REST API, RPC

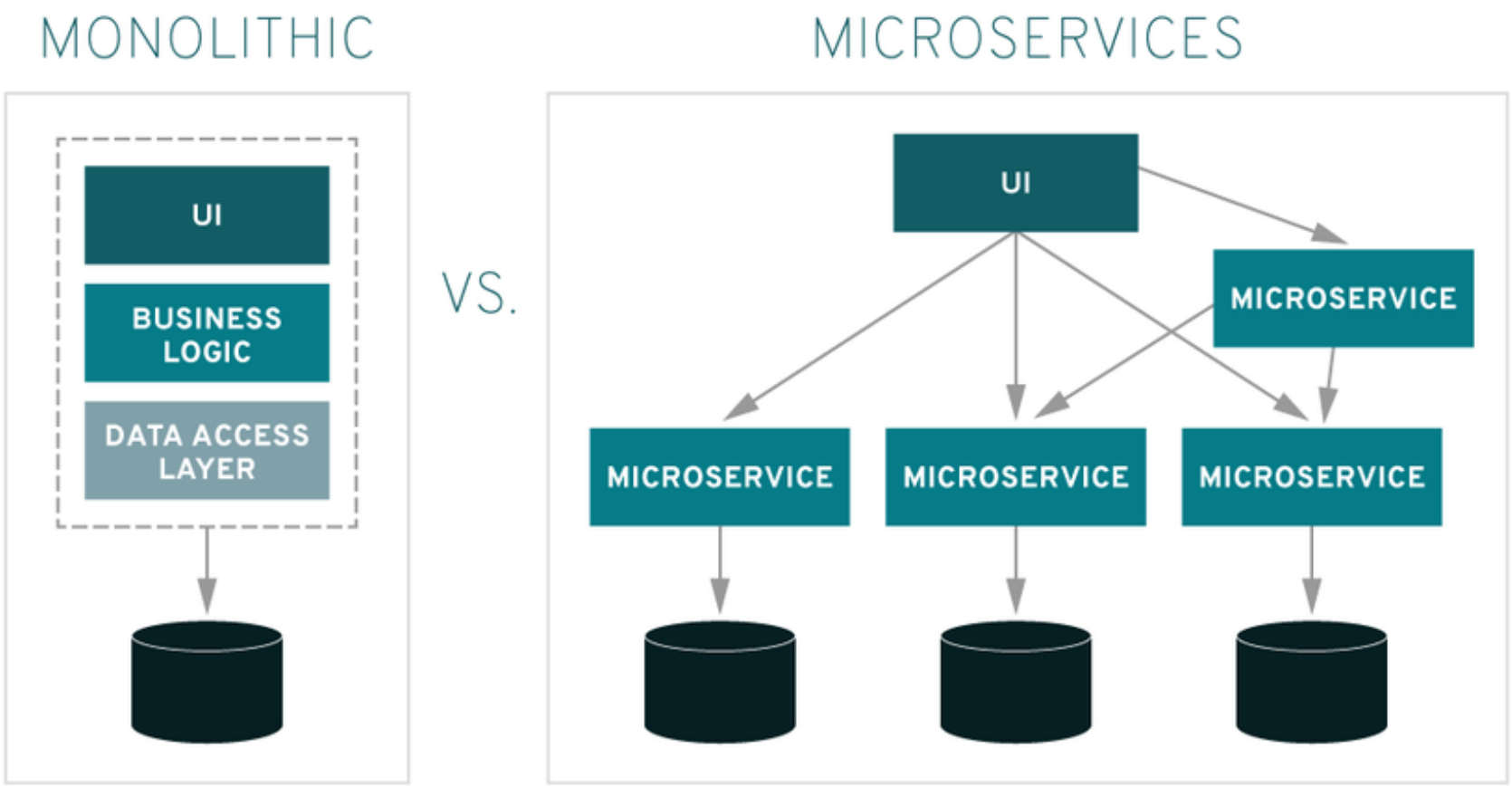
Борисенко Глеб

17.10.2024

Архитектура

- Монолит рождает сильных разработчиков.
- Сильные разработчики рождают микросервисы.
- Микросервисы рождают слабых разработчиков.
- Слабые разработчики рождают монолит.
- Монолит рождает…

Монолит и Микросервисы



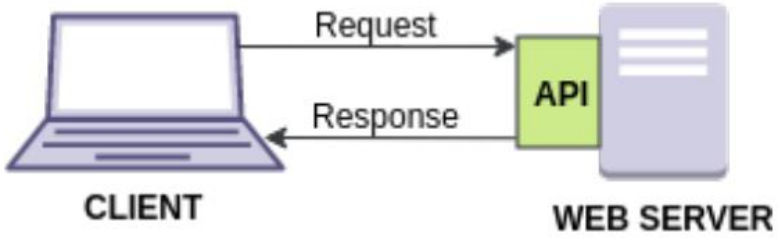
Архитектурный стиль микросервисов — это подход, при котором единое приложение строится как набор небольших сервисов, каждый из которых работает в собственном процессе и коммуницирует с остальными используя легковесные механизмы, как правило HTTP.

Общаются сервисы (точнее, раньше общались) обычно по REST API.

Сейчас все чаще переходят на gRPC для внутреннего API.

API

Application Programming Interface



Это способ коммуникации программных компонентов

Внутренние API (собственных библиотек) - для коммуникации микросервисов внутри приложения/компании

Внешние API (веб-сервисов) - позволяют получить доступ к сервису сторонним разработчикам через интернет, используя HTTP или другие протоколы.

REST

- Как расшифровывается? - **RE**presentational **S**tate **T**ransfer
- Что это? - Набор правил, которые позволяют разного рода системам обмениваться данными и масштабировать приложение. Соответствие этим правилам - нестрогое
- Как? - Используется HTTP протокол - прикладной уровень обмена данными по сети. Если при проектировании правила REST соблюдены - такой протокол называется *RESTful*
- REST это только то, что у нас “в голове”.

Ограничения, которые накладывает REST

1. **Модель клиент-сервер**
2. **Отсутствие состояния**
3. Кэширование
4. **Единообразие интерфейса**
5. Слои (промежуточные серверы)

CRUD

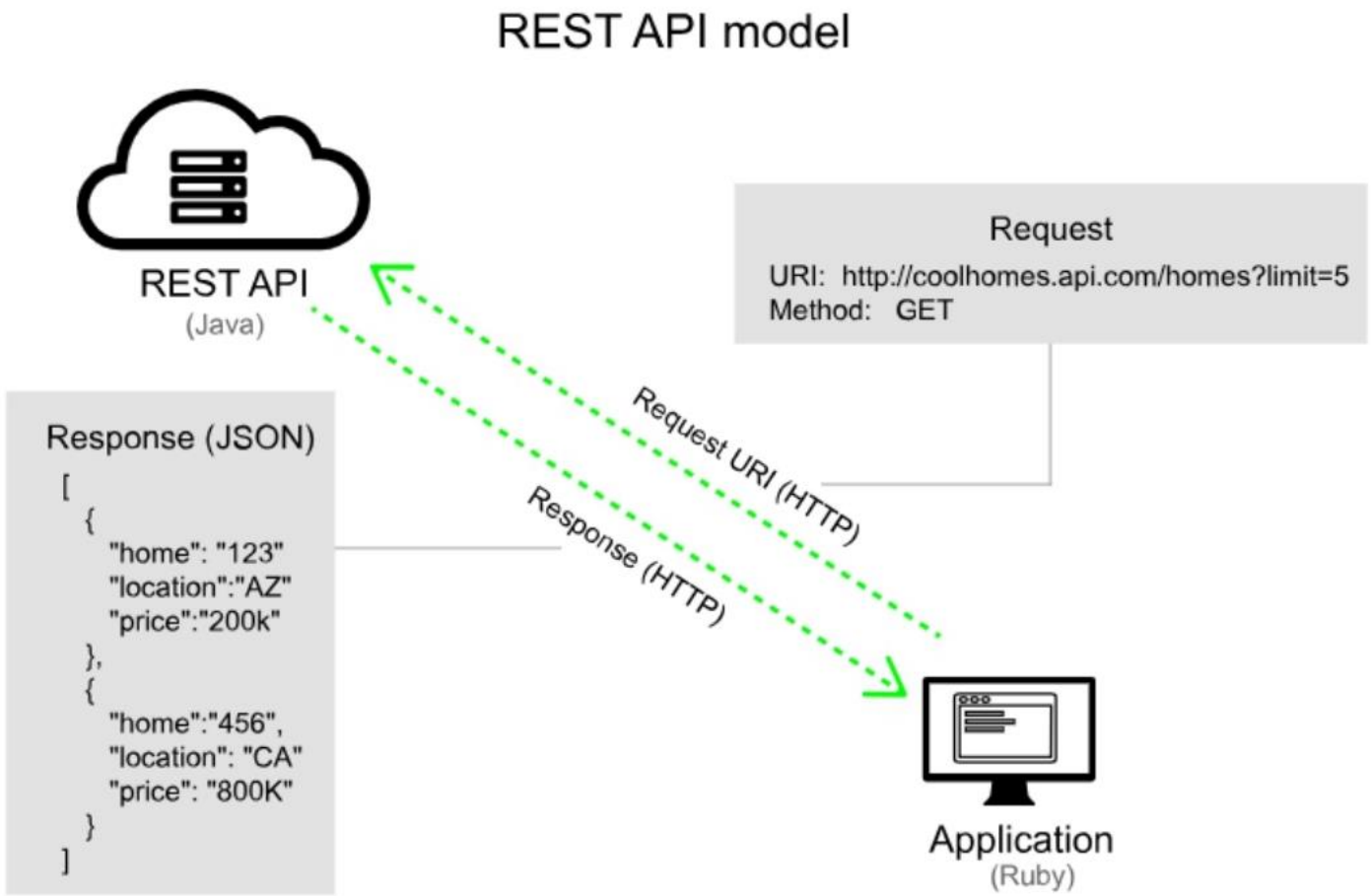
- RESTful API позволяет производить CRUD операции над всеми объектами, представленными в системе.
- **CRUD** - аббревиатура, которая описывает четыре базовых действия аббревиатура
  - C – create
  - R – read
  - U – update
  - D – delete

- Пример CRUD справа

Endpoint

- Конечная точка (endpoint) - это ресурс, расположенный на веб-сервере по определенному пути
- Endpoint приложения может выглядеть так:

- В REST API CRUD соответствуют *post*, *get*, *put*, *delete*. Ответ (Response) возвращается, как правило, в формате JSON или XML(реже).



- **get** - вернуть список объектов:

Request:

```
GET /api/train_samples
```

Response:

```
[
    {id:0, password: 'qwerty', times: 1601},
    {id:1, password: 'admin', times: 1920}
    ...
]
```

- **post** - добавить объект:

Request:

```
POST /api/train_samples/
```

Request object:

```
{password: '0000', times: 1000}
```

Response:

```
{id:9, password: '0000', times: 1000}
```

id - назначится сам

- **put** - обновить выбранную запись:

Request:

```
PUT /api/train_samples/1
```

Request object:

```
{id:1, password: 'admin', times: 2000}
```

Response:

```
{id:1, password: 'admin', times: 2000}
```

- **delete** - удалить выбранный объект:

Request:

```
DELETE /api/train_samples/1
```

Коды ответов

Код	Название	Описание
200	OK	Запрос выполнен успешно
201	Created	Возвращается при каждом создании ресурса в коллекции
204	No Content	Нет содержимого. Это ответ на успешный запрос, например, после DELETE)

Код	Название	Описание
400	Bad Request	Ошибка на стороне Клиента. Например, неправильный синтаксис запроса, неверные параметры запроса и т.д.
401	Unauthorized	Клиент пытается работать с закрытым ресурсом без предоставления данных авторизации
403	Forbidden	Сервер понял запрос, но отказывается его обрабатывать
404	Not found	Запрашивается несуществующий ресурс
405	Method Not Allowed	Клиент пытался использовать метод, который недопустим для ресурса. Например, указан метод DELETE, но такого метода у ресурса нет
500	Server error	Общий ответ об ошибкте на сервере, когда не подходит никакой другой код ошибки

Curl

- Что это? - Client URL, утилита командной строки
- Зачем? - позволяет выполнять запросы с различными параметрами и методами без перехода к веб-ресурсам в адресной строке браузера. Поддерживает протоколы HTTP, HTTPS, FTP, FTPS, SFTP и др.

Примеры запросов

- **Curl - GET запрос**

```
curl https://host.com
```

Метод GET - по умолчанию. Тот же результат получим, если вызовем так:

```
curl -X GET https://host.com
```

Чтобы получить ответ с заголовком:

```
curl https://host.com -i
```

Ответ будет содержать версию HTTP, код и статус ответа (например: HTTP/2 200 OK). Затем заголовки ответа, пустая строка и тело ответа.

- **Curl - POST запрос**

```
curl -X POST https://host.com
```

Используя передачу данных (URL-encoded):

```
curl -d "option=value_1&something=value_2"
     -X POST https://host.com/
```

Здесь -d или --data - флаг, обозначающий передачу данных

- **POST запрос, используя формат JSON**

```
curl -d '{"option": "val"}'
     -H "Accept:application/json"
     -X POST https://host.com/
```

Здесь -H или --header - флаг заголовока запроса на ресурс

Или можно передать json, как файл:

```
curl -d "@file.json"
     -X POST https://host.com/
```

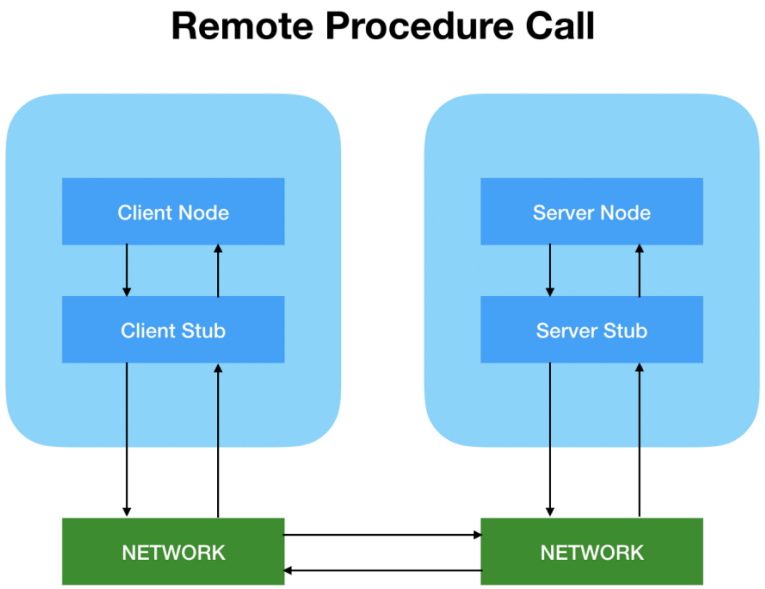
Swagger/OpenAPI

- Никто не будет лазить в код, чтобы посмотреть описание API.
- Общепринятым форматом для описания REST API на сегодняшний день является OpenAPI, который также известен как Swagger. Это по факту документация вашего API
- Спецификация представляет из себя единый файл в формате JSON или YAML, состоящий из трёх разделов:
  1. Заголовок, содержащий название, описание и версию API, а также дополнительную информацию;
  2. Описание всех ресурсов, включая их идентификаторы, HTTPметоды, все входные параметры, а также коды и форматы тела ответов;
  3. Определения объектов в формате JSON Schema, которые могут использоваться как во входных параметрах, так и в ответах.

Недостатки/особенности REST API:

- Для каждого языка необходимость разработки своего API. (Можно использовать Swagger - рассмотрим далее)
- JSON для передачи данных - не бинарный формат. Медленнее передача данных, но удобнее просматривать данные
- Протокол HTTP 1.1 - не поддерживает передачу потоковых данных
- Данные недостатки учтены в gRPC(Google Remote Procedure Call)

RPC – удаленный вызов процедур



- RPC - класс технологий, позволяющих программам вызывать функции или процедуры в другом адресном пространстве (на удалённых узлах, либо в независимой сторонней системе на том же узле).
- gRPC – одна из таких технологий (самая популярная сейчас)

gRPC

- Генерация кода стандартными средствами. Используется компилятор Protoc , который генерирует код для множетсва языков, включая python
- Бинарный формат данных Protobuf , использует сжатие -> быстрее передача данных
- Протокол HTTP 2 (2015 год) -> потоковая передача данных, бинарный формат, выше скорость и пр.
- Потоковая передача:
  - Один запрос – много ответов
  - Много запросов – один ответ
  - Много запросов – много ответов (двунаправленный поток)

Что выбрать?

- Если важна скорость - gRPC
- Если монолитное приложение с доступом извне или браузер - REST API
- Распределенная система на микросервисах - gRPC
- Потоковые данные (например, с датчиков) - gRPC

Flask + Gunicorn – зачем и почему

- Flask – Веб-фреймворк для Python
- Gunicorn – Python WSGI сервер

Flask

- Почему выбираем его по-умолчанию?
  - Минималистичный фреймворк
  - Быстрое прототипирование
  - Низкоуровневый фреймворк, после освоения будет проще разобраться с Django
- Также, лучшим решением будет выбрать Flask, если:
  - Разрабатывается микросерсисная архитектура
  - Реализуется REST API без фронтенда
  - Требуется гибкая кастомизация

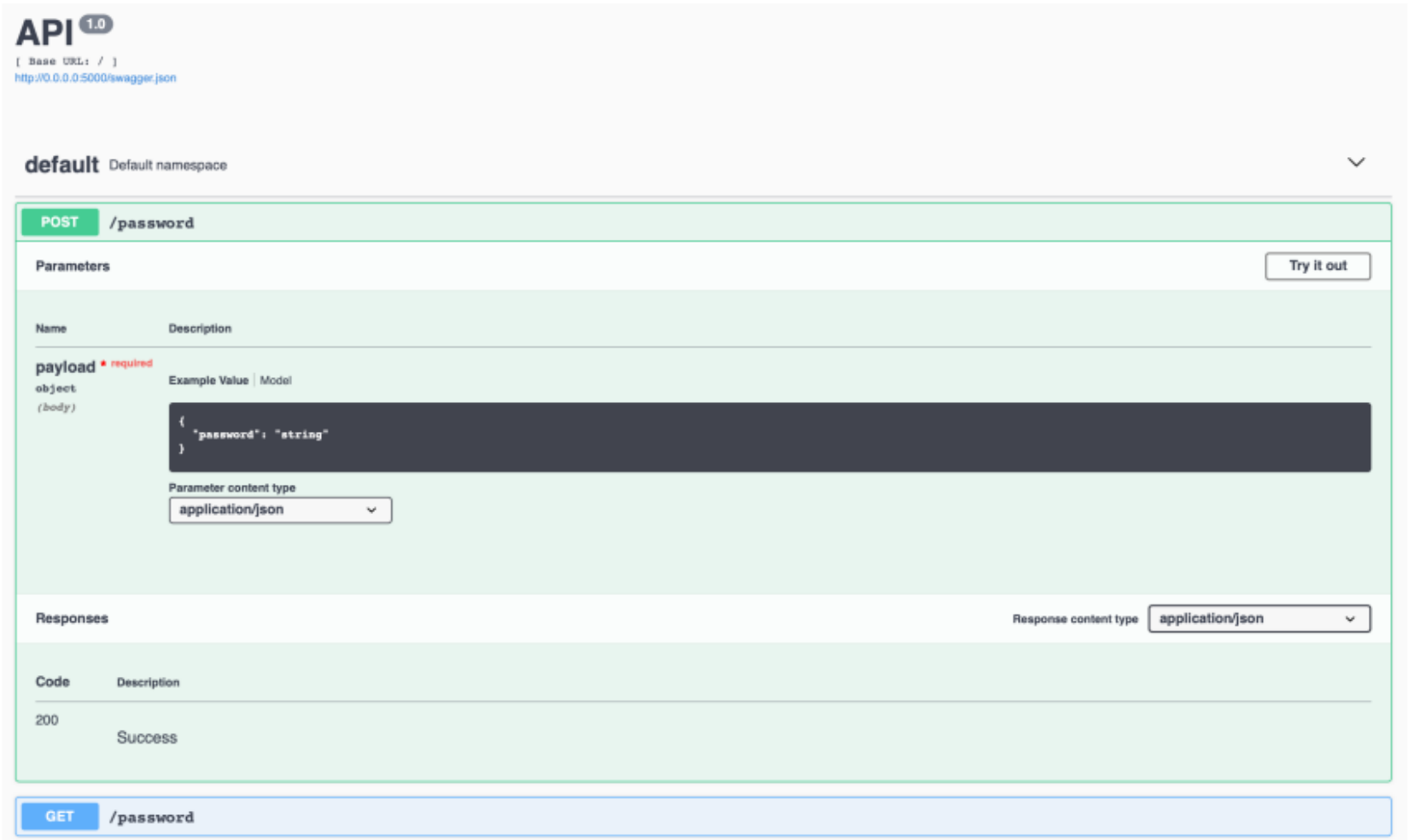
Минимальное Flask-приложение

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello, World!'

if __name__ == '__main__':
    app.run()
```

Flask API

- **Flask-RESTX** - это расширение для Flask, которое добавляет поддержку для быстрой разработки REST API. Форк flask-restplus
- Flask-RESTX предоставляет набор инструментов для генерации документации с использованием Swagger.

Документация **Swagger API** создается автоматически и доступна по корневому URL API:



Простой пример приложения, реализующий API на Flask:

```
3  from flask import Flask
4  from flask_restx import Api, Resource, fields
5
6  app = Flask(__name__)
7  api = Api(app)
8
9  passwords = []
10 a_password = api.model('Resource', {'password': fields.String})
11
12 @api.route('/password')
13 class Prediction(Resource):
14     def get(self):
15         return passwords
16
17     @api.expect(a_password)
18     def post(self):
19         passwords.append(api.payload)
20         return {'Result': 'pass added'}, 201
```

Gunicorn

- WSGI-сервер, то есть та самая штука, которая принимает HTTP запросы от клиента и передает их в питоновский код на фласке (например). И, соответственно, принимает от питоновского кода ответы и отправляет их клиенту.
- Синтаксис запуска крайне просто. Мы просто запускаем наш сервис на flask не питоном, а gunicron, то есть:
- gunicorn –bind 0.0.0.0:5000 my_app:app
- Где bind – адрес сервера и порта, my_app – название .py файла, а app – название нашего сервиса в нем.
- Есть куча вариантов для запуска и конфигурирования (например, автоматически распараллелить)

FastAPI + Uvicorn

- FastAPI – Веб-фреймворк для Python
- Uvicorn – Python ASGI сервер

FastAPI

- Почему выбираем его?
  - Асинхронный
  - Лучше документация
  - Проще поддерживать
  - Авто swagger
  - Куча фишичек дополнительных типа того же Dependency Injection
- Когда выбирать?
  - Всегда :3
  - Ну ладно, если вам нужно очень быстро, дальше поддерживать не будете, в FastAPI не шарите вообще, то и flask можно

Чем отличается от Flask?

- Типизация через Pydantic и встроенные штучки – валидация данных, генерация Swagger, проще поддерживать
- Асинхронный – быстрее будет работать сервер (если сами не налажаете)

Минимальное приложение на FastAPI

```python
from fastapi import FastAPI

app = FastAPI()


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

Чуть побольше

```python
from typing import Annotated

from fastapi import FastAPI, Path
from pydantic import BaseModel

app = FastAPI()


class Item(BaseModel):
    name: str
    description: str | None = None
    price: float
    tax: float | None = None


@app.put("/items/{item_id}")
async def update_item(
    item_id: Annotated[int, Path(title="The ID of the item to get", ge=0, le=1000)],
    q: str | None = None,
    item: Item | None = None,
):
    results = {"item_id": item_id}
    if q:
        results.update({"q": q})
    if item:
        results.update({"item": item})
    return results
```

А что по ответам и статусам?

```python
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()


class Item(BaseModel):
    name: str
    description: str | None = None
    price: float
    tax: float | None = None
    tags: list[str] = []


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


@app.get("/items/")
async def read_items() -> list[Item]:
    return [
        Item(name="Portal Gun", price=42.0),
        Item(name="Plumbus", price=32.0),
    ]
```

```python
@app.post("/items/", status_code=201)
async def create_item(name: str):
    return {"name": name}
```

```python
from fastapi import FastAPI, HTTPException

app = FastAPI()

items = {"foo": "The Foo Wrestlers"}


@app.get("/items/{item_id}")
async def read_item(item_id: str):
    if item_id not in items:
        raise HTTPException(status_code=404, detail="Item not found")
    return {"item": items[item_id]}
```

Dependency Injection

- Внедрение зависимостей – это способ для вашего кода объявлять вещи, которые он требует для работы (зависимости).
- Конкретно в FastAPI – это способ определять параметры вашей функции эндпоинта динамически/неявно

Как это выглядит

```python
from typing import Annotated

from fastapi import Depends, FastAPI

app = FastAPI()


async def common_parameters(q: str | None = None, skip: int = 0, limit: int = 100):
    return {"q": q, "skip": skip, "limit": limit}


@app.get("/items/")
async def read_items(commons: Annotated[dict, Depends(common_parameters)]):
    return commons


@app.get("/users/")
async def read_users(commons: Annotated[dict, Depends(common_parameters)]):
    return commons
```

Зачем нужно?

- Легко прикрутить всякие штучки для безопасности, авторизации и т.п.
- Легче использовать всякую разделяемую логику и писать код more SOLID
- И другие сложные схемы

Пример с БД

```python
# Dependency
def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()


@app.post("/users/", response_model=schemas.User)
def create_user(user: schemas.UserCreate, db: Session = Depends(get_db)):
    db_user = crud.get_user_by_email(db, email=user.email)
    if db_user:
        raise HTTPException(status_code=400, detail="Email already registered")
    return crud.create_user(db=db, user=user)
```

Middleware

```python
import time

from fastapi import FastAPI, Request

app = FastAPI()


@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
```

CORS

```python
from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware

app = FastAPI()

origins = [
    "http://localhost.tiangolo.com",
    "https://localhost.tiangolo.com",
    "http://localhost",
    "http://localhost:8080",
]

app.add_middleware(
    CORSMiddleware,
    allow_origins=origins,
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

Django

- Web-framework для написания бэка на питоне
- Весьма высокоуровневый, удобный
- Для API его будет использовать как пушкой по воробьям
- Но для пет-проджекта или в работе резко понадобится нормальное Web приложение быстро забабахть (такое бывает) – самое то
- У простых людей, не бэков, используется обычно такая связка:
  - Django
  - Bootstrap (один из шаблонов)
  - Jinja2 templating
  - Database (PostgreSQL for example)

В чем суть

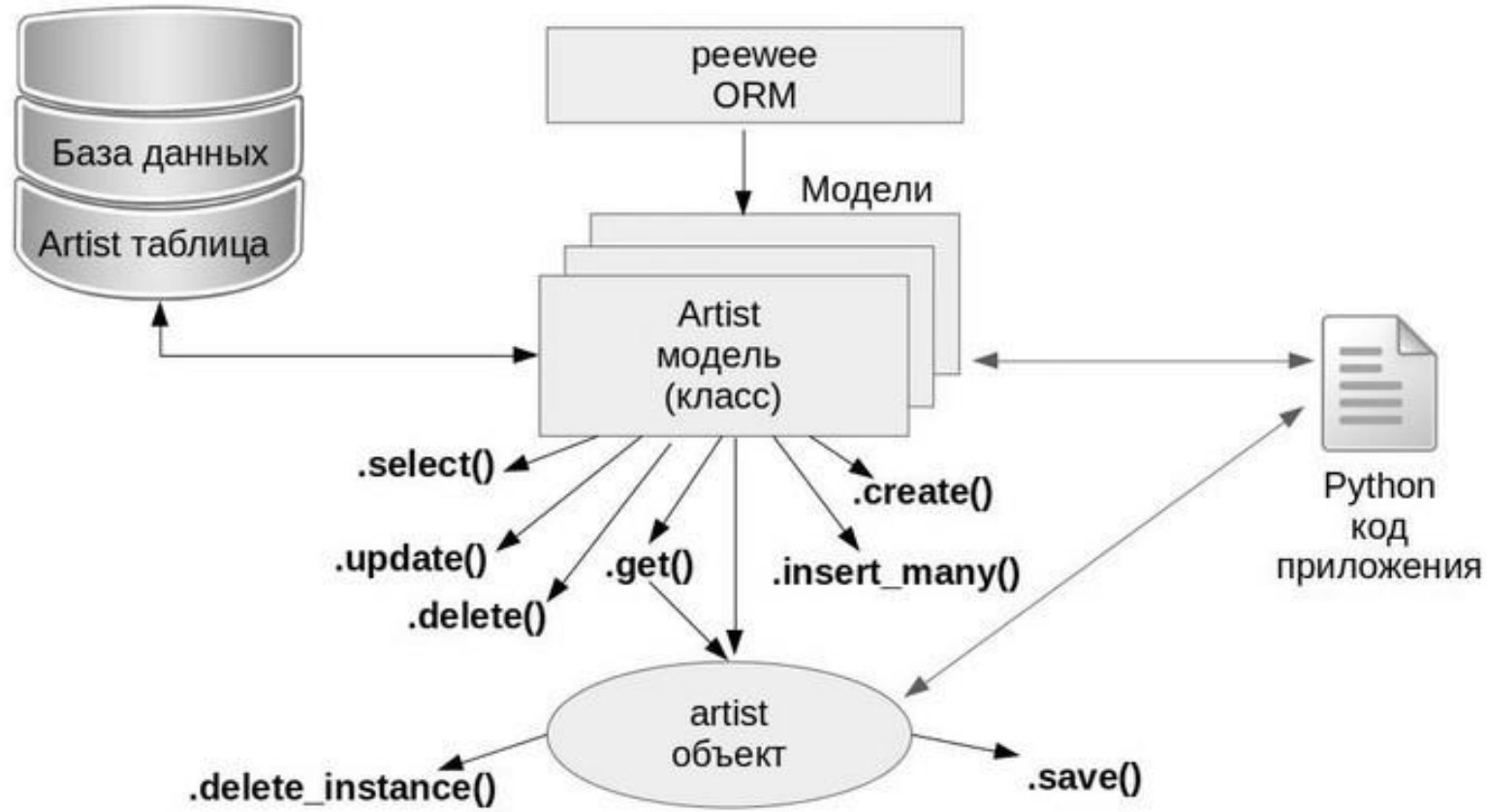
- Вы логику отображения контента пишите в виде view-шек
- В специальном файле urls.py связываете свои view-шки с нужными урлами (путями)
- Встроенная работа с БД через ORM и миграции
- Есть кайфовая админ панель из коробки

ORM

Это способ обращения к БД в коде. В чем суть:

- Каждая сущность/таблица/модель – это класс в питоне
- Атрибут сущности/колонка в таблице – это атрибут (параметр) класса в питоне
- Через методы этого класса мы общаемся с конкретной сущностью/таблицей/моделью
- В питоне чаще всего для этого используется SQLAlchemy

Схематично это выглядит так (при использовании либы peewee)



Как это выглядит в Django

```python
from django.db import models


class Question(models.Model):
    question_text = models.CharField(max_length=200)
    pub_date = models.DateTimeField('date published')


class Choice(models.Model):
    question = models.ForeignKey(Question, on_delete=models.CASCADE)
    choice_text = models.CharField(max_length=200)
    votes = models.IntegerField(default=0)
```

```
q = Question(question_text="What's new?", pub_date=timezone.now())
q.save()
q = Question.objects.get(pk=1)
```

А как это выглядит при прямом использовании DB-API (SQLite3)

```
import sqlite3
con = sqlite3.connect("tutorial.db")
cur = con.cursor()
res = cur.execute("SELECT name FROM sqlite_master")
res.fetchone()
```

Чувствуете разницу, да?

- С ORM работать удобнее и гибче.
- Но есть вопросики по оптимизации – нередко, при работе через ORM посылается сильно больше запросов, чем будет посылаться при вашем тупом и прямом «SELECT»

- Зачем это вам?
- Вы высоко вероятно будете работать с БД из кода

gRPC

- Описываем формат сообщений и процедуры сервера в proto-файле
- Компилируем этот файл
- Используем в своем коде скомпилированные на прошлом этапе классы

Пример proto-файла

```
// The greeting service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}
  // Sends another greeting
  rpc SayHelloAgain (HelloRequest) returns (HelloReply) {}
}

// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings
message HelloReply {
  string message = 1;
}
```

Компилируем его

- python -m grpc_tools.protoc -I../../protos --python_out=. --grpc_python_out=. ../../protos/helloworld.proto
- Будет два файла:
  - helloworld_pb2.py – классы для request-а и response-а
  - helloworld_pb2_grpc.py – классы клиента и сервера
- Эти файлы должны быть и на клиенте, и на сервере

Используем

- Сервер

```
class Greeter(helloworld_pb2_grpc.GreeterServicer):

  def SayHello(self, request, context):
    return helloworld_pb2.HelloReply(message='Hello, %s!' % request.name)

  def SayHelloAgain(self, request, context):
    return helloworld_pb2.HelloReply(message='Hello again, %s!' % request.name)
...
```

- Клиент

```
def run():
    with grpc.insecure_channel('localhost:50051') as channel:
        stub = helloworld_pb2_grpc.GreeterStub(channel)
        response = stub.SayHello(helloworld_pb2.HelloRequest(name='you'))
        print("Greeter client received: " + response.message)
        response = stub.SayHelloAgain(helloworld_pb2.HelloRequest(name='you'))
        print("Greeter client received: " + response.message)
```

Запуск сервера

```
def serve():
    port = '50051'
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
    helloworld_pb2_grpc.add_GreeterServicer_to_server(Greeter(), server)
    server.add_insecure_port('[::]:' + port)
    server.start()
    print("Server started, listening on " + port)
    server.wait_for_termination()
```