FastAPI

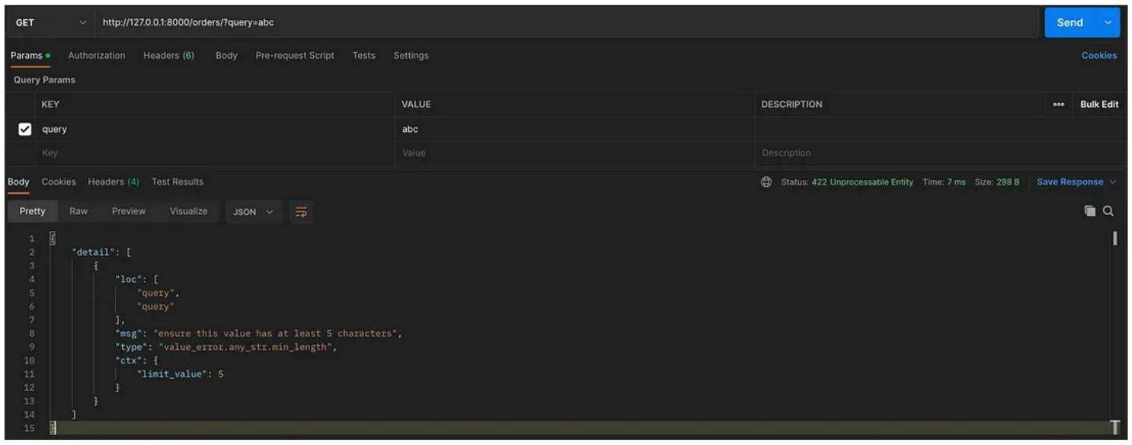
FastAPI is an enjoyable tool for building web applications in Python. It is well known for its integration with Pydantic models, which makes defining and validating data structures straightforward and efficient. In this guide, we explore how simple functions that return Pydantic models can seamlessly integrate with FastAPI.

FastAPI supports string and numeric parameter validations.

String validation example

In the example below, I have added a string constraint with a minimum 5 and maximum 100 character length. When I call this endpoint through Postman and pass a text with a size of 3, it throws an exception as below.

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

app = FastAPI()

@app.get("/orders/")
async def read_orders(query: Optional[str] = Query(None, min_length=5, max_length=100)):
#defining the constraints
    results = {"order": [{"order_id": "Order_1"}, {"order_id": "Order_2"}]}
    return results
    if query:
        results.update({"query": query})

if __name__ == "__main__":
    uvicorn.run("parameter-validations:app")
```



```
http://127.0.0.1:8000/orders/?query=abc
```

Query Params

KEY	VALUE	DESCRIPTION
query	abc	

Status: 422 Unprocessable Entity Time: 7 ms Size: 298 B

```
{
    "detail": [
        {
            "loc": [
                "query",
                "query"
            ],
            "msg": "ensure this value has at least 5 characters",
            "type": "value_error.any_str.min_length",
            "ctx": {
                "limit_value": 5
            }
        }
    ]
}
```

Serving Structured Data using Pydantic Models

One of Python's main attractions is that it's a dynamically typed language. Dynamic typing means that variable types are determined at runtime, unlike statically typed languages where they are explicitly declared at compile time. While dynamic typing is great for rapid development and ease of use, you often need more robust type checking and data validation for real-world applications.

Pydantic has quickly gained popularity, and it’s now the most widely used data validation library for Python.

Creating a Product Management API

To illustrate the capabilities of FastAPI and Pydantic, we’ll create a simple product management API.

This API will allow clients to create, retrieve, update, and delete products.

Defining the Product Model

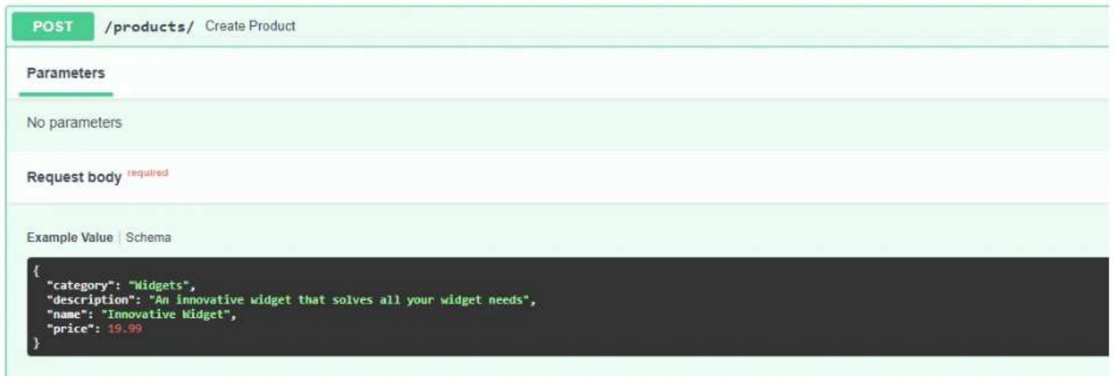
We define a Product class and a ProductRequest Pydantic model for our product data.

The Product class represents our data model, while ProductRequest is used for data validation and serialization of request data.

```
from typing import Optional
from pydantic import BaseModel, Field

class Product:
    id: int
    name: str
    category: str
    description: str
    price: float

    def __init__(self, id, name, category, description, price):
        self.id = id
        self.name = name
        self.category = category
        self.description = description
        self.price = price

class ProductRequest(BaseModel):
    id: Optional[int] = None
    name: str = Field(min_length=1)
    category: str = Field(min_length=1)
    description: str = Field(min_length=1, max_length=100)
    price: float = Field(gt=0)  # Price must be greater than 0

    class Config:
        json_schema_extra = {
            'example': {
                'name': 'Innovative Widget',
                'category': 'Widgets',
                'description': 'An innovative widget that solves all your widget needs',
                'price': 19.99
        }
    }
```

The Product class is a simple Python class that represents a product with attributes like id, name, category, description, and price. This class is used to create product instances which we'll store in an in-memory list for simplicity.

The ProductRequest class is a Pydantic model that defines the structure and validation rules for product data received from clients. It inherits from BaseModel, allowing it to benefit from Pydantic's data validation features. Fields such as name, category, description, and price are validated according to the rules defined (e.g., min_length, max_length, gt). This ensures that only valid data is accepted for creating or updating products.



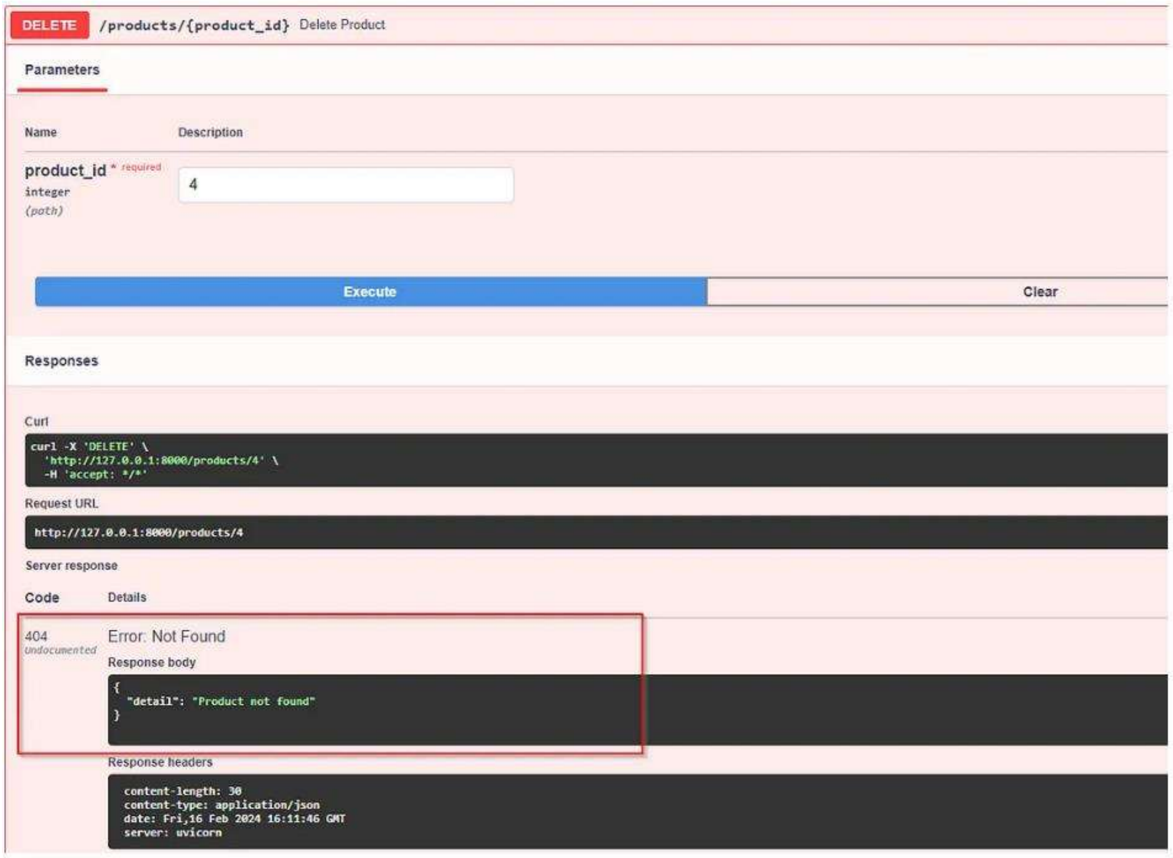
```
{
  "category": "Widgets",
  "description": "An innovative widget that solves all your widget needs",
  "name": "Innovative Widget",
  "price": 19.99
}
```

Implementing CRUD Operations

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

app = FastAPI()

# Example products list
PRODUCTS = [
    Product(1, 'Widget Pro', 'Widgets', 'A high-quality widget', 29.99),
    Product(2, 'Gadget Max', 'Gadgets', 'A versatile gadget for all your needs', 49.99),
]

# Create a product
@app.post('/products/', status_code=status.HTTP_201_CREATED)
async def create_product(product_request: ProductRequest):
    new_product = Product(**product_request.model_dump())
    PRODUCTS.append(new_product)
    return new_product

# Read all products
@app.get('/products/', status_code=status.HTTP_200_OK)
async def read_all_products():
    return PRODUCTS
```

```
# Read a product by ID
@app.get('/products/{product_id}', status_code=status.HTTP_200_OK)
async def read_product(product_id: int = Path(gt=0)):
    for product in PRODUCTS:
        if product.id == product_id:
            return product
    raise HTTPException(status_code=404, detail='Product not found')

# Update a product
@app.put('/products/{product_id}', status_code=status.HTTP_200_OK)
async def update_product(product_id: int, product_request: ProductRequest):
    for i, product in enumerate(PRODUCTS):
        if product.id == product_id:
            updated_product = Product(id=product_id, **product_request.model_dump())
            PRODUCTS[i] = updated_product
            return updated_product
    raise HTTPException(status_code=404, detail='Product not found')

# Delete a product
@app.delete('/products/{product_id}', status_code=status.HTTP_204_NO_CONTENT)
async def delete_product(product_id: int):
    for i, product in enumerate(PRODUCTS):
        if product.id == product_id:
            PRODUCTS.pop(i)
            return
    raise HTTPException(status_code=404, detail='Product not found')
```

Create a Product Endpoint: This endpoint creates a new product. The @app.post decorator specifies that it's a POST endpoint at the path /products/. The function takes a product_request of type ProductRequest. Using **product_request.model_dump(), we unpack the validated request data into the Product constructor, creating a new Product instance.



Curl

```
curl -X 'DELETE' \
  'http://127.0.0.1:8000/products/4' \
  -H 'accept: */*'
```

Request URL

```
http://127.0.0.1:8000/products/4
```

Server response

Code Details

404 *Undocumented* Error: Not Found

Response body

```
{
  "detail": "Product not found"
}
```

Response headers

```
content-length: 30
content-type: application/json
date: Fri,16 Feb 2024 16:11:46 GMT
server: uvicorn
```

Databases

Explanation

FastAPI works well with SQL and NoSQL databases. SQLAlchemy is commonly used for SQL databases, and Tortoise ORM is an example of a NoSQL ORM that works well with FastAPI.

Example Code: SQLAlchemy with SQLite

` database.py`

```
from sqlalchemy import create_engine, Column, Integer, String

from sqlalchemy.ext.declarative import declarative_base

from sqlalchemy.orm import sessionmaker


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

FastAPI

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

class User(Base):
    __tablename__ = "users"
    id = Column(Integer, primary_key=True, index=True)
    name = Column(String, index=True)
    email = Column(String, unique=True, index=True)
Base.metadata.create_all(bind=engine)
```

`main.py`

```
from fastapi import FastAPI, Depends, HTTPException
from sqlalchemy.orm import Session
from database import SessionLocal, User
app = FastAPI()
def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()

@app.post("/users/", response_model=User)
def create_user(user: User, db: Session = Depends(get_db)):
    db.add(user)
    db.commit()
    db.refresh(user)
    return user
```

```
@app.get("/users/{user_id}", response_model=User)
def read_user(user_id: int, db: Session = Depends(get_db)):
    user = db.query(User).filter(User.id == user_id).first()
    if user is None:
        raise HTTPException(status_code=404, detail="User not found")
    return user
```

Expected Output

When running the FastAPI app with Uvicorn, you can create and read users via the API endpoints:

- ` POST /users/ ` to create a user.
- ` GET /users/{user_id}` to retrieve a user by ID.

HTTP Methods

Explanation

FastAPI supports all standard HTTP methods: GET, POST, PUT, DELETE, etc.

Example Code

` main.py `

```
from fastapi import FastAPI
app = FastAPI()

@app.get("/")
def read_root():
    return {"Hello": "World"}

@app.post("/items/")
def create_item(name: str):
    return {"name": name}
```

```
@app.put("/items/{item_id}")
def update_item(item_id: int, name: str):
    return {"item_id": item_id, "name": name}


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

Expected Output

- ` **GET /** ` returns `**{"Hello": "World"}**`.
- ` **POST /items/** ` with a body parameter ` **name** ` returns `**{"name": "value"}**`**.**
- ` **PUT /items/{item_id}**` with a body parameter ` **name** ` updates and returns the item.
- ` **DELETE /items/{item_id}** ` deletes and returns the item ID.

Authentication

Explanation

FastAPI provides OAuth2 and JWT for secure authentication.

Example Code: OAuth2 with Password (and hashing), JWT Tokens

`**main.py**`

```
from fastapi import Depends, FastAPI, HTTPException, status
from fastapi.security import OAuth2PasswordBearer, OAuth2PasswordRequestForm
from passlib.context import CryptContext
from pydantic import BaseModel


app = FastAPI()
oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")


fake_users_db = {
    "johndoe": {
        "username": "johndoe",
        "full_name": "John Doe",
        "email": "johndoe@example.com",
        "hashed_password":
"$2b$12$KIXi1Ryk9SbhTH6G/0TgmeH8x6lK6wdqK6k0tl9y5WbbHdKCV9RNu",
        "disabled": False,
    }
}
pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")


class Token(BaseModel):
    access_token: str
    token_type: str


class User(BaseModel):
    username: str
    email: str
    full_name: str = None
    disabled: bool = None


class UserInDB(User):
    hashed_password: str


def verify_password(plain_password, hashed_password):
    return pwd_context.verify(plain_password, hashed_password)


def get_user(db, username: str):
    if username in db:
        user_dict = db[username]
        return UserInDB(**user_dict)


def authenticate_user(fake_db, username: str, password: str):
    user = get_user(fake_db, username)
    if not user:
        return False
    if not verify_password(password, user.hashed_password):
        return False
    return user


@app.post("/token", response_model=Token)
async def login_for_access_token(form_data: OAuth2PasswordRequestForm = Depends()):
    user = authenticate_user(fake_users_db, form_data.username, form_data.password)
    if not user:
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Incorrect username or password",
            headers={"WWW-Authenticate": "Bearer"},
        )
    return {"access_token": user.username, "token_type": "bearer"}


@app.get("/users/me", response_model=User)
async def read_users_me(token: str = Depends(oauth2_scheme)):
    username = token
    user = get_user(fake_users_db, username)
    if user is None:
        raise HTTPException(status_code=400, detail="Invalid authentication credentials")
    return user
```

Expected Output

- **POST /token** with valid credentials returns a JWT token.
- **GET /users/me** with a valid token returns the current user's information.

Testing

Explanation

FastAPI supports testing with the standard unittest or pytest libraries.

Example Code: Pytest

`test_main.py`

```
from fastapi.testclient import TestClient
from main import app

client = TestClient(app)

def test_read_root():
    response = client.get("/")
    assert response.status_code == 200
    assert response.json() == {"Hello": "World"}

def test_create_item():
    response = client.post("/items/", json={"name": "Item 1"})
    assert response.status_code == 200
    assert response.json() == {"name": "Item 1"}
```

Expected Output

Running ` **pytest** `on the test file will check if the endpoints return the correct status codes and responses.

Deployment

Explanation

Deploying FastAPI can be done using various methods such as Uvicorn, Docker, or on cloud platforms.

Example Code: Using Uvicorn

uvicorn main:app --host 0.0.0.0 --port 8000 à in shell

Example Code: Docker

` Dockerfile `

```
FROM tiangolo/uvicorn-gunicorn-fastapi:python3.8
COPY ./app /app
RUN pip install -r /app/requirements.txt
```

FastAPI

CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]

` requirements.txt `

fastapi

uvicorn

sqlalchemy

Expected Output

Running the Docker container will start the FastAPI application accessible at ` **http://localhost:8000**`.

U can go through these references to get better understanding

https://fastapi.tiangolo.com/tutorial/first-steps/

https://devdocs.io/fastapi/

Thank you