

BUILDING REST API ENDPOINT WITH FASTAPI

Chris Choy
PYCON HK 2020
November

Code Available at Github

<https://github.com/christlc/pyconhk2020fastapi>

Today's Content



REST API

Quick introduction of REST API

DEMO

Getting Parameter: path, query, body
Data Validation

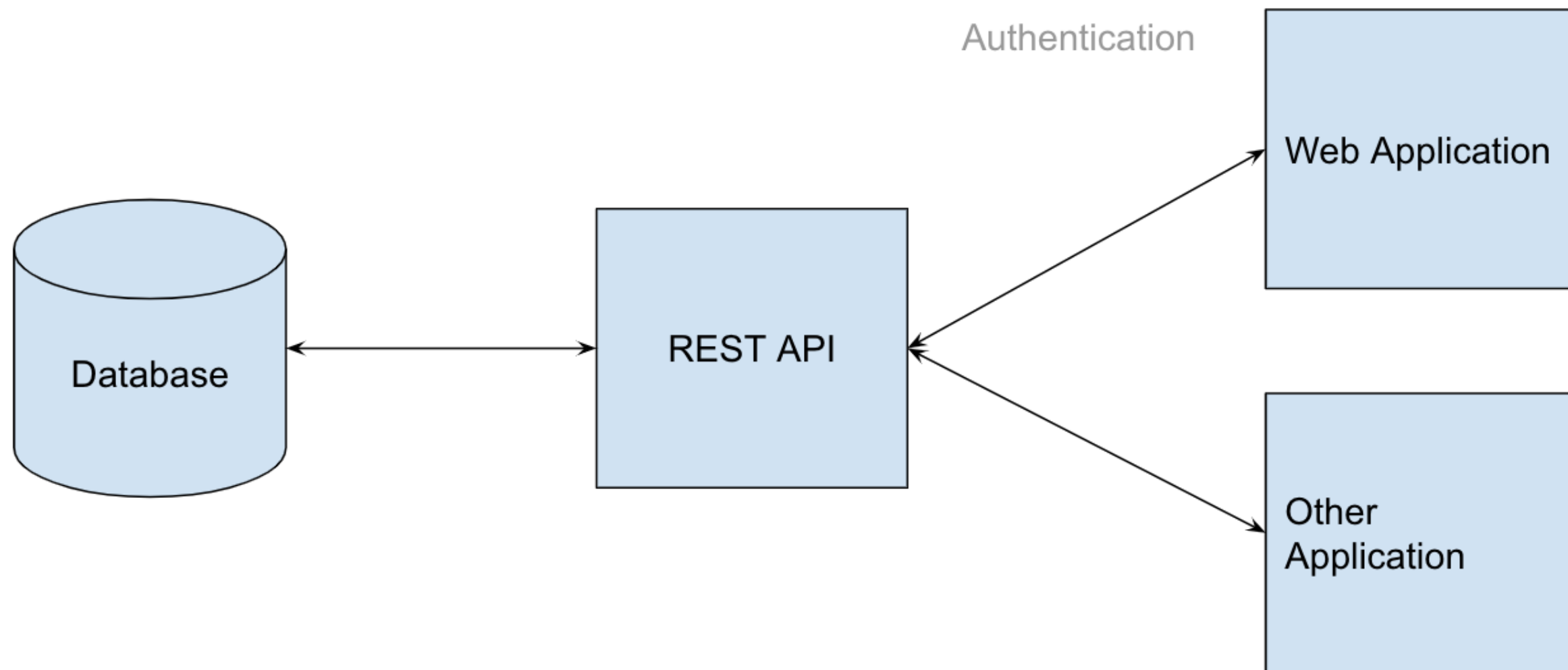
COMPARISON AND TIPS

Comparison with Flask

A black and white photograph of a notebook, a pen, and a flower on a wooden surface. The notebook is open, showing a blank page. A pen lies next to it. A flower is also visible. The image is part of a presentation slide with a green background.

What is a REST API?

Typical Architecture



FastAPI with (Some) Batteries Included

PARAMETER VALIDATION

Using type hint to automatically validate input format and provide useful error messages

API DOCUMENTATION

In accessible Swagger doc format with authentication support in-built

MULTIPLE FILES ROUTERS

Much easier than in vanilla Flask

Hello World

```
1  from fastapi import FastAPI
2
3  app = FastAPI()
4
5
6  @app.get("/")
7  def read_root():
8      return {"Hello": "World"}
9
10 if __name__ == "__main__":
11     import uvicorn
12     uvicorn.run("main:app", host="0.0.0.0", port=8000, log_level="info", workers=1)
13
```

← → ↻ ⓘ localhost:8000

```
{"Hello": "World"}
```

First API

GET at <http://127.0.0.1:8000>

Lets do a simple endpoint

GET /message/

Return all message in dictionary

POST /MESSAGE/{ID}

Write message to in memory
dictionary at key ID

GET /MESSAGE/{ID}

Return message of key ID

See it in action

← → ↻ ⓘ 127.0.0.1:8000/docs

FastAPI 0.1.0 OAS3

</openapi.json>

default

GET	/	Read Root
GET	/messages/	List Items
GET	/messages/{item_id}	Read Item
POST	/messages/{item_id}	Post Item

API DOC

127.0.0.1:8000/docs

Implementation

```
9
10     local_dict = {}
11
12
13     @app.get("/messages/")
14     def list_items():
15         return local_dict
16
17
18     @app.get("/messages/{item_id}")
19     def read_item(item_id: int):
20         return {'item_id': item_id, 'message': local_dict.get(item_id)}
21
22
23     @app.post("/messages/{item_id}")
24     def post_item(item_id: int, message: str):
25         local_dict[item_id] = message
26
```

Implemented in a few lines!

local_dict is for simple demo purpose

Type validation

← → ↻ ⓘ localhost:8000/messages/hello

```
{"detail": [{"loc": ["path", "item_id"], "msg": "value is not a valid integer", "type": "type_error.integer"}]}
```

**Useful Error Message when
type is incorrect**

<http://localhost:8000/messages/hello>

```
1  from fastapi.testclient import TestClient
2  from main import app
3
4  # Create a test client
5  client = TestClient(app)
6
7
8  ▶ def test_root():
9      result = client.get('/')
10     print(result.json())
11     assert result.status_code == 200
12     result_dict = result.json()
13     assert result_dict['Hello'] == 'World'
14
```

Testing —

Testing is simple too

Requests based testing

```
1 from typing import Optional
2 from pydantic import BaseModel
3
4
5 class Message(BaseModel):
6     message: str
7     meta: Optional[dict] = None
8
```

```
27
28 from schema import Message
29
30
31 @app.post("/messages/body/{item_id}")
32 def post_item_json(item_id: int, message: Message):
33     local_dict[item_id] = message
34
```

Body Parameter

Define a schema

Auto mapping of body to object

```

1 from fastapi import FastAPI
2 from fastapi import Depends, HTTPException
3 from fastapi.security import OAuth2PasswordBearer, OAuth2PasswordRequestForm
4
5
6 app = FastAPI()
7
8 oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")
9
10
11 @app.post("/token")
12 async def login(form_data: OAuth2PasswordRequestForm = Depends()):
13     if form_data.username != 'root' or form_data.password != 'password': # should replace with hashed password
14         raise HTTPException(status_code=400, detail="Incorrect username or password")
15     return {"access_token": 'mytoken', "token_type": "bearer"}
16
17
18 @app.get("/")
19 def read_root(token: str = Depends(oauth2_scheme)):
20     assert token == 'mytoken'
21     return {"Hello": "World"}

```

Authentication

A few lines to add basic authentication

Can easily integrate with any OAuth2 authentication flow

Authorize



default



POST

/token Login

GET

/ Read Root



Parameters

Cancel

No parameters

Execute

Clear

Authentication

Authentication in test interface

Usable authenticated test interface

Comparison

Key Benefits Over Vanilla Flask

01

PARAMETER VALIDATION

It's possible to use marshmallow in Flask, but it's very time consuming.

02

AUTO DOCUMENTATIONS GENERATION

It's also possible to generate docs for Flask API, but it needs extensive annotations.

03

MULTI-FILE ROUTER

No need to use another library.

A black and white photograph of a notebook, a pen, and a flower on a wooden surface. The notebook is open, showing a blank page. A pen lies diagonally across the bottom right. A flower is positioned near the top right. The background is a light-colored wooden surface.

Tips and Tricks



JSON Encoder

**Python default encoder generate
non-standard JSON**


Swapping implementation would solve most of these issues.

Async Support

```
1  from fastapi import FastAPI
2  import time
3  import asyncio
4
5  app = FastAPI()
6
7
8  @app.get("/benchmark1/")
9  async def benchmark():
10      await asyncio.sleep(1)
11      return {"message": "ok"}
12
13
14  @app.get("/benchmark2/")
15  def benchmark2():
16      time.sleep(1)
17      return {"message": "ok"}
18
```

Not as much benefits

Some benefits under high concurrency and long processing time. Not much differences under normal use case.

 FastAPI

Search

JSON Compatible Encoder

Body - Updates

Dependencies >

Security >

Middleware

CORS (Cross-Origin Resource Sharing)

SQL (Relational) Databases

Bigger Applications - Multiple Files

Background Tasks

Metadata and Docs URLs

Static Files

Testing

Debugging

Advanced User Guide >

SQL (Relational) Databases

FastAPI doesn't require you to use a SQL (relational) database.

But you can use any relational database that you want.

Here we'll see an example using [SQLAlchemy](#) [↔].

You can easily adapt it to any database supported by SQLAlchemy, like:

- PostgreSQL
- MySQL
- SQLite
- Oracle
- Microsoft SQL Server, etc.

Excellent
Documentations and
Worked Examples

With good production
worthy advice

<https://fastapi.tiangolo.com/>

Questions?

Code Available at Github

<https://github.com/christlc/pyconhk2020fastapi>