FastAPI

Path & Query Parameters

Advanced API Parameter Handling

A Comprehensive Guide to Dynamic URL Routing,
Data Validation, and Parameter-Based Operations

Sujil S

sujil9480@gmail.com

December 25, 2025

Contents

1 Introduction and Session Overview

1.1 What We'll Learn Today

In this session, we continue building our Patient Management System API. Today's focus is on two extremely important concepts in FastAPI:

1. **Path Parameters**: Dynamic URL segments for identifying specific resources
2. **Query Parameters**: Optional URL parameters for filtering and sorting data

Session Goals

By the end of this session, you will:

- Understand what path parameters are and when to use them
- Implement endpoints with dynamic URL routing
- Learn HTTP status codes and their meanings
- Master query parameters for advanced data operations
- Build sorting and filtering functionality
- Enhance API documentation with parameter metadata

1.2 Recap: Where We Left Off

In the previous session, we built a basic Patient Management API with the following endpoints:

Endpoint	Method	Purpose
/	GET	Home/Welcome message
/about	GET	API description
/view	GET	View all patient records

Current Limitation:

The `/view` endpoint returns ALL patient data. We have no way to:

- View a specific patient's record
- Filter or sort the patient list
- Perform targeted operations

Solution: We'll add dynamic parameters to overcome these limitations!

2 Understanding Path Parameters

2.1 What Are Path Parameters?

Formal Definition

Path parameters are dynamic segments of a URL path used to identify a specific resource.

2.2 Simple Explanation with Example

Let's understand this concept through our Patient Management System.

2.2.1 Current Situation

From the previous session, we have this endpoint:

```
http://localhost:8000/view
```

When you hit this URL, it returns ALL patient records at once.

2.2.2 The New Requirement

Question: What if we want to view data for a SPECIFIC patient instead of all patients?

For example:

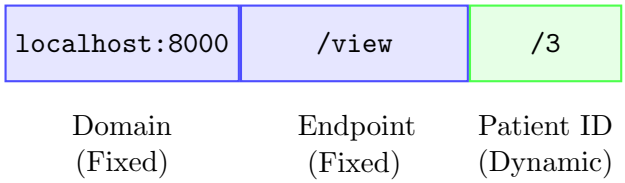
- We have 5 patients in our database
- We want to see only the 3rd patient's data
- We don't care about the others

2.2.3 The Solution: Dynamic URL

We can modify our URL to include a dynamic part:

```
http://localhost:8000/view/3
```

Breaking down this URL:



- `localhost:8000` - **Cannot change** (domain)
- `/view` - **Cannot change** (endpoint path)
- `/3` - **Can change** (dynamic segment)

2.3 The Dynamic Portion

Important Note

The `/3` part is dynamic! Different clients can request different patient IDs:

- Client A requests: `/view/3`
- Client B requests: `/view/4`
- Client C requests: `/view/5`

This dynamic portion is called a **Path Parameter**.

2.4 Purpose of Path Parameters

Path parameters serve ONE primary purpose:

Core Purpose

To locate and identify a SPECIFIC resource on the server

In our example:

- The number `3` identifies which patient record to retrieve
- The server uses this to find that specific patient
- Only that patient's data is returned

2.5 Common Use Cases for Path Parameters

Path parameters are extremely useful for several operations:

2.5.1 1. Retrieve Operations

Viewing Specific Resources

Social Media Profile:

`GET /users/12345`

Retrieves profile of user with ID 12345 from all users in the database.

2.5.2 2. Update Operations

Modifying Specific Resources

Updating Patient Weight:

`PUT /patient/P003`

Updates the record of patient P003 specifically, not all patients.

2.5.3 3. Delete Operations

Removing Specific Resources

Deleting an Order:

```
DELETE /orders/9876
```

Deletes only order 9876, leaving other orders untouched.

Important Note

Key Observation:

Path parameters are used extensively in three CRUD operations:

- **Retrieve**: Get specific resource
- **Update**: Modify specific resource
- **Delete**: Remove specific resource

They help you work with ONE specific resource among many!

3 Implementing Path Parameters in FastAPI

3.1 Building the Endpoint

Now let's create an endpoint that accepts a path parameter to view a specific patient.

3.1.1 Step 1: Define the Route

```
@app.get('/patient/{patient_id}')
```

Listing 1: Defining Route with Path Parameter

Breaking down the route:

- `/patient` - Fixed path segment
- {`patient_id`} - Dynamic path parameter
- Curly braces {} indicate this is a variable

Important Note

The syntax {`patient_id`} defines a path parameter. The name inside the braces becomes a variable that your function can access.

3.1.2 Step 2: Create Handler Function

```
def view_patient(patient_id: str):
```

Important points:

- Function parameter name MUST match the route parameter name
- We specify the data type (`str` in this case)
- FastAPI automatically extracts the value from URL and passes it here

Why String Type?

Question: Why is `patient_id` a string?

Answer: Looking at our `patients.json`:

```
{
  "P001": {...},
  "P002": {...},
  "P003": {...}
}
```

Our patient IDs are formatted as "P001", "P002", etc. - these are strings, not integers!

3.1.3 Step 3: Implement Function Logic

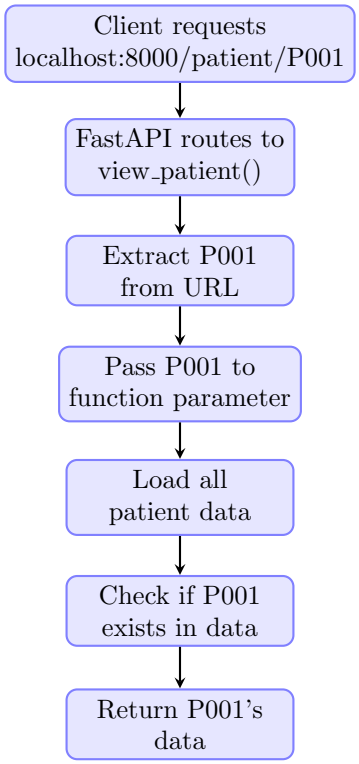
```python
@app.get('/patient/{patient_id}')
def view_patient(patient_id: str):
    # Load all patient data
    data = load_data()

    # Check if patient ID exists
    if patient_id in data:
        # Return specific patient data
        return data[patient_id]

    # Return error if patient not found
    return {'error': 'patient not found'}
```

Listing 2: Complete View Patient Function

3.2 Understanding the Flow



3.3 Testing the Endpoint

3.3.1 Method 1: Browser

Simply navigate to:

```
http://localhost:8000/patient/P001
http://localhost:8000/patient/P003
http://localhost:8000/patient/P005
```

Each URL will return that specific patient's data!

3.3.2 Method 2: Interactive Documentation

Navigate to `http://localhost:8000/docs`

1. Find the `GET /patient/{patient_id}` endpoint
2. Click to expand it
3. Notice it shows "patient_id" as a **path parameter** (required)
4. Click "Try it out"
5. Enter "P001" in the patient_id field
6. Click "Execute"
7. View the response below

Response Example

Request: `GET /patient/P001`

Response:

```
{
  "name": "Ananya Verma",
  "city": "Guwahati",
  "age": 28,
  "gender": "female",
  "height": 1.65,
  "weight": 90.0,
  "bmi": 33.06,
  "verdict": "Obese"
}
```

4 Improving Code with the Path() Function

4.1 Current Problem with Documentation

If you check the auto-generated documentation, you'll notice:

- It shows "patient_id" is a path parameter
- It shows it's required
- But NO additional information is provided

Why is this a problem?

- Clients don't know what format the patient ID should be
- No examples are provided
- No description explains what this parameter does
- Makes API harder to use

4.2 The Path() Function

What is Path()?

The Path() function in FastAPI is used to provide metadata, validation rules, and documentation hints for path parameters in your API endpoints.

4.3 Capabilities of Path()

The Path() function allows you to:

Feature	Description
Title	Add a title for the parameter
Description	Provide detailed explanation
Example	Show sample values
ge, gt, le, lt	Greater/less than validation (for numbers)
min_length	Minimum string length
max_length	Maximum string length
regex	Pattern matching validation

4.4 Implementing Path() Function

4.4.1 Step 1: Import Path

```
1 from fastapi import FastAPI, Path
```

4.4.2 Step 2: Use Path() in Function

```python
@app.get('/patient/{patient_id}')
def view_patient(
    patient_id: str = Path(
        ...,
        description="The ID of the patient in DB",
        example='P001'
    )
):
    data = load_data()

    if patient_id in data:
        return data[patient_id]

    return {'error': 'patient not found'}
```

Listing 3: Enhanced Path Parameter with Path()

4.5 Understanding the Path() Parameters

4.5.1 The Ellipsis (...)

```python
patient_id: str = Path(...)
```

What does ... mean?

The three dots `...` (ellipsis) indicate that this parameter is **required**.

Note: All path parameters are required by default, but using `...` is considered best practice for clarity.

4.5.2 Description

```python
description="The ID of the patient in DB"
```

This text appears in the documentation, explaining what the parameter expects.

4.5.3 Example

```python
example='P001'
```

Provides a sample value that clients can see and use as reference.

4.6 Additional Validation Options

While not needed in our case, here are other validation options:

```python
# For numeric IDs
patient_id: int = Path(
    ...,
    ge=0,         # Greater than or equal to 0
    le=1000,      # Less than or equal to 1000
    description="Patient ID must be between 0 and 1000"
)

```

```python
# For string length validation
patient_name: str = Path(
    ...,
    min_length=2,
    max_length=50,
    description="Patient name (2-50 characters)"
)

# For pattern matching
patient_code: str = Path(
    ...,
    regex="^P[0-9]{3}$",  # Must match P followed by 3 digits
    description="Patient code in format P001, P002, etc."
)
```

Listing 4: Validation Examples

4.7 Benefits After Adding Path()

4.7.1 Improved Documentation

Navigate to `http://localhost:8000/docs` again:

Enhanced Documentation

Before Path():

- Shows: "patient_id" (required)
- No other information

After Path():

- Shows: "patient_id" (required)
- Description: "The ID of the patient in DB"
- Example: "P001"
- Much clearer for API consumers!

Important Note

Key Takeaway:

Using the Path() function helps you create professional, well-documented APIs that are easy for clients to understand and use. It's a best practice in API development!

5 HTTP Status Codes: A Critical Concept

Before we improve our code further, we need to understand an important concept: HTTP Status Codes.

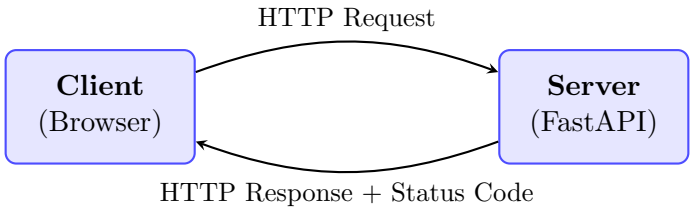
5.1 What Are HTTP Status Codes?

Definition

HTTP status codes are three-digit numbers returned by a web server to indicate the result of a client's request.

5.2 Understanding the Request-Response Flow

Let's revisit how client-server communication works:



The Communication Protocol:

1. Client sends HTTP request to server
2. Server processes the request
3. Server prepares HTTP response
4. **Server adds HTTP status code to response**
5. Response sent back to client

Important Note

Important: Every HTTP response includes a status code. This is a fundamental part of the HTTP protocol!

5.3 Purpose of Status Codes

Status codes tell the client:

- Was the request successful?
- If something went wrong, what type of error occurred?
- Does the client need to take any further action?

5.4 Categories of Status Codes

Status codes are grouped into 5 categories based on their first digit:

Range	Category	Meaning
2xx	Success	Request was successful
3xx	Redirection	Further action needed
4xx	Client Error	Error on client's side
5xx	Server Error	Error on server's side

Quick Recognition

If you see a status code starting with:

- **2**: Everything went well ✓
- **3**: Redirect or additional action needed
- **4**: You (client) did something wrong
- **5**: Server has a problem

5.5 Common HTTP Status Codes

5.5.1 2xx - Success Codes

Code	Name	Meaning
200	OK	Request succeeded, response returned properly
201	Created	Resource successfully created on server
204	No Content	Success, but no data to return

200 OK - Most Common Success Code

When you successfully retrieve data from an API:

```
GET /patient/P001

Response:
Status: 200 OK
Body: { "name": "Ananya Verma", "age": 28, ... }
```

The 200 code tells you everything worked perfectly!

201 Created - For POST Requests

When you create a new patient record:

```
POST /patient

Response:
Status: 201 Created
Body: { "id": "P006", "message": "Patient created" }
```

The 201 specifically indicates successful creation.

204 No Content - For DELETE Requests

When you delete a patient:

```
DELETE /patient/P001

Response:
Status: 204 No Content
Body: (empty)
```

Success, but nothing to return since data was deleted.

5.5.2 4xx - Client Error Codes

Code	Name	Meaning
400	Bad Request	Invalid request data, wrong format
401	Unauthorized	Authentication required
403	Forbidden	Logged in but not allowed
404	Not Found	Resource doesn't exist

400 Bad Request

Client sent wrong data format:

```
POST /patient
Body: { "age": "twenty-five" }  // Should be number, not string

Response:
Status: 400 Bad Request
Body: { "error": "Age must be a number" }
```

404 Not Found - Very Common!

Client requested something that doesn't exist:

```
GET /patient/P999  // This patient doesn't exist

Response:
Status: 404 Not Found
Body: { "error": "Patient not found" }
```

This is the famous "404 error" you've probably seen on websites!

5.5.3 5xx - Server Error Codes

Code	Name	Meaning
500	Internal Server Error	Something went wrong on server
502	Bad Gateway	Communication broken between servers
503	Service Unavailable	Server down or overloaded

5.6 Seeing Status Codes in Action

Let's check our current endpoint in the documentation:

1. Go to `http://localhost:8000/docs`
2. Find `GET /patient/{patient_id}`
3. Try it out with "P001"
4. Execute
5. Look at the response

Successful Request

Request: `GET /patient/P001`

Response:

```
Status Code: 200
Body: {
  "name": "Ananya Verma",
  "city": "Guwahati",
  ...
}
```

The 200 status code indicates success!

6 Identifying the Problem: Incorrect Status Codes

6.1 Testing with Non-Existent Patient

Let's test what happens when we request a patient that doesn't exist:

1. Go to `http://localhost:8000/docs`
2. Open `GET /patient/{patient_id}`
3. Try it out with "P007" (doesn't exist in our database)
4. Execute
5. Observe the response

6.2 The Problem Revealed

Warning

What we see:

```
Status Code: 200 OK

Response Body:
{
  "error": "patient not found"
}
```

The Problem: Status code is 200 (Success), but we're returning an error!

Why this is wrong:

- 200 means "request was successful"
- But the patient was NOT found
- This is misleading for API consumers
- Should be 404 (Not Found) instead

6.3 Why Status Codes Matter

- API clients (apps, websites) rely on status codes
- They don't always read the response body
- Automated systems check status codes first
- 200 tells them "everything's fine" even when it's not
- Can cause bugs in client applications

Real-World Impact

Imagine a mobile app using our API:

```
1 # App code checking our API
```

```
response = fetch_patient("P999")

if response.status_code == 200:
    # App thinks everything is fine
    display_patient_info(response.data)
    # But data contains error message!
    # App might crash or show wrong info
```

Proper status codes prevent such issues!

6.4 The Solution: HTTPException

FastAPI provides a special class to handle errors properly: `HTTPException`

What is HTTPException?

HTTPException is a special built-in exception in FastAPI used to return custom HTTP error responses when something goes wrong in your API.

Benefits:

- Gracefully raise errors with proper status codes
- Include custom error messages
- Automatic error response formatting
- Better than returning error dictionaries

6.5 Implementing HTTPException

6.5.1 Step 1: Import HTTPException

```
from fastapi import FastAPI, Path, HTTPException
```

6.5.2 Step 2: Raise Exception Instead of Returning Error

```
@app.get('/patient/{patient_id}')
def view_patient(
    patient_id: str = Path(
        ...,
        description="The ID of the patient in DB",
        example='P001'
    )
):
    data = load_data()

    if patient_id in data:
        return data[patient_id]

    # Instead of: return {'error': 'patient not found'}
    # We raise an HTTPException
    raise HTTPException(
        status_code=404,
        detail="Patient not found"
```

```
    )
```

Listing 5: Improved Code with HTTPException

6.6 Understanding the Parameters

Parameter	Description
status_code	The HTTP status code to return (404, 400, 500, etc.)
detail	The error message to include in response

6.7 Testing the Improved Code

6.7.1 Test 1: Valid Patient

```
GET /patient/P001

Response:
Status Code: 200 OK
Body: { "name": "Ananya Verma", ... }
```

✓ Works as expected!

6.7.2 Test 2: Invalid Patient

```
GET /patient/P007

Response:
Status Code: 404 Not Found
Body: { "detail": "Patient not found" }
```

✓ Now returns the correct 404 status code!

Important Note

Going Forward:

From now on in this project, whenever something goes wrong, we'll use `HTTPException` with proper status codes instead of returning error dictionaries.

This is a professional API development practice!

6.8 Complete Updated Code

```python
from fastapi import FastAPI, Path, HTTPException
import json

app = FastAPI()

def load_data():
    with open("patients.json", "r") as f:
        data = json.load(f)
    return data

@app.get("/")
```

```python
def hello():
    return {"message": "Patient Management System API"}

@app.get("/about")
def about():
    return {"message": "A fully functional API for Patient Management"}

@app.get("/view")
def view():
    data = load_data()
    return data

@app.get('/patient/{patient_id}')
def view_patient(
    patient_id: str = Path(
        ...,
        description="The ID of the patient in DB",
        example='P001'
    )
):
    data = load_data()

    if patient_id in data:
        return data[patient_id]

    raise HTTPException(
        status_code=404,
        detail="Patient not found"
    )
```

Listing 6: Complete Code with All Improvements

7 Understanding Query Parameters

7.1 The New Requirement

Let's introduce a new business requirement for our API.

7.1.1 Current Situation

We have the `/view` endpoint that returns ALL patients:

```
GET /view

Returns: All 5 patients in chronological order
```

7.1.2 New Feature Request

Requirement: Give users the option to view patients in SORTED order!

Specifically, allow sorting by:

- Weight (ascending or descending)
- Height (ascending or descending)
- BMI (ascending or descending)

Important: This should be OPTIONAL!

- If user doesn't specify sorting → show data normally
- If user specifies sorting → show sorted data

7.2 Why Not Path Parameters?

Warning

Question: Can we use path parameters for this?

Answer: NO! Here's why:

- Path parameters are REQUIRED
- Sorting is OPTIONAL
- Path parameters identify specific resources
- Sorting is a way to FILTER existing resources
- Path parameters change the endpoint structure
- We want to keep the same endpoint

We need something that's optional and doesn't change the endpoint → **Query Parameters**!

7.3 What Are Query Parameters?

Formal Definition

Query parameters are optional key-value pairs appended to the end of a URL, used to pass additional data to the server in an HTTP request.

They are typically employed for operations like:

- Filtering
- Sorting
- Searching
- Pagination

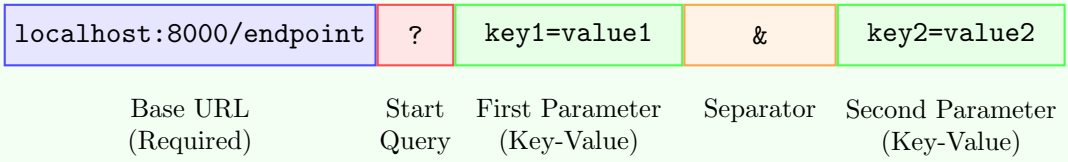
WITHOUT altering the endpoint path itself!

7.4 Query Parameter Syntax

URL Structure with Query Parameters

`http://localhost:8000/endpoint?key1=value1&key2=value2`

Breaking it down:



Key Elements:

1. **? (Question Mark)**: Starts the query parameters section
2. **key=value**: Each parameter as key-value pair
3. **& (Ampersand)**: Separates multiple parameters

7.5 Query Parameters vs Path Parameters

Aspect	Path Parameters	Query Parameters
Syntax	`/endpoint/{value}`	`/endpoint?key=value`
Required	Yes (always)	No (optional)
Purpose	Identify resource	Filter/modify results
Use case	Get specific item	Sort/filter items
Multiple values	Need multiple paths	Use & separator
URL clarity	Part of path	After ? mark

7.6 Real-World Examples

7.6.1 Example 1: E-commerce Search

Product Search with Filters

`https://shop.com/products?category=electronics&price_max=500&sort=rating`

Query Parameters:

- `category=electronics`: Filter by category
- `price_max=500`: Maximum price filter
- `sort=rating`: Sort by rating

All these are optional modifications to the /products endpoint!

7.6.2 Example 2: Social Media Feed

Paginated Feed

`https://social.com/feed?page=2&limit=20&filter=friends`

Query Parameters:

- `page=2`: Show page 2
- `limit=20`: Show 20 items per page
- `filter=friends`: Show only friends' posts

7.7 Our Use Case: Patient Sorting

For our Patient Management System, we'll use query parameters to enable sorting:

Patient Sorting URLs

Sort by BMI in descending order:

`http://localhost:8000/sort?sort_by=bmi&order=desc`

Sort by height in ascending order:

`http://localhost:8000/sort?sort_by=height&order=asc`

Just sort by weight (ascending by default):

`http://localhost:8000/sort?sort_by=weight`

Query Parameters:

- `sort_by`: Which field to sort on (weight/height/bmi)
- `order`: Sorting order (asc/desc) - OPTIONAL

8 Implementing Query Parameters in FastAPI

8.1 The Query() Function

Just like we have `Path()` for path parameters, FastAPI provides `Query()` for query parameters.

What is Query()?

Query() is a utility function provided by FastAPI to declare, validate, and document query parameters in your API endpoints.

Capabilities:

- Set default values
- Add validation rules (gt, lt, min_length, max_length)
- Provide metadata (title, description, examples)
- Make parameters optional or required

8.2 Creating the Sort Endpoint

8.2.1 Step 1: Import Query

```
from fastapi import FastAPI, Path, HTTPException, Query
```

8.2.2 Step 2: Define the Endpoint

```
@app.get('/sort')
def sort_patients(
    sort_by: str = Query(
        ...,
        description='Sort on the basis of height, weight or BMI'
    ),
    order: str = Query(
        'asc',
        description="Sort in ascending or descending order"
    )
):
    # Implementation will follow
    pass
```

Listing 7: Sort Endpoint with Query Parameters

8.3 Understanding the Parameters

8.3.1 Required Query Parameter

```
sort_by: str = Query(...)
```

- `...` (ellipsis) makes it REQUIRED
- User MUST provide this parameter
- If missing, FastAPI returns 422 error

8.3.2 Optional Query Parameter with Default

```
order: str = Query('asc', ...)
```

- `'asc'` is the default value
- If user doesn't provide `order`, it defaults to 'asc'
- Makes this parameter OPTIONAL

Important Note

Key Difference:

- `Query(...)`: Parameter is required
- `Query('default_value')`: Parameter is optional

This is how you control whether query parameters are mandatory or not!

8.4 Implementing the Sorting Logic

8.4.1 Step 1: Define Valid Fields

```
def sort_patients(sort_by: str = Query(...), order: str = Query('asc
    ')):
    # Define which fields can be sorted
    valid_fields = ['height', 'weight', 'bmi']

    # Validation will follow
```

8.4.2 Step 2: Validate sort_by Parameter

```
    # Check if sort_by is valid
    if sort_by not in valid_fields:
        raise HTTPException(
            status_code=400,
            detail=f'Invalid field, select from {valid_fields}'
        )
```

Why 400?

- 400 = Bad Request
- Client sent invalid data
- Client chose field that doesn't exist
- This is a client-side error

8.4.3 Step 3: Validate order Parameter

```
    # Check if order is valid
    if order not in ['asc', 'desc']:
        raise HTTPException(
            status_code=400,
            detail='Invalid order, select from asc or desc'
        )
```

8.4.4 Step 4: Load and Sort Data

```
    # Load all patient data
    data = load_data()

    # Determine sort order (True = descending, False = ascending)
    sort_order = True if order == 'desc' else False

    # Sort the data
    sorted_data = sorted(
        data.values(),
        key=lambda x: x[sort_by],
        reverse=sort_order
    )

    # Return sorted data
    return sorted_data
```

8.5 Understanding the Sorting Code

8.5.1 Extracting Values

```
data.values()
```

- `data` is a dictionary `{"P001": {...}, "P002": {...}}`
- `.values()` extracts just the patient records
- Returns a list we can sort

8.5.2 Lambda Function for Sorting Key

```
key=lambda x: x[sort_by]
```

- Lambda creates anonymous function
- `x` represents each patient dictionary
- `x[sort_by]` extracts the sorting field
- If `sort_by='bmi'`, it extracts BMI value

8.5.3 Reverse Parameter

```
reverse=sort_order
```

- `reverse=False`: Ascending order (default)
- `reverse=True`: Descending order

8.6 Complete Sort Function

```python
@app.get('/sort')
def sort_patients(
    sort_by: str = Query(
        ...,
        description='Sort on the basis of height, weight or BMI'
    ),
    order: str = Query(
        'asc',
        description="Sort in ascending or descending order"
    )
):
    # Define valid sorting fields
    valid_fields = ['height', 'weight', 'bmi']

    # Validate sort_by parameter
    if sort_by not in valid_fields:
        raise HTTPException(
            status_code=400,
            detail=f'Invalid field, select from {valid_fields}'
        )

    # Validate order parameter
    if order not in ['asc', 'desc']:
        raise HTTPException(
            status_code=400,
            detail='Invalid order, select from asc or desc'
        )

    # Load patient data
    data = load_data()

    # Determine sort order
    sort_order = True if order == 'desc' else False

    # Sort the data
    sorted_data = sorted(
        data.values(),
        key=lambda x: x[sort_by],
        reverse=sort_order
    )

    # Return sorted data
    return sorted_data
```

Listing 8: Complete Sorting Implementation

9 Testing the Sort Endpoint

9.1 Method 1: Direct URL Testing

9.1.1 Test 1: Sort by BMI in Descending Order

```
http://localhost:8000/sort?sort_by=bmi&order=desc
```

Expected Result: Patients sorted from highest BMI to lowest

```
[
  {
    "name": "Ananya Verma",
    "bmi": 33.06,    // Highest BMI first
    ...
  },
  {
    "name": "Neha Sinha",
    "bmi": 31.22,
    ...
  },
  {
    "name": "Ravi Mehta",
    "bmi": 27.76,
    ...
  },
  // ... continues in descending order
]
```

9.1.2 Test 2: Sort by Height in Ascending Order

```
http://localhost:8000/sort?sort_by=height&order=asc
```

Expected Result: Patients sorted from shortest to tallest

9.1.3 Test 3: Sort by Weight (Default Ascending)

```
http://localhost:8000/sort?sort_by=weight
```

Note: We didn't provide `order` parameter!

Result: Since `order` has default value 'asc', it sorts in ascending order automatically.

9.2 Method 2: Interactive Documentation

9.2.1 Testing Valid Request

1. Navigate to `http://localhost:8000/docs`
2. Find `GET /sort` endpoint
3. Click to expand
4. Click "Try it out"
5. Notice TWO query parameters:

- `sort_by`: Shows as required
- `order`: Shows with default value 'asc'

6. Enter "bmi" for sort_by
7. Enter "desc" for order
8. Click "Execute"
9. View response below

Successful Response

Status Code: 200 OK

Response Body: Array of patient objects sorted by BMI in descending order

9.2.2 Testing Validation Errors

Test Invalid Field:

1. Try it out again
2. Enter "age" for sort_by (not a valid field!)
3. Enter "asc" for order
4. Execute

Validation Error Response

Status Code: 400 Bad Request

Response Body:

```
{
  "detail": "Invalid field, select from ['height', 'weight', 'bmi']"
}
```

✓ Our validation is working!

Test Invalid Order:

1. Enter "bmi" for sort_by
2. Enter "ascending" for order (should be 'asc' or 'desc'!)
3. Execute

Order Validation Error

Status Code: 400 Bad Request

Response Body:

```
{
  "detail": "Invalid order, select from asc or desc"
}
```

✓ Validation catching invalid order values!

9.3 Verifying Optional Parameters

Test Without Order Parameter:

1. Try it out
2. Enter "weight" for sort_by
3. Leave order EMPTY or with default value
4. Execute

Result: Still works! Defaults to ascending order.

Important Note

This demonstrates the power of optional query parameters:

- Required parameters MUST be provided
- Optional parameters use defaults if not provided
- API remains flexible yet predictable

10 Complete Project Code

10.1 Final main.py

```python
from fastapi import FastAPI, Path, HTTPException, Query
import json

app = FastAPI()

def load_data():
    """Load patient data from JSON file"""
    with open("patients.json", "r") as f:
        data = json.load(f)
    return data

@app.get("/")
def hello():
    """Home endpoint"""
    return {"message": "Patient Management System API"}

@app.get("/about")
def about():
    """About endpoint"""
    return {
        "message": "A fully functional API for Patient Management System"
    }

@app.get("/view")
def view():
    """View all patients endpoint"""
    data = load_data()
    return data

@app.get('/patient/{patient_id}')
def view_patient(
    patient_id: str = Path(
        ...,
        description="The ID of the patient in DB",
        example='P001'
    )
):
    """
    View specific patient by ID

    Path Parameter:
        patient_id: Unique identifier for the patient

    Returns:
        Patient data if found

    Raises:
        404: Patient not found
    """
    data = load_data()

    if patient_id in data:
        return data[patient_id]
```

```python

    raise HTTPException(
        status_code=404,
        detail="Patient not found"
    )

@app.get('/sort')
def sort_patients(
    sort_by: str = Query(
        ...,
        description='Sort on the basis of height, weight or BMI'
    ),
    order: str = Query(
        'asc',
        description="Sort in ascending or descending order"
    )
):
    """
    Sort patients by specified field

    Query Parameters:
        sort_by: Field to sort on (height/weight/bmi) - REQUIRED
        order: Sort order (asc/desc) - OPTIONAL, defaults to 'asc'

    Returns:
        List of patients sorted by specified criteria

    Raises:
        400: Invalid field or order value
    """
    # Define valid sorting fields
    valid_fields = ['height', 'weight', 'bmi']

    # Validate sort_by parameter
    if sort_by not in valid_fields:
        raise HTTPException(
            status_code=400,
            detail=f'Invalid field, select from {valid_fields}'
        )

    # Validate order parameter
    if order not in ['asc', 'desc']:
        raise HTTPException(
            status_code=400,
            detail='Invalid order, select from asc or desc'
        )

    # Load patient data
    data = load_data()

    # Determine sort order
    sort_order = True if order == 'desc' else False

    # Sort the data
    sorted_data = sorted(
        data.values(),
        key=lambda x: x[sort_by],
        reverse=sort_order
```

```
    )

    # Return sorted data
    return sorted_data
```

Listing 9: Complete Patient Management API with Parameters

10.2 API Endpoints Summary

Endpoint	Method	Purpose
/	GET	Welcome message
/about	GET	API description
/view	GET	Get all patients
/patient/{id}	GET	Get specific patient (path param)
/sort	GET	Get sorted patients (query params)

10.3 Project Structure

```
patient-management-api/
+-- env/                    # Virtual environment
+-- main.py                 # FastAPI application
+-- patients.json           # Patient database
+-- requirements.txt        # Python dependencies
```

11 Key Concepts Summary

11.1 Path Parameters Recap

Path Parameters - Quick Reference

Definition: Dynamic segments of URL path to identify specific resources

Syntax: `/endpoint/{parameter_name}`

Characteristics:

- Always REQUIRED
- Part of the URL path
- Used to identify specific resources
- Common in Retrieve, Update, Delete operations

Example:

```
/patient/P001  ← P001 is path parameter
```

11.2 Query Parameters Recap

Query Parameters - Quick Reference

Definition: Optional key-value pairs at end of URL for filtering/modifying

Syntax: `/endpoint?key1=value1&key2=value2`

Characteristics:

- Can be OPTIONAL or REQUIRED
- After ? in URL
- Used for filtering, sorting, pagination
- Don't change endpoint path

Example:

```
/sort?sort_by=bmi&order=desc  ← Query parameters
```

11.3 Path() vs Query() Functions

Feature	Path()	Query()
Used for	Path parameters	Query parameters
Required by default	Yes (always)	No (can set default)
Default values	Not applicable	Supported
Validation	Supported	Supported
Documentation	Enhanced	Enhanced

11.4 HTTP Status Codes Reference

Code	Name	When to Use
200	OK	Successful GET/PUT/PATCH
201	Created	Successful POST (creation)
204	No Content	Successful DELETE
400	Bad Request	Invalid client data
401	Unauthorized	Authentication needed
403	Forbidden	Not allowed (even if authenticated)
404	Not Found	Resource doesn't exist
500	Server Error	Server-side problem

11.5 When to Use Each Parameter Type

Decision Guide

Use Path Parameters When:

- Identifying a SPECIFIC resource
- Parameter is REQUIRED
- Working with single resource (retrieve/update/delete)
- Example: `/users/123`, `/orders/456`

Use Query Parameters When:

- Filtering MULTIPLE resources
- Parameter is OPTIONAL
- Sorting, searching, or paginating
- Example: `/products?category=books&sort=price`

12 Combining Both Parameter Types

Important Note

Can you use BOTH Path and Query Parameters in the same endpoint?

Answer: YES — this is not only valid, it is extremely common in real-world APIs.

Example Request:

```
GET /users/{user_id}/posts?status=published&limit=10
```

- **Path parameter**: `user_id` — identifies *which user*
- **Query parameters**: `status`, `limit` — filter that user's posts

This pattern is widely used in REST APIs such as social media feeds, e-commerce orders, and dashboards.

```python
from fastapi import FastAPI, Path, Query, HTTPException

app = FastAPI()

# Dummy database
posts_db = {
    "u101": [
        {"id": 1, "title": "FastAPI Basics", "status": "published"},
        {"id": 2, "title": "Draft Post", "status": "draft"},
        {"id": 3, "title": "Advanced FastAPI", "status": "published"
    }
    ]
}

@app.get("/users/{user_id}/posts")
def get_user_posts(
    user_id: str = Path(
        ...,
        description="Unique ID of the user",
        example="u101"
    ),
    status: str = Query(
        None,
        description="Filter posts by status (published or draft)",
        example="published"
    ),
    limit: int = Query(
        10,
        ge=1,
        le=50,
        description="Maximum number of posts to return",
        example=10
    )
):
    if user_id not in posts_db:
        raise HTTPException(
            status_code=404,
            detail="User not found"
        )
```

```
39
40      posts = posts_db[user_id]
41
42      if status:
43          posts = [post for post in posts if post["status"] == status]
44
45      return posts[:limit]
```

Listing 10: Using Path and Query Parameters Together

How FastAPI Interprets This Endpoint

- `user_id` is extracted from the URL path
- `status` and `limit` are extracted from the query string
- Path parameters identify the resource
- Query parameters modify or filter the response

This separation keeps URLs clean, readable, and scalable.

Important Note

Golden Rule:

- Path parameters decide *WHAT* resource you want.
- Query parameters decide *HOW* you want that resource.

13 Best Practices and Guidelines

13.1 Parameter Validation

Always Validate User Input

Never trust user input blindly!

1. Use Path() and Query() for automatic validation
2. Define acceptable values explicitly
3. Return 400 status code for invalid input
4. Provide clear error messages

Good Validation Example

```python
@app.get('/items')
def get_items(
    category: str = Query(
        ...,
        regex="^(books|electronics|clothing)$",
        description="Item category"
    ),
    min_price: float = Query(
        0,
        ge=0,
        le=10000,
        description="Minimum price filter"
    )
):
    # Input is validated before reaching here
    pass
```

13.2 Documentation Best Practices

1. **Always add descriptions** to parameters
2. **Provide examples** of valid values
3. **Document error responses** in docstrings
4. **Use type hints** consistently
5. **Write clear function docstrings**

Well-Documented Endpoint

```python
@app.get('/search')
def search_patients(
    name: str = Query(
        None,
        min_length=2,
        description="Search by patient name",
```

```
        example="John"
    ),
    min_age: int = Query(
        None,
        ge=0,
        le=150,
        description="Minimum age filter",
        example=18
    )
):
    """
    Search patients by name and/or age

    Query Parameters:
        name: Patient name (partial match)
        min_age: Minimum age filter

    Returns:
        List of matching patients
    """
    pass
```

13.3 Error Handling Best Practices

Warning

Use HTTPException for All Errors

Don't:

```
return {"error": "Something went wrong"}  # Wrong
```

Do:

```
raise HTTPException(
    status_code=400,
    detail="Something went wrong"
)  # Correct
```

Why?

- Proper HTTP status codes
- Consistent error format
- Better client experience
- Follows REST standards

13.4 Optional vs Required Parameters

Required Parameter	Optional Parameter

`param: str = Query(...)`
# OR 	
`param: str = Query()`
	`param: str = Query("default")`
	# OR
	`param: str = Query(None)`

Must be provided by client	Can be omitted
No default value	Has default value
FastAPI returns 422 if missing	Uses default if missing

13.5 Naming Conventions

Parameter Naming Guidelines

Path Parameters:

- Use singular nouns: `user_id`, `product_id`
- Describe the resource: `patient_id`, not just `id`
- Use snake_case

Query Parameters:

- Be descriptive: `sort_by`, `filter_status`
- Use snake_case consistently
- Keep concise but clear

14 Common Pitfalls and How to Avoid Them

14.1 Pitfall 1: Parameter Name Mismatch

Warning

Wrong:

```python
@app.get('/patient/{patient_id}')
def view_patient(id: str):  # Names don't match!
    pass
```

Error: FastAPI won't find the parameter!

Correct:

```python
@app.get('/patient/{patient_id}')
def view_patient(patient_id: str):  # Names match
    pass
```

14.2 Pitfall 2: Forgetting Question Mark in Query Parameters

Warning

Wrong URL:

```
/sort&sort_by=bmi&order=desc  # Missing ?
```

Correct URL:

```
/sort?sort_by=bmi&order=desc  # ✓ Has ?
```

Remember: Query parameters ALWAYS start with ?

14.3 Pitfall 3: Returning Wrong Status Codes

Warning

Wrong:

```python
if not found:
    return {"error": "Not found"}  # Returns 200!
```

Correct:

```python
if not found:
    raise HTTPException(
        status_code=404,
        detail="Not found"
    )  # Returns 404
```

14.4 Pitfall 4: Not Validating User Input

Warning

Vulnerable Code:

```
@app.get('/sort')
def sort_patients(sort_by: str, order: str):
    # No validation!
    sorted_data = sorted(data.values(), key=lambda x: x[sort_by
    ])
    # What if sort_by is invalid? KeyError!
```

Safe Code:

```
@app.get('/sort')
def sort_patients(sort_by: str = Query(...), order: str = Query
    ('asc')):
    valid_fields = ['height', 'weight', 'bmi']
    if sort_by not in valid_fields:
        raise HTTPException(status_code=400, detail="Invalid
    field")
    # Now safe to use sort_by
```

14.5 Pitfall 5: Confusing Path and Query Parameters

When to Use Which?

Wrong Choice:

```
# Using path param for optional filtering
@app.get('/products/{category}/{min_price}/{max_price}')
# All become required, URL gets messy
```

Better Choice:

```
# Using query params for optional filtering
@app.get('/products')
def get_products(
    category: str = Query(None),
    min_price: float = Query(None),
    max_price: float = Query(None)
):
    # All optional, clean URL
    pass
```

15 Troubleshooting Guide

15.1 Issue: 422 Unprocessable Entity

Error Message:

```
{
  "detail": [
    {
      "loc": ["query", "sort_by"],
      "msg": "field required",
      "type": "value_error.missing"
    }
  ]
}
```

Cause: Required parameter not provided

Solution:

- Ensure you're providing all required parameters
- Check parameter names match exactly
- For query params, use `?key=value` format

15.2 Issue: Parameter Not Being Recognized

Symptoms:

- Function receives None instead of value
- Parameter doesn't appear in docs

Common Causes & Solutions:

1. **Name Mismatch**

```python
# Wrong
@app.get('/user/{user_id}')
def get_user(id: str):

# Correct
@app.get('/user/{user_id}')
def get_user(user_id: str):

```

2. **Missing Type Annotation**

```python
# Wrong
def get_user(user_id):  # No type

# Correct
def get_user(user_id: str):  # Has type

```

15.3 Issue: Validation Not Working

Problem: Invalid values passing through

Check:

```
# Are you using Path() or Query()?
@app.get('/endpoint')
def handler(param: str):  # No validation

@app.get('/endpoint')
def handler(param: str = Query(...)):  # With validation
```

15.4 Issue: Status Codes Not Changing

Problem: Always getting 200 even for errors

Cause: Returning dictionaries instead of raising exceptions

Fix:

```
# Replace this:
return {"error": "Not found"}

# With this:
raise HTTPException(status_code=404, detail="Not found")
```

16 Practice Exercises

16.1 Exercise 1: Add Age Filter

Task: Create an endpoint to filter patients by age range

Requirements:

- Endpoint: `GET /filter-by-age`
- Query parameters:
  - `min_age`: Minimum age (optional, default 0)
  - `max_age`: Maximum age (optional, default 150)
- Return patients within age range
- Validate: min_age $\leq$ max_age

Expected Usage:

```
/filter-by-age?min_age=25&max_age=40
```

16.2 Exercise 2: Search by City

Task: Add search functionality for patients by city

Requirements:

- Endpoint: `GET /search-city`
- Query parameter: `city` (required)
- Return all patients from that city
- Case-insensitive search
- Return 404 if no patients found in that city

16.3 Exercise 3: Update Patient Weight

Task: Create endpoint to update a patient's weight

Requirements:

- Endpoint: `PUT /patient/{patient_id}/weight`
- Path parameter: `patient_id`
- Query parameter: `new_weight` (float, required)
- Update weight and recalculate BMI
- Return 404 if patient not found
- Return updated patient data

16.4 Exercise 4: Pagination

Task: Add pagination to view all patients

Requirements:

- Endpoint: `GET /patients`
- Query parameters:
  - `page`: Page number (default 1)
  - `limit`: Items per page (default 10, max 100)
- Return slice of patients based on pagination
- Include metadata: total_count, page, limit

Expected Response:

```
{
  "data": [ /* patient objects */ ],
  "page": 1,
  "limit": 10,
  "total_count": 5
}
```

16.5 Solutions Approach

Important Note

Try solving these exercises yourself first! They combine concepts from this session:

- Path parameters for resource identification
- Query parameters for filtering
- HTTPException for proper error handling
- Data validation
- Status codes

Solutions will use the same patterns we learned today!

17 Quick Reference Guide

17.1 Command Reference

Essential Commands

```
# Start server
uvicorn main:app --reload

# Access API
http://127.0.0.1:8000/

# Access documentation
http://127.0.0.1:8000/docs

# Test with path parameter
http://127.0.0.1:8000/patient/P001

# Test with query parameters
http://127.0.0.1:8000/sort?sort_by=bmi&order=desc
```

17.2 Code Templates

17.2.1 Path Parameter Template

```python
@app.get('/resource/{resource_id}')
def get_resource(
    resource_id: str = Path(
        ...,
        description="Resource identifier",
        example="123"
    )
):
    # Load data
    data = load_data()

    # Check if exists
    if resource_id in data:
        return data[resource_id]

    # Return 404
    raise HTTPException(
        status_code=404,
        detail="Resource not found"
    )
```

17.2.2 Query Parameter Template

```python
@app.get('/search')
def search(
    field: str = Query(
        ...,
        description="Field to search"
    ),
```

```
    value: str = Query(
        None,
        description="Search value"
    ),
    limit: int = Query(
        10,
        ge=1,
        le=100,
        description="Results limit"
    )
):
    # Validation
    if value is None:
        raise HTTPException(
            status_code=400,
            detail="Search value required"
        )

    # Search logic
    results = []  # Implement search

    return results[:limit]
```

17.3 Import Statements

```
# All imports needed for this session
from fastapi import FastAPI, Path, Query, HTTPException
import json
```

17.4 Status Code Quick Reference

Most Common Status Codes

```
200 OK              - Successful GET/PUT/PATCH
201 Created         - Successful POST
204 No Content      - Successful DELETE
400 Bad Request     - Invalid client data
404 Not Found       - Resource doesn't exist
422 Unprocessable   - Validation error
500 Server Error    - Server problem
```

17.5 URL Format Reference

URL Patterns

```
# Path parameter
/resource/{id}
Example: /patient/P001

# Query parameters
/resource?key1=value1&key2=value2
Example: /sort?sort_by=bmi&order=desc

# Combined (covered in future)
/resource/{id}?filter=value
Example: /patient/P001?include=history
```

End of FastAPI Path & Query Parameters Guide

"The only way to do great work is to love what you do."
— Steve Jobs

Keep building, keep learning!
Ready for the next session on POST requests and Pydantic models!