AI VIETNAM
All-in-One Course
(TA Session)

Web Deployment Using FastAPI and Gradio

Extra Class: MLOps

Minh-Duc Bui – TA

Year 2024

Outline

- Introduction
- API
- FastAPI
- Integration
- Question

Introduction

Introduction

❖ Getting Started

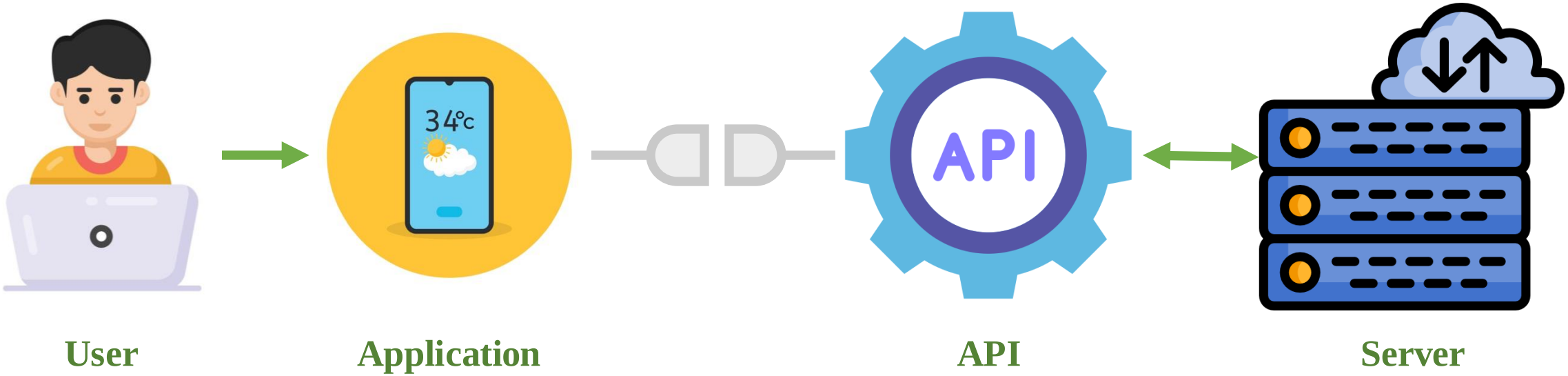
How to communicate?

API

API

❖ Getting Started

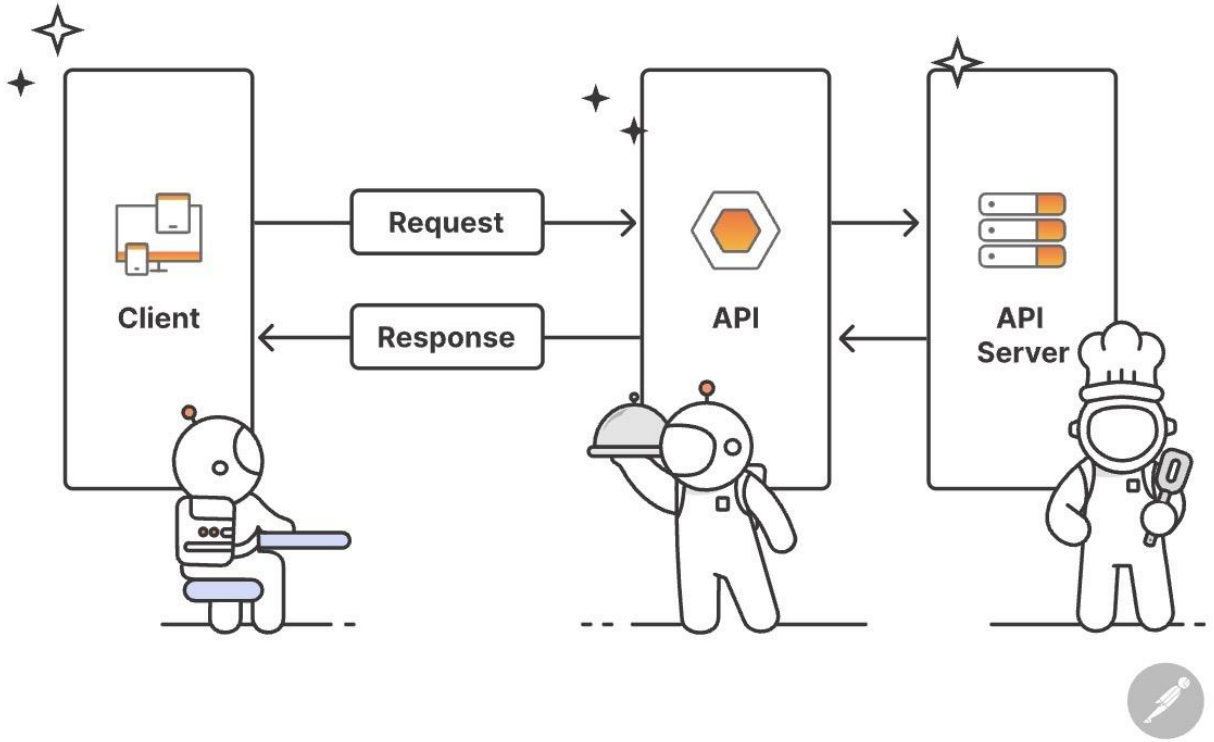
- **API (Application Programming Interface)** is a facilitator that enables apps, databases, software and IoT devices to communicate with each other.



Introduction

❖ Getting Started

- **API = Waiter:** Connects users and servers, like a waiter connects customers and the kitchen.
- **How it works:** Users send requests (orders), and the API relays them to the server, which processes and sends back responses.
- **Example:** Searching online is like ordering food. The API facilitates the communication between the user's request and the server's response.
- **Key Components:**
  - **User:** Customer
  - **API:** Waiter
  - **Server:** Kitchen



API

❖ Conclusion

	REST	GraphQL	SOAP	RPC
Structure	Follows six architectural constraints	Schema and type	Message structure	Local procedural calls
Format	Json, XLM, HTML, plain text	Json	XML	Json, XML, Flatbuffers, etc
Advantages	Flexible terms of data format and structure	Sovles over-fetching and under-fetching	Highly secure and extensible	Lightweight payloads make it high performing
Use cases	Resources based apps	Mobile APIs	Payment gateways	Command-focused systems

A Step Back

❖ Non-Concurrency

```python
import time


def fetch_data():
    print("Hi")
    time.sleep(3)
    print("Bye")


def main():
    start_time = time.perf_counter()  # Start the timer

    # Execute tasks sequentially
    for _ in range(2):
        fetch_data()

    end_time = time.perf_counter()  # End the timer
    print(f"Total time taken: {end_time - start_time:.2f} seconds")


# Run the main function
if __name__ == "__main__":
    main()
```

```
Hi
Bye
Hi
Bye
Total time taken: 6.02 seconds
```

The program runs sequentially and takes 6 seconds to complete.

A Step Back

❖ Non-Concurrency

```python
import threading
import time


def fetch_data():
    print(f"Start fetching data on thread: {threading.get_ident()}")
    time.sleep(3)
    print(f"Finished fetching data on thread: {threading.get_ident()}")


def main():
    start_time = time.perf_counter()  # Start the timer

    # Execute tasks sequentially
    for _ in range(2):
        fetch_data()

    end_time = time.perf_counter()  # End the timer
    print(f"Total time taken: {end_time - start_time:.2f} seconds")


# Run the main function
if __name__ == "__main__":
    main()
```

```
Start fetching data on thread: 8432333376
Finished fetching data on thread: 8432333376
Start fetching data on thread: 8432333376
Finished fetching data on thread: 8432333376
Total time taken: 6.00 seconds
```

The program runs sequentially and takes 6 seconds to complete.

A Step Back

❖ Async/await (Concurrency)

```python
import asyncio
import time


async def fetch_data():
    print("Hi")
    await asyncio.sleep(3)
    print("Bye")


async def main():
    start_time = time.perf_counter()  # Start the timer

    # Create a list of tasks
    tasks = [fetch_data() for _ in range(2)]

    # Run tasks concurrently
    await asyncio.gather(*tasks)

    end_time = time.perf_counter()  # End the timer
    print(f"Total time taken: {end_time - start_time:.2f} seconds")


# Run the main function
asyncio.run(main())
```

```
Hi
Hi
Bye
Bye
Total time taken: 3.00 seconds
```

The program runs concurrently and takes 3 seconds to complete.

A Step Back

❖ Async/await (Concurrency)

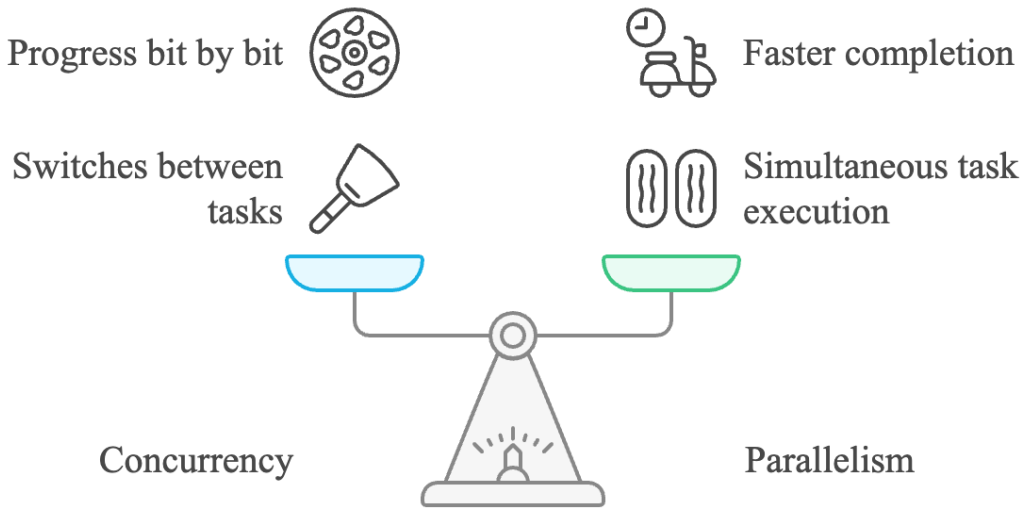
```python
import asyncio
import threading
import time


async def fetch_data():
    print(f"Start fetching data on thread: {threading.get_ident()}")
    await asyncio.sleep(3)
    print(f"Finished fetching data on thread: {threading.get_ident()}")


async def main():
    start_time = time.perf_counter()  # Start the timer

    # Create a list of tasks
    tasks = [fetch_data() for _ in range(2)]

    # Run tasks concurrently
    await asyncio.gather(*tasks)

    end_time = time.perf_counter()  # End the timer
    print(f"Total time taken: {end_time - start_time:.2f} seconds")


# Run the main function
asyncio.run(main())
```

```
Start fetching data on thread: 8432333376
Start fetching data on thread: 8432333376
Finished fetching data on thread: 8432333376
Finished fetching data on thread: 8432333376
Total time taken: 3.00 seconds
```

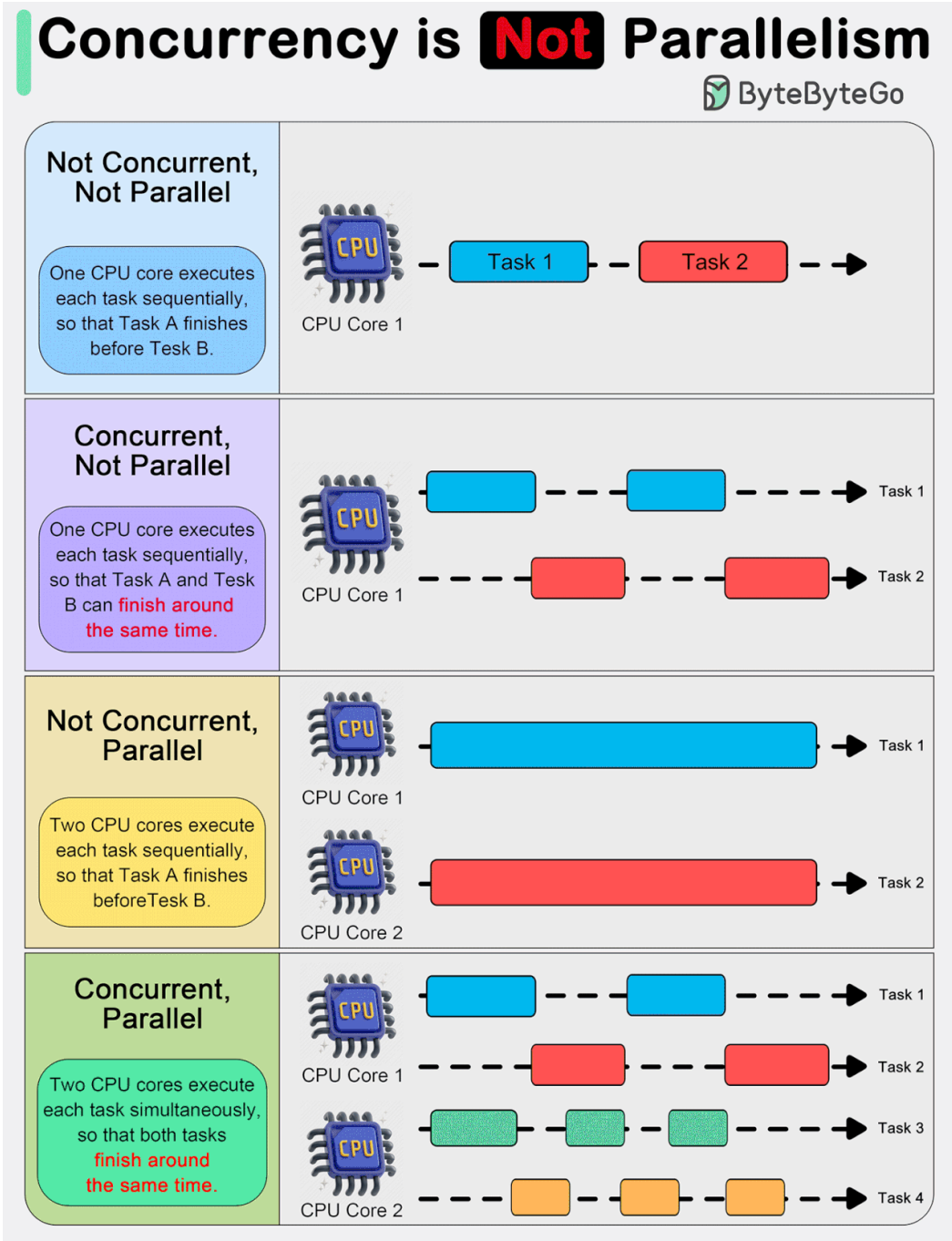
The program runs concurrently and takes 3 seconds to complete.

A Step Back

❖ Async/await (Concurrency)



Comparing concurrency and parallelism in task handling.



FastAPI

FastAPI

❖ Getting Started

<

<

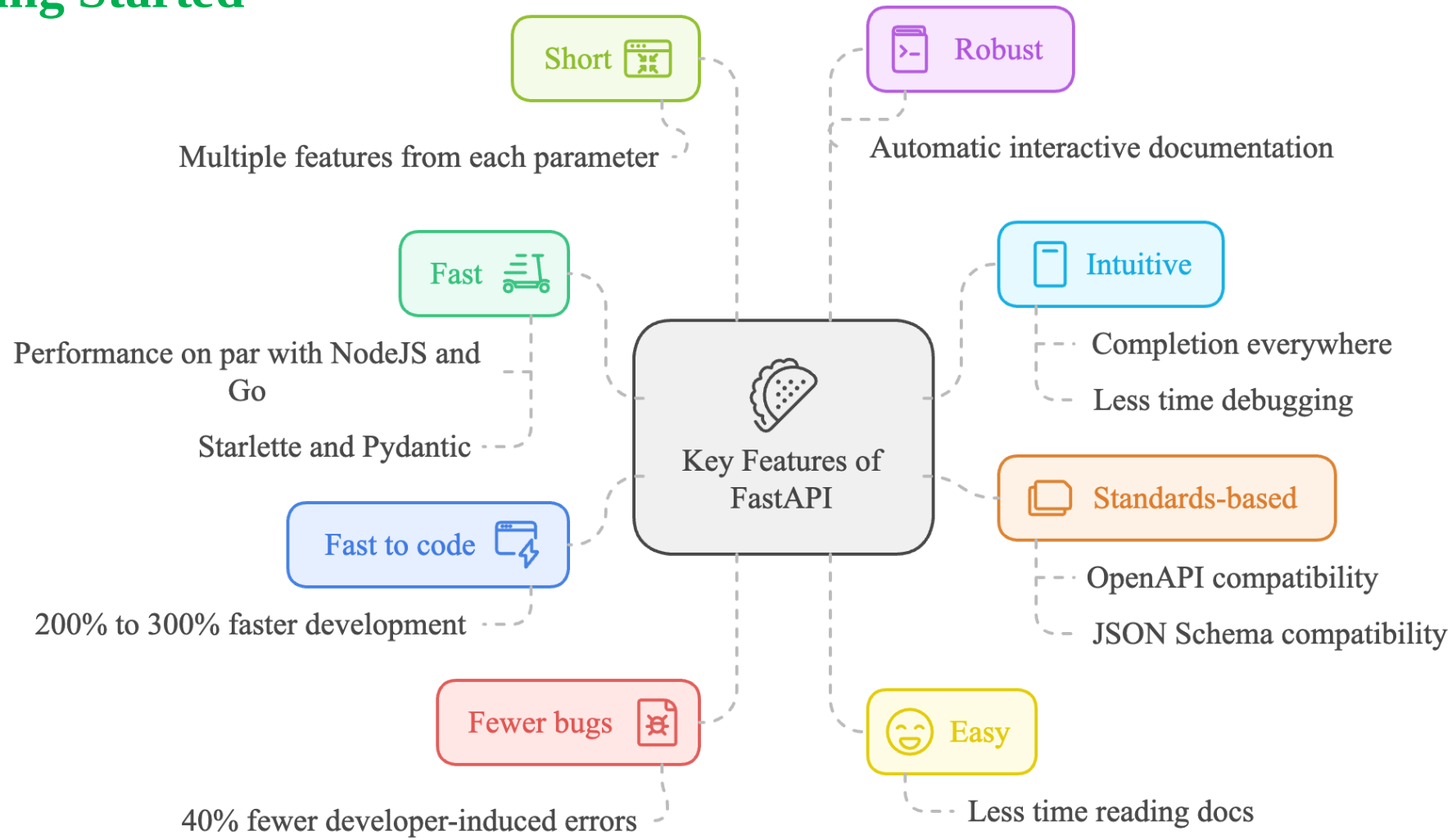
No features at all

Right in the middle

Too much features
(for most use cases)

FastAPI

❖ Getting Started



*estimation based on tests on an internal development team, building production applications.

FastAPI

❖ Django vs Flask vs FastAPI

	django	Flask	FastAPI
Performance speed	Normal	Faster than Django	The fastest out there
Async support	**YES** with restricted latency	**NO** needs Asyncio	**YES** native async support
Packages	**Plenty** for robust web apps	**Less than Django** for minimalistic apps	**The least of all** for building web apps faster
Popularity	The most popular	The second popular	Relatively new
Learning	Hard to learn	Easier to learn	The easiest to learn

FastAPI

❖ First API

```python
from fastapi import FastAPI

app = FastAPI()
```

```
(fastapi) ➜  code git:(main) ✗ uvicorn 001-first_api_1:app
INFO:     Started server process [54212]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO:     127.0.0.1:54562 - "GET /docs HTTP/1.1" 200 OK
INFO:     127.0.0.1:54562 - "GET /openapi.json HTTP/1.1" 200 OK
```

- Create a Python file (001-first_api_1.py in this case)
- This is the simplest form of a Fast application.

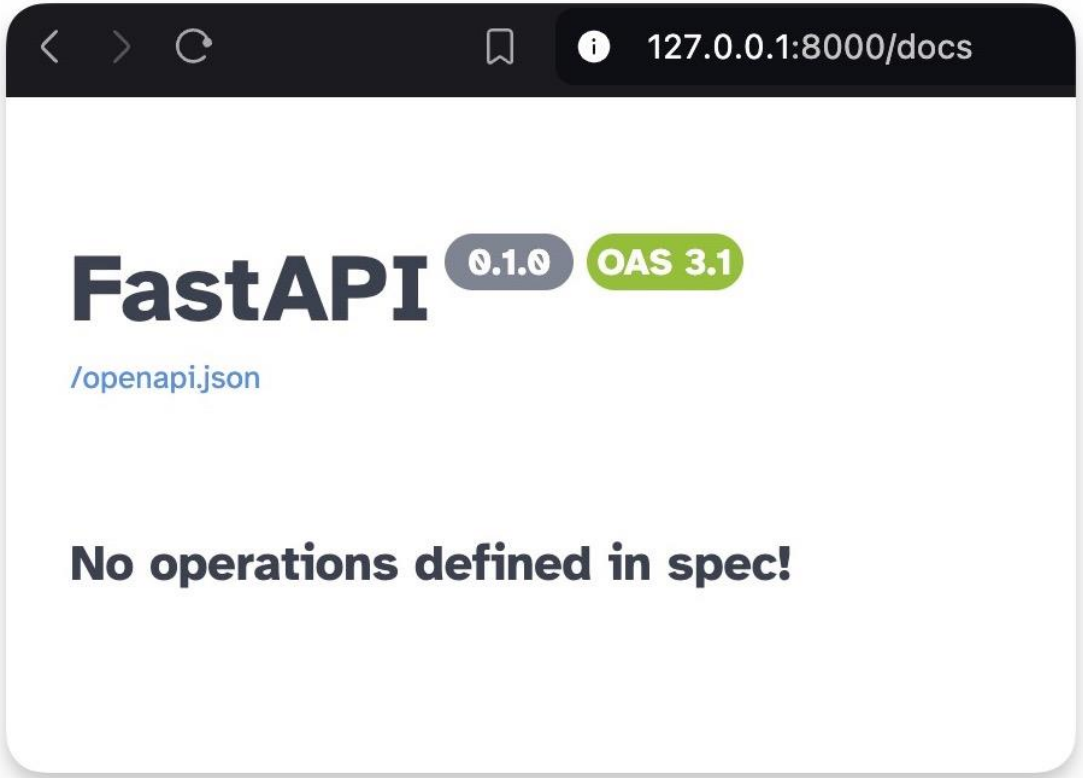
Open a new terminal and run the following command to start FastAPI:
uvicorn 001-first_api_1:app

001-first_api_1.py

FastAPI

❖ First API

Then locate localhost:8000/docs to see the Swagger UI.



FastAPI

❖ First API

```python
from fastapi import FastAPI

app = FastAPI()


if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app)
```

```
(fastapi) ➜  code git:(main) ✗ python 001-first_api_2.py
INFO:     Started server process [54614]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO:     127.0.0.1:54806 - "GET /docs HTTP/1.1" 200 OK
INFO:     127.0.0.1:54806 - "GET /openapi.json HTTP/1.1" 200 OK
```

We can also start a FastAPI app like a normal Python program.

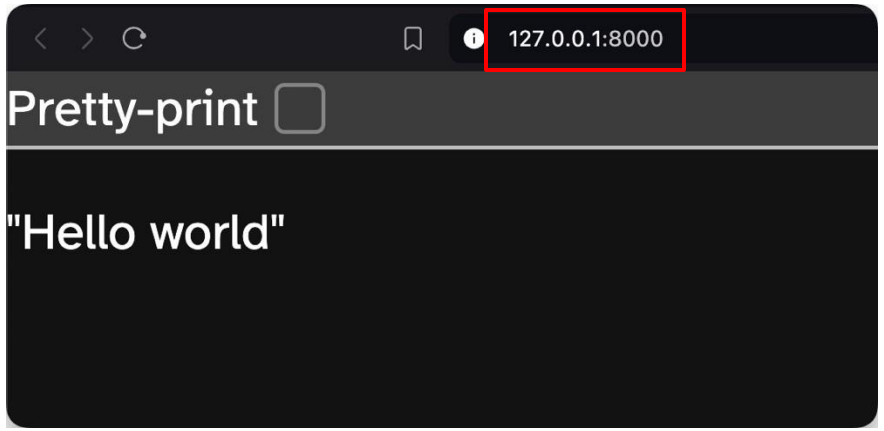
Then simply run:
python 001-first_api_2.py

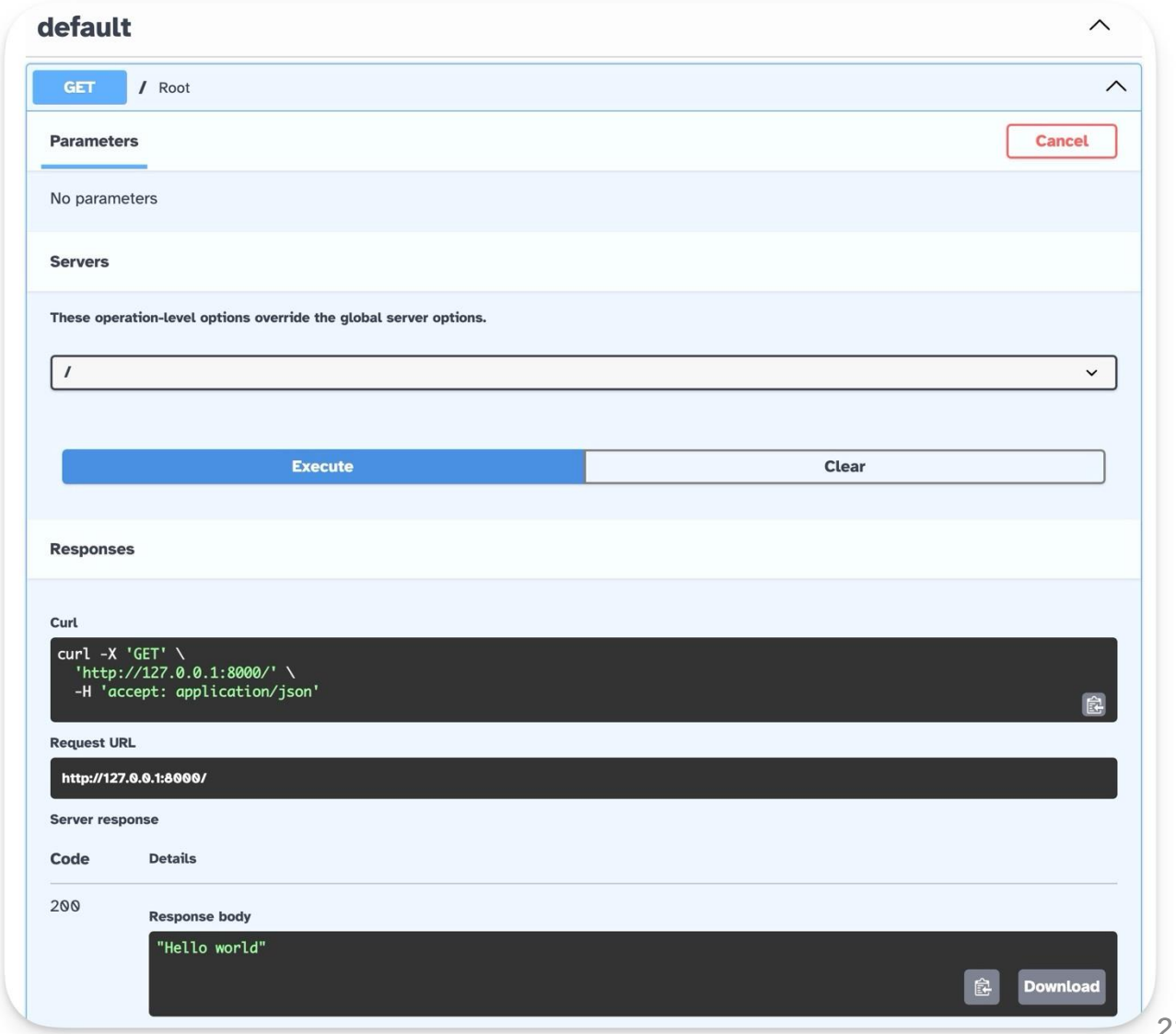
001-first_api_2.py

FastAPI

❖ First API

```python
from fastapi import FastAPI

app = FastAPI()

@app.get ('/')
def root():
    return "Hello world"

```





001-first_api_3.py

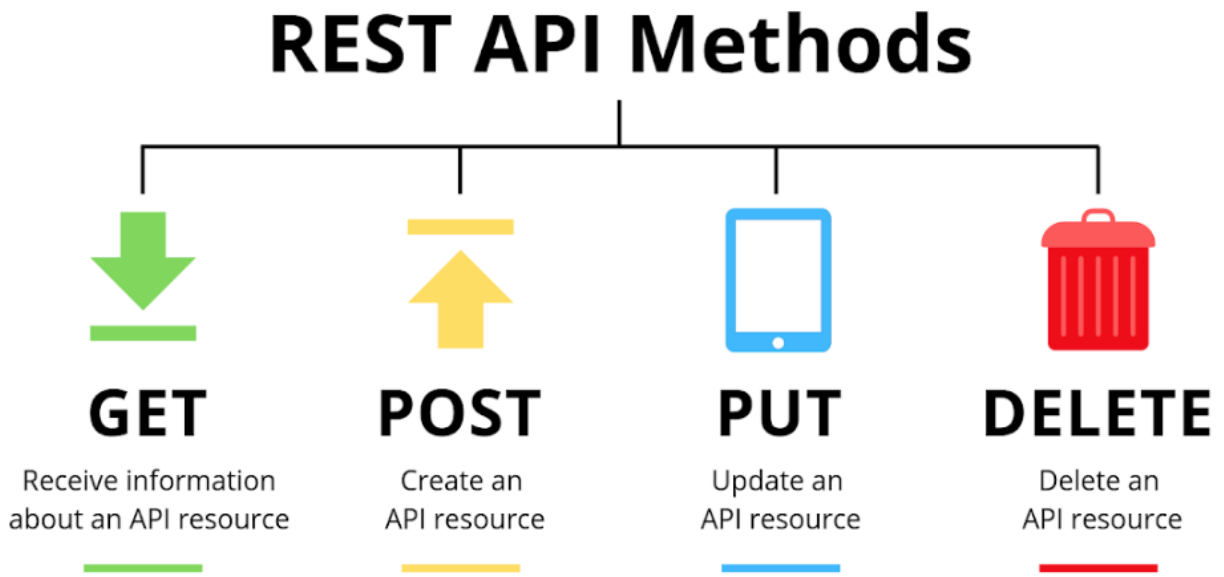
What is @app.get(‘/’)?

FastAPI

❖ Path Operations

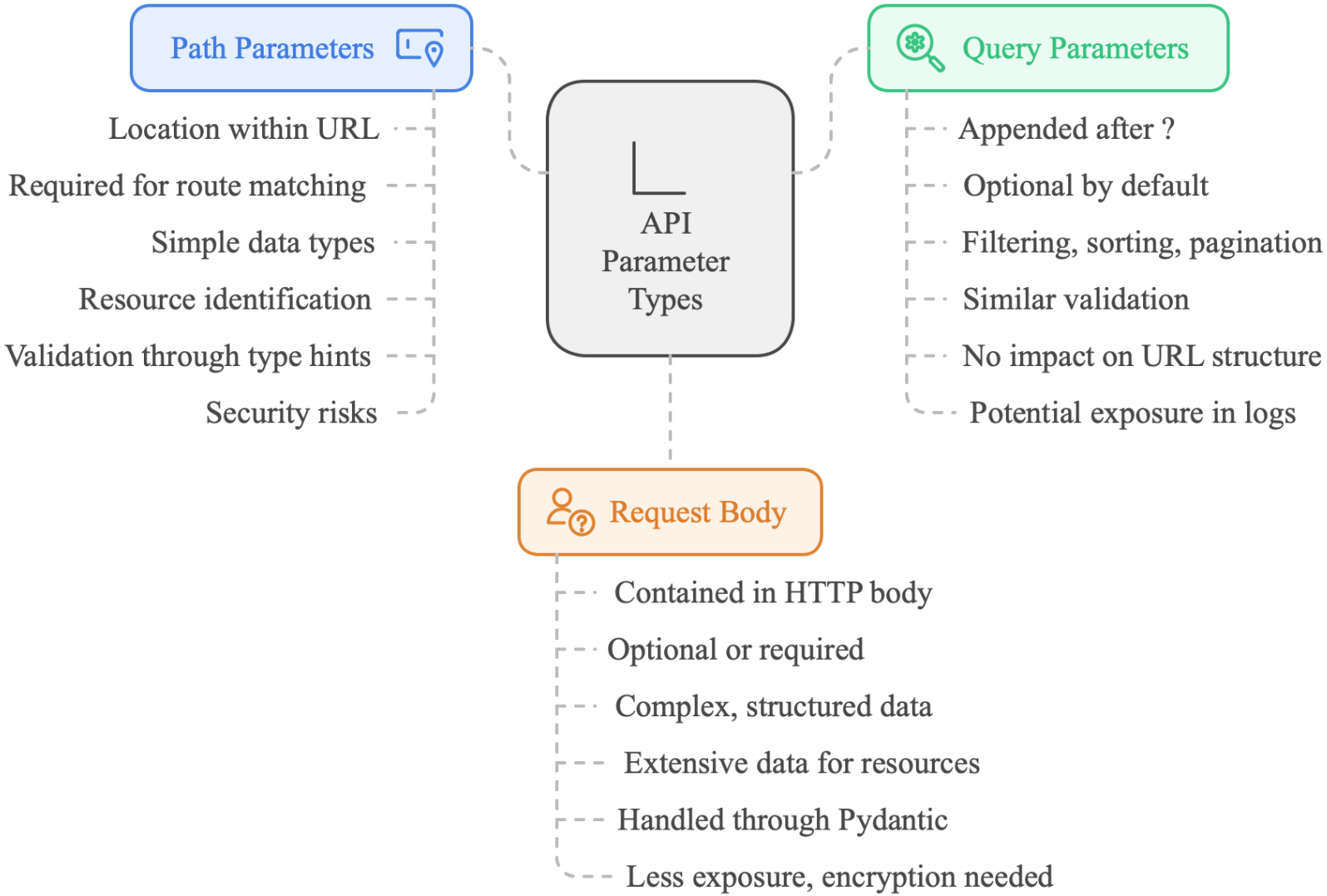
Path operations are the core of a FastAPI application. They define how clients interact with your API by specifying:

- **The URL path**: Where the resource is located.
- **The HTTP method**: The action to be performed (GET, POST, PUT, DELETE).
- **The function**: The code that will be executed to handle the request.



FastAPI

❖ How to send data to the API endpoints



FastAPI

❖ Path Operations: Query Parameter

```python
from fastapi import FastAPI

app = FastAPI()

fake_db = [
    {
        "id": 0,
        "username": "Nam",
    },
    {
        "id": 1,
        "username": "Lan",
    },
]


@app.get("/get_user")
def get_user(user_id: int):
    return fake_db[user_id]


@app.post("/post_user")
def post_user(username: str):
    new_user = {"id": len(fake_db), "username": username}
    fake_db.append(new_user)
    return new_user
```

```python
@app.put("/update_user")
def update_user(user_id: int, username: str):
    fake_db[user_id]["username"] = username
    return fake_db[user_id]


@app.delete("/delete_user")
def delete_user(user_id: int):
    del fake_db[user_id]
    return {"message": "User deleted successfully"}
```

This example shows 4 popular operations in FastAPI.

002-path_operation_crud-1.py

FastAPI

❖ Path Operations: Query Parameter and Path Parameter

```python
from fastapi import FastAPI

app = FastAPI()

fake_db = [
    {
        "id": 0,
        "username": "Nam",
    },
    {
        "id": 1,
        "username": "Lan",
    },
]


@app.get("/get_user/{user_id}")
def get_user(user_id: int):
    return fake_db[user_id]


@app.post("/post_user")
def post_user(username: str):
    new_user = {"id": len(fake_db), "username": username}
    fake_db.append(new_user)
    return new_user
```

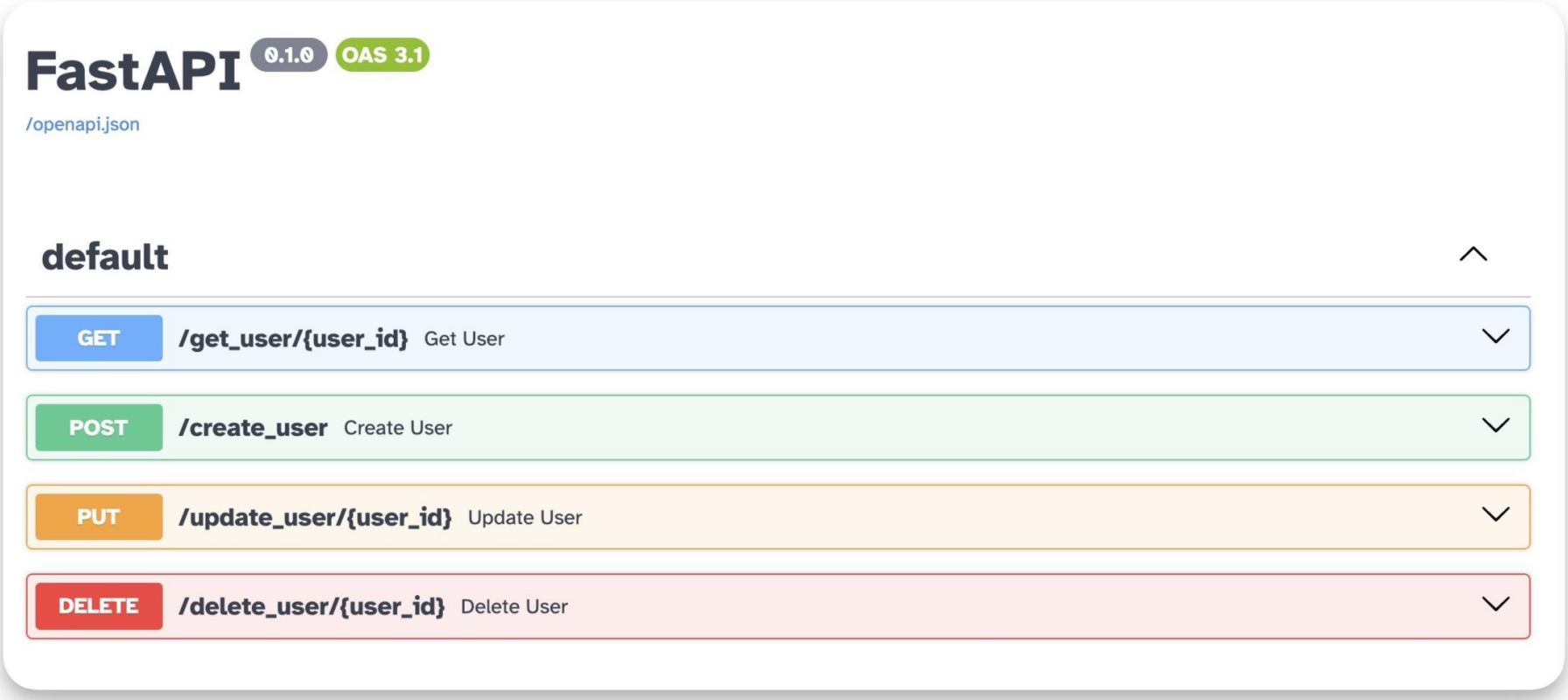
```python
@app.put("/update_user/{user_id}")
def update_user(user_id: int, username: str):
    fake_db[user_id]["username"] = username
    return fake_db[user_id]


@app.delete("/delete_user/{user_id}")
def delete_user(user_id: int):
    del fake_db[user_id]
    return {"message": "User deleted successfully"}
```

Let's update some operations with path parameters.

002-path_operation_crud-2.py

FastAPI

❖ Path Operations: Query Parameter and Path Parameter



FastAPI

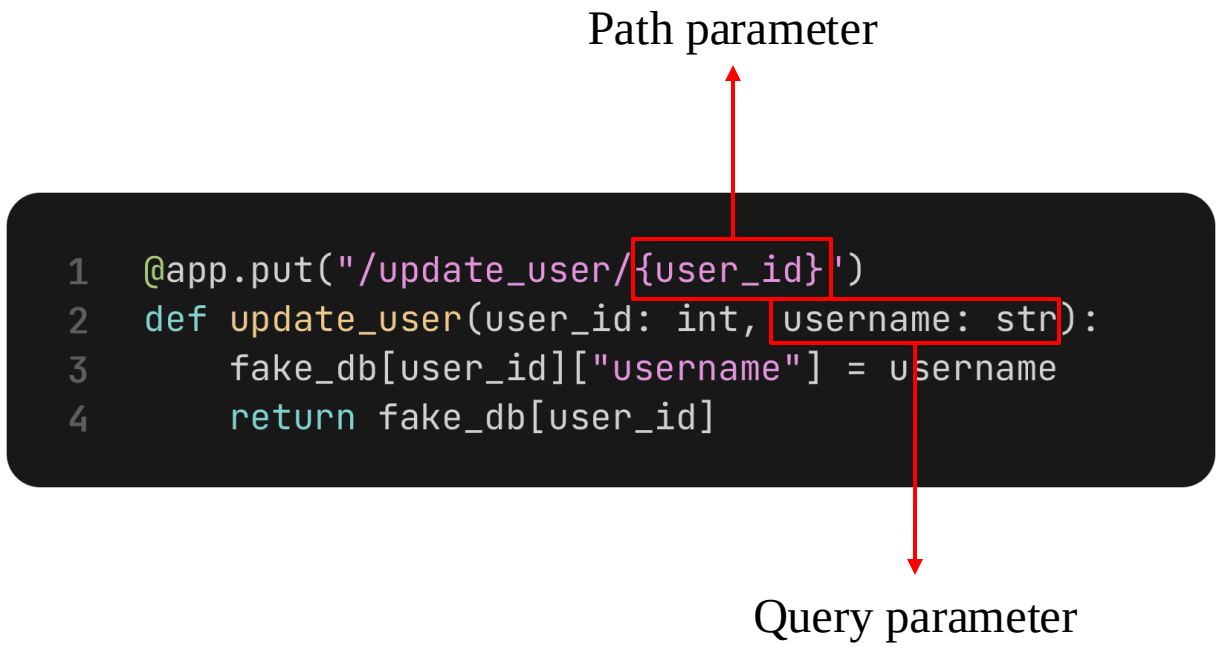
❖ Path Parameters and Query Parameters

Path parameters are part of the URL path. They are typically used to identify a specific resource.

Query parameters are appended to the URL after a question mark. They are used to filter or sort resources.

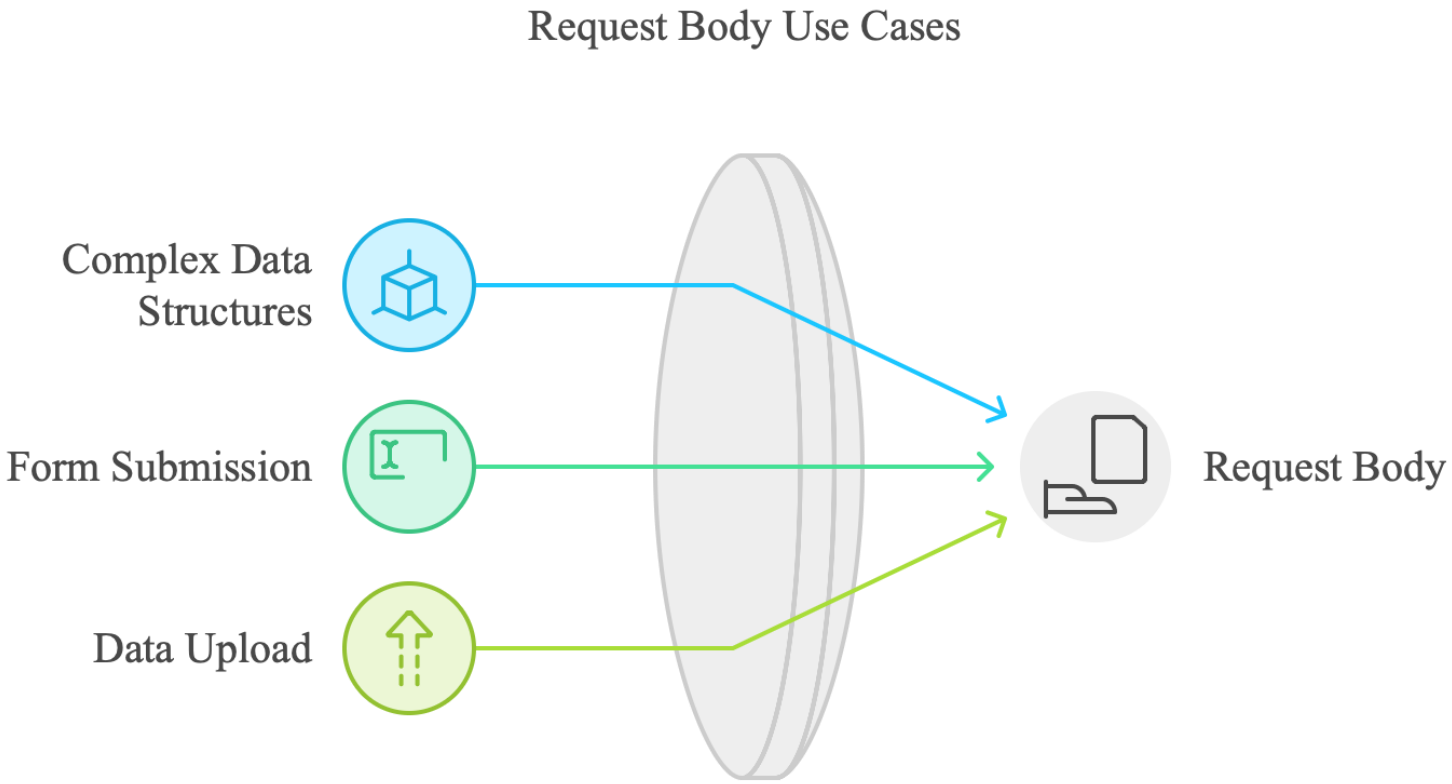
Example:

- /users/{user_id} - user_id is a path parameter.
- /users?username=lan - username is a query parameter.



FastAPI

❖ Request Body



The request body is used to send complex data structures to the API, typically in JSON format.

FastAPI

❖ Pydantic Model

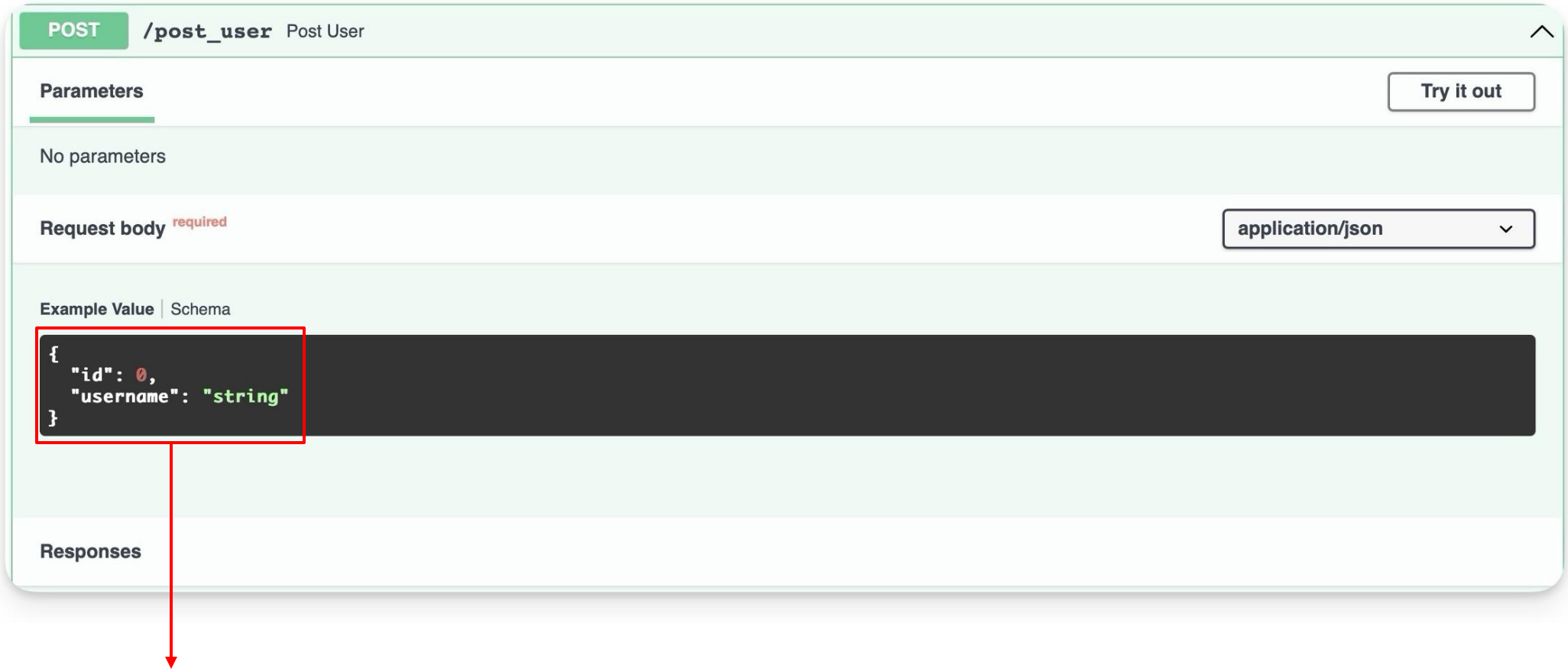
```python
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

fake_db = [
    {
        "id": 0,
        "username": "Nam",
    },
    {
        "id": 1,
        "username": "Lan",
    },
]


class User(BaseModel):
    id: int | None = None
    username: str
```

```python
@app.get("/get_user/{user_id}")
def get_user(user_id: int):
    return fake_db[user_id]


@app.post("/post_user")
def post_user(user: User):
    user.id = len(fake_db)
    fake_db.append(user)
    return user


@app.put("/update_user/{user_id}")
def update_user(user_id: int, user: User):
    fake_db[user_id]["username"] = user.username
    return fake_db[user_id]


@app.delete("/delete_user/{user_id}")
def delete_user(user_id: int):
    del fake_db[user_id]
    return {"message": "User deleted successfully"}
```


003-request-body-pydantic-1.py

FastAPI

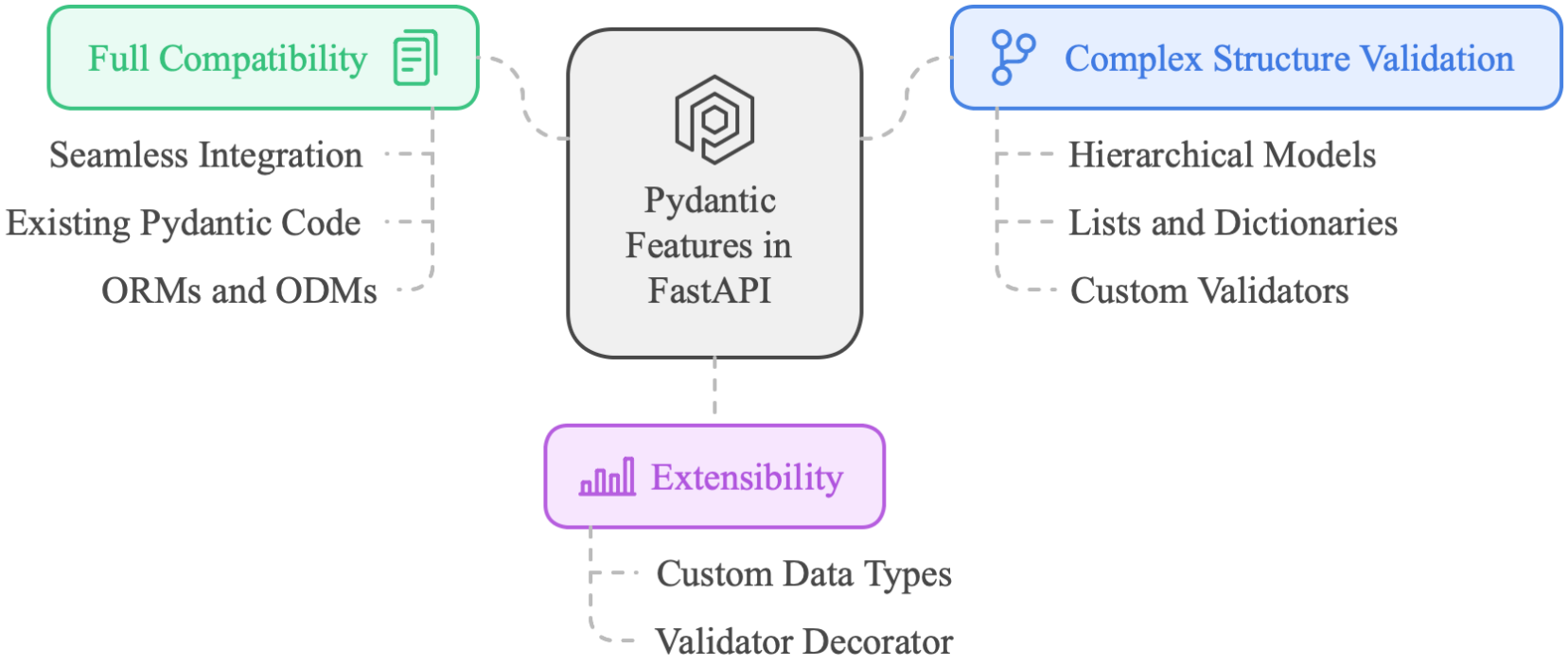
❖ Pydantic Model



Request body

FastAPI

❖ Pydantic Model



FastAPI

❖ Pydantic Model: Nested Loop

Nested loop

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

app = FastAPI()


class UserDetails(BaseModel):
    full_name: str
    age: int | None = None


class User(BaseModel):
    id: int | None = None
    username: str
    details: UserDetails
```

```python
# Updated fake database with nested structure
fake_db = [
    {
        "id": 0,
        "username": "Nam",
        "details": {
            "full_name": "Nam Nguyen",
            "age": 30,
        },
    },
    {
        "id": 1,
        "username": "Lan",
        "details": {
            "full_name": "Lan Tran",
            "age": 28,
        },
    },
]


@app.get("/get_user/{user_id}", response_model=User)
def get_user(user_id: int):
    return fake_db[user_id]


@app.post("/create_user", response_model=User)
def create_user(user: User):
    user.id = len(fake_db)
    fake_db.append(user)
    return user
```

003-request-body-pydantic-2.py

FastAPI

❖ How to send data to the API endpoints

Aspect	**Path Parameters**	**Query Parameters**	**Request Body**
Location in Request	Part of the URL path	Appended to the URL after "?"	Included in the HTTP request body
Required/Optional	**Required** for the route to match	**Optional** by default (can have defaults)	**Optional/Required** based on endpoint design
Data Type Complexity	Simple, typically primitive types (e.g., int, str)	Simple, typically primitive types (e.g., int, str)	Complex, structured data (e.g., JSON objects)
Use Cases	Identifying specific resources (e.g., resource ID)	Filtering, sorting, pagination, optional parameters	Creating or updating resources with extensive data
Typical HTTP Methods	All methods (GET, POST, PUT, DELETE, etc.)	Usually GET but can be used with other methods	Primarily POST, PUT
Default Values	Not applicable (always required)	Can have default values to make them optional	Defaults can be set within the request body schema
Data Validation	Handled via type hints and FastAPI validation	Handled via type hints and FastAPI validation	Handled via Pydantic models for complex validation
Security	Potentially exposed in logs and URL history	Potentially exposed in logs and URL history	Less exposed in logs and URL history; encryption may be necessary

FastAPI

❖ Return Type

- We can declare the return type of a path operation with Python type hints.
- Using → list[Item] tells FastAPI that the function read_items will return a list of objects of type Item.

FastAPI will:

- Validate the data returned by the function.
- Document the return type in the corrected schema.

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

app = FastAPI()


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


@app.get("/items/")
async def read_items() -> list[Item]:
    return [
        Item(name="Banh trung thu thap cam", price=45.0),
        Item(name="Banh trung thu dau xanh", price=40.0),
    ]
```

004-return-type.py

FastAPI

❖ Response Model

This example demonstrates defining and using response models:

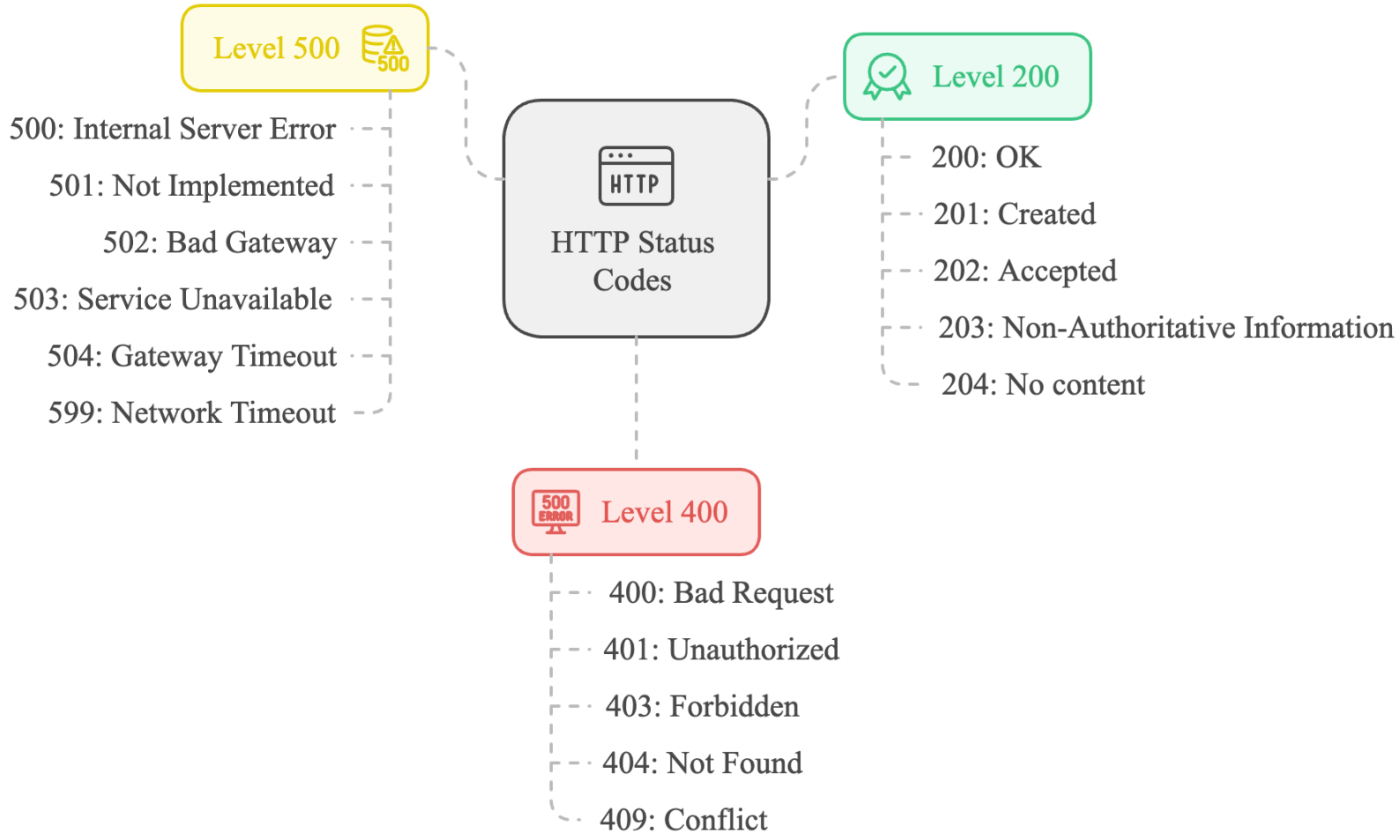
- We define a Pydantic `Item` model to structure the response data.
- We use `response_model` in the path operation decorators to:
  - Enforce the response structure and data types.
  - Automatically generate OpenAPI documentation.
- We can use `list[Item]` to indicate a list of Item objects in the response.

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

app = FastAPI()


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


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


@app.get("/items/", response_model=list[Item])
async def read_items():
    return [
        Item(name="Banh trung thu thap cam", price=45.0),
        Item(name="Banh trung thu dau xanh", price=40.0),
    ]
```

004-response-model.py

FastAPI

❖ Status Code



HTTP response status codes indicate whether a specific HTTP request has been successfully completed.

FastAPI

❖ Status Code Examples

```
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO:     Started reloader process [8153] using WatchFiles
INFO:     Started server process [8155]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     127.0.0.1:51823 - "GET /docs HTTP/1.1" 200 OK
INFO:     127.0.0.1:51823 - "GET /openapi.json HTTP/1.1" 200 OK
INFO:     127.0.0.1:51823 - "GET /items/ HTTP/1.1" 200 OK
```


```
INFO:     Application startup complete.
INFO:     127.0.0.1:51898 - "GET /items/ HTTP/1.1" 500 Internal Server Error
ERROR:    Exception in ASGI application
Traceback (most recent call last):
  File "/opt/homebrew/lib/python3.11/site-packages/uvicorn/protocols/http/httptools_impl.py", line 435, in run_asgi
    result = await app(  # type: ignore[func-returns-value]
             ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/opt/homebrew/lib/python3.11/site-packages/uvicorn/middleware/proxy_headers.py", line 78, in __call__
    return await self.app(scope, receive, send)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/opt/homebrew/lib/python3.11/site-packages/fastapi/applications.py", line 276, in __call__
    await super().__call__(scope, receive, send)
  File "/opt/homebrew/lib/python3.11/site-packages/starlette/applications.py", line 122, in __call__
    await self.middleware_stack(scope, receive, send)
```

FastAPI

❖ Annotated and Field

```python
from typing import Annotated

from fastapi import FastAPI, Query
from pydantic import BaseModel, Field

app = FastAPI()

items = [
    {
        "name": "Banh thap cam",
        "price": 45.0,
        "description": "Description of the banh trung thu thap cam",
    },
    {
        "name": "Banh dau xanh",
        "price": 40.0,
        "description": "Description of the banh trung thu dau xanh",
    },
]


class Item(BaseModel):
    name: str = Field(..., min_length=1, max_length=100)
    price: float = Field(..., gt=0, description="Price of the mooncake")
    description: str | None = Field(
        None, max_length=500, description="Optional description"
    )
```

```python
@app.get("/search_items/")
async def search_items(
    q: Annotated[str, Query(min_length=3, max_length=50, description="Search query")],
    max_price: Annotated[float | None, Query(gt=0, description="Maximum price")] = None,
):
    results = [item for item in items if q.lower() in item["name"].lower()]

    if max_price:
        results = [item for item in results if item["price"] ⩽ max_price]

    return results
```

- **Purpose:** Add metadata to Pydantic models
- **Annotated:** Used for adding custom metadata to types
- **Field:** Used for:
  - Defining model field details (e.g., default values, validation)
  - Including metadata for documentation or serialization

FastAPI

❖ File and Upload File

This example shows two ways to handle file uploads in FastAPI:

- **Direct File Access:** Receive the file as bytes and access its size.
- **UploadFile Class:** Use the UploadFile class for more metadata like accessing filename and content type.

```python
from typing import Annotated

from fastapi import FastAPI, File, UploadFile

app = FastAPI()


@app.post("/files/")
async def create_file(file: Annotated[bytes, File()]):
    return {
        "file_size": len(file),
    }


@app.post("/uploadfile/")
async def create_upload_file(file: UploadFile):
    return {
        "filename": file.filename,
        "content_type": file.content_type,
    }
```

006-file-and-upload-file.py

FastAPI

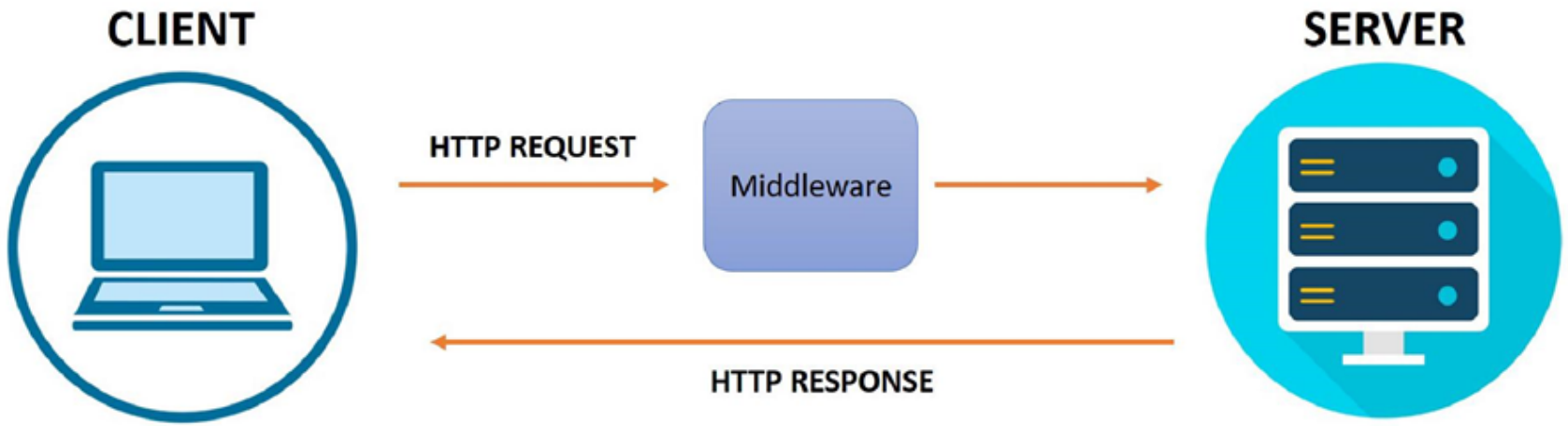
❖ File and Upload File

Feature/Aspect	File	UploadFile
Purpose	Used to read the entire file content into memory.	Used to handle file uploads efficiently without loading the entire file into memory.
Data Type	bytes	UploadFile object
Memory Usage	High, as it reads the entire file into memory.	Low, as it streams the file, reducing memory usage.
Use Case	Suitable for small files where the entire content needs to be processed at once.	Suitable for large files or when you need to handle files without loading them entirely into memory.
Methods Available	Standard bytes operations.	read(), write(), seek(), close(), etc.
File Metadata Access	Not directly available.	Access to filename, content type, and other metadata.
Performance	Can be slower for large files due to memory constraints.	More efficient for large files due to streaming.

FastAPI

❖ Middleware

A “middleware” is a function that works with every **request** before any specific path operation processes it. And also with every **response** before returning it.



FastAPI

❖ Middleware

- **FastAPI Middleware**: Measures HTTP request processing time.
- **Records Start/End Times**: Uses time.perf_counter() to track request duration.
- **Adds "X-Process-Time" Header**: Includes processing time in response headers.
- **Calls Next Handler**: Passes the request to the next middleware or route.

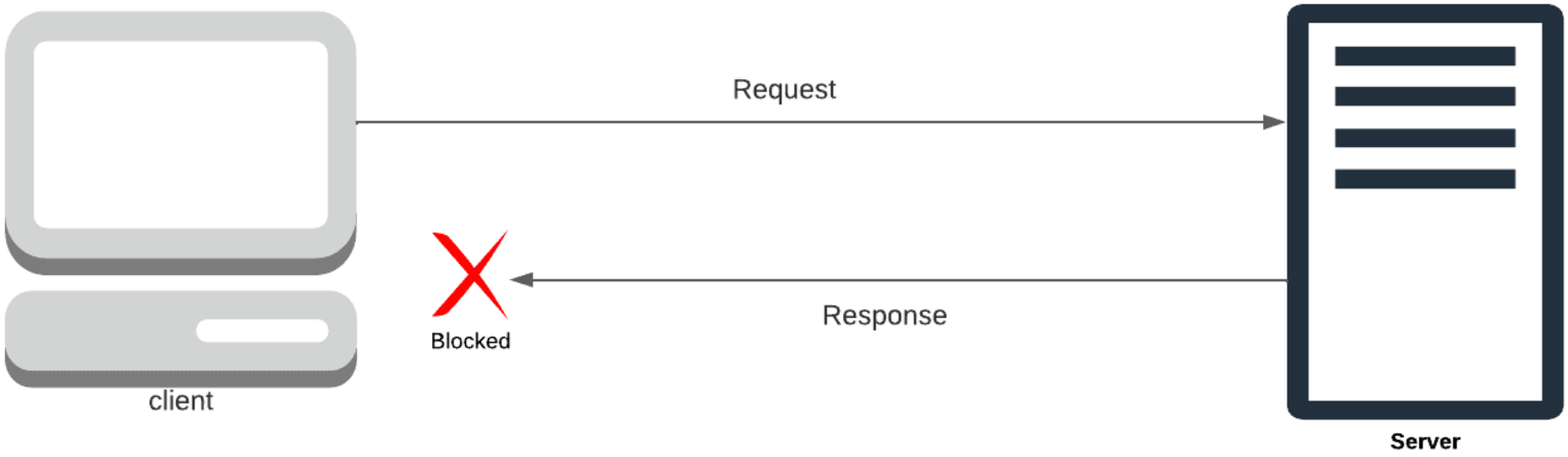
```python
import time

from fastapi import FastAPI, Request

app = FastAPI()


@app.middleware("http")
# This decorator registers a middleware function that will be executed
# for every incoming HTTP request.
async def add_process_time_header(request: Request, call_next):
    # Record the start time of the request processing
    start_time = time.perf_counter()
    # Call the next middleware or route handler in the chain
    response = await call_next(request)
    # Calculate the time taken to process the request
    process_time = time.perf_counter() - start_time
    # Add a custom header to the response containing the process time
    response.headers["X-Process-Time"] = str(process_time)
    # Return the modified response
    return response
```


007-middleware.py

FastAPI

❖ CORS (Cross-Origin Resource Sharing)

CORS is a method that **permits** resources to be loaded from one origin to another while **preserving** the security and integrity of the website. Popular web browsers like Chrome and Mozilla Firefox use this security technique to determine which cross-site requests are secure.



FastAPI

❖ CORS (Cross-Origin Resource Sharing)

- Allow requests from specific origins

  (e.g., http://localhost.tiangolo.com)
- Allow credentials (cookies, authorization headers, etc.)
- Allow all HTTP methods (GET, POST, etc.)
- Enable all headers.

```python
from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware

app = FastAPI()

origins = [
    "http://localhost.tiangolo.com",
    "https://localhost.tiangolo.com",
    "http://localhost",
    "http://localhost:8080",
]

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


@app.get("/")
async def main():
    return {"message": "Hello World"}
```


008-cors.py

FastAPI

❖ Static Files

- Serve static files automatically from a directory using StaticFiles.
- What is "Mounting”?

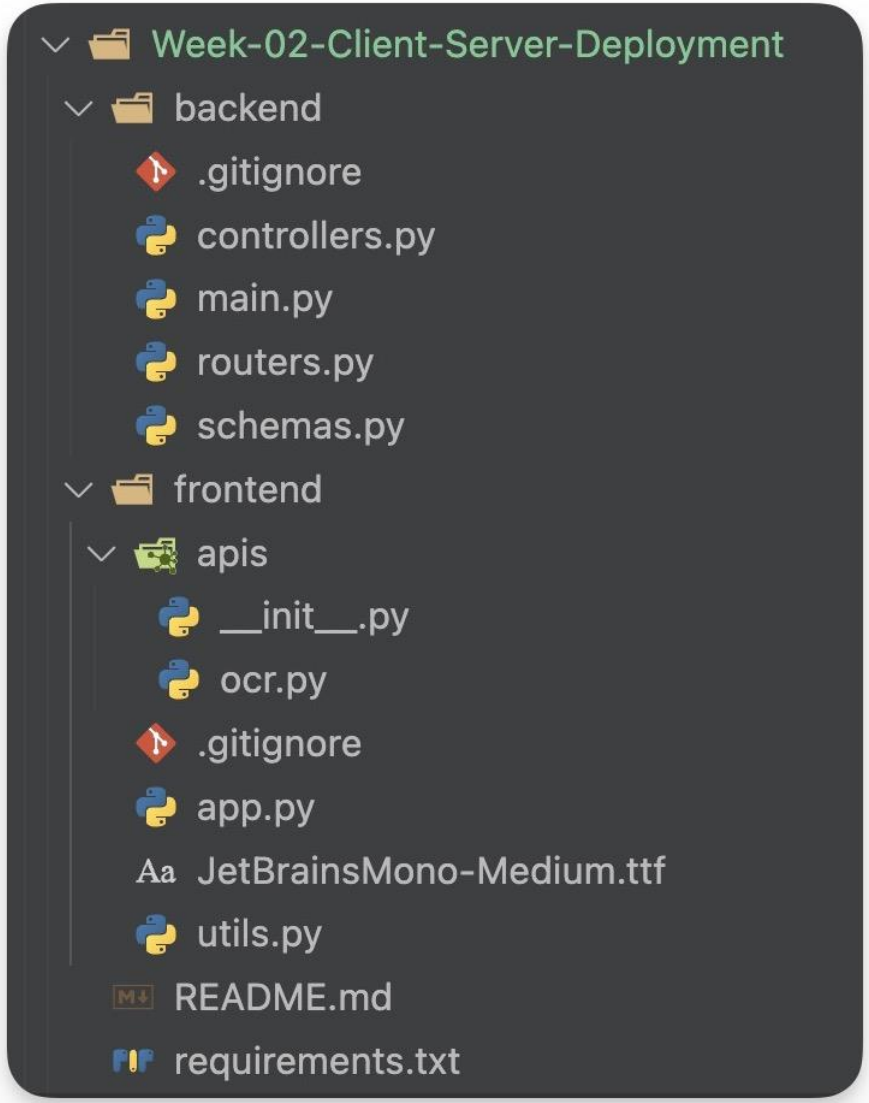
  "Mounting" means adding a complete "independent" application in a specific path, that then takes care of handling all the sub-paths.

```python
from fastapi import FastAPI
from fastapi.staticfiles import StaticFiles

app = FastAPI()

# Mount the static files directory
app.mount("/static", StaticFiles(directory="static"), name="static")


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


010-static-files.py

Integration

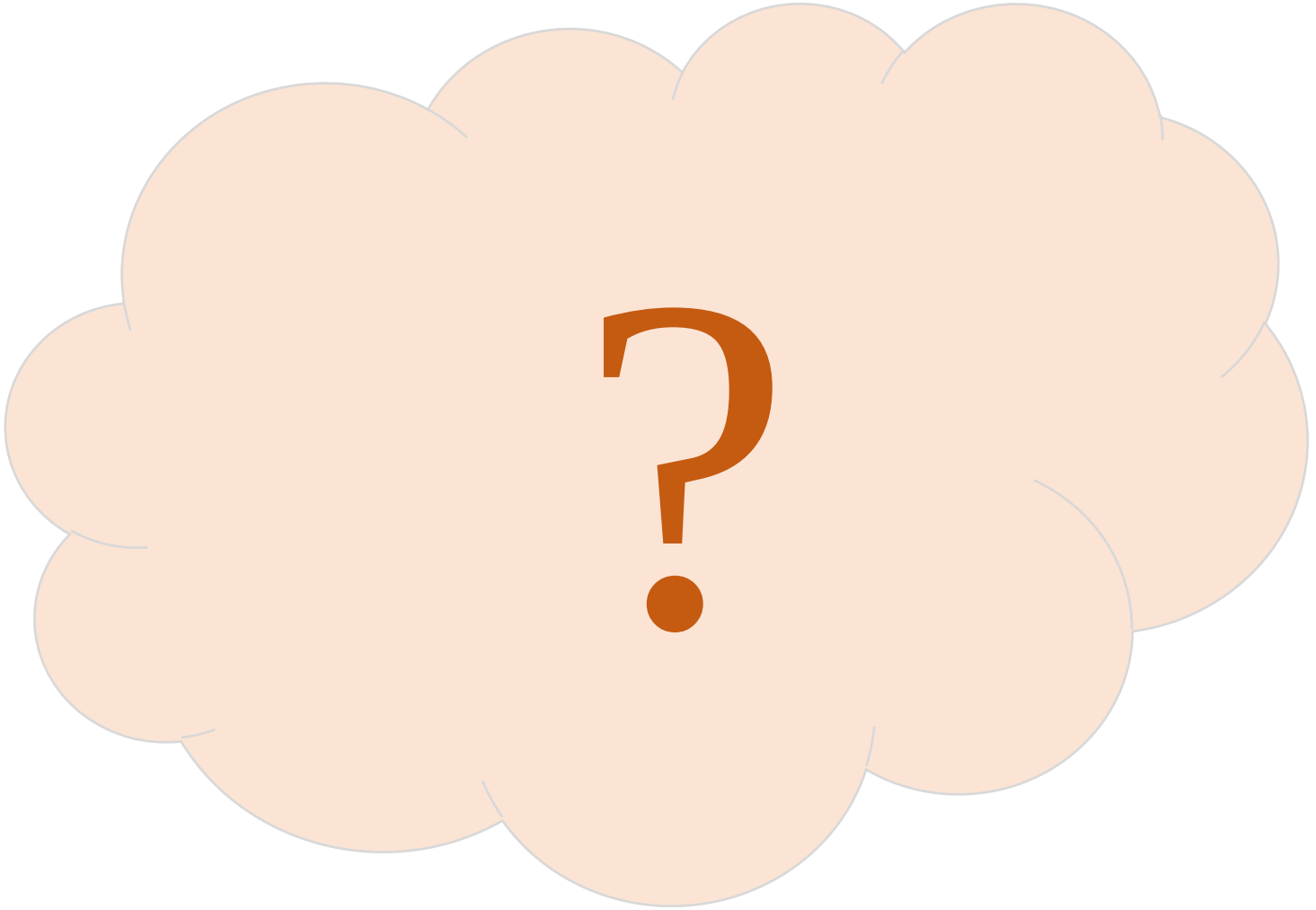
Integration



Deploy the applications using the client-server architecture.

- **The backend**: server will be built using the FastAPI package.
- **The frontend**: application will still use the Gradio package but the prediction process will be done on the backend server.

Question




Thank you!