A **server** is **a computer (or program)** whose **job is to listen for requests and respond with information or actions**.

A server is a program running on a computer/cloud that receives requests, processes them, and sends responses.

Server

A **server** is a *running program* that waits for requests, processes them using rules or logic, and sends back responses. It must be running continuously and listening (usually on a **network port**). A server does not have to be special hardware; it can be any computer running a program. When your FastAPI app is running and waiting for requests, it is acting as a server.

Service

A **service** is software that performs *one specific responsibility* and can be used by other programs. A service usually runs on a server, but the idea of a service is about **what it does**, not where it runs. For example, a "data preprocessing service" takes raw data and returns transformed data. Services communicate through APIs, which means they accept requests and return responses.

➡️ Relationship: **A service needs a server to stay alive and accept requests.**

Microservice

A **microservice** is a *small, independent service* that focuses on a single task and can be developed, deployed, and scaled independently. Instead of one large application doing everything, the system is broken into multiple microservices that talk to each other. In an ML system, preprocessing can be one microservice and prediction another.

➡️ Relationship: **Every microservice is a service, and every service runs on a server.**

API (important bridge concept)

An **API** is the contract that defines how a service can be used. It specifies what input the service expects and what output it returns. APIs are how clients or other services communicate with servers. Without an API, a service cannot be accessed externally.

➡️ Relationship: **A service exposes its functionality through an API, and the server delivers that API.**

FastAPI

FastAPI is a Python framework that helps you create APIs easily, so your Python code can act as a service and run as a server. FastAPI handles request parsing, data validation, routing, and response formatting, allowing you to focus on business logic like data preprocessing or model prediction. When you run a FastAPI app, it becomes a server that exposes services through APIs.

➡️ Relationship: **FastAPI turns Python functions into services, exposes them via APIs, and runs them as servers.**

How everything connects (big picture)

A **FastAPI application** runs as a **server**, that server hosts one or more **services**, each service exposes its functionality through **APIs**, and when those services are small and independent, they are called **microservices**. In data science, this allows you to separate preprocessing, prediction, and post-processing into different services that can evolve independently.

One-line memory chain 🧠

FastAPI → builds APIs → APIs expose services → services run on servers → small independent services are microservices

How Amazon Works:

1. **User Service** – Handles user login, profiles, authentication.
2. **Product Service** – Manages product details and inventory.
3. **Order Service** – Handles shopping carts and orders.
4. **Payment Service** – Processes payments.
5. **Recommendation Service** – Suggests products based on your history.
6. **Notification Service** – Sends emails or alerts about orders.

Youtube Micro service:

1. **Video Upload Service** – Handles uploading and storing videos.
2. **Streaming/Playback Service** – Streams videos efficiently.
3. **Comments Service** – Manages likes, comments, and replies.
4. **Recommendation Service** – Suggests videos based on history.
5. **Ads Service** – Shows ads personalized to users.

Type hints validate simple function-level inputs and outputs, while Pydantic validates structured request and response data before it reaches the function. Together, they make FastAPI **robust, self-documenting, and safe**.

1️⃣ Type Hints

- **Definition:** Type hints specify the expected data types of function arguments and return values. In FastAPI, they help **validate query/path parameters** and **generate API documentation automatically**.
- **Scope:** Function-level (input arguments and return types).
- **FastAPI usage:** Works for query parameters, path parameters, and function return values.

Example: Type Hints for query parameter

```
from fastapi import FastAPI
from typing import Optional

app = FastAPI()

@app.get("/square")
def square(number: int) -> int:
    # This function takes 'number' as a query parameter
    # Example URL: http://127.0.0.1:8000/square?number=5
    return number ** 2
```

Explanation:

- number: int → FastAPI checks that the query parameter is an integer.
- -> int → Indicates the function returns an integer.
- If the client sends invalid data (e.g., ?number=abc), FastAPI responds with **400 Bad Request** automatically.

2️⃣ Pydantic

- **Definition:** Pydantic is a Python library for **data parsing and validation**. In FastAPI, it is used to **validate request bodies, JSON data, and structured objects** before the function executes.
- **Scope:** Request/response level (especially for JSON body).
- **FastAPI usage:** Define **Pydantic models** to describe the expected structure of request or response data.

Example: Pydantic for request body

```
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

# 1. Define a Pydantic model for incoming data (The Blueprint)
class Item(BaseModel):
    name: str
    price: float
    in_stock: bool

# 2. Define the POST endpoint (The Receiver)
@app.post("/items/")
def create_item(item: Item):
    # This takes the JSON data from the request body
    return {
        "item_name": item.name,
        "item_price": item.price,
        "available": item.in_stock
    }
```

Explanation:

- Item defines the expected JSON structure.
- FastAPI uses **Pydantic** to check:
    - name must be a string
    - price must be a float
    - in stock must be a boolean
- Invalid data automatically triggers a **422 Unprocessable Entity** response.
- Your function receives a **clean, validated item object**, so you don't need to check types manually.

"From a programming perspective, imagine I have a **Microservices architecture** where my application is split into multiple independent services.

- **HTTP** is the **communication protocol** (the language and the road) they use to talk to each other.
- The **API** is the **defined interface** (the door) for each service. It acts as a contract that tells other services exactly what data they can request and what they will get back, ensuring that the internal logic of the service remains secure and private."

HTTP methods:

Get – Fetch Data | no changes in server
POST – Create Data | Create New resource
PUT – Update Entire Data | Replace Existing data
Patch – Partial update, only few fields
Delete – Delete Resource

HTTP Status Codes (Must Know)

1xx: Informational (The "Hold On" Codes)

These are rare in basic web dev but important for high-speed AI streaming.

- 101 Switching Protocols: Used when you move from standard HTTP to a WebSocket (like a live AI chat window that stays open for real-time streaming).

2xx: Success (The "I Got You" Codes)

- **200 OK:** The most common.
  - *Example:* You visit http://127.0.0.1:8000/products and get the list.
- **201 Created:** * *Example:* When you use your @app.post("/items/") function and a new product is successfully saved to your database.
- **204 No Content:** The request worked, but there is nothing to send back.
  - *Example:* You successfully delete a product from your list.

3xx: Redirection (The "Go Over There" Codes)

- **301 Moved Permanently:**
  - *Example:* You changed your API version. If someone hits /old-api/square, you redirect them to /v2/square.
- **304 Not Modified:**
  - *Example:* Your browser asks for the openapi.json file, but since the code hasn't changed, the server says, "Just use the one you already have in your cache."

4xx: Client Errors (The "You Messed Up" Codes)

These happen because the user (or the Agent script) sent something wrong.

- **400 Bad Request: General error.**
  - *Example:* You sent a string to your /square endpoint instead of an integer.
- **401 Unauthorized:**
  - *Example:* You tried to access OpenAI's API but forgot to provide your API Key.
- **403 Forbidden: You are logged in, but you don't have permission.**
  - *Example:* A regular user tries to access the /admin/delete-all-products path.
- **404 Not Found:**
  - *Example:* You typed http://127.0.0.1:8000/producTS (wrong spelling/case).

- **422 Unprocessable Entity (FastAPI Special):**
    - *Example:* Your Pydantic model expects price: float, but you sent price: "Ten Dollars". FastAPI uses this to tell you exactly which field failed.

5xx: Server Errors (The "I Messed Up" Codes)
The code is broken, or the computer running it crashed.

- **500 Internal Server Error:**
    - *Example:* You have a typo in your Python logic (like 1/0) that causes the script to crash while trying to process a request.
- **502 Bad Gateway:** * *Example:* Your Uvicorn server is off, but your Nginx or Cloud provider is still trying to talk to it.
- **503 Service Unavailable:**
    - *Example:* Your AI model is so busy it cannot handle another request, or the server is undergoing maintenance.
- **504 Gateway Timeout:**
    - *Example:* Your AI agent is taking too long to generate a response (e.g., 60+ seconds), and the web server gives up on waiting.

GET /users/123/orders?status=shipped&limit=10
/users/123/orders → Orders for user 123
status=shipped → Only shipped orders
limit=10 → Only first 10 results
Path = **resource identity**
Query = **search filter, pagination, optional parameter, sorting, toggle/flag or additional info**