FastAPI

FastAPI ek modern, high-performance Python framework hai jo RESTful APIs banane ke liye use hota hai. Ye asynchronous programming aur Python type hints par based hai, jo isse fast, scalable, aur developer-friendly banata hai. Iska automatic documentation feature (Swagger UI) aur Pydantic integration isse unique banata hai.

Key Features

- **High Performance**: Starlette ke async capabilities ki wajah se FastAPI Node.js aur Go ke comparable hai.
- **Automatic Documentation**: `/docs` endpoint par Swagger UI milta hai jo APIs ko test aur document karta hai.
- **Type Safety**: Python type hints ke saath data validation aur error handling.
- **Easy to Learn**: Python developers ke liye intuitive syntax.
- **Scalability**: Async support ke saath high-concurrency apps ke liye ideal.

Basic Example

```
from fastapi import FastAPI

app = FastAPI()

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

- Yahan `/hello` ek GET endpoint hai jo JSON response `{"message": "Hello, World!"}` return karta hai.
- Server run karne ke liye: `uvicorn main:app --reload`
- Browser mein `http://localhost:8000/docs` kholkar endpoint test kar sakte ho.

Why Use FastAPI?

- FastAPI APIs ko quickly develop karne ke liye perfect hai, khaas kar microservices ya data-heavy backends ke liye.
- Swagger UI development aur testing ko bohot simple banata hai.
- Pydantic ke saath data validation aur serialization built-in hota hai.

Pydantic

Pydantic ek Python library hai jo data validation aur serialization ke liye use hoti hai. FastAPI mein Pydantic models request body aur response data ko validate aur structure karne ke liye use hote hain. Ye type hints ke saath kaam karta hai aur runtime validation deta hai.

Key Features

- **Type Safety**: Data types (str, int, float, etc.) ko strictly enforce karta hai.
- **Automatic Validation**: Invalid data ke liye detailed error messages.
- **Serialization**: Models ko JSON ya dict mein convert karna easy.
- **Nested Models**: Complex data structures ke liye models ke andar models.
- **Built-in Validators**: Email, URL, aur custom validation ke liye support.

Example

```
from pydantic import BaseModel
from fastapi import FastAPI

app = FastAPI()

class User(BaseModel):
    username: str
    email: str
    age: int | None = None  # Optional field

@app.post("/users/")
async def create_user(user: User):
    return user
```

- `User` model define karta hai ki incoming JSON mein `username` (string), `email` (string), aur `age` (optional integer) hona chahiye.

Valid JSON input:

```
{
    "username": "john_doe",
    "email": "john@example.com",
    "age": 25
}
```

-
- Invalid JSON (jaise `"age": "twenty-five"`) 422 error dega with error details.

Nested Models

```
class Address(BaseModel):
    city: str
    country: str

class User(BaseModel):
    username: str
    address: Address

@app.post("/users/")
async def create_user(user: User):
    return user

JSON input:
{
    "username": "john_doe",
    "address": {
        "city": "New York",
        "country": "USA"
    }
}
```

-

Why Use Pydantic?

- FastAPI ke saath tightly integrated, jo validation aur documentation ko seamless banata hai.
- Complex data structures ke liye nested models aur custom validators.
- Error handling aur type safety development ko fast aur reliable banati hai.

API Parameters

API parameters woh data hote hain jo client API endpoint ko bhejta hai ya jo API expect karta hai. FastAPI mein parameters ko type hints ke saath define kiya jata hai, jo automatic validation aur documentation deta hai.

Types of Parameters

- **Path Parameters**: URL ke parts (e.g., `/items/123` mein `123`).
- **Query Parameters**: URL ke query string mein (e.g., `/items/?category=electronics`).
- **Request Body**: JSON data jo POST, PUT requests ke saath bheja jata hai.
- **Header Parameters**: HTTP headers mein (e.g., Authorization token).

- **Cookie Parameters**: Browser cookies se data.

Examples

Path Parameters

```
from fastapi import FastAPI

app = FastAPI()

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

- `item_id` ek path parameter hai jo integer hona chahiye.
- URL: `http://localhost:8000/items/123`
- Invalid input (jaise `/items/abc`) 422 error dega.

Query Parameters

```
@app.get("/items/")
async def get_items(category: str = None, limit: int = 10):
    return {"category": category, "limit": limit}
```

- `category` aur `limit` optional query parameters hain.
- URL: `http://localhost:8000/items/?category=electronics&limit=5`

Request Body

```
from pydantic import BaseModel

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

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

JSON input:
{
    "name": "Laptop",
    "price": 999.99,
    "description": "A high-end laptop"
}
```

-

Header Parameters

```
from fastapi import Header

@app.get("/secure/")
async def read_secure(authorization: str = Header()):
    return {"authorization": authorization}
```

- `authorization` header se value li jati hai.

Why Use API Parameters?

- Type hints ke saath parameters define karna validation aur error handling ko easy banata hai.
- Swagger UI mein parameters automatically document hote hain, jo testing ke liye helpful hai.
- Flexible syntax se path, query, body, aur headers ko handle karna simple hai.

Dependency Injection

Dependency Injection (DI) ek design pattern hai jisme ek function ya endpoint apne dependencies (jaise database connection, authentication logic, ya settings) ko khud create nahi karta, balki FastAPI se expect karta hai ki wo provide karega. Ye reusable, modular, aur testable code likhne ke liye use hota hai.

How It Works

- Dependencies ko functions ya classes ke roop mein define kiya jata hai.
- `Depends()` ke through dependencies endpoints mein inject hoti hain.
- FastAPI dependencies ko automatically resolve karta hai aur unke results endpoint ke arguments mein pass karta hai.
- Agar dependency mein error aata hai, to endpoint execute hone se pehle error response milta hai.

Key Features

- **Reusability**: Ek dependency ko multiple endpoints use kar sakte hain.
- **Hierarchical Dependencies**: Ek dependency doosre dependency par depend kar sakti hai.

- **Error Handling**: Dependencies mein errors raise karne se client ko immediate response milta hai.
- **Testing**: Dependencies ko mock karna easy hai.
- **Automatic Documentation**: Dependencies Swagger UI mein document hote hain.

Real-Life Examples

Example 1: User Authentication (E-Commerce App)

Scenario: Ek e-commerce app mein user ko authenticated hone par order history dekhna hai. Authentication logic ko har endpoint mein repeat karne ke bajaye, DI ka use karte hain.

```
from fastapi import FastAPI, Depends, HTTPException, Header

app = FastAPI()

# Dependency for authentication
async def get_current_user(authorization: str = Header()):
    if authorization != "Bearer fake-token":
        raise HTTPException(status_code=401, detail="Invalid token")
    return {"user_id": 1, "username": "john_doe"}

# Endpoint using dependency
@app.get("/orders/")
async def get_orders(current_user: dict = Depends(get_current_user)):
    return {"user": current_user["username"], "orders": ["order1", "order2"]}

# Reusing the same dependency
@app.get("/profile/")
async def get_profile(current_user: dict = Depends(get_current_user)):
    return {"user": current_user["username"], "email": "john@example.com"}
```

- `get_current_user` token validate karta hai aur user details return karta hai.
- `Depends(get_current_user)` se authentication logic dono endpoints mein reuse hota hai.
- Invalid token par 401 error milta hai.

Benefit: Authentication logic ek jagah define hota hai, aur agar logic change karna ho (jaise JWT library add karna), to bas dependency update karo.

Example 2: Database Connection (Blog App)

Scenario: Ek blog app mein posts ko database se fetch karna hai. Database connection ko har endpoint mein manually manage karne ke bajaye, DI ka use karte hain.

```
from fastapi import FastAPI, Depends
```

```
from databases import Database

app = FastAPI()
database = Database("sqlite:///blog.db")

# Dependency for database connection
async def get_db():
    await database.connect()
    try:
        yield database
    finally:
        await database.disconnect()

# Endpoint using dependency
@app.get("/posts/")
async def get_posts(db: Database = Depends(get_db)):
    posts = await db.fetch_all("SELECT * FROM posts")
    return posts

# Reusing the same dependency
@app.get("/posts/{post_id}")
async def get_post(post_id: int, db: Database = Depends(get_db)):
    post = await db.fetch_one("SELECT * FROM posts WHERE id = :id", {"id": post_id})
    return post
```

- `get_db` database connection provide karta hai aur `yield` se cleanup ensure karta hai.
- Dono endpoints (`get_posts`, `get_post`) same dependency use karte hain.

Benefit: Database connection ka lifecycle automatic manage hota hai, aur resource leaks prevent hote hain.

Example 3: Configuration Settings (SaaS App)

Scenario: Ek SaaS app mein API keys ya settings (jaise max file upload size) ko endpoints mein use karna hai. Settings ko dependency mein load karte hain.

```
from fastapi import FastAPI, Depends
import os
from pydantic import BaseModel

app = FastAPI()

# Pydantic model for settings
class AppSettings(BaseModel):
    max_file_size: int
    api_key: str
```

```
# Dependency for settings
def get_settings():
    return AppSettings(
        max_file_size=int(os.getenv("MAX_FILE_SIZE", 1048576)),  # 1MB default
        api_key=os.getenv("API_KEY", "default-key")
    )

# Endpoint using dependency
@app.get("/upload/")
async def upload_file(settings: AppSettings = Depends(get_settings)):
    return {"max_file_size": settings.max_file_size}

# Reusing the same dependency
@app.get("/api-info/")
async def api_info(settings: AppSettings = Depends(get_settings)):
    return {"api_key": settings.api_key}
```

- `get_settings` environment variables se settings load karta hai.
- Pydantic model settings ke structure aur types ko validate karta hai.

Benefit: Settings centralized hote hain, aur agar source change ho (jaise env variables ki jagah config file), to bas dependency update karo.

Hierarchical Dependencies

Dependencies ek doosre par depend kar sakti hain, jaise:

```
async def get_db():
    return Database("sqlite:///example.db")

async def get_repository(db: Database = Depends(get_db)):
    return Repository(db)

@app.get("/items/")
async def get_items(repo: Repository = Depends(get_repository)):
    return repo.get_all()
```

- `get_repository` `get_db` par depend karta hai, aur FastAPI hierarchy ko automatically resolve karta hai.

Why Use Dependency Injection?

- **Modular Code**: Common logic ko dependencies mein alag rakha jata hai.
- **Reusability**: Ek dependency multiple endpoints mein use hoti hai.

- **Easy Testing**: Dependencies ko mock karna simple hai.
- **Clean Error Handling**: Errors dependency mein handle hote hain.
- **Scalability**: Large projects ke liye DI code ko organized rakhta hai.

Practical Project Idea

Ek todo list API banao jisme:

- Users authenticated hone chahiye (token ke saath).
- Todos database mein store honge (SQLite).
- Settings (jaise max todos per user) environment variables se load honge.

```
from fastapi import FastAPI, Depends, HTTPException, Header
from databases import Database
from pydantic import BaseModel
import os

app = FastAPI()

# Pydantic Models
class Todo(BaseModel):
    title: str
    completed: bool = False

class AppSettings(BaseModel):
    max_todos: int
    api_key: str

# Dependencies
async def get_current_user(authorization: str = Header()):
    if authorization != "Bearer fake-token":
        raise HTTPException(status_code=401, detail="Invalid token")
    return {"user_id": 1, "username": "john_doe"}

async def get_db():
    db = Database("sqlite:///todos.db")
    await db.connect()
    try:
        yield db
    finally:
        await db.disconnect()

def get_settings():
    return AppSettings(
        max_todos=int(os.getenv("MAX_TODOS", 100)),
        api_key=os.getenv("API_KEY", "default-key")
    )
```

```
# Endpoints
@app.post("/todos/", response_model=Todo)
async def create_todo(
    todo: Todo,
    user: dict = Depends(get_current_user),
    db: Database = Depends(get_db),
    settings: AppSettings = Depends(get_settings)
):
    todos_count = await db.fetch_val(
        "SELECT COUNT(*) FROM todos WHERE user_id = :user_id",
        {"user_id": user["user_id"]}
    )
    if todos_count >= settings.max_todos:
        raise HTTPException(status_code=400, detail="Max todos limit reached")
    await db.execute(
        "INSERT INTO todos (title, completed, user_id) VALUES (:title, :completed, :user_id)",
        {"title": todo.title, "completed": todo.completed, "user_id": user["user_id"]}
    )
    return todo

@app.get("/todos/")
async def get_todos(
    user: dict = Depends(get_current_user),
    db: Database = Depends(get_db)
):
    todos = await db.fetch_all(
        "SELECT * FROM todos WHERE user_id = :user_id",
        {"user_id": user["user_id"]}
    )
    return todos
```

- Run: `uvicorn main:app --reload`
- Test: `/docs` par endpoints try karo with `Authorization: Bearer fake-token`.