# FastAPI Python Concepts Study Notes

---

## Table of Contents


1. Introduction to FastAPI
    - What is FastAPI?
    - Key Features
2. Installation
3. Basic Concepts
    - Request and Response
    - Path Parameters
    - Query Parameters
    - Request Body
4. Data Validation
    - Pydantic Models
    - Validation in FastAPI
5. Dependency Injection
    - What is Dependency Injection?
    - Creating Dependencies
6. Middleware
    - Definition of Middleware
    - How to Create Middleware
7. Routes and Routers
    - Creating Routes
    - Using Routers for Modular Design
8. Asynchronous Programming
    - Understanding Asynchronous in Python
    - Async Functions in FastAPI

9. Error Handling

- Exception Handling
- Custom Error Responses

10. Summary

---

# 1. Introduction to FastAPI

### What is FastAPI?

- FastAPI is a modern web framework for building APIs with Python 3.6+ based on standard Python type hints.
- It is designed to create high-performance APIs.

### Key Features

- **Fast**: High performance, comparable to NodeJS and Go.
- **Easy**: Designed for simplicity and ease of use.
- **Pythonic**: Leverages Python type hints for easy validation.
- **Automatic documentation**: Interactive API documentation (Swagger UI).
- **Asynchronous Support**: Built-in support for async and await.

---

# 2. Installation

To install FastAPI with the supporting ASGI server, use pip:

pip install fastapi[all]

- `fastapi`: The core library.
- `uvicorn`: ASGI server used to run FastAPI apps.

---

# 3. Basic Concepts

### Request and Response

- **Request**: The data sent by the client (e.g., a web browser).
- **Response**: The data returned by the server.

### Path Parameters

- Dynamic parameters in the route path.
- Example of defining a path parameter in FastAPI:

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

### Query Parameters

- Key-value pairs sent in the URL to further specify the request.

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

### Request Body

- Used to send complex data structures (e.g., JSON) in POST requests.

```
from pydantic import BaseModel

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

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

return item

---

## 4. Data Validation

### Pydantic Models

- FastAPI uses Pydantic for data validation and settings management.
- Helps ensure that data is of the expected format.

### Validation in FastAPI

- FastAPI validates the data automatically based on specified models.

Example:

from pydantic import BaseModel

class User(BaseModel):

name: str

age: int

email: str

@app.post("/users/")

async def create_user(user: User):

return user

---

## 5. Dependency Injection

### What is Dependency Injection?

- A way to manage dependencies and reduce tight coupling in your code.

### Creating Dependencies

- Use `Depends` to declare dependencies in FastAPI.

Example:

```
from fastapi import Depends

def get_query_param(q: str = None):
return q

@app.get("/items/")
async def read_items(q: str = Depends(get_query_param)):
return {"query": q}
```

---

## 6. Middleware

### Definition of Middleware

- Middleware is a function that runs before or after every request.

### How to Create Middleware

- Use `starlette.middleware`.

Example:

```
from starlette.middleware.cors import CORSMiddleware

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

---

## 7. Routes and Routers

### Creating Routes

- Define endpoints using decorators.

Example:

```
@app.get("/health/")
async def health_check():
return {"status": "healthy"}
```

### Using Routers for Modular Design

- Use `APIRouter` for organizing routes.

Example:

```
from fastapi import APIRouter

router = APIRouter()

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

app.include_router(router)
```

---

## 8. Asynchronous Programming

### Understanding Asynchronous in Python

- Allows writing concurrent code using the `async` and `await` syntax.

### Async Functions in FastAPI

- Define async functions to handle requests for better performance.

Example:

```
@app.get("/async-items/")
async def async_get_items():
await asyncio.sleep(1) # Simulates asynchronous processing
return ["item1", "item2"]
```

---

## 9. Error Handling

### Exception Handling

- FastAPI allows you to manage exceptions and provide custom responses.

Example:

```
from fastapi import HTTPException

@app.get("/items/{item_id}")
async def get_item(item_id: int):
if item_id > 10: # Example condition
raise HTTPException(status_code=404, detail="Item not found")
return {"item_id": item_id}
```

### Custom Error Responses

- You can customize error responses globally.

Example:

```
from fastapi.responses import JSONResponse

@app.exception_handler(HTTPException)
```

```
async def http_exception_handler(request, exc):
return JSONResponse(
status_code=exc.status_code,
content={"message": exc.detail},
)
```

---

## 10. Summary

- FastAPI is an efficient framework for building RESTful APIs.
- It integrates data validation using Pydantic and handles asynchronous requests gracefully.
- Dependency injection promotes clean code architecture.
- Middleware can be used for cross-cutting concerns, and error handling can be customized.
- FastAPI auto-generates interactive API documentation, facilitating ease of use.

FastAPI is a powerful tool that allows developers to create high-performance APIs with minimal effort while maintaining code clarity and simplicity.

---

Use these notes to understand the concepts and functionalities of FastAPI, and revise them for better retention of knowledge!