

data = {
    'Nama': ['Alice', 'Bob', 'Charlie'],
    'Umur': [24, 27, 22]
}
df = pd.DataFrame(data)
print(df)
# Output:
#       Nama  Umur
# 0    Alice    24
# 1      Bob    27
# 2  Charlie    22
```

Contoh 3: Menggunakan matplotlib untuk visualisasi data:

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y = [1, 4, 9, 16, 25]

plt.plot(x, y)
plt.xlabel('x')
plt.ylabel('y')
plt.title('Grafik x vs y')
plt.show()
```

Dengan memahami cara mengimpor dan menginstal library serta package, Anda dapat memanfaatkan berbagai fungsionalitas tambahan yang disediakan oleh komunitas Python. Ini akan memperluas kemampuan program Anda dan memungkinkan Anda untuk menyelesaikan tugas-tugas kompleks dengan lebih efisien. Selanjutnya, kita akan membahas tentang Pemrograman Berorientasi Objek (OOP) dalam Python.

Pemrograman Berorientasi Objek (OOP)

Pemrograman Berorientasi Objek (OOP) adalah paradigma pemrograman yang menggunakan "objek" dan "kelas" untuk menciptakan model berdasarkan konsep dunia nyata. OOP memudahkan pengelolaan kode dengan memecah program menjadi bagian-bagian kecil yang

disebut objek. Python mendukung OOP dan memungkinkan Anda untuk membuat kelas dan objek.

1. Kelas dan Objek

Kelas adalah cetak biru atau template untuk membuat objek. Kelas mendefinisikan atribut dan metode yang dimiliki oleh objek.

Objek adalah instance dari kelas. Setiap objek memiliki atribut dan metode yang didefinisikan oleh kelasnya.

Contoh: Membuat kelas dan objek

```
class Anjing:
    def __init__(self, nama, umur):
        self.nama = nama
        self.umur = umur

    def menggonggong(self):
        return f"{self.nama} menggonggong!"

# Membuat objek dari kelas Anjing
anjing1 = Anjing("Buddy", 3)
anjing2 = Anjing("Lucy", 5)

print(anjing1.nama)  # Output: Buddy
print(anjing2.umur)  # Output: 5
print(anjing1.menggonggong())  # Output: Buddy menggonggong!
```

2. Atribut dan Metode

Atribut adalah variabel yang dimiliki oleh objek. Atribut didefinisikan dalam metode `__init__` (konstruktor) kelas.

Metode adalah fungsi yang dimiliki oleh objek. Metode didefinisikan di dalam kelas dan biasanya memanipulasi atau mengakses atribut objek.

Contoh:

```
class Mobil:
    def __init__(self, merk, model, tahun):
        self.merk = merk
```

```
        self.model = model
        self.tahun = tahun

    def deskripsi(self):
        return f"{self.merk} {self.model}, tahun {self.tahun}"

# Membuat objek dari kelas Mobil
mobil1 = Mobil("Toyota", "Camry", 2020)
print(mobil1.deskripsi())  # Output: Toyota Camry, tahun 2020
```

3. Enkapsulasi

Enkapsulasi adalah konsep OOP yang membatasi akses langsung ke atribut atau metode objek dan hanya memungkinkan manipulasi melalui metode yang didefinisikan. Ini melindungi data dari perubahan yang tidak diinginkan.

Contoh:

```
class BankAccount:
    def __init__(self, nama, saldo):
        self.__nama = nama  # Atribut privat
        self.__saldo = saldo  # Atribut privat

    def deposit(self, jumlah):
        if jumlah > 0:
            self.__saldo += jumlah
        else:
            print("Jumlah harus positif!")

    def withdraw(self, jumlah):
        if 0 < jumlah <= self.__saldo:
            self.__saldo -= jumlah
        else:
            print("Jumlah tidak valid atau saldo tidak cukup!")

    def cek_saldo(self):
        return self.__saldo

# Membuat objek dari kelas BankAccount
akun = BankAccount("Alice", 1000)
akun.deposit(500)
print(akun.cek_saldo())  # Output: 1500
akun.withdraw(2000)  # Output: Jumlah tidak valid atau saldo tidak cukup!
```

4. Pewarisan (Inheritance)

Pewarisan adalah konsep OOP yang memungkinkan sebuah kelas untuk mewarisi atribut dan metode dari kelas lain. Kelas yang diwarisi disebut kelas induk (parent class), sedangkan kelas yang mewarisi disebut kelas anak (child class).

Contoh:

```
class Hewan:
    def __init__(self, nama):
        self.nama = nama

    def berbicara(self):
        pass  # Metode abstrak

class Anjing(Hewan):
    def berbicara(self):
        return "Guk guk!"

class Kucing(Hewan):
    def berbicara(self):
        return "Meong!"

# Membuat objek dari kelas Anjing dan Kucing
anjing = Anjing("Buddy")
kucing = Kucing("Whiskers")

print(anjing.nama)  # Output: Buddy
print(anjing.berbicara())  # Output: Guk guk!
print(kucing.nama)  # Output: Whiskers
print(kucing.berbicara())  # Output: Meong!
```

5. Polimorfisme (Polymorphism)

Polimorfisme adalah konsep OOP yang memungkinkan metode yang sama untuk memiliki perilaku yang berbeda pada kelas yang berbeda. Polimorfisme memungkinkan penggunaan metode yang sama pada objek dari berbagai kelas.

Contoh:

```
class Burung:
    def bersuara(self):
        return "Cuit cuit!"

class Ayam(Burung):
```

```
    def bersuara(self):
        return "Kukuruyuk!"

class Itik(Burung):
    def bersuara(self):
        return "Kwek kwek!"

# Polimorfisme dengan menggunakan metode bersuara()
burung = Burung()
ayam = Ayam()
itik = Itik()

for hewan in (burung, ayam, itik):
    print(hewan.bersuara())
    # Output:
    # Cuit cuit!
    # Kukuruyuk!
    # Kwek kwek!
```

Dengan memahami konsep-konsep OOP seperti kelas dan objek, atribut dan metode, enkapsulasi, pewarisan, dan polimorfisme, Anda dapat menulis kode yang lebih modular, terstruktur, dan mudah dipelihara. OOP memungkinkan Anda untuk menciptakan model yang lebih dekat dengan konsep dunia nyata, membuat program Anda lebih intuitif dan fleksibel. Selanjutnya, kita akan membahas lebih dalam tentang penggunaan Python dalam pengolahan data.

Bab 2 Pengolahan data menggunakan Python

Pengolahan Data Array dengan NumPy

NumPy adalah library Python yang digunakan untuk pengolahan data berbasis array. NumPy menyediakan objek array multidimensi yang kuat serta berbagai fungsi matematis untuk melakukan operasi pada array tersebut. Berikut adalah pengenalan dasar penggunaan NumPy.

1. Instalasi NumPy

Untuk menggunakan NumPy, Anda perlu menginstalnya terlebih dahulu. Anda dapat menginstalnya menggunakan `pip`.

```
pip install numpy
```

2. Membuat dan Mengakses Array

Membuat Array

Array NumPy dapat dibuat menggunakan fungsi `array()` dari library NumPy. Anda juga dapat menggunakan fungsi lain seperti `zeros()`, `ones()`, dan `arange()` untuk membuat array.

Contoh:

```
import numpy as np

# Membuat array dari daftar Python
array_1d = np.array([1, 2, 3, 4, 5])
print(array_1d)  # Output: [1 2 3 4 5]

# Membuat array 2D (matriks)
array_2d = np.array([[1, 2, 3], [4, 5, 6]])
print(array_2d)
# Output:
# [[1 2 3]
#  [4 5 6]]

# Membuat array berisi nol
array_zeros = np.zeros((2, 3))
print(array_zeros)
```

```
# Output:
# [[0. 0. 0.]
#  [0. 0. 0.]]

# Membuat array berisi satu
array_ones = np.ones((3, 2))
print(array_ones)
# Output:
# [[1. 1.]
#  [1. 1.]
#  [1. 1.]]

# Membuat array dengan nilai dalam rentang tertentu
array_arange = np.arange(1, 10, 2)
print(array_arange)  # Output: [1 3 5 7 9]
```

Mengakses Elemen Array

Anda dapat mengakses elemen dalam array menggunakan indeks. NumPy menggunakan indeks berbasis nol (indeks pertama dimulai dari 0).

Contoh:

```
array = np.array([10, 20, 30, 40, 50])

# Mengakses elemen dengan indeks
print(array[0])  # Output: 10
print(array[4])  # Output: 50

# Mengakses elemen dalam array 2D
array_2d = np.array([[1, 2, 3], [4, 5, 6]])
print(array_2d[0, 1])  # Output: 2
print(array_2d[1, 2])  # Output: 6
```

3. Operasi pada Array

NumPy menyediakan berbagai operasi aritmatika yang dapat dilakukan pada array.

Contoh:

```
array_a = np.array([1, 2, 3])
array_b = np.array([4, 5, 6])

# Penjumlahan
```

```
print(array_a + array_b)  # Output: [5 7 9]

# Pengurangan
print(array_a - array_b)  # Output: [-3 -3 -3]

# Perkalian
print(array_a * array_b)  # Output: [4 10 18]

# Pembagian
print(array_a / array_b)  # Output: [0.25 0.4  0.5 ]
```

4. Operasi Matematika Lanjutan

NumPy juga menyediakan berbagai fungsi matematis untuk operasi lanjutan.

Contoh:

```
array = np.array([1, 4, 9, 16])

# Akar kuadrat
print(np.sqrt(array))  # Output: [1. 2. 3. 4.]

# Logaritma
print(np.log(array))  # Output: [0.         1.38629436 2.19722458 2.77258872]

# Sinus
print(np.sin(array))  # Output: [0.84147098 -0.7568025   0.41211849 -0.28790332]
```

5. Manipulasi Bentuk Array

Anda dapat mengubah bentuk (reshape) array menggunakan fungsi `reshape()`.

Contoh:

```
array = np.arange(1, 13)
print(array)  # Output: [ 1  2  3  4  5  6  7  8  9 10 11 12]

# Mengubah bentuk array menjadi 3x4
array_reshaped = array.reshape((3, 4))
print(array_reshaped)
# Output:
# [[ 1  2  3  4]
```

```
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
```

6. Indexing dan Slicing

Anda dapat menggunakan teknik indexing dan slicing untuk mengakses sebagian elemen dari array.

Contoh:

```
array = np.array([10, 20, 30, 40, 50, 60])

# Mengambil elemen dari indeks 1 sampai 4
print(array[1:5])  # Output: [20 30 40 50]

# Mengambil elemen dari awal sampai indeks 3
print(array[:4])  # Output: [10 20 30 40]

# Mengambil elemen dari indeks 2 sampai akhir
print(array[2:])  # Output: [30 40 50 60]

# Mengambil elemen dengan langkah 2
print(array[::2])  # Output: [10 30 50]
```

7. Operasi Linier dan Statistik

NumPy menyediakan fungsi untuk operasi linier dan statistik.

Contoh:

```
array = np.array([1, 2, 3, 4, 5])

# Mean (rata-rata)
print(np.mean(array))  # Output: 3.0

# Median
print(np.median(array))  # Output: 3.0

# Standar Deviasi
print(np.std(array))  # Output: 1.4142135623730951

# Matriks dot product
matrix_a = np.array([[1, 2], [3, 4]])
```

```
matrix_b = np.array([[5, 6], [7, 8]])
print(np.dot(matrix_a, matrix_b))
# Output:
# [[19 22]
#  [43 50]]
```

Dengan memahami dasar-dasar pengolahan data array menggunakan NumPy, Anda dapat melakukan operasi matematis dan manipulasi data dengan lebih efisien. NumPy adalah alat yang sangat kuat untuk analisis data, dan memahami penggunaannya adalah langkah penting dalam pengolahan data menggunakan Python. Selanjutnya, kita akan membahas tentang pengolahan data CSV menggunakan pandas.

Pengolahan Data CSV dengan Pandas

Pandas adalah library Python yang sangat populer untuk pengolahan data. Library ini menyediakan struktur data dan alat analisis data yang mudah digunakan, termasuk untuk pengolahan data CSV. CSV (Comma-Separated Values) adalah format file yang sering digunakan untuk menyimpan data tabular. Berikut adalah pengenalan dasar penggunaan pandas untuk pengolahan data CSV.

1. Instalasi Pandas

Untuk menggunakan pandas, Anda perlu menginstalnya terlebih dahulu. Anda dapat menginstalnya menggunakan `pip`.

```
pip install pandas
```

2. Membaca Data dari File CSV

Anda dapat membaca data dari file CSV ke dalam DataFrame pandas menggunakan fungsi `read_csv()`.

Contoh:

```
import pandas as pd

# Membaca data dari file CSV
data = pd.read_csv('data.csv')
```

```
# Menampilkan 5 baris pertama dari DataFrame
print(data.head())
```

3. Menulis Data ke File CSV

Anda dapat menyimpan DataFrame ke file CSV menggunakan fungsi `to_csv()`.

Contoh:

```
import pandas as pd

# Membuat DataFrame
data = pd.DataFrame({
    'Nama': ['Alice', 'Bob', 'Charlie'],
    'Umur': [24, 27, 22]
})

# Menyimpan DataFrame ke file CSV
data.to_csv('data_output.csv', index=False)
```

4. Manipulasi Data dengan DataFrame

Pandas menyediakan berbagai fungsi untuk manipulasi data dalam DataFrame.

Contoh:

```
import pandas as pd

# Membaca data dari file CSV
data = pd.read_csv('data.csv')

# Menampilkan informasi umum tentang DataFrame
print(data.info())

# Mengganti nama kolom
data = data.rename(columns={'OldName': 'NewName'})

# Menambahkan kolom baru
data['KolomBaru'] = data['Kolom1'] + data['Kolom2']

# Menghapus kolom
data = data.drop(columns=['KolomBaru'])
```

```
# Menampilkan statistik deskriptif
print(data.describe())
```

5. Seleksi dan Filter Data

Anda dapat memilih dan memfilter data dalam DataFrame menggunakan indexing, slicing, dan kondisi logis.

Contoh:

```
import pandas as pd

# Membaca data dari file CSV
data = pd.read_csv('data.csv')

# Seleksi kolom
print(data['Kolom1'])
print(data[['Kolom1', 'Kolom2']])

# Seleksi baris dengan kondisi logis
filtered_data = data[data['Umur'] > 25]
print(filtered_data)

# Seleksi baris dan kolom dengan iloc (index based)
print(data.iloc[0, 1])  # Baris pertama, kolom kedua
print(data.iloc[0:3, 0:2])  # Tiga baris pertama, dua kolom pertama

# Seleksi baris dan kolom dengan loc (label based)
print(data.loc[0, 'Kolom1'])  # Baris pertama, kolom 'Kolom1'
print(data.loc[0:2, ['Kolom1', 'Kolom2']])  # Tiga baris pertama, dua kolom pertama
```

6. Menggabungkan dan Menggabungkan Data

Pandas memungkinkan Anda untuk menggabungkan DataFrame menggunakan operasi seperti `concat`, `merge`, dan `join`.

Contoh:

```
import pandas as pd

# Membuat DataFrame
data1 = pd.DataFrame({
    'Nama': ['Alice', 'Bob'],
```

```
    'Umur': [24, 27]
})
data2 = pd.DataFrame({
    'Nama': ['Charlie', 'David'],
    'Umur': [22, 23]
})

# Menggabungkan DataFrame secara vertikal
data_vertikal = pd.concat([data1, data2], ignore_index=True)
print(data_vertikal)

# Menggabungkan DataFrame secara horizontal
data3 = pd.DataFrame({
    'Nama': ['Alice', 'Bob'],
    'Kota': ['Jakarta', 'Bandung']
})
data_horizontal = pd.merge(data1, data3, on='Nama')
print(data_horizontal)
```

7. Mengelompokkan Data

Pandas menyediakan fungsi `groupby()` untuk mengelompokkan data dan melakukan operasi agregasi.

Contoh:

```
import pandas as pd

# Membuat DataFrame
data = pd.DataFrame({
    'Nama': ['Alice', 'Bob', 'Charlie', 'Alice', 'Bob'],
    'Nilai': [85, 90, 88, 92, 85]
})

# Mengelompokkan data berdasarkan kolom 'Nama'
grouped_data = data.groupby('Nama').mean()
print(grouped_data)
```

8. Menghandle Missing Data

Pandas menyediakan fungsi untuk menangani data yang hilang (missing data).

Contoh:

```
import pandas as pd

# Membuat DataFrame dengan missing values
data = pd.DataFrame({
    'Nama': ['Alice', 'Bob', 'Charlie'],
    'Umur': [24, None, 22],
    'Nilai': [85, 90, None]
})

# Mengisi missing values dengan nilai tertentu
data_filled = data.fillna({'Umur': 25, 'Nilai': 0})
print(data_filled)

# Menghapus baris dengan missing values
data_dropped = data.dropna()
print(data_dropped)
```

Dengan memahami dasar-dasar pengolahan data CSV menggunakan pandas, Anda dapat melakukan berbagai operasi manipulasi dan analisis data dengan efisien. Pandas adalah alat yang sangat kuat dan fleksibel untuk bekerja dengan data tabular dalam Python. Selanjutnya, kita akan membahas tentang pengolahan data gambar menggunakan OpenCV dan Pillow.

Pengolahan Data Gambar dengan OpenCV dan Pillow

Pengolahan data gambar adalah proses manipulasi dan analisis gambar digital. Python memiliki beberapa library yang kuat untuk pengolahan gambar, termasuk OpenCV dan Pillow. Berikut adalah pengenalan dasar penggunaan OpenCV dan Pillow untuk pengolahan data gambar.

1. Instalasi OpenCV dan Pillow

Untuk menggunakan OpenCV dan Pillow, Anda perlu menginstalnya terlebih dahulu. Anda dapat menginstalnya menggunakan `pip`.

```
pip install opencv-python-headless pillow
```

2. Membaca dan Menyimpan Gambar

Dengan OpenCV:

```
import cv2
```

```
# Membaca gambar
image = cv2.imread('image.jpg')

# Menyimpan gambar
cv2.imwrite('output.jpg', image)
```

Dengan Pillow:

```
from PIL import Image

# Membaca gambar
image = Image.open('image.jpg')

# Menyimpan gambar
image.save('output.jpg')
```

3. Menampilkan Gambar

Dengan OpenCV:

```
import cv2

# Membaca gambar
image = cv2.imread('image.jpg')

# Menampilkan gambar
cv2.imshow('Image', image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Dengan Pillow:

```
from PIL import Image

# Membaca gambar
image = Image.open('image.jpg')

# Menampilkan gambar
image.show()
```

4. Mengubah Ukuran Gambar

Dengan OpenCV:

```
import cv2

# Membaca gambar
image = cv2.imread('image.jpg')

# Mengubah ukuran gambar
resized_image = cv2.resize(image, (width, height))

# Menyimpan gambar yang telah diubah ukurannya
cv2.imwrite('resized_image.jpg', resized_image)
```

Dengan Pillow:

```
from PIL import Image

# Membaca gambar
image = Image.open('image.jpg')

# Mengubah ukuran gambar
resized_image = image.resize((width, height))

# Menyimpan gambar yang telah diubah ukurannya
resized_image.save('resized_image.jpg')
```

5. Memotong (Crop) Gambar

Dengan OpenCV:

```
import cv2

# Membaca gambar
image = cv2.imread('image.jpg')

# Memotong gambar (x, y, width, height)
cropped_image = image[y:y+height, x:x+width]

# Menyimpan gambar yang telah dipotong
cv2.imwrite('cropped_image.jpg', cropped_image)
```

Dengan Pillow:

```
from PIL import Image

# Membaca gambar
image = Image.open('image.jpg')

# Memotong gambar (left, upper, right, lower)
cropped_image = image.crop((left, upper, right, lower))

# Menyimpan gambar yang telah dipotong
cropped_image.save('cropped_image.jpg')
```

6. Mengubah Warna Gambar

Dengan OpenCV:

```
import cv2

# Membaca gambar
image = cv2.imread('image.jpg')

# Mengubah gambar menjadi grayscale
gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

# Menyimpan gambar grayscale
cv2.imwrite('gray_image.jpg', gray_image)
```

Dengan Pillow:

```
from PIL import Image

# Membaca gambar
image = Image.open('image.jpg')

# Mengubah gambar menjadi grayscale
gray_image = image.convert('L')

# Menyimpan gambar grayscale
gray_image.save('gray_image.jpg')
```

7. Rotasi Gambar

Dengan OpenCV:

```
import cv2

# Membaca gambar
image = cv2.imread('image.jpg')

# Mendapatkan ukuran gambar
(h, w) = image.shape[:2]

# Mendefinisikan pusat rotasi
center = (w // 2, h // 2)

# Mendefinisikan matriks rotasi
matrix = cv2.getRotationMatrix2D(center, angle, 1.0)

# Melakukan rotasi gambar
rotated_image = cv2.warpAffine(image, matrix, (w, h))

# Menyimpan gambar yang telah diputar
cv2.imwrite('rotated_image.jpg', rotated_image)
```

Dengan Pillow:

```
from PIL import Image

# Membaca gambar
image = Image.open('image.jpg')

# Melakukan rotasi gambar
rotated_image = image.rotate(angle)

# Menyimpan gambar yang telah diputar
rotated_image.save('rotated_image.jpg')
```

8. Menambahkan Teks pada Gambar

Dengan OpenCV:

```
import cv2

# Membaca gambar
image = cv2.imread('image.jpg')

# Menambahkan teks pada gambar
cv2.putText(image, 'Hello, OpenCV!', (x, y), cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 255,
255), 2)
```

```
# Menyimpan gambar yang telah diberi teks
cv2.imwrite('text_image.jpg', image)
```

Dengan Pillow:

```
from PIL import Image, ImageDraw, ImageFont

# Membaca gambar
image = Image.open('image.jpg')

# Membuat objek ImageDraw
draw = ImageDraw.Draw(image)

# Menentukan font (pastikan font path sesuai dengan sistem Anda)
font = ImageFont.truetype('arial.ttf', size=45)

# Menambahkan teks pada gambar
draw.text((x, y), 'Hello, Pillow!', fill='white', font=font)

# Menyimpan gambar yang telah diberi teks
image.save('text_image.jpg')
```

Dengan memahami dasar-dasar pengolahan data gambar menggunakan OpenCV dan Pillow, Anda dapat melakukan berbagai operasi manipulasi gambar dengan efisien. OpenCV dan Pillow adalah alat yang sangat kuat untuk bekerja dengan gambar dalam Python, dan memahami penggunaannya adalah langkah penting dalam pengolahan data gambar. Selanjutnya, kita akan membahas tentang bagaimana mengintegrasikan semua konsep yang telah dipelajari untuk membangun aplikasi pengolahan data yang lebih kompleks.

Bab 3 Pembuatan REST API dengan FastAPI

Pengenalan FastAPI

FastAPI adalah framework modern untuk membangun aplikasi web dan API menggunakan Python. Dikembangkan oleh Sebastián Ramírez, FastAPI dirancang untuk menawarkan kinerja tinggi dengan penulisan kode yang bersih dan mudah dipahami.

Apa itu FastAPI?

FastAPI adalah framework web untuk Python yang memungkinkan Anda untuk membangun API RESTful dengan cepat dan efisien. FastAPI menggunakan tipe anotasi Python untuk menghasilkan dokumentasi API secara otomatis dan menawarkan validasi input secara otomatis menggunakan Pydantic.

Fitur Utama FastAPI:

- **Kinerja Tinggi:** Dibangun di atas ASGI (Asynchronous Server Gateway Interface) dan menggunakan Uvicorn sebagai server ASGI default, FastAPI mampu menangani permintaan dengan sangat cepat.
- **Penggunaan Tipe Anotasi:** Menggunakan tipe anotasi Python untuk menghasilkan dokumentasi API yang lengkap dan interaktif secara otomatis.
- **Validasi Input Otomatis:** Menggunakan Pydantic untuk validasi input dan penanganan kesalahan secara otomatis.
- **Dokumentasi API Otomatis:** Secara otomatis menghasilkan dokumentasi API menggunakan Swagger UI dan Redoc.
- **Asinkron:** Mendukung operasi asynchronous, memungkinkan penanganan I/O-bound dan operasi network-bound dengan efisien.

Keunggulan FastAPI Dibandingkan Framework Lain

- **Kinerja yang Cepat:** FastAPI menawarkan kinerja yang setara dengan Node.js dan Go, menjadikannya salah satu framework Python tercepat.
- **Mudah Digunakan:** Penulisan kode dengan FastAPI intuitif dan mudah dipahami, dengan dukungan dokumentasi yang sangat baik.

- **Validasi Input yang Kuat:** Dengan Pydantic, FastAPI memastikan bahwa data yang diterima oleh API selalu divalidasi dengan benar.
- **Dokumentasi Interaktif:** Swagger UI dan Redoc menyediakan antarmuka pengguna interaktif untuk menguji dan memahami endpoint API Anda.
- **Pengembangan Cepat:** Dengan fitur seperti dokumentasi otomatis dan validasi input, FastAPI memungkinkan pengembangan aplikasi dengan cepat dan efisien.

Konsep Dasar dalam FastAPI

1. **Aplikasi FastAPI:** Sebuah aplikasi FastAPI adalah instance dari kelas `FastAPI`.
2. **Route dan Endpoint:** Route adalah URL yang digunakan untuk mengakses endpoint tertentu dalam aplikasi API. Endpoint adalah fungsi yang menangani permintaan HTTP ke route tertentu.
3. **Path Parameters:** Parameter yang ditentukan dalam path URL. Contoh: `/items/{item_id}`.
4. **Query Parameters:** Parameter yang ditentukan setelah tanda tanya `?` dalam URL. Contoh: `/items?skip=0&limit=10`.
5. **Request Body:** Data yang dikirim oleh klien dalam permintaan HTTP POST, PUT, atau PATCH.
6. **Response Model:** Model yang digunakan untuk menentukan struktur data yang akan dikembalikan oleh API.

Contoh Aplikasi FastAPI Sederhana:

```
from fastapi import FastAPI

app = FastAPI()

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

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

Penjelasan:

- `FastAPI()` menciptakan instance aplikasi FastAPI.

- `@app.get("/")` adalah dekorator yang mendefinisikan endpoint GET pada root URL (`/`).
- `read_root()` adalah fungsi yang menangani permintaan GET ke root URL dan mengembalikan pesan "Hello, World!".
- `@app.get("/items/{item_id}")` adalah dekorator yang mendefinisikan endpoint GET dengan path parameter `item_id` .
- `read_item(item_id: int, q: str = None)` adalah fungsi yang menangani permintaan GET ke `/items/{item_id}` dan mengembalikan `item_id` serta parameter query `q` (jika ada).

Dengan memahami konsep dasar ini, Anda dapat mulai membangun API menggunakan FastAPI. Selanjutnya, kita akan membahas tentang instalasi dan persiapan lingkungan untuk memulai proyek FastAPI.

Instalasi dan Persiapan Lingkungan

Instalasi FastAPI dan Uvicorn

FastAPI dan Uvicorn adalah dua komponen utama yang Anda perlukan untuk memulai pembangunan aplikasi API. FastAPI adalah framework web, sedangkan Uvicorn adalah server ASGI yang cepat untuk menjalankan aplikasi FastAPI.

Langkah-langkah instalasi:

Buat Virtual Environment:

1. Membuat virtual environment untuk proyek Anda memastikan bahwa dependensi proyek terisolasi dari sistem Python Anda.

```
python -m venv myenv
```

Aktifkan Virtual Environment:

1. Untuk mengaktifkan virtual environment, jalankan perintah berikut:
   - Di Windows:

```
myenv\Scripts\activate
```

- Di macOS/Linux:

```
source myenv/bin/activate
```

Instal FastAPI dan Uvicorn:

1. Gunakan `pip` untuk menginstal FastAPI dan Uvicorn.

```
pip install fastapi uvicorn
```

Menyiapkan Proyek FastAPI

Setelah menginstal FastAPI dan Uvicorn, langkah berikutnya adalah menyiapkan proyek FastAPI.

Buat Struktur Direktori Proyek:

1. Buat direktori untuk proyek Anda dan buat file utama untuk aplikasi FastAPI.

```
mkdir my_fastapi_project
cd my_fastapi_project
touch main.py
```

Menulis Aplikasi FastAPI Pertama:

1. Buka `main.py` dan tulis aplikasi FastAPI pertama Anda.

```
from fastapi import FastAPI

app = FastAPI()

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

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

Menjalankan Aplikasi FastAPI:

1. Jalankan aplikasi FastAPI menggunakan Uvicorn.

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

2. Penjelasan:
   - `main:app` menunjukkan file `main.py` dan objek `app` di dalamnya.
   - `--reload` memungkinkan server untuk otomatis memuat ulang ketika ada perubahan pada kode.

Mengakses Aplikasi:

1. Buka browser dan akses `http://127.0.0.1:8000` untuk melihat pesan "Hello, World!". Anda juga dapat mengakses dokumentasi otomatis yang dihasilkan oleh FastAPI di `http://127.0.0.1:8000/docs` (Swagger UI) dan `http://127.0.0.1:8000/redoc` (Redoc).

Struktur Direktori Proyek FastAPI

Untuk proyek FastAPI yang lebih besar dan kompleks, penting untuk memiliki struktur direktori yang terorganisir. Berikut adalah contoh struktur direktori yang disarankan:

```
my_fastapi_project/
├── app/
│   ├── __init__.py
│   ├── main.py
│   ├── models.py
│   ├── schemas.py
│   ├── crud.py
│   ├── database.py
│   ├── routers/
│   │   ├── __init__.py
│   │   ├── items.py
│   │   └── users.py
│   └── core/
│       ├── __init__.py
│       ├── config.py
│       └── security.py
├── requirements.txt
└── README.md
```

Penjelasan:

- `app/` - Direktori utama yang berisi kode aplikasi.
- `main.py` - File utama untuk menjalankan aplikasi FastAPI.
- `models.py` - Berisi definisi model database.
- `schemas.py` - Berisi skema Pydantic untuk validasi input dan output.
- `crud.py` - Berisi operasi CRUD (Create, Read, Update, Delete).
- `database.py` - Berisi konfigurasi dan koneksi database.
- `routers/` - Berisi modul router untuk endpoint aplikasi.
- `core/` - Berisi konfigurasi dan pengaturan keamanan aplikasi.
- `requirements.txt` - Berisi daftar dependensi proyek.
- `README.md` - Berisi dokumentasi proyek.

Dengan struktur direktori yang terorganisir, Anda dapat mengelola proyek FastAPI dengan lebih mudah dan memastikan bahwa kode tetap terstruktur dan dapat dipelihara. Selanjutnya, kita akan mulai membangun endpoint dasar dalam aplikasi FastAPI.

Membuat Endpoint Dasar

Endpoint dalam FastAPI adalah titik akhir (URL) yang dapat diakses oleh klien untuk berinteraksi dengan aplikasi. Endpoint didefinisikan menggunakan decorator seperti `@app.get()`, `@app.post()`, dan sebagainya, yang menunjukkan metode HTTP yang digunakan.

Membuat Aplikasi FastAPI Sederhana

Mulailah dengan membuat file baru bernama `main.py` dan mendefinisikan aplikasi FastAPI dasar.

```
from fastapi import FastAPI

app = FastAPI()

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

Penjelasan:

- `FastAPI()` menciptakan instance aplikasi FastAPI.
- `@app.get("/")` adalah decorator yang mendefinisikan endpoint GET pada root URL (`/`).
- `read_root()` adalah fungsi yang menangani permintaan GET ke root URL dan mengembalikan pesan "Hello, World!".

Menyusun Route dan Endpoint Dasar

Menambahkan Endpoint GET:

Tambahkan endpoint untuk mendapatkan item berdasarkan ID.

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

Penjelasan:

- `@app.get("/items/{item_id}")` mendefinisikan endpoint GET dengan path parameter `item_id`.
- `item_id: int` menunjukkan bahwa `item_id` harus berupa integer.

Menambahkan Endpoint POST:

Tambahkan endpoint untuk membuat item baru.

```
from pydantic import BaseModel

class Item(BaseModel):
    name: str
    description: str = None
    price: float
    tax: float = None

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

Penjelasan:

- `BaseModel` dari Pydantic digunakan untuk mendefinisikan model data `Item`.
- `@app.post("/items/")` mendefinisikan endpoint POST untuk membuat item baru.
- `create_item(item: Item)` menerima objek `Item` sebagai parameter dan mengembalikan informasi tentang item yang dibuat.

Menggunakan Dekorator untuk Mendefinisikan Endpoint

FastAPI menyediakan beberapa decorator untuk mendefinisikan endpoint dengan metode HTTP yang berbeda, seperti `@app.get()`, `@app.post()`, `@app.put()`, `@app.delete()`, dan lain-lain.

Contoh Endpoint PUT dan DELETE:

Tambahkan endpoint untuk memperbarui dan menghapus item.

```
@app.put("/items/{item_id}")
def update_item(item_id: int, item: Item):
    return {"item_id": item_id, "item_name": item.name, "item_price": item.price}

@app.delete("/items/{item_id}")
def delete_item(item_id: int):
    return {"item_id": item_id, "status": "deleted"}
```

Penjelasan:

- `@app.put("/items/{item_id}")` mendefinisikan endpoint PUT untuk memperbarui item berdasarkan `item_id`.
- `update_item(item_id: int, item: Item)` menerima parameter `item_id` dan objek `Item`, lalu mengembalikan informasi tentang item yang diperbarui.
- `@app.delete("/items/{item_id}")` mendefinisikan endpoint DELETE untuk menghapus item berdasarkan `item_id`.
- `delete_item(item_id: int)` mengembalikan status penghapusan item.

Contoh Aplikasi Lengkap

Berikut adalah contoh aplikasi FastAPI lengkap dengan endpoint GET, POST, PUT, dan DELETE.

```
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

class Item(BaseModel):
    name: str
    description: str = None
    price: float
    tax: float = None

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

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

@app.post("/items/")
```

```
def create_item(item: Item):
    return {"item_name": item.name, "item_price": item.price}

@app.put("/items/{item_id}")
def update_item(item_id: int, item: Item):
    return {"item_id": item_id, "item_name": item.name, "item_price": item.price}

@app.delete("/items/{item_id}")
def delete_item(item_id: int):
    return {"item_id": item_id, "status": "deleted"}
```

Menjalankan Aplikasi:
Untuk menjalankan aplikasi, buka terminal dan jalankan perintah berikut:

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

Akses aplikasi di browser melalui URL `http://127.0.0.1:8000`. Anda dapat mengakses dokumentasi otomatis di `http://127.0.0.1:8000/docs` (Swagger UI) dan `http://127.0.0.1:8000/redoc` (Redoc).

Dengan memahami cara membuat endpoint dasar menggunakan FastAPI, Anda dapat mulai membangun API yang lebih kompleks dan menangani berbagai permintaan HTTP. Selanjutnya, Anda dapat menjelajahi cara menangani request dan response secara lebih mendetail.

Handling Request dan Response

Dalam pengembangan API dengan FastAPI, memahami cara menangani request dan response adalah hal yang penting. FastAPI menyediakan berbagai fitur untuk mempermudah proses ini, termasuk penggunaan Pydantic untuk validasi data dan tipe anotasi untuk memastikan tipe data yang tepat.

Menangani Request Body dengan Pydantic

FastAPI menggunakan Pydantic untuk validasi dan parsing request body. Anda dapat mendefinisikan model data dengan Pydantic untuk memastikan bahwa data yang dikirimkan oleh klien sesuai dengan yang diharapkan.

Contoh:

```
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

class Item(BaseModel):
    name: str
    description: str = None
    price: float
    tax: float = None

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

Penjelasan:

- `Item` adalah model Pydantic yang mendefinisikan struktur data yang diterima oleh endpoint.
- Endpoint `/items/` menerima request body berupa objek `Item` dan mengembalikan data yang sama.

Mengelola Path dan Query Parameters

FastAPI memungkinkan Anda untuk mengelola path parameters dan query parameters dengan mudah.

Path Parameters:
Path parameters adalah bagian dari URL yang digunakan untuk mengidentifikasi sumber daya tertentu.

Contoh:

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

Penjelasan:

- `item_id: int` mendefinisikan path parameter `item_id` yang harus berupa integer.

Query Parameters:
Query parameters adalah bagian dari URL yang digunakan untuk mengirimkan data non-hierarkis.

Contoh:

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

Penjelasan:

- `skip` dan `limit` adalah query parameters dengan nilai default masing-masing 0 dan 10.

Menangani Response Model dan Status Code

Anda dapat menggunakan Pydantic untuk mendefinisikan model data yang akan dikembalikan sebagai response. Anda juga dapat menentukan status code yang dikembalikan oleh endpoint.

Contoh:

```
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel

app = FastAPI()

class Item(BaseModel):
    name: str
    description: str = None
    price: float
    tax: float = None

class ItemResponse(BaseModel):
    name: str
    price_with_tax: float

@app.post("/items/", response_model=ItemResponse, status_code=201)
def create_item(item: Item):
    item_response = ItemResponse(
        name=item.name,
        price_with_tax=item.price + (item.tax if item.tax else 0)
```

```
    )
    return item_response
```

Penjelasan:

- `ItemResponse` adalah model Pydantic yang mendefinisikan struktur data response.
- `response_model=ItemResponse` menunjukkan bahwa response dari endpoint harus sesuai dengan model `ItemResponse`.
- `status_code=201` menunjukkan bahwa status code untuk response adalah 201 (Created).

Menangani Request Body yang Kompleks

FastAPI memungkinkan Anda untuk menangani request body yang kompleks dengan menggunakan nested models dan list.

Contoh:

```
from fastapi import FastAPI
from pydantic import BaseModel
from typing import List

app = FastAPI()

class SubItem(BaseModel):
    name: str
    description: str = None

class Item(BaseModel):
    name: str
    description: str = None
    price: float
    tax: float = None
    sub_items: List[SubItem] = []

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

Penjelasan:

- `SubItem` adalah model Pydantic yang digunakan sebagai nested model dalam `Item`.

- `sub_items` adalah daftar (`List`) dari objek `SubItem`.

Dengan memahami cara menangani request dan response dalam FastAPI, Anda dapat membangun API yang robust dan efisien. FastAPI memberikan fleksibilitas dan kemudahan dalam menangani berbagai jenis data dan memastikan bahwa API Anda berfungsi dengan baik dan aman. Selanjutnya, Anda dapat mempelajari cara mengimplementasikan operasi CRUD dalam aplikasi FastAPI.

Operasi CRUD dengan FastAPI

Operasi CRUD (Create, Read, Update, Delete) adalah dasar dari aplikasi yang mengelola data. FastAPI memudahkan implementasi operasi CRUD dengan menggunakan model data yang divalidasi oleh Pydantic dan integrasi yang baik dengan berbagai database.

Membuat Model Data dengan Pydantic

Pydantic digunakan untuk mendefinisikan model data yang akan divalidasi saat request masuk dan keluar dari API.

Contoh:

```
from pydantic import BaseModel

class Item(BaseModel):
    name: str
    description: str = None
    price: float
    tax: float = None
```

Penjelasan:

- `Item` adalah model Pydantic yang mendefinisikan struktur data dengan tipe data yang jelas untuk setiap atribut.

Implementasi Operasi Create

Operasi `Create` digunakan untuk menambah data baru ke dalam sistem.

Contoh:

```
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

class Item(BaseModel):
    name: str
    description: str = None
    price: float
    tax: float = None

items = []

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

Penjelasan:

- `create_item` menerima objek `Item` sebagai request body dan menambahkannya ke daftar `items`.

Implementasi Operasi Read

Operasi `Read` digunakan untuk membaca atau mengambil data dari sistem.

Contoh:

```
@app.get("/items/", response_model=List[Item])
def read_items():
    return items

@app.get("/items/{item_id}", response_model=Item)
def read_item(item_id: int):
    return items[item_id]
```

Penjelasan:

- `read_items` mengembalikan seluruh daftar `items`.
- `read_item` mengembalikan item berdasarkan `item_id`.

Implementasi Operasi Update

Operasi `Update` digunakan untuk memperbarui data yang ada dalam sistem.

Contoh:

```
@app.put("/items/{item_id}", response_model=Item)
def update_item(item_id: int, item: Item):
    items[item_id] = item
    return item
```

Penjelasan:

- `update_item` menerima `item_id` dan objek `Item` sebagai request body dan memperbarui item dalam daftar `items` berdasarkan `item_id`.

Implementasi Operasi Delete

Operasi `Delete` digunakan untuk menghapus data dari sistem.

Contoh:

```
@app.delete("/items/{item_id}", response_model=Item)
def delete_item(item_id: int):
    return items.pop(item_id)
```

Penjelasan:

- `delete_item` menghapus item dari daftar `items` berdasarkan `item_id` dan mengembalikan item yang dihapus.

Contoh Aplikasi CRUD Sederhana

Berikut adalah contoh lengkap aplikasi FastAPI dengan operasi CRUD:

```
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
```

```
from typing import List

app = FastAPI()

class Item(BaseModel):
    name: str
    description: str = None
    price: float
    tax: float = None

items = []

@app.post("/items/", response_model=Item)
def create_item(item: Item):
    items.append(item)
    return item

@app.get("/items/", response_model=List[Item])
def read_items():
    return items

@app.get("/items/{item_id}", response_model=Item)
def read_item(item_id: int):
    if item_id >= len(items):
        raise HTTPException(status_code=404, detail="Item not found")
    return items[item_id]

@app.put("/items/{item_id}", response_model=Item)
def update_item(item_id: int, item: Item):
    if item_id >= len(items):
        raise HTTPException(status_code=404, detail="Item not found")
    items[item_id] = item
    return item

@app.delete("/items/{item_id}", response_model=Item)
def delete_item(item_id: int):
    if item_id >= len(items):
        raise HTTPException(status_code=404, detail="Item not found")
    return items.pop(item_id)
```

Penjelasan:

- `create_item` menambahkan item baru ke daftar `items`.
- `read_items` mengembalikan semua item dalam daftar `items`.
- `read_item` mengembalikan item berdasarkan `item_id`, dengan pengecekan apakah item ada atau tidak.

- `update_item` memperbarui item berdasarkan `item_id`, dengan pengecekan apakah item ada atau tidak.
- `delete_item` menghapus item berdasarkan `item_id`, dengan pengecekan apakah item ada atau tidak.

Dengan memahami dan mengimplementasikan operasi CRUD ini, Anda dapat mengelola data dalam aplikasi FastAPI dengan efektif. Operasi CRUD adalah dasar dari banyak aplikasi web dan API, memungkinkan interaksi penuh dengan data dalam sistem. Selanjutnya, Anda dapat menjelajahi cara mengintegrasikan aplikasi FastAPI dengan database untuk menyimpan dan mengambil data dengan lebih efisien.

Integrasi dengan Database

Integrasi dengan database adalah langkah penting dalam membangun aplikasi yang persisten, dimana data disimpan di database dan dapat diakses kembali. FastAPI dapat dengan mudah diintegrasikan dengan berbagai jenis database, salah satunya menggunakan SQLAlchemy untuk database relasional.

Menghubungkan FastAPI dengan Database menggunakan SQLAlchemy

SQLAlchemy adalah toolkit SQL dan Object-Relational Mapping (ORM) untuk Python yang memungkinkan Anda untuk berinteraksi dengan database menggunakan objek Python.

Instalasi SQLAlchemy

Instal SQLAlchemy dan `asyncpg` (untuk PostgreSQL) atau `aiomysql` (untuk MySQL) untuk mendukung operasi asinkron.

```
pip install sqlalchemy databases asyncpg
```

Mengkonfigurasi Koneksi Database

Buat file baru bernama `database.py` untuk mengkonfigurasi koneksi ke database.

Contoh:

```
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

SQLALCHEMY_DATABASE_URL = "postgresql://user:password@localhost/dbname"

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

Base = declarative_base()
```

Penjelasan:

- `create_engine` membuat instance koneksi database.
- `SessionLocal` menyediakan sesi untuk berinteraksi dengan database.
- `Base` digunakan sebagai dasar untuk membuat model ORM.

Membuat Model Database

Buat model database dengan mewarisi `Base`. Model ini akan direpresentasikan sebagai tabel dalam database.

Contoh:

```
from sqlalchemy import Column, Integer, String, Float
from .database import Base

class Item(Base):
    __tablename__ = "items"

    id = Column(Integer, primary_key=True, index=True)
    name = Column(String, index=True)
    description = Column(String, index=True)
    price = Column(Float)
    tax = Column(Float)
```

Penjelasan:

- `__tablename__` menentukan nama tabel dalam database.
- Setiap kolom direpresentasikan oleh instance `Column` dengan tipe data yang sesuai.

Membuat dan Mengelola Skema Database

Tambahkan fungsi untuk membuat skema database berdasarkan model yang telah didefinisikan.

Contoh:

```
from .database import engine, Base

def init_db():
    Base.metadata.create_all(bind=engine)
```

Panggil fungsi ini saat aplikasi mulai untuk memastikan skema database dibuat.

Menggunakan Dependency Injection untuk Sesi Database

Gunakan dependency injection untuk mengelola sesi database di FastAPI.

Contoh:

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

models.Base.metadata.create_all(bind=engine)

app = FastAPI()

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

Penjelasan:

- `get_db` adalah fungsi dependency yang menghasilkan sesi database dan menutupnya setelah digunakan.

Membuat Operasi CRUD dengan Database

Implementasikan operasi CRUD menggunakan sesi database.

Contoh:

```
@app.post("/items/", response_model=schemas.Item)
def create_item(item: schemas.ItemCreate, db: Session = Depends(get_db)):
    db_item = models.Item(**item.dict())
    db.add(db_item)
    db.commit()
    db.refresh(db_item)
    return db_item

@app.get("/items/{item_id}", response_model=schemas.Item)
def read_item(item_id: int, db: Session = Depends(get_db)):
    db_item = db.query(models.Item).filter(models.Item.id == item_id).first()
    if db_item is None:
        raise HTTPException(status_code=404, detail="Item not found")
    return db_item

@app.put("/items/{item_id}", response_model=schemas.Item)
def update_item(item_id: int, item: schemas.ItemCreate, db: Session = Depends(get_db)):
    db_item = db.query(models.Item).filter(models.Item.id == item_id).first()
    if db_item is None:
        raise HTTPException(status_code=404, detail="Item not found")
    for key, value in item.dict().items():
        setattr(db_item, key, value)
    db.commit()
    db.refresh(db_item)
    return db_item

@app.delete("/items/{item_id}", response_model=schemas.Item)
def delete_item(item_id: int, db: Session = Depends(get_db)):
    db_item = db.query(models.Item).filter(models.Item.id == item_id).first()
    if db_item is None:
        raise HTTPException(status_code=404, detail="Item not found")
    db.delete(db_item)
    db.commit()
    return db_item
```

Penjelasan:

- `create_item`: Menambahkan item baru ke database.
- `read_item`: Mengambil item dari database berdasarkan `item_id`.

- `update_item`: Memperbarui item di database berdasarkan `item_id`.
- `delete_item`: Menghapus item dari database berdasarkan `item_id`.

Contoh Aplikasi CRUD dengan Database

Berikut adalah contoh lengkap aplikasi FastAPI dengan operasi CRUD yang terintegrasi dengan database:

```
from fastapi import FastAPI, Depends, HTTPException
from sqlalchemy.orm import Session
from typing import List

from . import models, schemas
from .database import SessionLocal, engine

models.Base.metadata.create_all(bind=engine)

app = FastAPI()

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

@app.post("/items/", response_model=schemas.Item)
def create_item(item: schemas.ItemCreate, db: Session = Depends(get_db)):
    db_item = models.Item(**item.dict())
    db.add(db_item)
    db.commit()
    db.refresh(db_item)
    return db_item

@app.get("/items/", response_model=List[schemas.Item])
def read_items(skip: int = 0, limit: int = 10, db: Session = Depends(get_db)):
    items = db.query(models.Item).offset(skip).limit(limit).all()
    return items

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

```
@app.put("/items/{item_id}", response_model=schemas.Item)
def update_item(item_id: int, item: schemas.ItemCreate, db: Session = Depends(get_db)):
    db_item = db.query(models.Item).filter(models.Item.id == item_id).first()
    if db_item is None:
        raise HTTPException(status_code=404, detail="Item not found")
    for key, value in item.dict().items():
        setattr(db_item, key, value)
    db.commit()
    db.refresh(db_item)
    return db_item

@app.delete("/items/{item_id}", response_model=schemas.Item)
def delete_item(item_id: int, db: Session = Depends(get_db)):
    db_item = db.query(models.Item).filter(models.Item.id == item_id).first()
    if db_item is None:
        raise HTTPException(status_code=404, detail="Item not found")
    db.delete(db_item)
    db.commit()
    return db_item
```

Dengan integrasi ini, Anda dapat menyimpan dan mengambil data dari database, memastikan bahwa data tetap persisten dan dapat diakses kembali di masa mendatang. Integrasi dengan database adalah langkah penting untuk membangun aplikasi yang skalabel dan dapat diandalkan. Selanjutnya, Anda dapat mengeksplorasi autentikasi dan otorisasi untuk mengamankan aplikasi Anda.

Autentikasi dan Otorisasi

Autentikasi dan otorisasi adalah dua konsep penting dalam keamanan aplikasi. Autentikasi memastikan bahwa pengguna yang mengakses aplikasi adalah pengguna yang sah, sedangkan otorisasi menentukan apa yang dapat dilakukan oleh pengguna tersebut dalam aplikasi.

Menyiapkan Autentikasi Menggunakan OAuth2 dengan Password Flow

OAuth2 adalah protokol standar untuk otentikasi dan otorisasi. FastAPI menyediakan dukungan bawaan untuk OAuth2.

Langkah-langkah:

1. **Instalasi FastAPI OAuth2:**

```
pip install python-multipart
```

Membuat Model Pengguna dan Skema:

1. Buat model pengguna dan skema untuk memvalidasi data pengguna.

```
from sqlalchemy import Column, Integer, String
from .database import Base

class User(Base):
    __tablename__ = "users"

    id = Column(Integer, primary_key=True, index=True)
    username = Column(String, unique=True, index=True)
    hashed_password = Column(String)

from pydantic import BaseModel

class UserCreate(BaseModel):
    username: str
    password: str

class UserInDB(UserCreate):
    hashed_password: str

class UserOut(BaseModel):
    id: int
    username: str
```

Membuat Fungsi Hashing Password:

1. Gunakan `bcrypt` untuk hashing password.

```
pip install bcrypt
from passlib.context import CryptContext

pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")

def get_password_hash(password):
    return pwd_context.hash(password)

def verify_password(plain_password, hashed_password):
    return pwd_context.verify(plain_password, hashed_password)
```

Mengonfigurasi OAuth2 dan Token JWT:

1. Buat token JWT untuk autentikasi.

```
pip install pyjwt
from datetime import datetime, timedelta
from jose import JWTError, jwt

SECRET_KEY = "your_secret_key"
ALGORITHM = "HS256"
```

```
ACCESS_TOKEN_EXPIRE_MINUTES = 30

def create_access_token(data: dict, expires_delta: timedelta = None):
    to_encode = data.copy()
    if expires_delta:
        expire = datetime.utcnow() + expires_delta
    else:
        expire = datetime.utcnow() + timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)
    to_encode.update({"exp": expire})
    encoded_jwt = jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)
    return encoded_jwt

def verify_access_token(token: str):
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        username: str = payload.get("sub")
        if username is None:
            raise credentials_exception
        return username
    except JWTError:
        raise credentials_exception
```

Membuat Endpoint Login:

1. Buat endpoint untuk login dan menghasilkan token.

```
from fastapi import Depends, FastAPI, HTTPException, status
from fastapi.security import OAuth2PasswordBearer, OAuth2PasswordRequestForm

app = FastAPI()

oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")

@app.post("/token", response_model=Token)
async def login_for_access_token(form_data: OAuth2PasswordRequestForm = Depends()):
    user = authenticate_user(fake_users_db, form_data.username, form_data.password)
    if not user:
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Incorrect username or password",
            headers={"WWW-Authenticate": "Bearer"},
        )
    access_token_expires = timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)
    access_token = create_access_token(
        data={"sub": user.username}, expires_delta=access_token_expires
    )
    return {"access_token": access_token, "token_type": "bearer"}

def authenticate_user(fake_db, username: str, password: str):
    user = get_user(fake_db, username)
```

```
        if not user:
            return False
        if not verify_password(password, user.hashed_password):
            return False
        return user

    def get_user(db, username: str):
        if username in db:
            user_dict = db[username]
            return UserInDB(**user_dict)
```

Membuat Dependensi untuk Pengguna Terkait:

1. Buat dependensi untuk mendapatkan pengguna yang terautentikasi.

```
    from fastapi import Depends, HTTPException, status
    from fastapi.security import OAuth2PasswordBearer

    oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")

    def get_current_user(token: str = Depends(oauth2_scheme)):
        credentials_exception = HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Could not validate credentials",
            headers={"WWW-Authenticate": "Bearer"},
        )
        return verify_access_token(token, credentials_exception)
```

Mengimplementasikan Otorisasi Berbasis Peran

Otorisasi berbasis peran memungkinkan Anda untuk mengatur akses ke endpoint tertentu berdasarkan peran pengguna.

Contoh:

```
from typing import List

class UserRole(BaseModel):
    username: str
    roles: List[str]

fake_roles_db = {
    "alice": ["admin"],
    "bob": ["user"]
}

def get_user_roles(username: str):
```

```
    if username in fake_roles_db:
        roles = fake_roles_db[username]
        return UserRole(username=username, roles=roles)
    return None

def get_current_active_user(current_user: User = Depends(get_current_user)):
    user_roles = get_user_roles(current_user.username)
    if user_roles is None or "active" not in user_roles.roles:
        raise HTTPException(status_code=400, detail="Inactive user")
    return current_user
```

Penjelasan:

- `get_user_roles` mengembalikan peran pengguna berdasarkan nama pengguna.
- `get_current_active_user` memastikan bahwa pengguna memiliki peran "active".

Menggunakan JSON Web Token (JWT) untuk Autentikasi

JWT adalah token yang digunakan untuk autentikasi. Token ini dienkripsi dan dikirimkan dalam header HTTP untuk mengautentikasi permintaan.

Contoh:

```
from fastapi import FastAPI, Depends, HTTPException
from fastapi.security import OAuth2PasswordBearer
from jose import JWTError, jwt

app = FastAPI()

oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")

SECRET_KEY = "your_secret_key"
ALGORITHM = "HS256"

def get_current_user(token: str = Depends(oauth2_scheme)):
    credentials_exception = HTTPException(
        status_code=401,
        detail="Could not validate credentials",
        headers={"WWW-Authenticate": "Bearer"},
    )
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        username: str = payload.get("sub")
        if username is None:
```

```
            raise credentials_exception
    except JWTError:
        raise credentials_exception
    return username

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

Penjelasan:

- `get_current_user` mendekode token JWT dan mengautentikasi pengguna.
- Endpoint `/users/me` mengembalikan informasi pengguna yang terautentikasi.

Dengan memahami dan mengimplementasikan autentikasi dan otorisasi, Anda dapat membangun aplikasi FastAPI yang aman dan dapat diandalkan. Ini memungkinkan Anda untuk mengontrol akses pengguna ke berbagai bagian aplikasi Anda, melindungi data sensitif, dan memastikan bahwa hanya pengguna yang sah yang dapat mengakses sumber daya tertentu.

Validasi dan Penanganan Error

Validasi input dan penanganan error adalah aspek penting dalam pengembangan API yang andal. FastAPI menyediakan alat yang kuat untuk validasi input menggunakan Pydantic dan mekanisme untuk menangani error secara efektif.

Validasi Input dengan Pydantic

FastAPI menggunakan Pydantic untuk validasi input secara otomatis. Pydantic memastikan bahwa data yang diterima oleh API sesuai dengan tipe data dan batasan yang telah ditentukan.

Contoh:

```
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel, Field

app = FastAPI()

class Item(BaseModel):
    name: str
    description: str = Field(None, max_length=300)
    price: float = Field(..., gt=0)
```

```
    tax: float = Field(None, ge=0)

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

Penjelasan:

- `Field` digunakan untuk menambahkan validasi tambahan pada atribut model, seperti `max_length`, `gt` (greater than), dan `ge` (greater or equal).
- FastAPI akan secara otomatis memvalidasi input berdasarkan definisi model `Item`.

Menangani Error dengan HTTPException

FastAPI menyediakan kelas `HTTPException` untuk menangani error HTTP. Anda dapat menggunakan `HTTPException` untuk mengembalikan kode status dan pesan error khusus.

Contoh:

```
@app.get("/items/{item_id}")
def read_item(item_id: int):
    if item_id < 0:
        raise HTTPException(status_code=400, detail="Invalid item ID")
    return {"item_id": item_id}
```

Penjelasan:

- `HTTPException` digunakan untuk mengembalikan error dengan kode status 400 (Bad Request) jika `item_id` kurang dari 0.

Membuat Custom Exception Handler

Anda dapat membuat custom exception handler untuk menangani jenis error tertentu secara khusus.

Contoh:

```
from fastapi import Request
```

```
class ItemNotFoundException(Exception):
    def __init__(self, item_id: int):
        self.item_id = item_id

@app.exception_handler(ItemNotFoundException)
def item_not_found_exception_handler(request: Request, exc: ItemNotFoundException):
    return JSONResponse(
        status_code=404,
        content={"message": f"Item with ID {exc.item_id} not found"},
    )

@app.get("/items/{item_id}")
def read_item(item_id: int):
    if item_id >= len(items):
        raise ItemNotFoundException(item_id=item_id)
    return items[item_id]
```

Penjelasan:

- `ItemNotFoundException` adalah custom exception yang dibuat untuk menangani kasus item tidak ditemukan.
- `item_not_found_exception_handler` adalah custom exception handler yang mengembalikan response dengan status 404 dan pesan khusus.
- Endpoint `/items/{item_id}` akan memicu `ItemNotFoundException` jika `item_id` tidak ditemukan dalam daftar `items`.

Menggunakan Middleware untuk Penanganan Error Global

Middleware adalah komponen yang digunakan untuk memproses permintaan dan respons sebelum atau setelah endpoint dijalankan. Anda dapat menggunakan middleware untuk penanganan error global.

Contoh:

```
from fastapi.middleware import Middleware
from starlette.middleware.base import BaseHTTPMiddleware
from starlette.responses import JSONResponse

class ErrorHandlerMiddleware(BaseHTTPMiddleware):
    async def dispatch(self, request, call_next):
        try:
            response = await call_next(request)
```

```
        except Exception as e:
            return JSONResponse(status_code=500, content={"message": "Internal Server
Error"})
        return response

middleware = [
    Middleware(ErrorHandlerMiddleware)
]

app = FastAPI(middleware=middleware)

@app.get("/")
def read_root():
    raise Exception("This is a test error")
```

Penjelasan:

- `ErrorHandlerMiddleware` adalah middleware yang menangani semua error dan mengembalikan respons dengan status 500 (Internal Server Error).
- Middleware ditambahkan ke aplikasi FastAPI melalui parameter `middleware` saat membuat instance `FastAPI`.

Dengan memahami dan mengimplementasikan validasi dan penanganan error, Anda dapat memastikan bahwa aplikasi FastAPI Anda lebih robust dan dapat menangani berbagai kondisi error dengan baik. Validasi input yang kuat membantu mencegah data yang tidak valid masuk ke sistem, sementara penanganan error yang efektif memastikan bahwa pengguna mendapatkan feedback yang jelas dan tepat ketika terjadi masalah. Selanjutnya, Anda dapat mengeksplorasi dokumentasi otomatis yang dihasilkan oleh FastAPI.

Dokumentasi Otomatis dengan Swagger dan Redoc

FastAPI secara otomatis menghasilkan dokumentasi API yang interaktif menggunakan Swagger UI dan Redoc. Ini membantu pengembang dan pengguna API untuk memahami, menguji, dan menggunakan endpoint yang tersedia.

Memahami Dokumentasi Otomatis FastAPI

FastAPI menghasilkan dua jenis dokumentasi otomatis:

- **Swagger UI:** Antarmuka pengguna yang interaktif untuk menguji endpoint API secara

langsung dari browser.

- **Redoc:** Dokumentasi API yang terstruktur dan mudah dibaca.

Setelah menjalankan aplikasi FastAPI, Anda dapat mengakses dokumentasi di URL berikut:

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

Menggunakan Swagger UI dan Redoc untuk Dokumentasi API

Swagger UI:
Swagger UI memberikan antarmuka interaktif yang memungkinkan Anda untuk menguji endpoint API. Anda dapat mengirimkan permintaan HTTP (GET, POST, PUT, DELETE) dan melihat respons secara langsung.

Redoc:
Redoc menyediakan dokumentasi API yang rapi dan terstruktur. Ini sangat berguna untuk memahami hierarki endpoint dan model data yang digunakan oleh API.

Menyesuaikan Dokumentasi API

Anda dapat menyesuaikan dokumentasi API dengan menambahkan metadata pada aplikasi FastAPI.

Contoh:

```
from fastapi import FastAPI

app = FastAPI(
    title="My API",
    description="This is a sample API using FastAPI",
    version="1.0.0",
    terms_of_service="http://example.com/terms/",
    contact={
        "name": "API Support",
        "url": "http://example.com/contact/",
        "email": "support@example.com",
    },
    license_info={
        "name": "Apache 2.0",
```

```
        "url": "https://www.apache.org/licenses/LICENSE-2.0.html",
    },
)

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

Penjelasan:

- `title`: Menentukan judul API.
- `description`: Menentukan deskripsi API.
- `version`: Menentukan versi API.
- `terms_of_service`: URL untuk halaman syarat dan ketentuan.
- `contact`: Informasi kontak untuk dukungan API.
- `license_info`: Informasi lisensi API.

Menambahkan Tag pada Endpoint

Anda dapat mengelompokkan endpoint menggunakan tag untuk membuat dokumentasi lebih terstruktur.

Contoh:

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/items/", tags=["items"])
def read_items():
    return [{"name": "Item 1"}, {"name": "Item 2"}]

@app.post("/items/", tags=["items"])
def create_item(item: dict):
    return item

@app.get("/users/", tags=["users"])
def read_users():
    return [{"username": "User 1"}, {"username": "User 2"}]
```

Penjelasan:

- `tags`: Menentukan tag untuk mengelompokkan endpoint di dokumentasi.

Menggunakan Deskripsi dan Response Model pada Endpoint

Anda dapat menambahkan deskripsi pada endpoint dan mendefinisikan model response untuk memperkaya dokumentasi.

Contoh:

```
from fastapi import FastAPI
from pydantic import BaseModel
from typing import List

app = FastAPI()

class Item(BaseModel):
    name: str
    description: str = None
    price: float
    tax: float = None

@app.post("/items/", response_model=Item, summary="Create an item", description="Create
a new item with all the information provided")
def create_item(item: Item):
    return item

@app.get("/items/", response_model=List[Item], summary="Read items",
description="Retrieve all items in the database")
def read_items():
    return [
        {"name": "Item 1", "description": "This is item 1", "price": 10.0, "tax": 1.0},
        {"name": "Item 2", "description": "This is item 2", "price": 20.0, "tax": 2.0}
    ]
```

Penjelasan:

- `response_model`: Menentukan model response yang akan ditampilkan di dokumentasi.
- `summary`: Menambahkan ringkasan singkat untuk endpoint.
- `description`: Menambahkan deskripsi detail untuk endpoint.

Menyembunyikan Endpoint dari Dokumentasi

Anda dapat menyembunyikan endpoint tertentu dari dokumentasi jika tidak ingin ditampilkan.

Contoh:

```
@app.get("/hidden-endpoint", include_in_schema=False)
def hidden_endpoint():
    return {"message": "This endpoint is hidden from the documentation"}
```

Penjelasan:

- `include_in_schema`: Menentukan apakah endpoint akan dimasukkan dalam dokumentasi (default: `True`).

Dengan memahami dan menggunakan dokumentasi otomatis yang dihasilkan oleh FastAPI, Anda dapat membuat API Anda lebih mudah digunakan dan dipahami oleh pengembang lain. Dokumentasi yang baik membantu dalam pengembangan, pemeliharaan, dan pengujian API secara efektif. Selanjutnya, Anda dapat mengeksplorasi cara menulis dan menjalankan pengujian untuk API FastAPI Anda.

Testing API dengan FastAPI

Pengujian adalah langkah penting dalam pengembangan aplikasi untuk memastikan bahwa fungsionalitas yang diimplementasikan bekerja sesuai yang diharapkan. FastAPI mendukung berbagai jenis pengujian, termasuk unit testing dan integration testing.

Menulis Unit Test dan Integration Test untuk Endpoint

Unit test memeriksa fungsionalitas individu dari unit kecil kode, sedangkan integration test memeriksa bagaimana beberapa unit bekerja bersama-sama.

Menyiapkan Lingkungan Pengujian:

Instalasi Pytest:

1. Pytest adalah framework pengujian yang kuat dan mudah digunakan untuk Python.

```
pip install pytest
```

Struktur Direktori Pengujian:

1. Buat direktori `tests` untuk menempatkan file pengujian Anda.

```
mkdir tests
touch tests/test_main.py
```

Contoh Pengujian dengan Pytest:

1. Membuat file `tests/test_main.py`:

```
from fastapi.testclient import TestClient
from main import app

client = TestClient(app)

def test_read_root():
    response = client.get("/")
    assert response.status_code == 200
    assert response.json() == {"message": "Hello, World!"}

def test_create_item():
    response = client.post(
        "/items/",
        json={"name": "Item 1", "description": "This is item 1", "price": 10.0, "tax":
1.0}
    )
    assert response.status_code == 200
    assert response.json() == {
        "name": "Item 1",
        "description": "This is item 1",
        "price": 10.0,
        "tax": 1.0
    }
```

Penjelasan:

- `TestClient` dari FastAPI digunakan untuk mengirimkan permintaan HTTP ke aplikasi FastAPI.
- `test_read_root` menguji endpoint root (`/`) dan memeriksa status kode dan respons JSON.
- `test_create_item` menguji endpoint untuk membuat item (`/items/`) dan memeriksa status kode serta respons JSON.

Menggunakan pytest untuk Pengujian

Untuk menjalankan pengujian, gunakan perintah berikut di terminal:

```
pytest
```

Pytest akan secara otomatis menemukan dan menjalankan semua fungsi yang dimulai dengan `test_`.

Mocking dan Pengujian Asinkron

Mocking digunakan untuk menggantikan bagian dari sistem dengan tiruan yang dikendalikan selama pengujian. FastAPI mendukung pengujian asinkron yang memungkinkan Anda untuk menguji fungsi asinkron secara efektif.

Contoh Mocking:

```
from unittest.mock import patch

def test_create_item_with_mock():
    with patch("main.get_db") as mock_get_db:
        mock_get_db.return_value = iter([mock_db_session])
        response = client.post(
            "/items/",
            json={"name": "Item 1", "description": "This is item 1", "price": 10.0,
"tax": 1.0}
        )
        assert response.status_code == 200
        assert response.json() == {
            "name": "Item 1",
            "description": "This is item 1",
            "price": 10.0,
            "tax": 1.0
        }
```

Penjelasan:

- `patch` digunakan untuk menggantikan fungsi `get_db` dengan mock yang mengembalikan sesi database tiruan (`mock_db_session`).

Contoh Pengujian Asinkron:

```
import pytest
from httpx import AsyncClient

@pytest.mark.asyncio
async def test_read_item():
    async with AsyncClient(app=app, base_url="http://test") as ac:
        response = await ac.get("/items/1")
    assert response.status_code == 200
    assert response.json() == {"item_id": 1}
```

Penjelasan:

- `pytest.mark.asyncio` digunakan untuk menandai fungsi pengujian sebagai asinkron.
- `AsyncClient` dari `httpx` digunakan untuk mengirimkan permintaan HTTP asinkron ke aplikasi FastAPI.

Menangani Database dalam Pengujian

Untuk menguji aplikasi yang terhubung dengan database, Anda perlu menyiapkan lingkungan pengujian yang terisolasi.

Contoh:

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
from fastapi.testclient import TestClient
from main import app, get_db
from database import Base

SQLALCHEMY_DATABASE_URL = "sqlite:///./test.db"

engine = create_engine(SQLALCHEMY_DATABASE_URL, connect_args={"check_same_thread":
False})
TestingSessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

Base.metadata.create_all(bind=engine)

def override_get_db():
    try:
        db = TestingSessionLocal()
        yield db
    finally:
        db.close()
```

```
app.dependency_overrides[get_db] = override_get_db

client = TestClient(app)

def test_create_item():
    response = client.post(
        "/items/",
        json={"name": "Item 1", "description": "This is item 1", "price": 10.0, "tax":
1.0}
    )
    assert response.status_code == 200
    assert response.json() == {
        "name": "Item 1",
        "description": "This is item 1",
        "price": 10.0,
        "tax": 1.0
    }
```

Penjelasan:

- `create_engine` dan `sessionmaker` digunakan untuk membuat engine dan sesi database untuk pengujian.
- `override_get_db` menggantikan fungsi `get_db` dengan versi yang menggunakan database pengujian.
- `TestingSessionLocal` digunakan untuk membuat sesi database khusus pengujian.

Dengan memahami dan menerapkan pengujian menggunakan pytest, mocking, dan pengujian asinkron, Anda dapat memastikan bahwa aplikasi FastAPI Anda berfungsi dengan benar dan menangani berbagai kondisi dengan baik. Pengujian yang baik membantu dalam menjaga kualitas kode dan mencegah regresi saat menambahkan fitur baru.

Deployment Aplikasi FastAPI

Deployment adalah proses menempatkan aplikasi yang telah Anda buat ke server agar dapat diakses oleh pengguna. FastAPI mendukung berbagai metode deployment yang memungkinkan Anda untuk menjalankan aplikasi di berbagai lingkungan.

Men-deploy Aplikasi FastAPI dengan Uvicorn

Uvicorn adalah server ASGI yang cepat dan ringan untuk menjalankan aplikasi FastAPI. Uvicorn dapat digunakan untuk menjalankan aplikasi secara lokal atau di server produksi.

Menjalankan Aplikasi Secara Lokal:

1. **Instal Uvicorn:**

   ```
   pip install uvicorn
   ```

2. **Menjalankan Aplikasi:**

   ```
   uvicorn main:app --host 0.0.0.0 --port 8000 --reload
   ```

3. Penjelasan:
   - `main:app` menunjukkan file `main.py` dan objek `app` di dalamnya.
   - `--host 0.0.0.0` memungkinkan akses dari semua alamat IP.
   - `--port 8000` menentukan port yang akan digunakan.
   - `--reload` memungkinkan server untuk otomatis memuat ulang ketika ada perubahan pada kode (hanya untuk pengembangan).