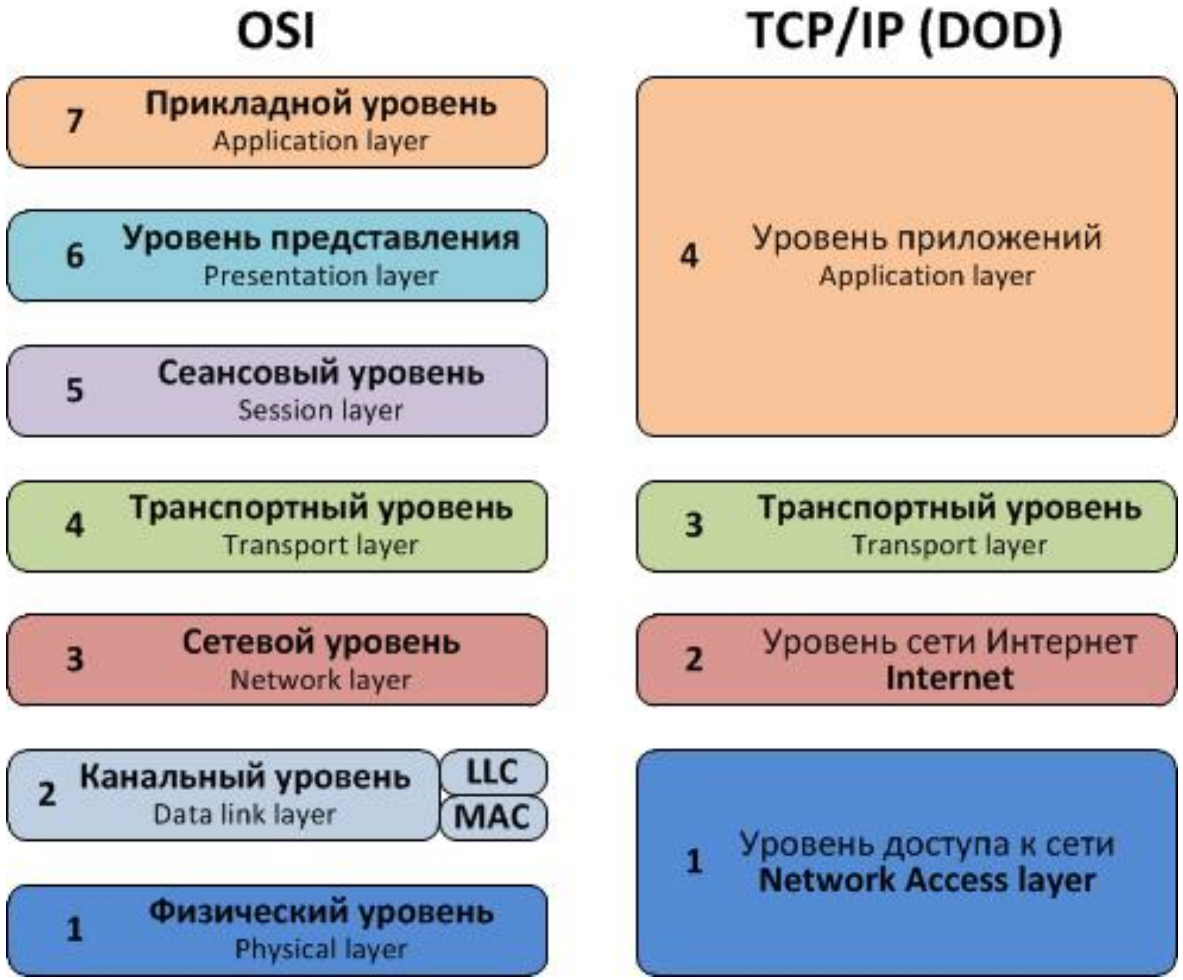
REST API, RPC

Борисенко Глеб

ФТиАД2021

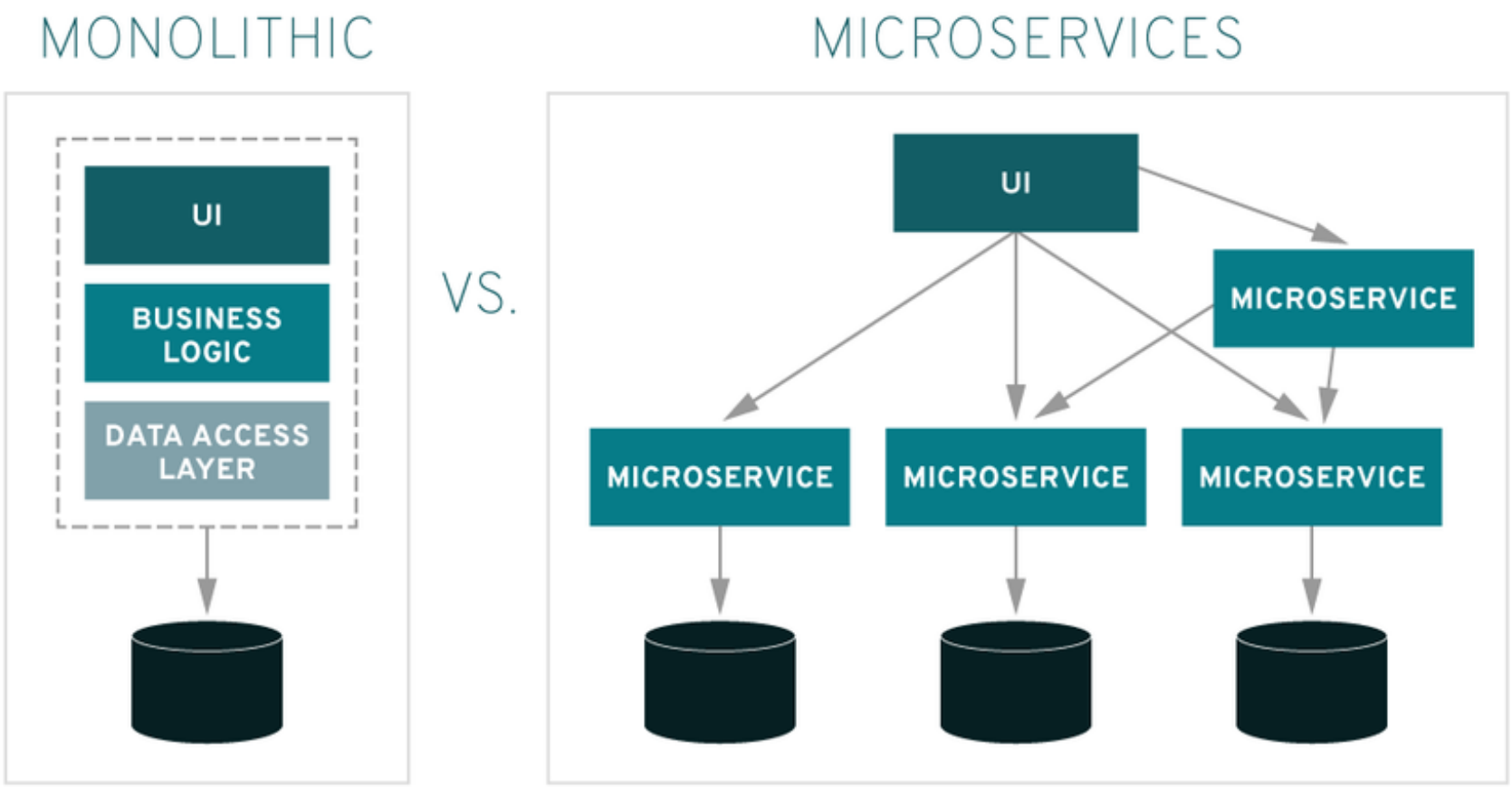
О чем говорили в прошлый раз



Архитектура

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

Монолит и Микросервисы



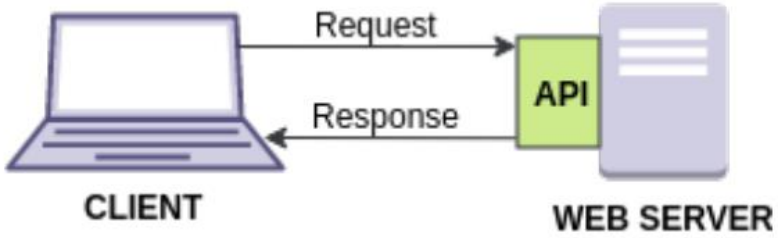
Архитектурный стиль микросервисов — это подход, при котором единое приложение строится как набор небольших сервисов, каждый из которых работает в собственном процессе и коммуницирует с остальными используя легковесные механизмы, как правило HTTP.

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

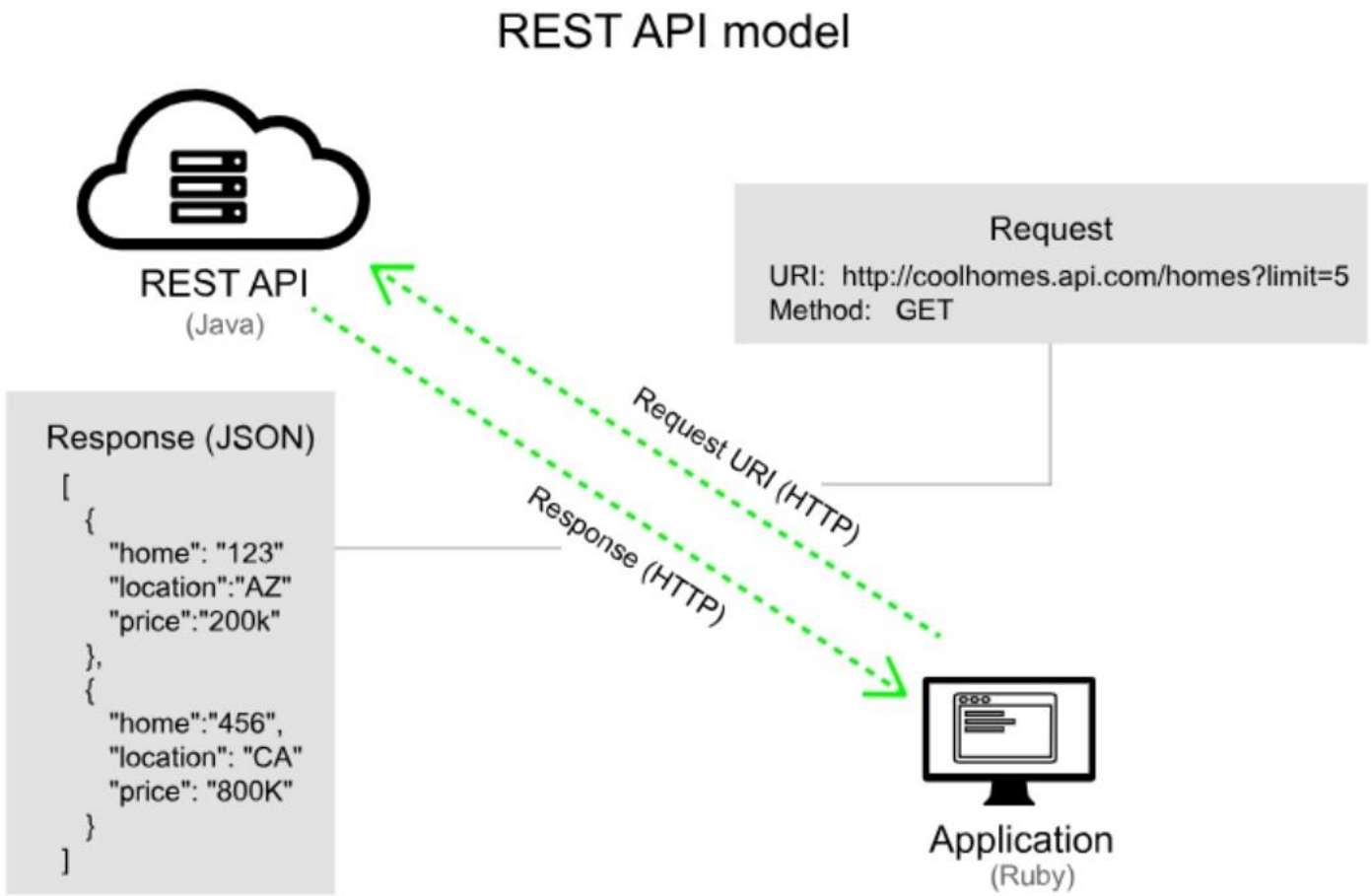
- RESTful API позволяет производить CRUD операции над всеми объектами, представленными в системе.
- **CRUD** - аббревиатура, которая описывает четыре базовых действия аббревиатура
  - C – create
  - R – read
  - U – update
  - D – delete

- 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/
```

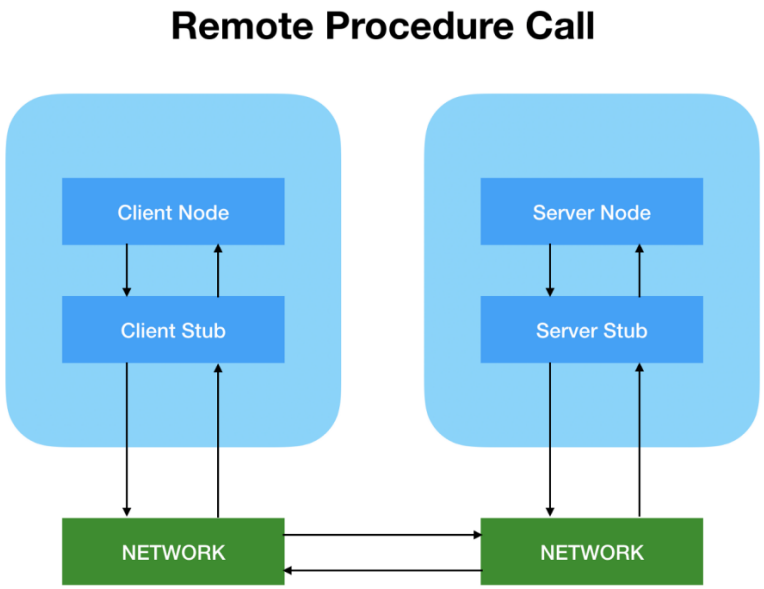
Swagger/OpenAPI

- 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
- Gunicron – 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

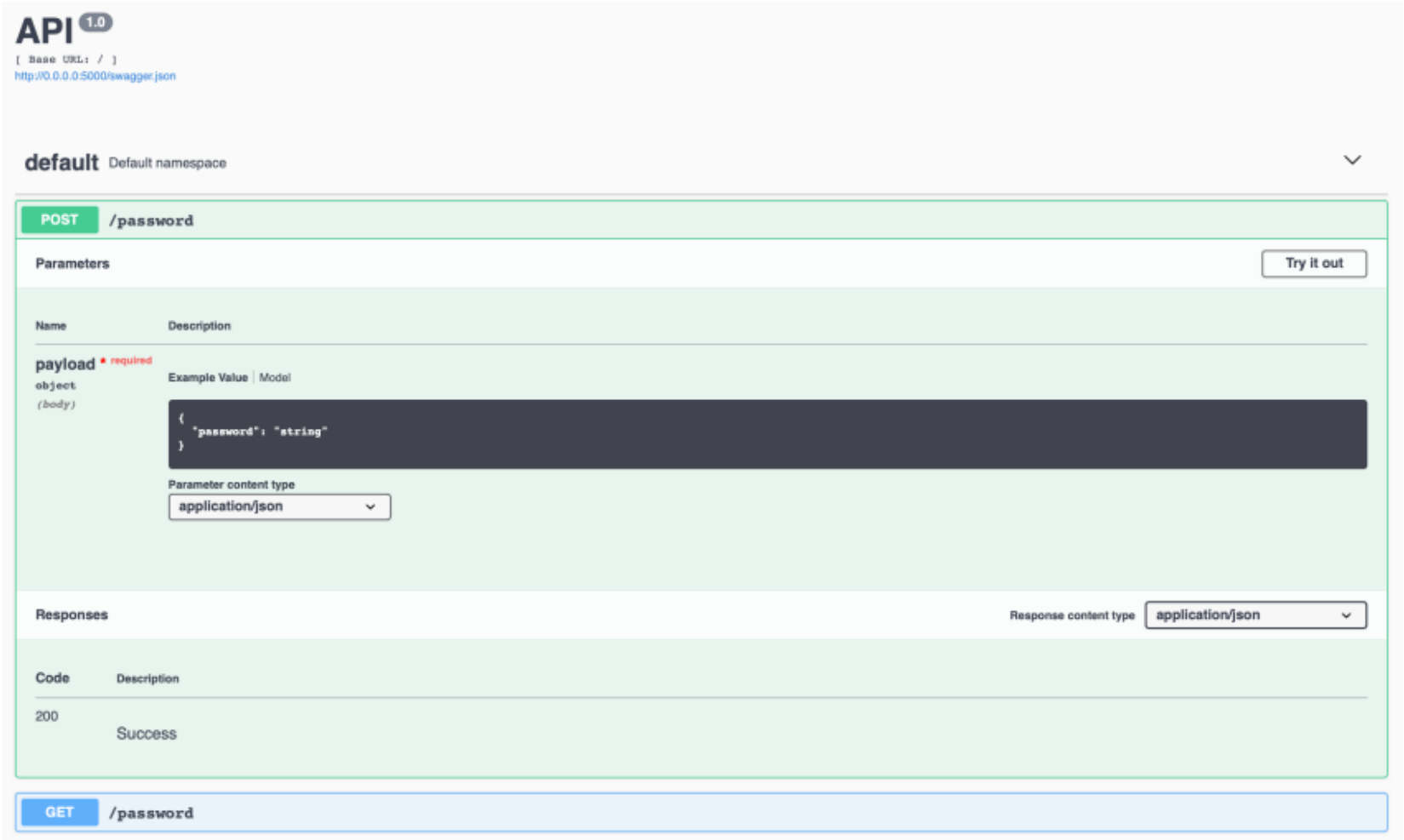
Минимальное Flask-приложение

```
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)

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()
```

Nginx

- nginx [engine x] — это HTTP-сервер и обратный прокси-сервер, почтовый прокси-сервер, а также TCP/UDP прокси-сервер общего назначения
- Функциональности – до жопы
- Используется много где для высоконагруженного прода
- Нас интересует сейчас конкретно балансировка нагрузки
- Подробнее тут: https://nginx.org/ru/

Зачем он нужен

- Клиентов много – один сервер не справляется. Что делать?
- Добавить больше серверов!
- Но как к ним направлять запросы? По какому принипу?
- Поднимаем Nginx! Шлеп запросы ему, он сам по выбранному алгоритму пересылает их на нужный сервер

Пример конфигурации Nginx (вся его суть в файлах конфигурации)

```
http {
    upstream app{
        server 10.2.0.100 weight=5;
        server 10.2.0.101 weight=3;
        server 10.2.0.102 weight=1;
    }

    server {
        listen 80;

        location / {
            proxy_pass http://app;
        }
    }
}
```

Всё

Смотрите в следующей серии:

- HTML, CSS
- Шаблон, Bootstrap
- Django, ORM
- Streamlit