DGTL Raiffeisen

PYCON RUSSIA

FastAPI как основной framework для python бекендов

Название надо было покороче придумать, но я был не особо адекватен, когда его придумывал…

Представлюсь

Вдруг это кому-то важно

— Меня зовут Аникин Денис

— team lead в команде Chat, Raiffeisenbank

— community lead в Python Community

— Увлекаюсь fullstack: разрабатываю на **python** и typescript

— Занимаюсь развитием Dev(Sec?)Ops практик

— Мой сайт: https://xfenix.ru/

— В айти давно, но пока следов выгорания нет

(зато есть обугливание)

Издалека: какие задачи у питонистов в банке?

— Пишем бекенды для разных частей банка во множестве команд

— Занимаемся многими автоматизациями процессов (CRM и разными другими вещами, о которых я не могу рассказать, но которые звучат не так скучно как просто «автоматизация»)

— Делаем внутреннее облако аля AWS (поменьше, конечно, мы не такие безосные)

— Активно развиваем бекенды вместе с DS (которые тоже во многом бекендеры-питонисты ☺️)

Кратко:
у нас много бекендов

А теперь перейдём к теме доклада

2018 год — каков выбор фреймворков?

Ещё были живы рэп батлы и на машины не накидывали «допов» за 500к+…

Django + DRF

Flask

Aiohttp +

много всего

И тут бац!

Появился

Что же такого интересного в FastAPI?

— Революю… ну ладно, очень очень свежий подход

— Аннотации типов (привет, HUG?)

— Универсальный подход к обработке всего «приходящего»

— Schema-oriented: openapi.json и отсюда Swagger, Redoc, Pydantic

— Инверсия зависимостей

— Совмещение асинхронного подсета языка с синхронным

— Скорость разработки и выполнения

Больше про аннотации типов

— Можно задавать обязательные параметры

— Автоматически валидируются данные

— Автоматически конвертируются данные

— Автоматически выдаются ошибки

— Важно всё, что пишется в аргументах «вьюх»

Как выглядят аннотация

Не слишком важно что здесь происходит (ничего хорошего)

```python
from fastapi import Cookie, FastAPI, Header, Path, Query
from pydantic import BaseModel


APP_OBJ: FastAPI = FastAPI()


class Item(BaseModel):
    name: str
    value: int


@APP_OBJ.get("/some_rest_like/")
async def read_item(
    path_param: str = Path( ... ),
    query_param: str = Query( ... ),
    header_param: str = Header( ... ),
    cookie_param: str = Cookie( ... ),
):
    return Item(name="Hello", value=10)

```

Что можно сказать про фреймворк в общем?

— Лучше всего подходит для HTTP based API

  REST, JSON over HTTP — точно. Возможно JSON RPC

— Подходит для Websocket приложения и GraphQL приложений

— FastAPI за вас делает множество вещей с помощью «магии».

  Многие не любят магию, но здесь её применение того стоит

Погружаемся в FastAPI

```python
from fastapi import FastAPI


APP_OBJ = FastAPI()
FAKE_ITEMS_DB: list[dict[str, str]] = [
    {"item_name": "Foo"},
    {"item_name": "Bar"},
    {"item_name": "Baz"}
]


@APP_OBJ.get("/items/")
async def read_item(skip: int = 0, limit: int = 10):
    return FAKE_ITEMS_DB[skip : skip + limit]


# http://127.0.0.1:8000/items/?skip=0&limit=10

```

Чуть сложнее

```python
import typing

from fastapi import Cookie, FastAPI, Header, Path, Query


APP_OBJ: FastAPI = FastAPI()


@APP_OBJ.get("/items/{item_id}")
async def read_items(
    q: str,
    what_is_this: list = Query([])
    item_id: int = Path( ... , title="The ID of the item to get"),
    user_agent: str | None = Header(None),  # converts - to _, also duplicates is ok
    ads_id: str = Cookie(None),
):
    results: list[str, str | int] = {"item_id": item_id}
    if q:
        results.update({"q": q})
    return results

```

Как это писать?

— Полагается на аннотации типов, аргументы функции и дефолтные значения

— Нужен query параметр? Возьми Query или можно задать прямо аргументом

— Нужен кусок пути? Возьми Path

— Нужен заголовок? Возьми Header

— Нужны cookie? Возьми Cookie

— Для json тела можно взять Body (+ магия) или Field

— Для форм есть Form

In grave need of best player opinion

Нас очень важно мнение лучшего игрока на этом рынке!

Что там у Django?

```python
from django.http import Http404, HttpResponse, JsonResponse


# urlpatters = [path('/items/<int:item_id>', views.something_unreal)]


def something_unreal(request, item_id: int) -> JsonResponse:
    q = request.GET.get('q')
    if q is None:
        raise Http404('Not found q param in GET')
    what_is_this = request.GET.getlist('what_is_this')
    if what_is_this is None:
        raise Http404('Not found what is this param in GET')
    ads_id = request.COOKIES.get('ads_id')
    if ads_id is None:
        raise Http404('Wrong ads id')
    user_agent = request.headers.get('user_agent')
    if user_agent is None:
        raise Http404('Wrong user agent')
    results: list[str, str | int] = {"item_id": item_id}
    if q:
        results.update({"q": q})
    return JsonResponse(results)

```

И к FastAPI — у нас тут строковая валидация

```python
from typing import Optional

from fastapi import FastAPI, Query

app = FastAPI()


@app.get("/items/")
async def read_items(
    q: Optional[str] = Query(
        None,
        alias="item-query",
        title="Query string",
        description="Query string for the items to search in the database that have a good match",
        min_length=3,
        max_length=50,
        regex="^fixedquery$",
        deprecated=True,
    )
):
    results = {"items": [{"item_id": "Foo"}, {"item_id": "Bar"}]}
    if q:
        results.update({"q": q})
    return results

```

Числовая валидация

```python
from fastapi import FastAPI, Path, Query

app = FastAPI()


@app.get("/items/{item_id}")
async def read_items(
    *,
    item_id: int = Path(..., title="The ID of the item to get", ge=0, le=1000),
    q: str,
    size: float = Query(..., gt=0, lt=10.5)
):
    results = {"item_id": item_id}
    if q:
        results.update({"q": q})
    return results
```

Сразу может быть не очень ясно

— q: Optional[str] = Query(None) — опциональный параметр

— q: Optional[str] = None — тоже самое

— q: str = Query(..., min_length=3) — обязательный параметр (Ellipsis)

Внезапно!

Немного о Pydantic

Лучшая библиотека валидации и настроек?

Что представляет из себя?

— библиотека для валидации данных и менеджмента настроек

— очень вербозные модели, основанные на аннотациях типов

(их могут читать даже те, кто не знаком с python совсем)

— 12 factor поклонники будут в восторге!

Валидация

```python
from datetime import datetime
from typing import List, Optional

from pydantic import BaseModel


class User(BaseModel):
    id: int
    name = 'John Doe'
    signup_ts: Optional[datetime] = None
    friends: List[int] = []


user_obj: User = User(
    **{
        'id': '123',
        'signup_ts': '2019-06-01 12:22',
        'friends': [1, 2, '3'],
    }
)
print(user_obj.id)
# > 123
print(repr(user_obj.signup_ts))
# > datetime.datetime(2019, 6, 1, 12, 22)
print(user_obj.dict())
"""
{
    'id': 123,
    'signup_ts': datetime.datetime(2019, 6, 1, 12, 22),
    'friends': [1, 2, 3],
    'name': 'John Doe',
}
"""

```

Комплексная иерархия

```python
from typing import List
from pydantic import BaseModel


class Foo(BaseModel):
    count: int
    size: float = None


class Bar(BaseModel):
    apple = 'x'
    banana = 'y'


class Spam(BaseModel):
    foo: Foo
    bars: List[Bar]


m = Spam(foo={'count': 4}, bars=[{'apple': 'x1'}, {'apple': 'x2'}])
print(m)
#> foo=Foo(count=4, size=None) bars=[Bar(apple='x1', banana='y'),
#> Bar(apple='x2', banana='y')]
print(m.dict())
"""
{
    'foo': {'count': 4, 'size': None},
    'bars': [
        {'apple': 'x1', 'banana': 'y'},
        {'apple': 'x2', 'banana': 'y'},
    ],
}
"""
```

Можно рекурсивно

```python
from pydantic import BaseModel


class Foo(BaseModel):
    a: int = 123
    #: The sibling of `Foo` is referenced by string
    sibling: 'Foo' = None


Foo.update_forward_refs()

print(Foo())
#> a=123 sibling=None
print(Foo(sibling={'a': '321'}))
#> a=123 sibling=Foo(a=321, sibling=None)

```

Причём здесь настройки?

Это просто что-то нереальное...

```python
from typing import Set

from pydantic import BaseModel, BaseSettings, Field, PostgresDsn, PyObject, RedisDsn


class SubModel(BaseModel):
    foo = 'bar'
    apple = 1


class Settings(BaseSettings):
    auth_key: str
    api_key: str = Field( ... , env='my_api_key')
    redis_dsn: RedisDsn = 'redis://user:pass@localhost:6379/1'
    pg_dsn: PostgresDsn = 'postgres://user:pass@localhost:5432/foobar'
    special_function: PyObject = 'math.cos'
    domains: Set[str] = set()
    more_settings: SubModel = SubModel()

    class Config:
        env_prefix = 'project_'


print(Settings().dict())
"""
{
    'auth_key': 'xxx',
    'api_key': 'xxx',
    'redis_dsn': RedisDsn('redis://user:pass@localhost:6379/1',
scheme='redis', user='user', password='pass', host='localhost',
host_type='int_domain', port='6379', path='/1'),
    'pg_dsn': PostgresDsn('postgres://user:pass@localhost:5432/foobar',
scheme='postgres', user='user', password='pass', host='localhost',
host_type='int_domain', port='5432', path='/foobar'),
    'special_function': <built-in function cos>,
    'domains': set(),
    'more_settings': {'foo': 'bar', 'apple': 1},
}
"""

```

Ещё это быстро (ну так говорят их бенчмарки)

И чего бы я вдруг им не доверял?

Benchmarks

Below are the results of crude benchmarks comparing *pydantic* to other validation libraries.

Package	Version	Relative Performance	Mean validation time
pydantic	`1.7.3`		93.7µs
attrs + cattrs	`20.3.0`	1.5x slower	143.6µs
valideer	`0.4.2`	1.9x slower	175.9µs
marshmallow	`3.10.0`	2.4x slower	227.6µs
voluptuous	`0.12.1`	2.7x slower	257.5µs
trafaret	`2.1.0`	3.2x slower	296.7µs
schematics	`2.1.0`	10.2x slower	955.5µs
django-rest-framework	`3.12.2`	12.3x slower	1148.4µs
cerberus	`1.3.2`	25.9x slower	2427.6µs

Вернемся к FastAPI

Выходные модели с Pydantic

```python
import typing
from enum import Enum

from fastapi import FastAPI
from pydantic import BaseModel


class ChatbotBases(str, Enum):
    mobile = "mobile"
    desktop = "desktop"
    ivr = "ivr"


class BaseAnswer(BaseModel):
    base_name: str
    message: typing.Optional[str]
    answers: typing.Optional[int]
    sleep: typing.Optional[int]
    fallback: typing.Optional[bool]


app = FastAPI()


@app.get("/chatbot/{base_name}/", response_model=BaseAnswer)
async def get_model(base_name: ChatbotBases):
    if base_name == ChatbotBases.mobile:
        return BaseAnswer(base_name=base_name, answers=10, sleep=20)
    elif base_name == ChatbotBases.desktop:
        return BaseAnswer(base_name=base_name, fallback=True)
    return {"base_name": base_name, "message": "Good choice"}

```

Конвертация всего во всё с помощью pydantic

```python
from typing import Optional

from fastapi import FastAPI
from pydantic import BaseModel, EmailStr


app = FastAPI()


class UserIn(BaseModel):
    username: str
    password: str
    email: EmailStr
    full_name: Optional[str] = None


class UserOut(BaseModel):
    username: str
    email: EmailStr
    full_name: Optional[str] = None


@app.post("/user/", response_model=UserOut)
async def create_user(user: UserIn):
    return user

```

Поддержка датаклассов

```python
from dataclasses import dataclass
from typing import Optional

from fastapi import FastAPI


@dataclass
class Item:
    name: str
    price: float
    description: Optional[str] = None
    tax: Optional[float] = None


app = FastAPI()


@app.post("/items/")
async def create_item(item: Item):
    return item
```

Удивительная обработка Body

```python
from typing import Optional

from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()


class Dialog(BaseModel):
    id: int
    revision: str
    message: str


class Meta(BaseModel):
    phone_number: str
    extra: typing.Optional[str]


@app.post("/items/{item_id}/")
async def update_item(item_id: int, dialog: Dialog, meta_params: Meta):
    return {"dialog": dialog, "meta": meta_params}


"""
{
    "dialog": {
        "id": 10,
        "revision": "uuid4",
        "message": "Hello"
    },
    "meta": {
        "phone_number": "+79991112233"
    }
}
"""

```

Зачем нужен Body?

Нужен для случая обработки тела запроса в виде JSON

Инверсия зависимостей

```python
from typing import Optional

from fastapi import Depends, FastAPI


app = FastAPI()
fake_items_db = [{"item_name": "Foo"}, {"item_name": "Bar"}, {"item_name": "Baz"}]


class CommonQueryParams:
    def __init__(self, q: Optional[str] = None, skip: int = 0, limit: int = 100):
        self.q = q
        self.skip = skip
        self.limit = limit


@app.get("/items/")
async def read_items(commons: CommonQueryParams = Depends()):
    response = {}
    if commons.q:
        response.update({"q": commons.q})
    items = fake_items_db[commons.skip : commons.skip + commons.limit]
    response.update({"items": items})
    return response

```

Больше про инверсию

— Подходит, например, для выделения любых общих кусков логики

— Можно организовать подключение внешних ресурсов, например БД

— Depends можно использовать и в функциях, и в декораторах, и в самом «приложении»

— Можно вообще всю логику писать через инверсию

Промежуточные итоги

Классные штуки фреймворка

— отличная документация (но есть куда улучшать)

— великолепный pydantic + pydantic settings (12factor app зовёт, берет переменные из .env файла/просто окружения, все с валидацией, парсингом, тайпкастом)

— легкая интеграция с шаблонным движком, graphql

— поддержка websocket

— поддержка почти всего http протокола

— высокая скорость работы

— middleware

— security

Классные штуки фреймворка 2

— поддержка JWT, Oauth 2

— возможность масштабирования

— для тех, кому мало стандартной инверсии зависимостей — сторонние проекты неплохо интегрируются, например, отличный фреймворк dependency injector

— BackgroundTasks

— за ним можно поставить джангу или фласк 🤷‍♂️🤷‍♂️🤷‍♂️

— разные варианты ответов, в т.ч. Streaming

— (свежак) SqlModel от автора фреймворка — pydantic + sql alchemy

Почему выбрали как основной

— мы довольно быстро пришли к мысли, что многие другие фреймворки медленнее в разработке, чем FastAPI

— комбинирование синхронного и асинхронного кода позволяет даже новичкам спокойно начинать работу с фреймворком без головной боли

— хорошая дока позволяет быстро погружаться

— очень очень быстро пишется код

— очень надёжно

— очень быстро работают бекенды

Спорный момент

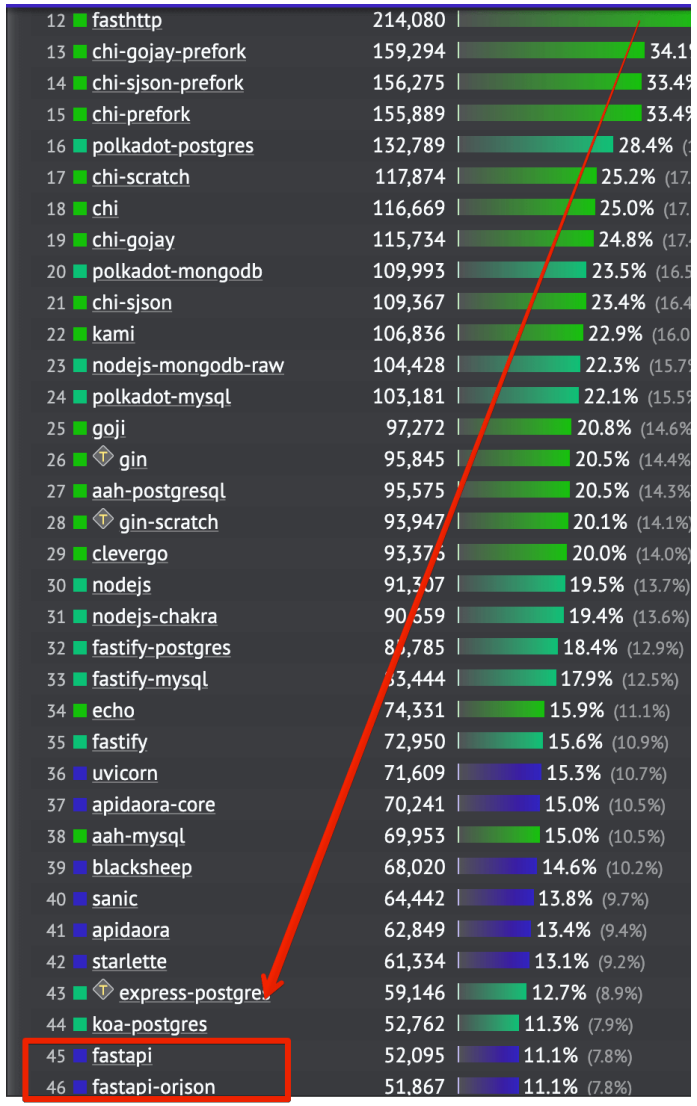
Самый быстрый фреймворк?

Цитата из документации FastAPI

> Fast: Very high performance, on par with **NodeJS** and **Go** (thanks to Starlette and Pydantic)

Не факт

Но фреймворк правда быстрый



Заметки из продакшена

Чего нам не хватает?

— нет версионирования

— нет ORM

— нет возможности генерить данные по pydantic моделям в тестах

— актуально для тех у кого много сервисов:

    — нет стандартного healtcheck

    — нет стандартного подхода к логгированию с request_id

Чего нам не хватает?

— нет версионирования <— написали пакет

— нет ORM <— взяли databases (сейчас автор завез SqlModel)

— нет возможности генерить данные по pydantic моделям в тестах <— не решено пока ☹

— актуально для тех у кого много сервисов:

— нет стандартного healtcheck <— написали пакет

— нет стандартного подхода к логгированию с request_id <— написали пакет

Опасайтесь


T.F.M.
RTFM!

❌ Не делайте так

— У uvicorn очень простой супервизор

— Он не умеет перезапускать после самоубийства воркеры

— Мы добавили limit-max-requests

— Мы не читатели, мы писатели: на доку мы забили

— В проде обнаружили количество рестартов подов > 9000

— Дебажили

— Дебажили

— Прочли доку

…

✅ Решение?

— Читать доку…

— На самом деле взяли gunicorn с воркером uvicorn ☺️

— Если вам нужен супервизор, это точно не uvicorn

❌ Не делайте так

В адрес вы можете нечаянно положить сенситивные данные.

Так вот потом в документации вы с удовольствием увидите как они спалились.

```python
MY_COOL_SECRET_TOKEN: str = '12321321312'


@app.put(f"/items/{MY_COOL_SECRET_TOKEN}")
async def update_item():
    ...
```

DGTL Raiffeisen

Спасибо. Вопросы?

https://github.com/xfenix

ad@xfenix.ru

xfenix.ru