FastAPI Basics

an Introduction to FastAPI

Vincent Errick Itucal

August 3, 2023

Introduction

What is FastAPI?

- Fast - onpar with NodeJS and Go
- Easy - Design to be easy to use and learn
- Supports python type hints
- Standards based: can produce OpenAPI (previously known as swagger) docs

First Steps

Simplest FastAPI App:

requirements.txt

```
fastapi
pydantic
uvicorn
```

main.py

```python
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
async def root():
    return {"message": "Hello World"}
```

Run the live server

```
uvicorn main:app --reload
```

Interactive API docs

go to http://127.0.0.1:5000/docs.

Review on components of HTTP Requests

- Methods - POST, GET, PUT, DELETE, etc.
- Path and Path Params -
  - in the form http://{host/host:port}/{path}
  - http://tianglo.com/client/{clientId}
- Query Params - key-value pairs preceeded by '?' in the url
  - http://tianglo.com/client/1?date=2023-07-19&age=26
  - key is date, age
  - value is 2023-07-19, 26
- Headers - additional info on your HTTP request/response (Authorization, etc)
- Request/Response Body - Main content sent/received to/from the server (usually in POST, PUT and GET requests). Can be anything like form-data, raw, binary, GraphQL, XML but by far the most popular is JSON

JSON object sample

```
{
  "mac": "10:B5:S3:06:C6:E9",
  "route":0,
  "latitude":53.834588,
  "longitude":10.704048,
  "time":"12.09.2015 13:45:00",
  "speed":3,
  "direction":"",
  "flag": true
}
```

HTTP Methods, Path and Status code

Code Snippet

```
app = FastAPI()

@app.get(path="/", status_code=200)
@app.post(path="/", status_code=200)
@app.put(path="/", status_code=200)
@app.delete(path="/", status_code=200)
```

by default if not defined return status code is 200
Other status codes are:

- Informational responses (100 – 199)
- Successful responses (200 – 299)
- Redirection messages (300 – 399)
- Client error responses (400 – 499)
- Server error responses (500 – 599)

Path Parameters

Code Snippet

```
@app.get(path="/path_params/{path_param_name}")
async def path_params(path_param_name: int):
    return {"path_param_name": path_param_name}
```

- expects a path param named "path_param_name" of type int
- then return as json with status code 200 OK

Path Parameters

Code Snippet

```
@app.get(path="/path_params/{path_param_name}")
async def path_params(path_param_name: int):
    return {"path_param_name": path_param_name}
```

- expects a path param named "path_param_name" of type int
- then return as json with status code 200 OK

Query Parameters

Code Snippet

```
from fastapi import Query
...
@app.get(path="/query_params")
async def query_params(
    needy: str,
    bbn: Union[str, None] = None,
    actual_date: Union[str, None] = Query(default=None),
):
    return {
        "bbn": bbn,
        "actual_date": actual_date,
        "needy": needy,
    }
```

- expects a query params named <u>needy</u>, <u>bbn</u>, <u>actual_date</u> all of type str
- <u>needy</u> variable, as the name describes is a required query param (based on the type hints) if not passed then it will return a 422 error
- on the other hand <u>bbn</u> and <u>actual_date</u> are optional query params as suggested by the type hints
- can also explicitly declare input as type Query (actual_date), but not needed it will be tackled why in the next slide

Headers

Code Snippet

```
from fastapi import Header
...
@app.post(path="/headers")
async def headers(
    sample_header1: Union[str, None] = Header(default=None),
    sample_header2: Union[str, None] = Header(default=None, convert_underscores=False)
):
    return {
        "sample_header1": sample_header1,
        "sample_header2": sample_header2
    }
```

- To declare headers, you need to use Header, because otherwise the parameters would be interpreted as query parameters.
- Headers are usually in the hyphenated form (e.g. Sample-Header1). FastAPI auto-converts this by default when you declare your param name by replacing the dash with an underscore automatically, also headers should be case-insensitive so variable name is lowercase.
- In cases Header is not in the usual format e.g. containing underscores you can disable convert underscores

Request Body

Code Snippet

```
from pydantic import BaseModel
...
class LoginCreds(BaseModel):
    username: str
    password: str
...
@app.post(path="/request_body", response_model=LoginCreds)
async def request_body(creds: LoginCreds):
    return creds
```

- params of type X inheriting from BaseModel are considered request body
- declare what your json schema in the pydantic model
- LoginCreds matches this json for example:

```
{
  "username": "vitucal",
  "password": "password"
}
```

Response Model

Code Snippet

```
from pydantic import BaseModel
...
class LoginCreds(BaseModel):
    username: str
    password: str
...
@app.post(path="/request_body", response_model=LoginCreds)
async def request_body(creds: LoginCreds):
    return creds
```

- Response Model of a type inheriting BaseModel will automatically return a json response body for it
- This will also produce a corresponding model for the response in the api docs

Request Body - Fields

Code Snippet

```
from pydantic import BaseModel, Field
...
class PersonModelWithFields(BaseModel):
    name: str
    description: Union[str, None] = Field(
        default="Person", title="description of this person", max_length=10,
    )
    age: float = Field(ge=0, description="age must be greater than or equal to zero")
...
@app.post(path="/request_body/fields", response_model=PersonModelWithFields)
async def request_body_fields(person: PersonModelWithFields):
    return person
```

- You can use Field to further describe model attributes as shown above
- You can set default values, add descriptions, assert leangth, if ge (greater than or equal) and many more.
- This will also be reflected in the api docs

Request Body - Multi Params

Code Snippet

```
from pydantic import BaseModel, Field
...
class LoginCreds(BaseModel):
    username: str
    password: str
...
class PersonModelWithFields(BaseModel):
    name: str
    description: Union[str, None] = Field(
        default="Person", title="description of this person", max_length=10,
    )
    age: float = Field(ge=0, description="age must be greater than or equal to zero")
...
@app.post(path="/request_body/multi_params", response_model=MultiParam)
async def request_body_multi_params(
    person: PersonModelWithFields,
    creds: LoginCreds
):
    return {
        "person": person,
        "creds": creds
    }
```

- defining two or more types inheriting BaseModels from param will automatically detect them

Sample JSON for multi param request body

```json
{
    "person": {
        "name": "vitucal",
        "description": "cool",
        "age": 26
    },
    "creds": {
        "username": "vitucal",
        "password": "password"
    }
}
```

Request Body - Nested Models

Code Snippet

```
from pydantic import BaseModel, Field
...
class NestedPersonWithCreds(BaseModel):
    name: str
    description: Union[str, None] = Field(
        default="Person", title="description of this person", max_length=10,
    )
    age: float = Field(ge=0, description="age must be greater than or equal to zero")
    creds: Union[LoginCreds, None]
...
@app.post(path="/request_body/nested")
async def request_body_nested(
    person_with_creds: NestedPersonWithCreds
):
    return person_with_creds
```

- Models can also be nested as shown

Sample JSON for nested model request body

```
{
    "name": "vitucal",
    "description": "cool",
    "age": 26,
    "creds": {
        "username": "vitucal",
        "password": "password"
    }
}
```

Handling Errors

Code Snippet

```
from fastpi import HTTPException
...
@app.post(path="/handling_errors")
async def handling_errors(
    error: bool
):
    if error:
        raise HTTPException(status_code=400, detail="I raised an error")
    return {
        "detail": "no errors"
    }
```

- In this example we are trying to route a response in case we want to handle errors.
- this can be done by raising an HTTPException and returning a desired status code and detail in the response body.
- This route expects an error query param, if true return an error else return a success with no errors.

Middlewares

Code Snippet

```
@app.middleware("http")
async def add_process_time_header(request: Request, call_next):
    start_time = time.time()
    response = await call_next(request)
    process_time = time.time() - start_time
    response.headers["X-Process-Time"] = str(process_time)
    return response
```

- A "middleware" is a function that works with every request before it is processed by any specific path operation and also with every response before returning it.
- The middleware above records the time initially before processing the request, when done it records the process time and is added to a X-Process-Time header

Practical Applications In DE?

Code Snippet

```
@app.get(path="/table_contents")
async def get_transactions(
    query_date: Union[str, None] = None,
    bbn: Union[str, None] = None,
):
    df = spark.read.format("csv").option("header", "true"). \
        load("/code/fast-api-basics/app/file.csv")

    if query_date is not None:
        df = df.filter(f.col("business_date") == query_date)
    if bbn is not None:
        df = df.filter(f.col("bbn") == bbn)

    result_list = df.rdd.map(lambda row: row.asDict()).collect()

    if not result_list:
        return HTTPException(status_code=404, detail="record not found!", headers=None)
    else:
        return result_list
```

- A GET HTTP API wherein we return the contents of a table depending on the filters provided in the query param
- Can be extended to adding pagination

References

Tiangolo (Sebastián Ramírez), *Fastapi Webpage*, `https://fastapi.tiangolo.com/`, Accessed: August 2, 2023.