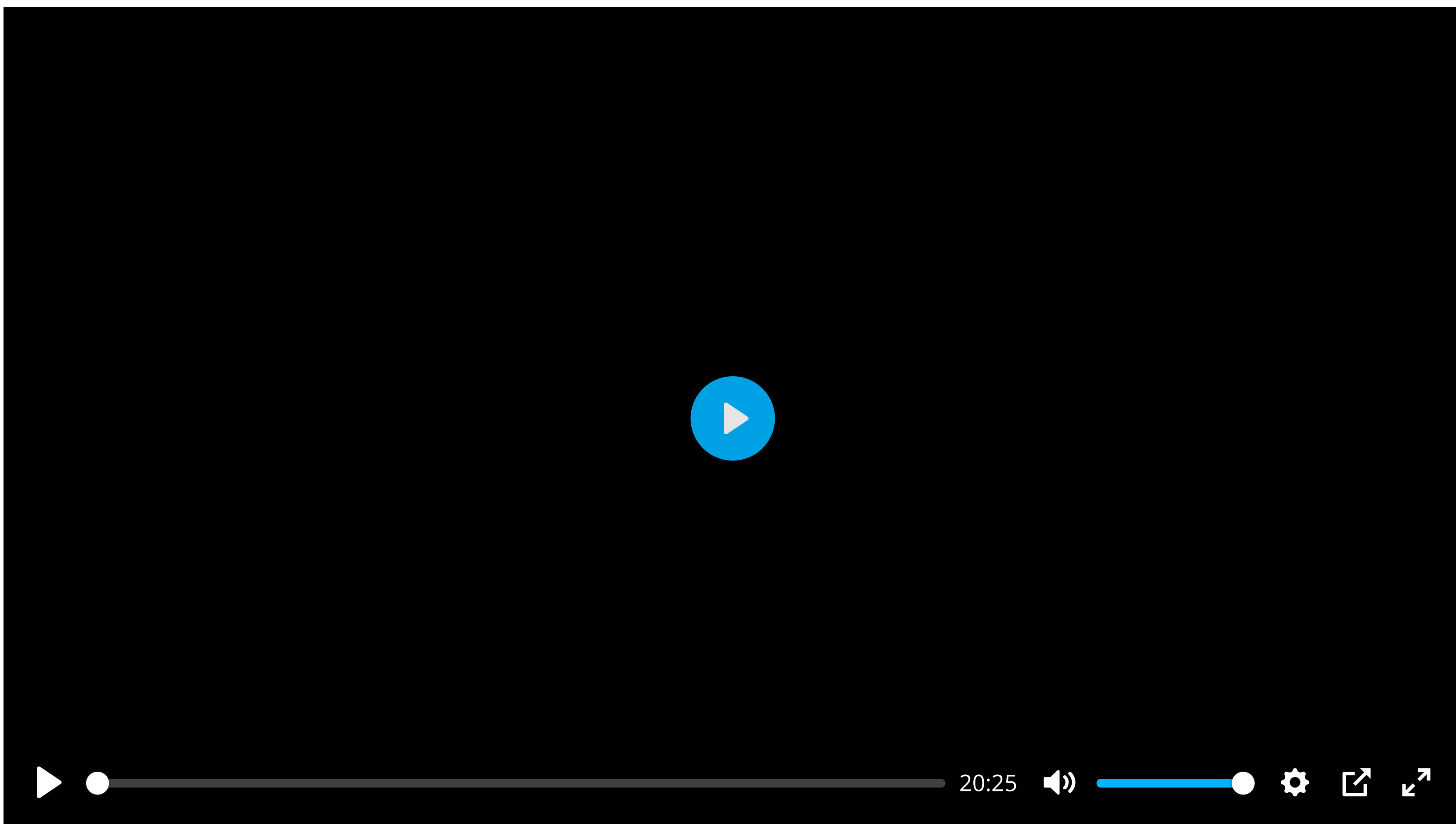


Текстовая расшифровка видео:

СОВРЕМЕННЫЕ АСИНХРОННЫЕ МИКРОСЕРВИСЫ

План:

- Что такое микросервис?;
- Кроулер для опроса URL'ов;
- Операции и эндпойнты;
- Глобальный реестр задач;
- Модель данных – Task;
- Отслеживание прогресса;
- Создание и запуск таска;
- FastAPI.

Что такое микросервис?

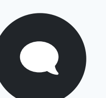
Микросервис – это сервис, который имеет внешний API и решает одну конкретную независимую небольшую задачу, часто со своей личной, независимой базой данных.

Основоположник – Крис Ричардсон.

Реализация:

На сегодняшний день микросервисы пишут на Python, GO, иногда на Java. Чаще всего их пишут в асинхронной манере из-за большого трафика.

Примером может послужить telegram-bot.



Кроулер для опроса URL'ов

Давайте попробуем представить, как в асинхронном мире может выглядеть кроулер на Python, для того чтобы опрашивать URL'ы.

Что делать?

- Необходим REST API;
- Необходимы таски и воркеры, так как мы будем общаться с внешними сервисами и, возможно, будем использовать Redis;

Получается довольно серьёзная система с большим количеством компонентов. Поэтому возьмём, во-первых, асинхронный Python; во-вторых, попробуем накидать минимальный экземпляр того, как это может выглядеть.

Спойлер: подобную задачу можно решить одним маленьким файлом на Python. В какой-то мере это будет продуктовым решением.

Рассмотрим код:

```
1. import asyncio
2. import uvloop
3. from aiohttp import web
4.
5.asyncio.set_event_loop_policy(uvloop.EventLoopPolicy())
6. app = web.Application()
7.
8. ...
9.
10. web.run_app(app, host=host, port=port)
11.
```

Для начала возьмем **aiohttp**. Он одновременно может выступать как сервером, так и клиентом. Соответственно, на нем можно писать не только приложения, но и асинхронные запросы. Добавим немного зависимостей, например, aiodns, а также используем uvloop (обертка вокруг libuv). Asyncio поддерживает нестандартные имплементации event loop'a. Далее, мы создаем экземпляр «Application» и сервим его.

Операции и эндпойнты

Для того, чтобы построить API нам потребуется несколько эндпойнтов.

Рассмотрим код:

```
1. def run_service(host, port):
2.   app.router.add_post('/submit', submit)
3.   app.router.add_get('/tasks', tasks)
4.   app.router.add_get('/tasks/{task_id}', task_status)
5.   app.router.add_delete('/tasks/{task_id}', task_delete)
6.   web.run_app(app, host=host, port=port)
7.
8.
9. if __name__ == "__main__":
10.    run_service("0.0.0.0", 7777)
```

Здесь есть три эндпойнта: submit для эндпоинта, куда мы кидаем URL'ы; /tasts – общая коллекция всех тасков. Мы можем посмотреть на конкретный таск при помощи /tasks - **/tast_id**. Также должен поддерживаться метод «delete». Далее, реализуем эндпоинты.

Глобальный реестр задач

В качестве базы, где будут храниться таски, используем встроенный в бэкенд словарь Python.

Делаем словарик **app[´TASKS´]**. Мы возвращаем json-ответ, которым проходимся по всем объектам «task» и для каждого возвращаем словарик, где лежит статус и текущий результат (если есть):

```
1. async def tasks(request):
2.    return web.json_response({
3.        k: {
4.            "status": v["status"],
5.            "result": v.get("result", {})
6.        } for k, v in request.app['TASKS'].items()
7.     })
```

Модель данных – Task

Сами по себе task'и выглядят следующим образом (как объекты):

```
1. {
2.     "status": "running",
3.     "workers": [],
4.     "result": {
5.     "https://example.com": {
6.          "timestamp": int(time.time()),
7.          "errors": {
8.              "http": 404
9.          }
10.         },
11.        "https://another-example.io": {
12.             "timestamp": int(time.time()),
13.             "errors": {
14.                 "http": "SSL_ERROR"
15.             }
16.         }
17.     }
19. }
```

- У них может быть статус: running, error, ready;
- У них может быть результат;
- У них может быть workers (список задач).

Данные объекты будут храниться в app[´TASKS´].

Отслеживание прогресса

```
1. async def task_status(request):
2.   task_id = request.match_info.get('task_id')
3.
4.   if not task_id or task_id not in request.app['TASKS']:
5.       return web.Response(status=404)
6.
7.   task = app['TASKS'].get(task_id)
8.   return web.json_response({
9.       "status": task.get("status"),
10.       "result": task.get("result", {})
11.   })



1. async def task_delete(request):
2.   task_id = request.match_info.get('task_id')
3.
4.   if not task_id or task_id not in request.app['TASKS']:
5.       return web.Response(status=404)
6.
7.   del app['TASKS'][task_id]
```

Это примитивная задача: мы проверяем task_id, если он присутствует, то возвращаем task в том же виде в API; в случае **delete** task сносим.

Создание и запуск таска

```
1. async def submit(request):
2.   data = await request.json()
3.   urls_count = len(data["urls"])
4.
5.   task_id = str(uuid.uuid4())
6.   request.app['TASKS'][task_id] = {
7.       "status": "running",
8.       "workers": [],
9.       "result": {}
10.   }
11.   logger.info(
12.       "Task '{0}' created: {1} URLs received".format(
13.           task_id,
14.           urls_count
15.       )
16.   )
17.
18.   asyncio.ensure_future(
19.       worker(task_id, data["urls"])
20.   )
21.
22.   return web.json_response({"task_id": task_id})
```

Здесь нам необходимо реализовать handler «submit», куда мы помещаем пачку URL. На бэкенде мы проходим по всем URL и возвращаем результат. Мы можем использовать обертку «a**syncio.ensure_future**».

Worker здесь – асинхронная функция, реализующая хождение по URL.

Главная идея: в асинхронном приложении мы можем запускать task'и, опрашивающие URL на том же ивент-лупе, что и сам бэкенд.

Итог: мы не можем рекомендовать сервисы, которые написаны подобным образом. Для продакшена лучше использовать внешний брокер и Swagger.

Наша цель – показать выгодные отличия асинхронного подхода перед синхронным.

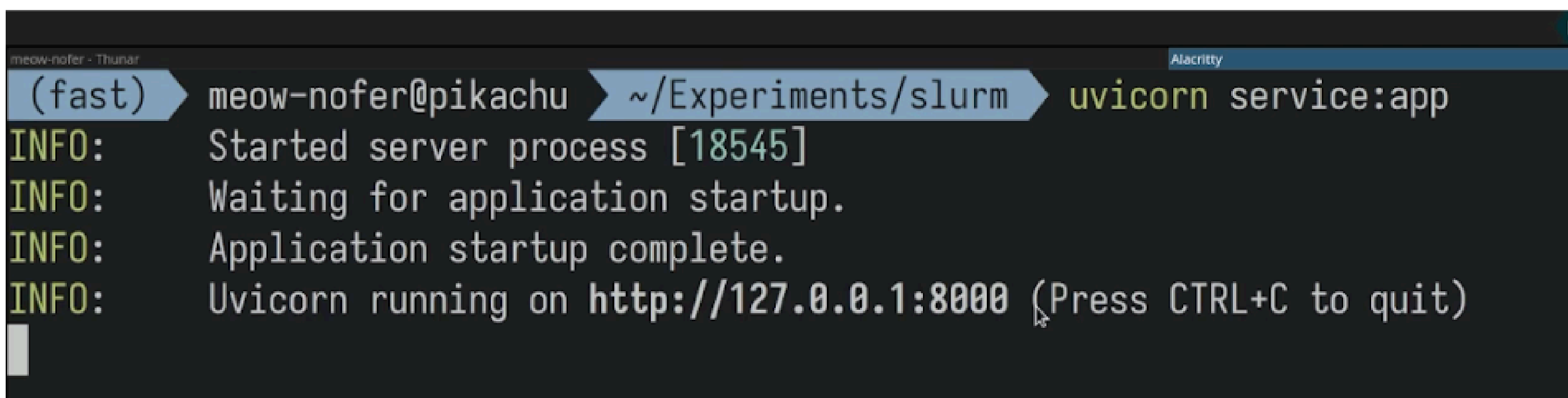
FastAPI

Если бы поступила задача написать асинхронный сервис на Python, мы бы взяли более современный фреймворк, например, FastAPI.

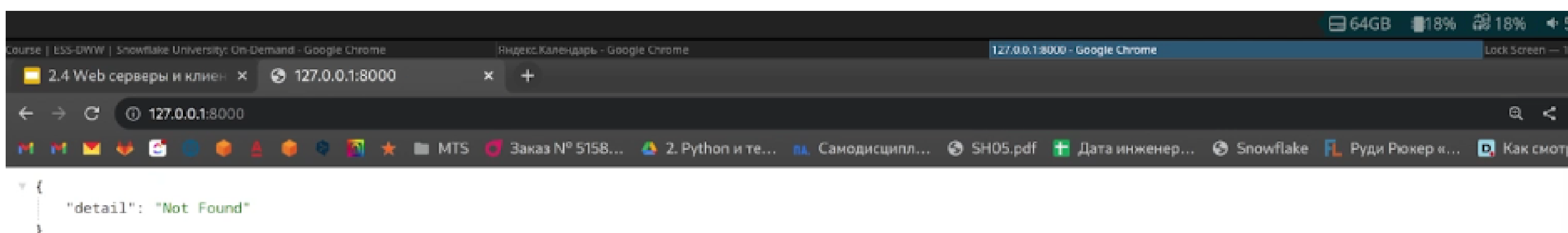
```
1. from fastapi import FastAPI, APIRouter
2. from pydantic import BaseModel
3.
4. app = FastAPI()
5. router = APIRouter(prefix="/api/v1")
6.
7. class Greeting(BaseModel):
8.   message: str
9.
10.
11. @router.get("/greeting", response_model=Greeting)
12. async def root():
13.   return Greeting(message="Hello, World!")
14.
15.
16. app.include_router
```

FastAPI – это асинхронный фреймворк для Python. Он делает маппинг URL'ов на handler и помогает добавлять сильную типизацию в код. Так, используя библиотеку «Pydantic», мы можем описывать структуру объектов, которые хотим передавать по API. Тогда из коробки автоматически начнет работать валидация.

Скопируем код сервиса и запустим через промежуточный application server.



Код запускается на порту 8000. Если мы пойдем на порт 8000, то нам вернется ошибка json. Это отличает этот подход и фреймворк от других:



Далее, мы используем специальный префикс. Поскольку мы реализуем API, стандартной практикой будет указание версии, используемой API в URL:

```
1. from fastapi import FastAPI, APIRouter
2. from pydantic import BaseModel
3.
4. app = FastAPI()
5. router = APIRouter(prefix="/api/v1")
6.
7. class Greeting(BaseModel):
8.     message: str
9.
10.
11. @router.get("/greeting", response_model=Greeting)
12. async def root():
13.     return Greeting(message="Hello, World!")
14.
15.
16. app.include_router(router)
```

Из интересного: FastAPI – один из тех фреймворков, умеющих забирать с собой Swagger из коробки. Например, если мы пойдем по /docs, то увидим Swagger.UI, а эндпоинт, который мы создадим будет также присутствовать:

No parameters

Execute | Clear

Responses

Curl

```
curl -X 'GET' \
  'http://127.0.0.1:8000/api/v1/greeting' \
  -H 'accept: application/json'
```

Request URL

```
http://127.0.0.1:8000/api/v1/greeting
```

Server response

Code	Details
200	Response body

```
{
  "message": "Hello, World!"
}
```

Download

Response headers

```
content-length: 27
content-type: application/json
```

Responses

Curl

```
curl -X 'GET' \
  'http://127.0.0.1:8000/api/v1/greeting' \
  -H 'accept: application/json'
```

Request URL

```
http://127.0.0.1:8000/api/v1/greeting
```

Server response

Code	Details
200	Response body

```
{
  "message": "Hello, World!"
}
```

Download

Response headers

```
content-length: 27
content-type: application/json
date: Fri,26 May 2023 15:22:08 GMT
server: uvicorn
```

Responses

Code	Description	Links

Это серьезный HTTP-клиент, который мы можем использовать в дальнейшем для разработки.

WSGI – стандарт по запуску синхронных приложений/бэкендов на Python, Django, Flask.

ASGI – другой популярный сервер. С ним вы можете подробнее ознакомиться с ASGI.

Как вам урок?

Далее

Слёрм ©

+7 (495) 248-05-80

Лицензия №ДЛ-1368 от 22.08.2019

Политика конфиденциальности

Публичная оферта