Introduction to FastAPI

What is FastAPI?

- A modern, fast (high-performance), web framework for building APIs with Python 3.6+ based on standard Python type hints.
- Built on top of **Starlette** (web toolkit) and **Pydantic** (data validation and settings management).

Key Features:

- Automatic API documentation (Swagger UI & ReDoc).
- High performance (comparable to Node.js and Go).
- Asynchronous support with `async` and `await`.
- Type safety with Python type hints.

1. Creating a Route

Routes define the paths or endpoints for the API. A route is associated with a URL path and a request method (like `GET`, `POST`, `PUT`, or `DELETE`).

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
def read_root():
    return {"message": "Welcome to FastAPI!"}

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

---

2. Path Parameters

Path parameters are dynamic segments of the route URL, captured and passed to the handler function.

- **Example**: `/items/{item_id}`
- The `item_id` parameter is extracted from the URL.

```
@app.get("/users/{user_id}")
def get_user(user_id: int):
    return {"user_id": user_id}
```

Path parameters can have type hints, ensuring FastAPI validates the data automatically.

---

3. Query Parameters

Query parameters are additional key-value pairs in the URL, usually after the `?`.

```
@app.get("/search")
def search_items(q: str, limit: int = 10):
    return {"query": q, "limit": limit}
```

Here:

- `q` is a required query parameter.
- `limit` is an optional query parameter with a default value of `10`.

---

4. Request Methods

FastAPI supports HTTP methods like:

- `GET` (read data)
- `POST` (create data)
- `PUT` (update data)
- `DELETE` (delete data)
- `PATCH` (partial update)

```
@app.post("/items")
def create_item(item: dict):
    return {"item": item}

@app.put("/items/{item_id}")
def update_item(item_id: int, item: dict):
    return {"item_id": item_id, "updated_item": item}
```

---

5. Request Bodies

FastAPI allows handling JSON data in request bodies, using Pydantic models for validation and type-checking.

```
from pydantic import BaseModel

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

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

---

6. Path and Query Parameter Validation

- You can validate path and query parameters with constraints.

```
from typing import Optional

@app.get("/items/{item_id}")
def read_item(item_id: int, limit: Optional[int] = None):
    if limit and limit < 1:
        return {"error": "Limit must be greater than 0"}
    return {"item_id": item_id, "limit": limit}
```

---

7. Grouping Routes with Routers

FastAPI provides `APIRouter` to modularize routes.

```
from fastapi import APIRouter

router = APIRouter()

@router.get("/users")
def get_users():
    return [{"user_id": 1, "name": "John"}]

@router.post("/users")
def create_user(user: dict):
    return {"message": "User created", "user": user}

app.include_router(router, prefix="/api")
```

- The `prefix` argument adds a common prefix to all routes in the router.

---

8. Route Dependencies

Dependencies can be added to routes to perform shared logic.

```
from fastapi import Depends

def common_dependency():
    return {"key": "value"}

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

---

9. Middleware and Route Preprocessing

Middleware can intercept requests/responses before they reach the route.

```
from fastapi.middleware.cors import CORSMiddleware

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

---

10. Handling Async Endpoints

```
import asyncio

@app.get("/async")
async def read_async():
    await asyncio.sleep(1)
    return {"message": "This is asynchronous!"}
```

---

11. Route Tags

Routes can be tagged for grouping and documentation purposes.

```
@app.get("/users", tags=["users"])
def get_users():
    return [{"user_id": 1, "name": "Alice"}]
```

---

12. Route Documentation and Metadata

You can add descriptions and summaries to routes for better API docs.

```
@app.get("/items/{item_id}", summary="Get an item", description="Retrieve an item by its ID")
def get_item(item_id: int):
    return {"item_id": item_id}
```

---

13. Testing with FastAPI

FastAPI is fully compatible with `pytest`.

```
from fastapi.testclient import TestClient

client = TestClient(app)

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

---

14. Advanced Topics

WebSockets:

```
from fastapi import WebSocket

@app.websocket("/ws")
async def websocket_endpoint(websocket: WebSocket):
    await websocket.accept()
    data = await websocket.receive_text()
    await websocket.send_text(f"Message text: {data}")
```

Background Tasks:

```
from fastapi import BackgroundTasks

def write_log(message: str):
    with open("log.txt", "a") as file:
        file.write(message + "\n")

@app.post("/log/")
def log_message(message: str, background_tasks: BackgroundTasks):
    background_tasks.add_task(write_log, message)
    return {"message": "Message logged"}
```