Лекция 8

Преподаватель: Евгений Володин

Итоги:

1. **API — лицо продукта**
2. **Нужно понимание технического аспекта, понимать, когда какой API применить и какие у него ограничения**
3. **Нужно следить за метриками**
4. **И быть аккуратными при изменении API**
5. **API должен быть максимально удобен пользователям, чтобы они его боготворили**

Бизнес логика и API

- Что такое API?

API — **Application Programming Interface** — это черный ящик, мы не знаем что происходит внутри, а только подаем на вход данные и получаем ответную реакцию/результат

Нужно понимать, что API можно применять даже при реализации архитектуры ввиде сервисов. API будет центральным узлом для взаимодействия между сервисами

- Зачем нужен API?

Повторное использование функциональности (VK Pay API)
Упрощение интеграции с внешними сервисами (Yandex Maps API)
Обеспечение модульности (микросервисы)
Сокращение времени разработки (встраивание VK Video API в сайт)
Унификация взаимодействия (все REST похожи - легко научиться)
Безопасность (ограничение прав)
Расширение функционала экосистемы (VK mini aps, telegram)

- Виды API

PRIVATE — внутри организации (менее строгие требования, легче всего применять)
PUBLIC — открытые(по контрактам)
PARTNER — для партнеров (похожа на публичные, но по запросу партнера открывается доступ)

- Виды API

Web API — Браузер/ Микросервисы (в основном сегодня про это !!!)
Библиотечные API — библиотеки OS/ библиотеки для языков программирования

Архитектуры API

REST — лучший выбор для **публичных API** и веб-приложений (простота, кеширование).

gRPC — идеален для **микросервисов** и высоконагруженных систем (бинарный формат, стримы).

WebSocket — для **реалтайм-приложений** (чаты, уведомления, игры).

GraphQL — если нужна **гибкость** в запросах данных (например, мобильные API с разными требованиями к данным).

SOAP — почти устарел, но используется в **корпоративных системах** (банки, госучреждения).

Rest

Пример: Получение данных пользователя через HTTP GET.

Запрос:

```http
GET /users/123 HTTP/1.1
Host: api.example.com
Accept: application/json
```

Ответ (JSON):

```json
{
  "id": 123,
  "name": "Alice",
  "email": "alice@example.com"
}
```

REST — это набор правил общения, например: правила именования URL /users/123 …

Особенности

- Правила для URL
- Есть протокол HTTP для взаимодействия
- Формат получаемых данных: JSON

gRPC — Remote Proccess Calling

- сделал Google/ Похожа на REST
- Запрос может вызывать множественные ответы
- Множественные запросы один ответ
- **Плюсы:** Он бинарный и из-за этого достаточно быстрый и быстрее текстового формата
- У него строгая типизации
- Поддержка Real time общения между клиентом и сервером

gRPC

Файл `user.proto`:

```protobuf
syntax = "proto3";

service UserService {
  rpc GetUser (UserRequest) returns (UserResponse);
}

message UserRequest {
  int32 id = 1;
}

message UserResponse {
  int32 id = 1;
  string name = 2;
  string email = 3;
}
```

Клиент (Python):

```python
import grpc
from user_pb2 import UserRequest
from user_pb2_grpc import UserServiceStub

channel = grpc.insecure_channel("localhost:50051")
stub = UserServiceStub(channel)
response = stub.GetUser(UserRequest(id=123))
print(response.name)  # "Alice"
```

* здесь очень важен порядок

WebSocket

- для **RealTime**

Пример:

Чат → Сообщение → доставка пользователю (Ждем ответа)

Нужно делать *Поулинг - раз в несколько секунд стучаться на сервер(Есть ли что-то для меня?)

Минус: мусорный трафик

Клиент (JavaScript):

```javascript
const socket = new WebSocket("wss://chat.example.com");

// Отправка сообщения
socket.send(JSON.stringify({ user: "Alice", text: "Hello!" }));

// Получение сообщений
socket.onmessage = (event) => {
  console.log("New message:", JSON.parse(event.data));
};
```

Сервер (Node.js):

```javascript
const WebSocket = require("ws");
const wss = new WebSocket.Server({ port: 8080 });

wss.on("connection", (ws) => {
  ws.on("message", (message) => {
    // Рассылка сообщения всем клиентам
    wss.clients.forEach(client => client.send(message));
  });
});
```

GraphQL

***** более гибкая

- когда нет стандарта отправки сообщения
- аналитическая система для работы с данными

Запрос:

```graphql
query {
  user(id: 123) {
    name
    email
  }
}
```

Ответ (JSON):

```json
{
  "data": {
    "user": {
      "name": "Alice",
      "email": "alice@example.com"
    }
  }
}
```

Сервер (Node.js + Apollo):

```javascript
const { ApolloServer, gql } = require("apollo-server");

const typeDefs = gql`
  type User {
    id: ID!
    name: String!
    email: String!
  }

  type Query {
    user(id: ID!): User
  }
`;

const server = new ApolloServer({ typeDefs, resolvers });
server.listen().then(({ url }) => console.log(`Server ready at ${url}`));
```

SOAP — устарел / в БАНКах м.б

затратный по данным

Запрос (HTTP POST + XML):

```http
POST /BankService HTTP/1.1
Host: soap.example.com
Content-Type: text/xml

<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <GetBalance xmlns="http://example.com/bank">
      <accountId>12345</accountId>
    </GetBalance>
  </soap:Body>
</soap:Envelope>
```

Ответ (XML):

```xml
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <GetBalanceResponse xmlns="http://example.com/bank">
      <balance>1000.00</balance>
    </GetBalanceResponse>
  </soap:Body>
</soap:Envelope>
```

API нужно писать правильно

Аутентификация и безопасность

- HTTPS
- Аутентификация:
  - API Key
  - JWT
  - OAuth
  - …
- Авторизация
- Rate limiting
- …

- **API key** — когда берем ключ, который бывает: бесплатны и платный.

Бесплатный — есть ограничения к запросам(например: количество запросов в сек)
Платный — к ключу привязана банковская карточка. И за определенные действия с карточки снимаются деньги

- Должны быть API для авторизации — для ролевой модели (роль для каждого пользователя) (Очень важно внедрять из-за того что если не внедрить, то дальше будет больно(у каждого пользователя проверять ключ что ли?))

- Разработчикам можно получать ключи из Секретницы, в которой храняться ключи для сервисов (Только важно понимать, что у разработчика есть роль: разработчик конкретного сервиса и он может взять ключ только для этого сервиса)

* **В API должна быть безопасность**

Инструменты для работы с API

- Postman — визуальное тестирование
- Insomnia — один из аналогов
- curl — командная строка
- Swagger

- Postman — стандарт индустрии/ мощный инструмент для управления с API запросами https://habr.com/ru/companies/vk/articles/750096/
- Insomnia — просто аналог, т. к. он OPEN SOURCE
- curl — когда подняли на сервере что-то и хотим отправить запрос на этот сервер

- **SWAGGER (часть ДЗ) -**
  gRPC — есть четкая структура / контракты
  REST — нет инстументов конкретных, можно взять любое IP и поднять его.
  Для **сервес — сервесного** взаимодействия, т. е. вы делаете API а кто-то им пользуется
  Вот тут то и понадобится SWAGGER:
  Пишем в OPEN-API стандарте - описываем API в yaml.
  Дальше генерим код, который отправит или примет по такой API сообщение.

т. е. мы как бы договариваемся вместе с людьми, которые пишут с нами проект, чтобы API сошлось.

API как продукт

API — это не просто интерфейс к данным или функциям, это *продукт*, который должен:

- удовлетворять потребности своих пользователей
- иметь четкое позиционирование и ценность
- развиваться по правилам продуктовой разработки

API — самостоятельный продукт

- API нужно рассматривать как отдельную сущность в продуктовой линейке.
- Он решает задачи интеграции, автоматизации, масштабирования.
- У него есть свой цикл жизни: планирование → реализация → релиз → сопровождение → эволюция или утилизация.

Есть продукт, а есть API и они параллельно развиваются.

Антипатерн:
- А если надо переделывать продукт, то обычно переделывают и API, а потребители могут быть не готовы к этому. И другие сервисы могут быть не готовы к этому изменению и они упадут.
- Или если мы поменяли текущее API во внешнее.

Пользователи API: кто они?

- **Основные пользователи** — разработчики. Их нужно воспринимать как "клиентов".
- Внутренние команды, партнеры, сторонние интеграторы, автоматические системы — всё это разные типы пользователей.
- Нужно проектировать API так, чтобы оно было:
  - понятным (документация, примеры)
  - предсказуемым (обратная совместимость, чёткая модель ошибок)
  - стабильным (долгосрочная поддержка, минимизация сбоев при обновлениях)

* обратная совместимости: - это разные версии продукта, должны быть совместимы
на url пишется v1 / v2 …

Продуктовое видение и развитие

- API требует продуктовой стратегии:
  - Зачем оно нужно?
  - Для кого?
  - Как измерять успех?
- Решения о новых фичах, депрекейтах, версиях должны приниматься с учетом обратной связи и бизнес-приоритетов.
- API должно быть управляемым — через дорожную карту, релизный процесс, пользовательскую поддержку.

Метрики успеха API

Метрики API — это способ понять, насколько хорошо API решает задачи пользователей. Примеры:

- **MAU (Monthly Active Users)** — сколько разработчиков активно используют API.
- **Ошибка 4xx/5xx rate** — насколько стабильно работает сервис.
- **Latency и throughput** — скорость и производительность.
- **Time to first call** — сколько времени уходит от изучения API до первого успешного запроса.
- **NPS разработчиков** — насколько разработчики довольны API.

Ошибки 4xx/5xx — надежность
Latency и throughput - требования зависят от того где применяется API
Time to first call — нужно для того, чтобы пользователей было больше, чем понятнее тем лучше

NPS - про счастье / можно спросить пользователей, все ли хорошо, нравится ли Вам?

Документация как часть продукта

Хорошая документация — это интерфейс вашего API.

Должна включать:

- Обзор и цели API (что оно делает)
- Примеры запросов/ответов
- SDK и инструменты (если есть)
- Политику версионирования и поддержку

*** интерфейс на интерфейс нашего продукта**

ПРАКТИКА

https://excalidraw.com/#room=658c23676756cf25ec4c,J1mjpjZFukhL2t8ks5VMRA

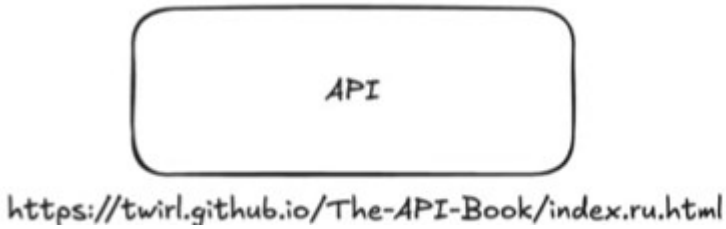
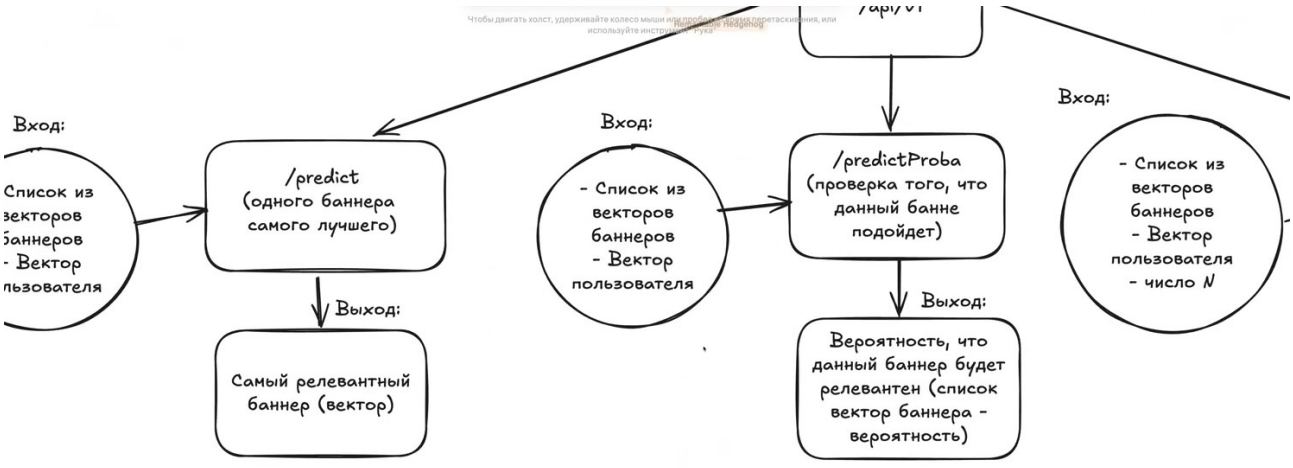


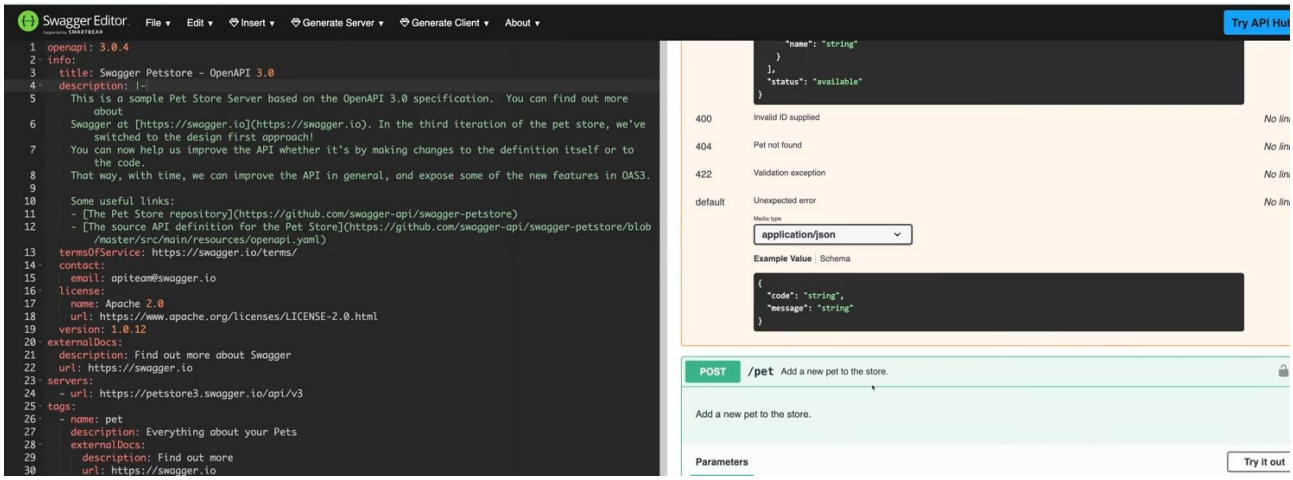
Рисунок 1: Ссылка на книгу по API от разработчика от ЯНДЕКСА

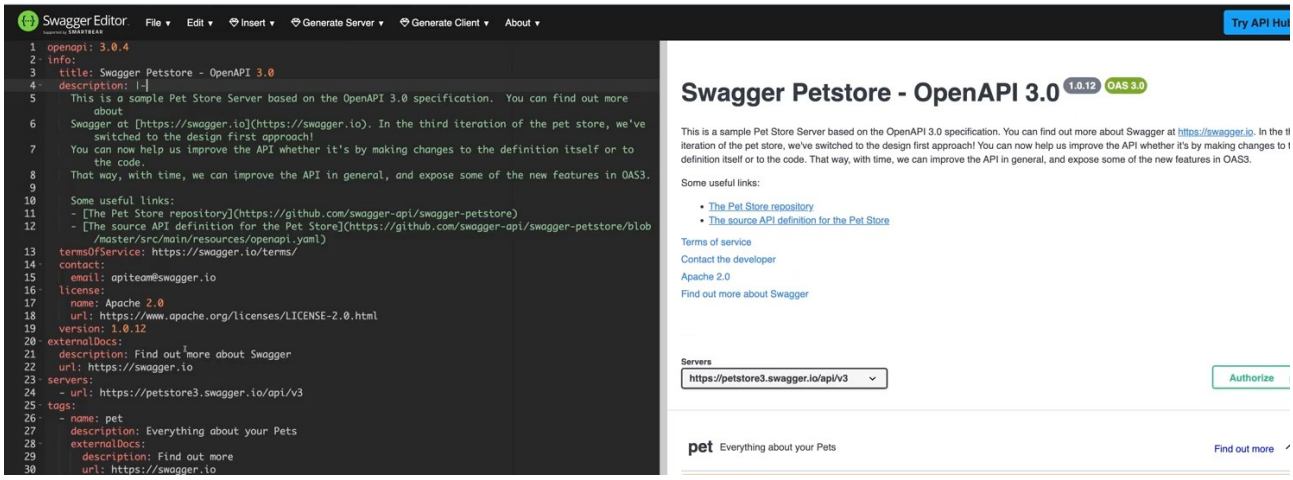
Шаг 1:

Шаг 2: добавляем что на вход подается в кругах



шаг 3: Объяснили про SWAGGER





yaml — слева
commit и потом забрали этот файл себе
Далее сгенерировали по этому файлу все url

ДЗ

Анализ API :
СПОСОБ_1: есть открытое API, и мы с помощью postmane или curl нужно потыкаться в него и посмотреть на что же оно отвечает на каждый запрос
СПОСОБ_2: смотреть документацию / и кусок стима не существующий нужно будет придумать это API.
Что-то там часть по существующим/ часть по не существующим API

второй пункт: Иследование API

МОЙ ВОПРОС: SWAGGER показывает документацию?