02 FastAPI Path and Query Parameters

FastAPI Path and Query Parameters

Path Parameters

Path parameters allow you to create dynamic routes in your API, making your endpoints more flexible and reusable. They're essential for RESTful API design and are commonly used to identify specific resources.

Basic Path Parameters

Path parameters are parts of the URL path that are variable and can be captured as function parameters. They're defined by enclosing the parameter name in curly braces `{}` within the path string.

```python
from fastapi import FastAPI

app = FastAPI()

@app.get('/property/{id}')
def get_property(id):
    return f"This is a page for property {id}"
```

With this single route, you can handle requests to `/property/1`, `/property/2`, etc., without writing separate route handlers for each property ID.

Path Parameters with Type Validation

One of the most powerful features of FastAPI is its automatic validation system. By adding type hints to your path parameters, FastAPI will:

1. Validate the input according to the type
2. Convert the parameter to the specified type
3. Generate OpenAPI documentation that includes the type information
4. Provide better editor support with autocomplete

```python
@app.get('/property/{property_id}')
def get_property(property_id: int):
    return f"This is a page for property {property_id}"
```

Now, if someone tries to access `/property/chennai`, they'll receive a validation error because "chennai" can't be converted to an integer. Only numeric IDs will be accepted.

Real-world Example: E-commerce Product Detail

```python
@app.get('/products/{product_id}')
def get_product_details(product_id: int):
    # In a real application, you would fetch the product from a database
    product = {
        "id": product_id,
        "name": f"Product {product_id}",
        "price": 29.99,
        "in_stock": True
    }
    return product
```

Advanced Path Parameters: Using Enums

For cases where you want to restrict path parameters to a specific set of values, you can use Python's `Enum` class:

```
from enum import Enum
from fastapi import FastAPI

class CategoryName(str, Enum):
    electronics = "electronics"
    clothing = "clothing"
    books = "books"
    home = "home"

app = FastAPI()

@app.get('/categories/{category}')
def get_category(category: CategoryName):
    return {"category": category, "items": f"List of {category.value} items"}
```

Query Parameters

Query parameters are used to filter, sort, or provide additional information to an API endpoint without being part of the resource identification. They're especially useful for optional parameters or when you need to include many parameters.

Basic Query Parameters

In FastAPI, any function parameter that isn't part of the path parameters is automatically interpreted as a query parameter.

```
@app.get('/products/')
def list_products(min_price: float = 0, max_price: float = 1000, sort_by: str =
"price"):
    return {
        "min_price": min_price,
        "max_price": max_price,
        "sort_by": sort_by,
        "products": f"List of products filtered by price between {min_price} and
{max_price}, sorted by {sort_by}"
    }
```

This endpoint can be called with:

- `/products/` (uses default values)
- `/products/?min_price=20` (overrides only min_price)

- `/products/?min_price=20&max_price=50&sort_by=name` (overrides all parameters)

Required vs Optional Query Parameters

Parameters with default values are optional. Parameters without default values are required:

```python
@app.get('/search/')
def search_items(query: str, category: str = None, page: int = 1, items_per_page:
int = 10):
    results = f"Search results for '{query}'"
    if category:
        results += f" in category '{category}'"

    results += f" (Page {page}, showing {items_per_page} items per page)"

    return {"results": results}
```

In this example:

- `query` is required (no default value)
- `category`, `page`, and `items_per_page` are optional (have default values)

Real-world Example: Product Filtering API

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

app = FastAPI()

@app.get('/products/filter/')
def filter_products(
    category: Optional[str] = None,
    min_price: float = 0,
    max_price: Optional[float] = None,
    brand: Optional[List[str]] = Query(None),
    in_stock: bool = True,
    sort_by: str = "popularity",
    page: int = 1,
    limit: int = 20
):
    filters = {
        "category": category,
        "price_range": {"min": min_price, "max": max_price},
        "brands": brand,
        "in_stock_only": in_stock,
        "sort_by": sort_by,
        "pagination": {"page": page, "limit": limit}
    }

    # In a real application, you would use these parameters to query a database
    return {
        "filters_applied": filters,
        "total_results": 42,
        "products": [
            {"id": 1, "name": "Sample Product", "price": 49.99}
            # More products would be here
        ]
    }
```

This endpoint allows for complex filtering with URL patterns like:

```
/products/filter/?
category=electronics&min_price=100&max_price=500&brand=Apple&brand=Samsung&in_stoc
k=true&sort_by=price&page=2&limit=10
```

Advanced Query Parameters: Validation with Query

FastAPI provides the `Query` class for adding more validation and metadata to query parameters:

```
from fastapi import FastAPI, Query

app = FastAPI()

@app.get('/users/')
def search_users(
    q: str = Query(
        None,
        min_length=3,
        max_length=50,
        description="Search query string",
        title="Search Query"
    ),
    status: str = Query(
        "active",
        regex="^(active|inactive|pending)$"
    ),
    offset: int = Query(0, ge=0),
    limit: int = Query(10, ge=1, le=100)
):
    return {
        "search": q,
        "status": status,
        "offset": offset,
        "limit": limit
    }
```

Combining Path and Query Parameters

In real-world applications, you'll often need to use both path and query parameters together.

Example: E-commerce API

```
@app.get('/users/{user_id}/orders/')
def get_user_orders(
    user_id: int,
```

```python
    status: Optional[str] = None,
    from_date: Optional[str] = None,
    to_date: Optional[str] = None,
    page: int = 1,
    limit: int = 10
):
    # In a real application, you would fetch orders from a database
    filters = {
        "status": status,
        "date_range": {"from": from_date, "to": to_date},
        "pagination": {"page": page, "limit": limit}
    }

    return {
        "user_id": user_id,
        "filters_applied": filters,
        "total_orders": 42,
        "orders": [
            {"id": 1, "status": "delivered", "total": 99.99, "date": "2023-01-15"}
            # More orders would be here
        ]
    }
```

Example: Social Media API

```python
@app.get('/profiles/{username}/posts/{post_id}/comments/')
def get_post_comments(
    username: str,
    post_id: int,
    sort_by: str = "newest",
    filter_by: Optional[str] = None,
    page: int = 1,
    limit: int = 20
):
    return {
        "username": username,
        "post_id": post_id,
        "comments": f"Comments for post {post_id} by {username}",
        "sort_by": sort_by,
        "filter_by": filter_by,
        "pagination": {"page": page, "limit": limit}
    }
```

Best Practices for Path and Query Parameters

1. **Use path parameters for resource identification**: If it identifies a specific resource (user ID, product ID, etc.), it should be a path parameter.
2. **Use query parameters for filtering, sorting, pagination**: Parameters that filter or customize the view of a resource should be query parameters.
3. **Be consistent with naming:** Use snake_case or camelCase consistently throughout your API.

4. **Always add type annotations:** They enhance documentation and provide automatic validation.
5. **Make non-essential parameters optional:** Use default values for parameters that aren't strictly required.
6. **Add proper validation:** Use FastAPI's validation tools to ensure your parameters meet your requirements.
7. **Document your parameters:** FastAPI generates documentation automatically, but you can enhance it with descriptions.
8. **Use proper HTTP methods:** GET for retrieving, POST for creating, PUT/PATCH for updating, DELETE for removing.

Error Handling for Parameters

FastAPI automatically handles validation errors, but you can customize the error responses:

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

app = FastAPI()

@app.get('/items/{item_id}')
def get_item(
    item_id: int = Path(... , ge=1, description="The ID of the item to get"),
    q: str = Query(None, min_length=3, max_length=50)
):
    items = {1: "Laptop", 2: "Phone", 3: "Tablet"}

    if item_id not in items:
        raise HTTPException(status_code=404, detail=f"Item with ID {item_id} not
found")

    item = items[item_id]

    if q:
        if q.lower() not in item.lower():
            raise HTTPException(status_code=404, detail=f"No item found matching
query '{q}'")

    return {"item_id": item_id, "item": item, "query": q}
```

Security Considerations

When working with path and query parameters, always be mindful of:

1. **Data validation:** Always validate and sanitize input data to prevent injection attacks.
2. **Sensitive information:** Never include sensitive data (passwords, tokens) in URLs.
3. **Parameter constraints:** Use FastAPI's validation to limit input to reasonable ranges or patterns.
4. **Rate limiting:** Implement rate limiting to prevent abuse of your API endpoints.

Conclusion

Path and query parameters are foundational concepts in API design that allow you to create flexible, dynamic endpoints. FastAPI's strong typing and validation features make working with these

parameters both safer and more developer-friendly than in many other frameworks.
By understanding when and how to use each type of parameter, you can design intuitive, RESTful APIs that are easy to use and maintain.