03 FastAPI Request Body - Beginner

Request Body in FastAPI

Understanding Request Bodies

In API development, a request body is data sent by the client to your API. Request bodies are crucial when clients need to send larger structured data sets to the server.

The key points to understand:

- A response body is what your API sends back to clients
- A request body is what clients send to your API
- APIs always need to send response bodies, but clients don't always need to send request bodies
- POST requests typically include request bodies for creating or submitting data

Creating a Simple POST Request

```python
@app.post('/adduser')
def adduser():
    return {'user': {'name': 'kirkyagami', 'email': 'kirkyagami99@gmail.com'}}
```

This example defines a POST endpoint that doesn't yet process any incoming data - it simply returns a hardcoded user object. To actually process client data, we need to implement request body handling.

Using Pydantic Models for Request Bodies

Pydantic is a data validation library that allows us to define custom data types with validation rules. It's the foundation of FastAPI's request body handling.

```python
from pydantic import BaseModel

class Profile(BaseModel):
    name: str
    email: str
    age: int

@app.post('/profile/')
def createprofile(profile: Profile):
    return profile
```

When this code runs:

1. FastAPI will see that the function parameter `profile` is a Pydantic model
2. It will read the request body as JSON
3. It will validate the incoming data against the Profile model requirements
4. If valid, it will create a Profile instance and pass it to your function
5. If invalid, it will return a detailed error message to the client

The beauty of this approach is that validation happens automatically - you don't need to write any validation code yourself.

Combining Request Bodies with Path Parameters

FastAPI can distinguish between different parameter sources in your function:

```
from pydantic import BaseModel

class Product(BaseModel):
    name: str
    price: int
    discount: int
    discounted_price: int


@app.post('/addproduct/{product_id}')
def addproduct(product: Product, product_id: int):
    product.discounted_price = product.price - (product.price * product.discount /
100)
    return {'product_id': product_id, 'product': product}
```

FastAPI knows that:

- `product_id` comes from the path because it matches the path parameter name
- `product` comes from the request body because it's a Pydantic model

This allows you to combine different data sources in a single endpoint, creating flexible and powerful APIs.

Adding Query Parameters to the Mix

You can also include query parameters alongside path parameters and request bodies:

```
@app.post('/addproduct/{product_id}')
def addproduct(product: Product, product_id: int, category: str):
    product.discounted_price = product.price - (product.price * product.discount /
100)
    return {'product_id': product_id, 'product': product, 'category': category}
```

Here, FastAPI recognizes that:

- `product_id` is a path parameter
- `product` is a request body
- `category` is a query parameter (since it's not in the path and not a Pydantic model)

This allows you to build complex endpoints that accept data in multiple formats.

Multiple Body Parameters

Sometimes you need to accept multiple objects in a single request. FastAPI supports this by allowing multiple Pydantic models:

```
class Product(BaseModel):
    name: str
    price: int
    discount: int
    discounted_price: int

class User(BaseModel):
    name: str
    email: str

@app.post('/purchase/')
def purchase(user: User, product: Product):
    return {"user": user, "product": product}
```

In this example, the client would send a JSON object containing both user and product data:

```
{
  "user": {
    "name": "John",
    "email": "john@example.com"
  },
  "product": {
    "name": "Smartphone",
    "price": 500,
    "discount": 10,
    "discounted_price": 450
  }
}
```

FastAPI will parse this and provide separate model instances to your function.

Enhancing Models with Field Metadata

The `Field` class from Pydantic allows you to add extra validation and documentation to your model fields:

```
from pydantic import BaseModel, Field

class Product(BaseModel):
    name: str
    price: int = Field(..., gt=0, title="Price of the item", description="Price
must be greater than zero")
    discount: int
    discounted_price: int
```

The `Field` function accepts:

- A default value (or `...` for required fields)
- Validation constraints (`gt=0` means "greater than zero")
- Documentation metadata (title and description)

These details will appear in your API documentation, making it more informative for developers using your API.

In the Swagger UI documentation, under the "Schemas" section, you'll see this additional metadata, which helps API consumers understand what data they need to provide.

This approach to request body handling gives you precise control over the data your API accepts while automating validation and documentation - a significant advantage of FastAPI over other frameworks.