Here is the **complete FastAPI project setup** for beginners — step-by-step, clean, and recommended for real projects.

✅ FastAPI Project Setup (Beginner → Advanced)

1️⃣ Create Project Folder

```
mkdir fastapi_project
cd fastapi_project
```

2️⃣ Create & Activate Virtual Environment

Linux/macOS:

```
python3 -m venv venv
source venv/bin/activate
```

Windows:

```
python -m venv venv
venv\Scripts\activate
```

3️⃣ Install FastAPI + Uvicorn

```
pip install fastapi uvicorn[standard]
```

⭐ `fastapi` → The framework
⭐ `uvicorn` → The server to run it

4️⃣ Create Folder Structure

Recommended project structure:

```
fastapi_project/
|── app/
|   ├── main.py
|   ├── routers/
|   |      └── user.py
|   ├── schemas/
|   |      └── user.py
|   ├── models/
|   |      └── user.py
|   ├── database.py
|── requirements.txt
```

You can start simple if you're a beginner:

```
app/
 └── main.py
```

5️⃣ Create `main.py`

`app/main.py`:

```python
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
def read_root():
    return {"message": "FastAPI setup successful!"}
```

6️⃣ Run the Project

Inside the project folder:

```
uvicorn app.main:app --reload
```

Now open:

- ▶ `http://127.0.0.1:8000`
- ▶ API Docs → `http://127.0.0.1:8000/docs`

🎯 Optional (Highly Recommended)

7️⃣ Create `requirements.txt`

```
pip freeze > requirements.txt
```

📌 (Advanced) Database Setup Example

If you want to connect with a database (MySQL/PostgreSQL/SQLite):

Install SQLAlchemy:

```
pip install SQLAlchemy
```

`app/database.py`:

```python
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker, declarative_base

DATABASE_URL = "sqlite:///./test.db"

engine = create_engine(DATABASE_URL, connect_args={"check_same_thread":
False})

SessionLocal = sessionmaker(autocommit=False, autoflush=False,
bind=engine)
Base = declarative_base()
```

Here is the **clearest and shortest explanation of Path Parameters in FastAPI**, with examples for beginners.

🔵 What Are Path Parameters?

Path parameters are **dynamic values inside the URL**.
They allow you to capture values like:

```
/users/5
/products/123
/blog/python
```

Here `5`, `123`, `python` are path parameters.

✅ Basic Path Parameter Example

```python
from fastapi import FastAPI

app = FastAPI()

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

👉 URL to test:

```
http://127.0.0.1:8000/items/10
```

Output:

```json
{
    "item_id": 10
}
```

🧠 FastAPI Auto Converts Types

If you set `item_id: int`, FastAPI will automatically:

- Convert string → int
- Validate it
- Return error if invalid

Example of invalid request:

```
/items/abc
```

You get:

```
422 Unprocessable Entity
```

🔥 Multiple Path Parameters

```
@app.get("/users/{user_id}/orders/{order_id}")
def get_order(user_id: int, order_id: int):
    return {"user": user_id, "order": order_id}
```

📌 Path Parameter + String

```
@app.get("/blog/{slug}")
def blog_detail(slug: str):
    return {"slug": slug}
```

🔐 Path Parameter with Validation (Constraints)

Example: age must be > 18

```
from fastapi import Path

@app.get("/age/{user_age}")
def check_age(user_age: int = Path(gt=18)):
    return {"age": user_age}
```

More constraints:

- `gt` → greater than
- `ge` → greater or equal
- `lt` → less than
- `le` → less or equal

⚡ Default Value Inside Path Parameter (Rare)

Only works for non-dynamic parts:

❌ Wrong:

```
/items/{item_id=50}   # Not allowed
```

✔ Recommended: use **query parameters** for defaults.

🌐 Using Enum as Path Parameter

```
from enum import Enum

class ModelName(str, Enum):
    alexnet = "alexnet"
    resnet = "resnet"

@app.get("/model/{model_name}")
def get_model(model_name: ModelName):
    return {"model": model_name}
```

Here is the **clearest explanation of Query Parameters in FastAPI**, with examples. This will help you understand how they work in real projects.

🔵 What Are Query Parameters?

Query parameters appear **after the `?` in the URL**, like:

```
/items?name=book&price=100
```

They are optional by default.

✅ Basic Query Parameter Example

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/items/")
def read_items(name: str, price: int):
    return {"name": name, "price": price}
```

Test URL:

```
http://127.0.0.1:8000/items/?name=phone&price=500
```

Output:

```
{
  "name": "phone",
  "price": 500
}
```

🔥 Query Parameters With Default Values

```
@app.get("/products/")
def list_products(limit: int = 10, category: str = "all"):
    return {"limit": limit, "category": category}
```

Test:

```
/products/?limit=5&category=electronics
```

👉 All query parameters become **optional** when default values are provided.

🧠 Optional Query Parameters

Use `Optional` from typing:

```
from typing import Optional

@app.get("/search/")
def search(keyword: Optional[str] = None):
    return {"keyword": keyword}
```

URL:

```
/search/
```

Output:

```
{"keyword": null}
```

🌟 Combine Path + Query Parameters

```
@app.get("/users/{user_id}")
def get_user(user_id: int, details: bool = False):
    return {"user_id": user_id, "details": details}
```

URL:

```
/users/10?details=true
```

🔒 Query Parameter Validation (Constraints)

Use `Query()`:

```
from fastapi import Query

@app.get("/items/")
def items(
    q: str = Query(min_length=3, max_length=10, regex="^[a-zA-Z]+$")
):
    return {"query": q}
```

Allows only letters, 3–10 characters.

Example:

```
/items/?q=hello
```

📦 Multiple Values in Query Parameters (List)

```
/tags/?tag=python&tag=fastapi&tag=api
```

```
from typing import List

@app.get("/tags/")
def get_tags(tag: List[str]):
    return {"tags": tag}
```

🔥 Query Parameters are Powerful Because:

- Automatically validated
- Optional by default
- Support type conversion
- Support multiple values
- Automatically show up in Swagger docs

Here is the **easiest explanation of Pydantic Models in FastAPI** with simple examples, validation, and best practices.

🔵 What Are Pydantic Models?

Pydantic models are **Python classes** that define the **structure, data types, and validation rules** for request data in FastAPI.

They help you handle:

✔ Request body
✔ Response models
✔ Data validation
✔ Serialization

🧩 1. Basic Pydantic Model Example

```
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

class Item(BaseModel):
    name: str
    price: float
    in_stock: bool

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

Test with JSON body:

```
{
  "name": "Laptop",
  "price": 85000,
  "in_stock": true
}
```

🔹 FastAPI automatically
→ validates data
→ converts types
→ shows fields in Swagger UI

🧠 2. Optional Fields

```python
from typing import Optional

class Item(BaseModel):
    name: str
    price: float
    description: Optional[str] = None
```

🔥 3. Pydantic Model with Validation

```python
from pydantic import BaseModel, Field

class User(BaseModel):
    username: str = Field(min_length=3, max_length=20)
    age: int = Field(gt=17, lt=100)
    email: str = Field(pattern=r"^[\w\.-]+@[\w\.-]+\.\w+$")
```

Constraints:

- `min_length`, `max_length`
- `gt`, `ge`, `lt`, `le`
- regex validation

🔄 4. Default Values

```python
class Item(BaseModel):
    name: str
    price: float = 0.0
```

🧩 5. Nested Pydantic Models

```python
class Address(BaseModel):
    city: str
    zipcode: str

class User(BaseModel):
    name: str
    address: Address
```

Request JSON:

```json
{
  "name": "Atiar",
  "address": {
    "city": "Dhaka",
    "zipcode": "1207"
  }
}
```

📦 6. Response Model (Filtering Output)

Use `response_model` to control what returns to the client:

```python
class ItemOut(BaseModel):
    name: str
    price: float

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

👉 Extra fields will be removed automatically.

🚀 7. Model Config (Advanced)

(Useful for JSON convert, ORM mode, aliases)

```
class Product(BaseModel):
    name: str
    price: float

    class Config:
        orm_mode = True
```

📚 8. Using Pydantic Model in Query + Path + Body

```
@app.put("/items/{item_id}")
def update_item(item_id: int, item: Item, q: str | None = None):
    return {"id": item_id, "item": item, "query": q}
```

🌟 Summary: Why Pydantic Models?

✔ Structured data
✔ Automatic validation
✔ Auto-documentation
✔ Converts types
✔ Error-free API development
