🧠🚀 FastAPI Parameters & Request Body – Full Deep Dive

FastAPI makes parameter handling intuitive, type-safe, and well-documented. Let's break down everything!

🔹 1. Path & Query Parameters

✅ Path Parameters

Defined in the URL path like `/items/{item_id}`.

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

- `item_id` is **mandatory**
- Automatically converted to `int`

✅ Query Parameters

Passed via the URL like `/items?name=phone`.

```
@app.get("/items/")
def read_item(name: str, price: float = 0.0):
    return {"name": name, "price": price}
```

- **Optional** if default value is provided
- FastAPI infers types and validates them

🔹 2. Request Body 📄

Used to send JSON, Form data, etc.

✅ Example with Pydantic Model:

```
from pydantic import BaseModel

class Item(BaseModel):
    name: str
```

```
    price: float
    in_stock: bool

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

- Sends JSON like:

```
{
  "name": "Phone",
  "price": 299.99,
  "in_stock": true
}
```

- Automatic validation + documentation

🔹 3. Parameter Metadata 🏷️

Use `Query()`, `Path()`, and `Body()` to add **validation and metadata**.

```
from fastapi import Query, Path

@app.get("/products/{product_id}")
def get_product(
    product_id: int = Path(..., title="The ID of the product", gt=0),
    name: str = Query(None, max_length=50)
):
    return {"product_id": product_id, "name": name}
```

🚀 Common metadata options:

- `title`: Adds a title in Swagger UI
- `description`: Tooltip in docs
- `alias`: Alternate key
- `deprecated=True`: Warns in docs
- `example`: Shown as input sample

🔹 4. Validators 🛡️

Validation happens **automatically** via:

- Type hints (`str`, `int`)

- `Pydantic` models
- `Query`, `Path`, `Body` metadata

You can also **custom validate** inside models:

```
from pydantic import BaseModel, validator

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

    @validator("price")
    def price_positive(cls, value):
        if value <= 0:
            raise ValueError("Price must be positive")
        return value
```

🔹 5. Multiple Values 📚

Query parameters can accept multiple values (like a list):

```
from typing import List

@app.get("/search/")
def search_items(tags: List[str] = Query([])):
    return {"tags": tags}
```

👉 Accessed via `/search?tags=tech&tags=fastapi`

🔹 6. Number Validators 🔢

Use built-in constraints for validation:

```
@app.get("/range/")
def get_range(
    num: int = Query(..., gt=10, lt=100)
):
    return {"num": num}
```

🔢 Available:

- `gt` / `ge`: Greater than / Greater or equal

- `lt` / `le`: Less than / Less or equal
- `multiple_of`: Must be divisible

🔸 7. Complex Subtypes 🧬

Nested models, deeply structured request bodies:

```
class Features(BaseModel):
    size: str
    color: str

class Product(BaseModel):
    name: str
    price: float
    features: Features

@app.post("/products/")
def create_product(product: Product):
    return {"product": product}
```

✅ Swagger UI automatically nests the structure.

✅ Bonus: Mixing All Three – Path, Query, and Body

```
@app.put("/products/{product_id}")
def update_product(
    product_id: int = Path(...),
    available: bool = Query(...),
    product: Product = Body(...)
):
    return {
        "id": product_id,
        "available": available,
        "details": product
    }
```

🧠 Summary Table

Feature	Example	Tool Used
Path Param	`/items/{id}`	`Path()`
Query Param	`/items?name=x`	`Query()`

Feature	Example	Tool Used
Request Body	JSON payload	`Body()`, Model
Metadata	Titles, defaults, constraints	`Path()`, `Query()`
List Values	`/tags?tag=x&tag=y`	`List[str]`
Validators	price > 0, name length, etc.	`Pydantic`, `Query()`
Number Constraints	gt, lt, multiple_of	`Query()`, `Path()`
Nested Structures	Models inside Models	Pydantic

🔥 Pro Tips

- Use `Optional[]` or default `None` to make parameters optional.
- Use `Body(embed=True)` if you want request body as a single object like: `{ "item": { ... } }`
- Use `Enum` for fixed choices in query/path params.
- Use `response_model=YourModel` to enforce response types.

📘 FastAPI: Parameter Metadata Deep Dive

FastAPI allows you to **add metadata** to your endpoint parameters using special helper functions:

- `Query()` – for query parameters
- `Path()` – for path parameters
- `Body()` – for request bodies

This metadata:

- Adds **validation rules**
- Enhances the **Swagger docs**
- Helps set **default values**, **examples**, **constraints**, and more

---

1️⃣ Using `Query()` – For Query Parameters

```
from fastapi import Query

@app.get("/items/")
def read_items(
    q: str = Query(
        default="test",
        title="Query string",
        description="Search string for the items",
        min_length=3,
```

```
        max_length=50,
        example="mobile"
    )
):
    return {"q": q}
```

✅ What it does:

- `q` is a **query param**: `/items?q=mobile`
- Metadata like `title`, `description`, etc., appear in the **Swagger UI**
- Validates string length (`min_length`, `max_length`)
- Sets default to `"test"` if not provided

2️⃣ Using `Path()` – For Path Parameters

```
from fastapi import Path

@app.get("/products/{product_id}")
def get_product(
    product_id: int = Path(
        ...,  # Required
        title="Product ID",
        description="The ID of the product to retrieve",
        gt=0,  # must be > 0
        example=100
    )
):
    return {"product_id": product_id}
```

✅ Notes:

- `...` = required (no default)
- Adds **validation** (`gt=0`)
- `example` is shown in Swagger

3️⃣ Using `Body()` – For Request Body Parameters

```
from fastapi import Body
from pydantic import BaseModel

class Blog(BaseModel):
    title: str
    content: str
```

```python
@app.post("/blogs/")
def create_blog(
    blog: Blog = Body(
        ...,
        title="Blog Body",
        description="The body of the blog post",
        example={
            "title": "FastAPI Metadata",
            "content": "Understanding parameter documentation."
        }
    )
):
    return {"blog": blog}
```

✅ What's happening:

- `Body(...)` accepts a **Pydantic model**
- Adds a **rich example** directly in docs
- Helps front-end users understand the expected JSON structure

---

🧠 Default Values

Parameter Type	Syntax Example
Query	`q: str = Query("default")`
Path	`id: int = Path(..., gt=0)`
Body	`item: Item = Body(...)`

📊 All Common Metadata Options

Option	Description
`default`	Default value for optional params
`title`	Shows in Swagger field title
`description`	Tooltip in Swagger
`example`	Example input for that param
`min_length`	Min length for strings
`max_length`	Max length for strings
`gt`, `ge`	Greater than / Greater or equal (numbers)
`lt`, `le`	Less than / Less or equal (numbers)

Option	Description
`regex`	Regular expression pattern (for strings)
`deprecated`	Marks param as deprecated in Swagger UI

📌 Summary Example (All Together)

```python
@app.get("/search/{id}")
def search_item(
    id: int = Path(..., title="Item ID", gt=0, example=123),
    q: str = Query("default", min_length=3, max_length=20, example="phone"),
    page: int = Query(1, ge=1, le=100, description="Page number"),
):
    return {"id": id, "q": q, "page": page}
```

🎯 Swagger UI Output

- Clear field titles
- Default values pre-filled
- Example values shown for guidance
- Auto-validation if user enters bad input

✅ Summary

Import Used	When to Use	Use Case Example
`Query`	Query string parameters	`/items?q=shoes`
`Path`	URL parameters	`/items/45`
`Body`	JSON body (usually a Pydantic model)	`{ "title": "X" }`

---x

🧠 What Does This Mean?

Parameter Type	Syntax Example in FastAPI	Explanation
`Query`	`q: str = Query("default")`	A **query string parameter** (e.g., `/items?q=test`). `"default"` is the fallback value if the client doesn't pass it.
`Path`	`id: int = Path(..., gt=0)`	A **path parameter** (e.g., `/items/42`). `...` means **required**, `gt=0` means it must be greater than 0.

Parameter Type	Syntax Example in FastAPI	Explanation
`Body`	`post: PostModel = Body(...)`	A **JSON body payload** (used in POST/PUT), usually validated using a **Pydantic model**.

📦 FastAPI vs MERN Stack Parameter Concepts

Let's map these concepts to how you do things in **MERN (MongoDB, Express, React, Node.js)**:

Concept	FastAPI (Python)	MERN Stack (Node.js + Express)
Path Params	`@app.get("/items/{id}") id: int = Path(...)`	`router.get('/items/:id', (req, res) => { const id = req.params.id })`
Query Params	`q: str = Query("default")`	`req.query.q` → from URL like `/items?q=test`
Request Body (JSON)	`post: PostModel = Body(...)`	`req.body` → used with `body-parser` / `express.json()` middleware
Validation	`gt=0`, `min_length`, `regex`, via `Pydantic`	`express-validator`, `joi`, or manual `if (!req.body.title)`
Automatic Docs (OpenAPI)	Built-in with Swagger UI	Needs manual Swagger integration using `swagger-jsdoc`
Default Values	`Query("default")`, `Path(..., gt=0)`	Set in route logic: `req.query.q	'default'`
Type Safety	Fully typed via function signatures + Pydantic	Not typed unless you use **TypeScript** or **JSDoc annotations**

🔁 Example: Same Endpoint in Both

✅ FastAPI:

```
from fastapi import FastAPI, Query, Path
app = FastAPI()

@app.get("/items/{id}")
def get_item(
    id: int = Path(..., gt=0),
    search: str = Query("default")
```

```
):
    return {"id": id, "search": search}
```

✅ Express.js (MERN):

```
const express = require("express");
const router = express.Router();

router.get("/items/:id", (req, res) => {
    const id = parseInt(req.params.id);
    const search = req.query.search || "default";

    if (id <= 0) return res.status(400).json({ error: "Invalid ID" });

    res.json({ id, search });
});
```

🚀 Key Differences

Feature	FastAPI	MERN (Express)
Validation	Auto with types + constraints	Manual or with middleware
Docs	Auto-generated (Swagger UI)	Manual with Swagger setup
Type Checking	Native in Python	Use TypeScript or external tools
Default values & metadata	Built-in (`Query`, `Path`)	Manual using JS logic

---

🔎 Summary

FastAPI	Express (Node.js)
`Path(...)`	`req.params`
`Query(...)`	`req.query`
`Body(...)`	`req.body`
Pydantic Models	`Joi`, `Yup`, `zod`
Auto Docs	Swagger (built-in) `swagger-jsdoc`, `swagger-ui-express`

---

how to pass multiple values

✅ 1. Query Parameter (recap)

```
from typing import List, Optional
from fastapi import FastAPI, Query

app = FastAPI()

@app.get("/items/")
def get_items(v: Optional[List[str]] = Query(None)):
    return {"query_values": v}
```

Try URL:

```
http://localhost:8000/items/?v=1&v=2&v=3
```

✅ 2. Path Parameters with Multiple Values

🔴 **Path parameters do not natively support multiple values** like query params do (e.g., you can't do `/items/1,2,3` and get a `List[int]` directly).

📦 But you can **split the string manually**:

```
from fastapi import FastAPI
from typing import List

app = FastAPI()

@app.get("/items/{values}")
def get_items(values: str):
    # Split by comma
    value_list = values.split(",")
    return {"path_values": value_list}
```

Try URL:

```
http://localhost:8000/items/10,20,30
```

📤 Output:

```
{
  "path_values": ["10", "20", "30"]
}
```

🔧 You can convert to `int` if needed:

```
value_list = [int(i) for i in values.split(",")]
```

✅ 3. **Body Parameters with Multiple Values**

For **POST** requests, you can send multiple values in the **request body** (JSON), like this:

🧩 Example Request Body:

```
{
  "values": [1, 2, 3]
}
```

✅ Code:

```
from fastapi import FastAPI
from pydantic import BaseModel
from typing import List

app = FastAPI()

class Item(BaseModel):
    values: List[int]

@app.post("/items/")
def post_items(item: Item):
    return {"body_values": item.values}
```

POST to: `/items/` With body:

```
{
  "values": [1, 2, 3]
}
```

📤 Response:

```
{
  "body_values": [1, 2, 3]
}
```

🔄 Summary Table

Method	Input Type	FastAPI Handling	URL/Example
Query	List[str]	`Query(None)` or `Query(...)`	`/items?v=1&v=2&v=3`
Path	str (parsed)	`values: str` → `split(",")`	`/items/1,2,3`
Body (POST)	List[int]	Use `Pydantic` model with `List[...]`	`{ "values": [1, 2, 3] }`

🧩 Scenario: Combine All Three

You want an endpoint like this:

```
POST /items/123?v=10&v=20
```

With a JSON body:

```
{
  "tags": ["electronics", "gadget"]
}
```

✅ Goal:

- `123` is a **Path parameter** (item ID)
- `v=10&v=20` are **Query parameters** (version list)
- `tags` is a **Body parameter** (list of tags)

✅ Full FastAPI Example

```
from fastapi import FastAPI, Query, Path
from pydantic import BaseModel
from typing import List, Optional
```

```python
app = FastAPI()

# Body Model
class ItemData(BaseModel):
    tags: List[str]

@app.post("/items/{item_id}")
def combined_params(
    item_id: int = Path(..., description="Item ID from path"),
    versions: Optional[List[int]] = Query(None, description="List of version
numbers"),
    item: ItemData = ...
):
    return {
        "item_id": item_id,
        "versions": versions,
        "tags": item.tags
    }
```

🧪 Example Request

URL:

```
POST http://localhost:8000/items/123?v=10&v=20
```

Body:

```
{
  "tags": ["electronics", "gadget"]
}
```

✅ Response

```
{
  "item_id": 123,
  "versions": [10, 20],
  "tags": ["electronics", "gadget"]
}
```

🔍 Breakdown:

Parameter Type	Variable	Type	Description
Path	`item_id`	`int`	Extracted from `/items/{item_id}`
Query	`versions`	`List[int]`	Supports multiple: `v=1&v=2&v=3`
Body (JSON)	`item`	`ItemData` model	Parsed from the request body (POST)

🛠 Bonus Tip – Make Body Optional Too:

If body is optional, modify:

```
item: Optional[ItemData] = None
```

Then access:

```
tags = item.tags if item else []
```
