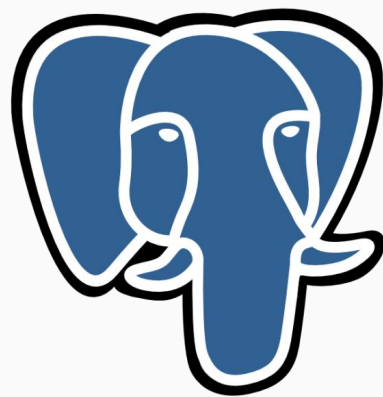


Wybrane technologie



PostgreSQL

Dlaczego Postgres?

Zdecydowaliśmy się na relacyjne podejście bazodanowe ponieważ, jest ono pewnym i dobrze ugruntowanym rozwiązaniem dla bardzo szerokiego zakresu systemów informatycznych. Zależało nam również, aby wybrane przez nas SZBD był open-source, był szeroko stosowany, miał dobre wsparcie społeczności i dużą bazę materiałów. W praktyce kryteria te zwężają wybór do dwóch rozwiązań. Ostatecznie wybór padł na Postgres-a, ponieważ część z nas ma z nim już doświadczenie.

Dlaczego Python?

Do tworzenia aplikacji komunikującej się z bazą danych wybraliśmy Python'a ze względu na jego wszechstronność. Każdy z członków naszej grupy miał z tym językiem mniejszą czy większą styczność podczas realizacji innych projektów. Klarowność oraz czystość pisanego w nim kodu przemawia za tym wyborem. Nawet w przypadku trudności jest stosunkowo łatwo trafić na wartościowy artykuł rozwijający wszelkie wątpliwości. Dodatkowo trzeba zaznaczyć, że język ten cieszy się wyjątkową popularnością przez co materiałów pozwalających rozwinąć własną wiedzę jest odpowiednio dużo.

To co sprawia, że takie połączenie się sprawdzi to m.in. **SQLAlchemy**.
Jest to rozbudowana biblioteka, która zapewnia komunikację z poziomu Python'a z Postgres'em oraz ORM.

```
from sqlalchemy import Column, String
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker
```

```
db_string = "postgres://admin:abc@xyz.com:15813/compose"
```

```
db = create_engine(db_string)
base = declarative_base()
```

```
class Film(base):
    __tablename__ = 'films'

    title = Column(String, primary_key=True)
    director = Column(String)
    year = Column(String)
```

```
Session = sessionmaker(db)
session = Session()
base.metadata.create_all(db)
```

```
# Create
```

```
doctor_strange = Film(title="Doctor Strange",
    director="Scott Derrickson", year="2016")
session.add(doctor_strange)
session.commit()
```

```
# Read
```

```
films = session.query(Film)
for film in films:
    print(film.title)
```

```
# Delete
```










```
session.delete(doctor_strange)
session.commit()
```

*Przykładowy kod (nie związany z projektem) mający na celu zilustrowanie prostoty komunikacji z poziomu języka Python z bazą PostgreSQL. Ma on na celu pokazanie motywacji naszego wyboru. Na kolejnych slajdach zostanie pokazany konkretny sposób (zgodnie potrzebami projektu) wraz z omówionym dalej FastAPI.

FastAPI - co skłoniło nas do tego wyboru?

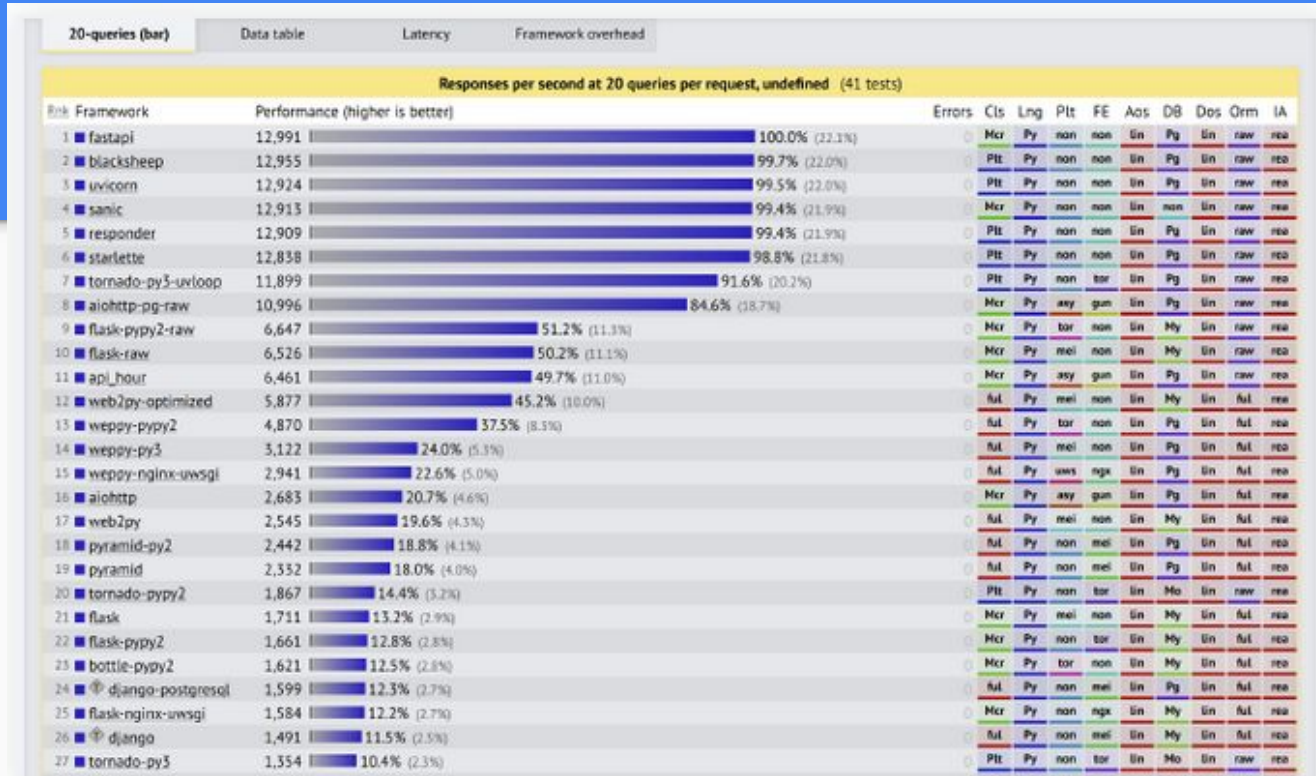
Dynamicznie rozwijający się projekt, który mimo że jest dość “młody” (2018) w porównaniu z innymi istniejącymi rozwiązaniami, przykładowo django (2003) czy flask (2010) zdążył zebrać już wielu zwolenników. Dane z Github’a:

- django
- flask
- fastapi

 Watch	2.3k	 Star	52.9k	 Fork	22.8k
 Watch	2.3k	 Star	52.4k	 Fork	13.8k
 Watch	407	 Star	22k	 Fork	1.5k

Prędkość

FastAPI nie jest
szybkie w działaniu
wyłącznie z nazwy.
Obok zamieszczamy
porównanie
względem innych
rozwiązań:



TechEmpower Benchmark Result

Źródło:

<https://livecodestream.dev/post/2020-08-05-quickly-develop-highly-performant-apis-with-fastapi-python/>

Chęć poznania nowej technologii

Nie mieliśmy do tej pory styczności z FastAPI. Pomysł by poznać ją w ramach tego przedmiotu wydał nam się ciekawy. Wiedza odnośnie tworzenia mikroserwisów oraz umiejętność zastosowania ich w połączeniu z bazami danych jest w naszej opinii niezwykle użyteczna oraz rozwijająca. Dodatkowym atutem jest poszerzenie naszego wachlarzu umiejętności na rynku pracy po pomyślnym zrealizowaniu tego projektu.

Łatwość podziału obowiązków

Dzięki architekturze mikroservisowej w łatwy sposób jesteśmy przydzielić między siebie zadania. Tak by każdy był odpowiedzialny za poszczególne serwisy (bądź część jednego większego - realizacja za pomocą APIRouter - pokazana obok)



```
from fastapi import APIRouter

router = APIRouter()

@router.get("/users/", tags=["users"])
async def read_users():
    return [{"username": "Foo"}, {"username": "Bar"}]

@router.get("/users/me", tags=["users"])
async def read_user_me():
    return {"username": "fakecurrentuser"}


@router.get("/users/{username}", tags=["users"])
async def read_user(username: str):
    return {"username": username}
```

Źródło: <https://fastapi.tiangolo.com/tutorial/bigger-applications/>

Bogactwo dokumentacji

FastAPI
[FastAPI](#)
Languages >
Features
Python Types Intro
Tutorial - User Guide >
Advanced User Guide >
Concurrency and async / await
Deployment
Project Generation - Template
Alternatives, Inspiration and Comparisons
History, Design and Future
External Links and Articles
Benchmarks
Help FastAPI - Get Help
Development - Contributing
Release Notes

FastAPI



FastAPI

FastAPI framework, high performance, easy to learn, fast to code, ready for production

Test passing coverage 100% pypi package v0.61.1 chat on github

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

Source Code: <https://github.com/tiangolo/fastapi>

Table of contents
Opinions
Typer, the FastAPI of CLIs
Requirements
Installation
Example
 Create it
 Run it
 Check it
 Interactive API docs
 Alternative API docs
Example upgrade
 Interactive API docs upgrade
 Alternative API docs upgrade
Recap
Performance
Optional Dependencies
License

Przykładowy serwis

Serwis testowy stworzony by sprawdzić poprawność konfiguracji Docker'a, Nginx oraz połączenia z bazą PostgreSQL. Posiada on wyłącznie jeden endpoint wyświetlający wszystkie stany z tabeli "us_states".

```
from fastapi import FastAPI
from databases import Database
import os

DATABASE_URI = os.getenv('DATABASE_URI')

database = Database(DATABASE_URI)

app = FastAPI(openapi_url="/api/test/openapi.json", docs_url="/api/test/docs")

@app.get('/api/test/get_states')
async def get_states():
    """Return all states stored in database."""
    return await database.fetch_all(query='SELECT * FROM us_states')

@app.on_event("startup")
async def startup():
    await database.connect()

@app.on_event("shutdown")
async def shutdown():
    await database.disconnect()
```

Użycie

Po uruchomieniu komendy “docker-compose up” w katalogu projektu możemy przejść pod adres w przeglądarce: <http://0.0.0.0:8080/api/test/docs> by zobaczyć dokumentację danego serwisu. Następnie po wybraniu endpointu “get_states” oraz przyciśnięciu “Try it out” i kolejno “Execute” otrzymamy odpowiedź z bazy wypisującą nam wszystkie stany.



default



GET

/api/test/get_states Get States

Return all states stored in database.

Parameters

Cancel

No parameters

Execute

Clear

Responses

Curl

```
curl -X GET "http://0.0.0.0:8080/api/test/get_states" -H "accept: application/json"
```



Request URL

```
http://0.0.0.0:8080/api/test/get_states
```

Server response

Code

Details

200

Response body

```
{
  {
    "state_id": 1,
    "state_name": "Alabama",
    "state_abbr": "AL",
    "state_region": "south"
  },
  {
    "state_id": 2,
    "state_name": "Alaska",
    "state_abbr": "AK",
    "state_region": "north"
  },
  {
    "state_id": 3,
    "state_name": "Arizona",
    "state_abbr": "AZ",
    "state_region": "west"
  }
}
```

Dziękujemy za uwagę!