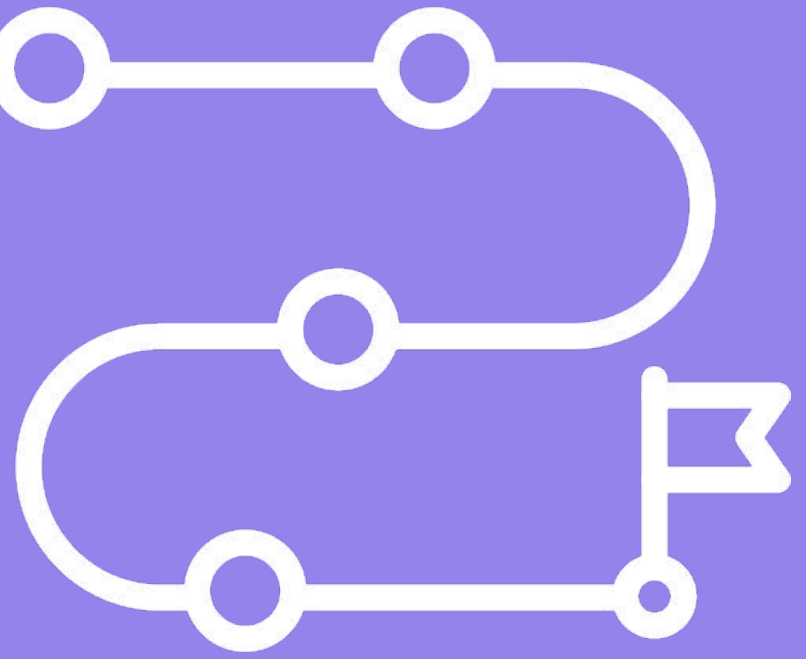
DGTL



Разбираемся с end-to-end type safety для наших бэков

Денис Аникин, https://xfenix.ru

Кто я?

Что я такое?

— Я техлид/комьюнити лид

— fullstack, python, typescript, devops, микросервисы, kubernetes

— Выступаю на конференциях

— Отвечаю за внутреннее сообщество питонистов

https://xfenix.ru

О чем будем говорить

— Об апишках, особенно REST

— О том как их «готовить» fullstack

— О проблеме «рассинхрона» моделей между беком и фронтом

— О способах её решения и, конкретно, о…

— end-to-end type safety

Оглавление

Идеального решения пока нет

но есть разные интересные варианты


СКИДКИ!
КРАБ ПАЛ ДЛЯ САЛАТА
53 74


The crab has fallen
ITS OVER
Billions must DIE 4 salad

Почему актуально не только для фронта

— Писали ли вы когда-то клиенты между микросервисами? (httpx, aiohttp, requests, niquests)
— Представим себе, что они общаются по REST
— И вот мы снова решаем ту же проблему со схемами и актуальными полями…

Обсудим REST

И что его гложет :)

Проблема REST

DGTL

Это ключевая проблема в докладе

Проблема — наш бек

```python
class VeryImportantDomainModel(pydantic.BaseModel):
    user_name: typing.Annotated[str, pydantic.Field(min_length=1, max_length=100)]
    sound_volume: typing.Annotated[int, pydantic.Field(gt=0, example=10)]
    score: int
    when: datetime.datetime


@fastapi_app.get("/rest/simple/")
async def test_simple_get() -> VeryImportantDomainModel:
    return VeryImportantDomainModel(...)
```

Как работать с этим на фронте?

вот тут и начинается интересное

Фронт — «наивный»

```
const response = await fetch('/rest/simple/');
const serverData = await response.json();
// в serverData неизвестно что, ну JSON дело такое... ¯\_(ツ)_/¯

...

// Никакой жалости! Никакого раскаяния! Никакого страха!
```

В процессе разработки вот так…

На деле

Список проблем (?)

— Новые поля

— Смена типов старых (мы ожидали number, стало string и наоборот)

— Удаление старых полей

— Полная смена схемы

Фронт — чуть сложнее

```
interface VeryImportantDomainModel {
  user_name: string;
  sound_volume: number;
  score: number;
  when: string;
}

async function fetchData(): Promise<VeryImportantDomainModel> {
  const response = await fetch('/rest/simple/');
  const data = await response.json();
  return await response.json();
}

fetchData()
  .then((data) => {
    // в data есть автокомплит
  })
```

```
fetchData()
    .then((data) => {
        // в data есть автокомплит
        data.
    })
```

- score
- sound_volume
- user_name
- when

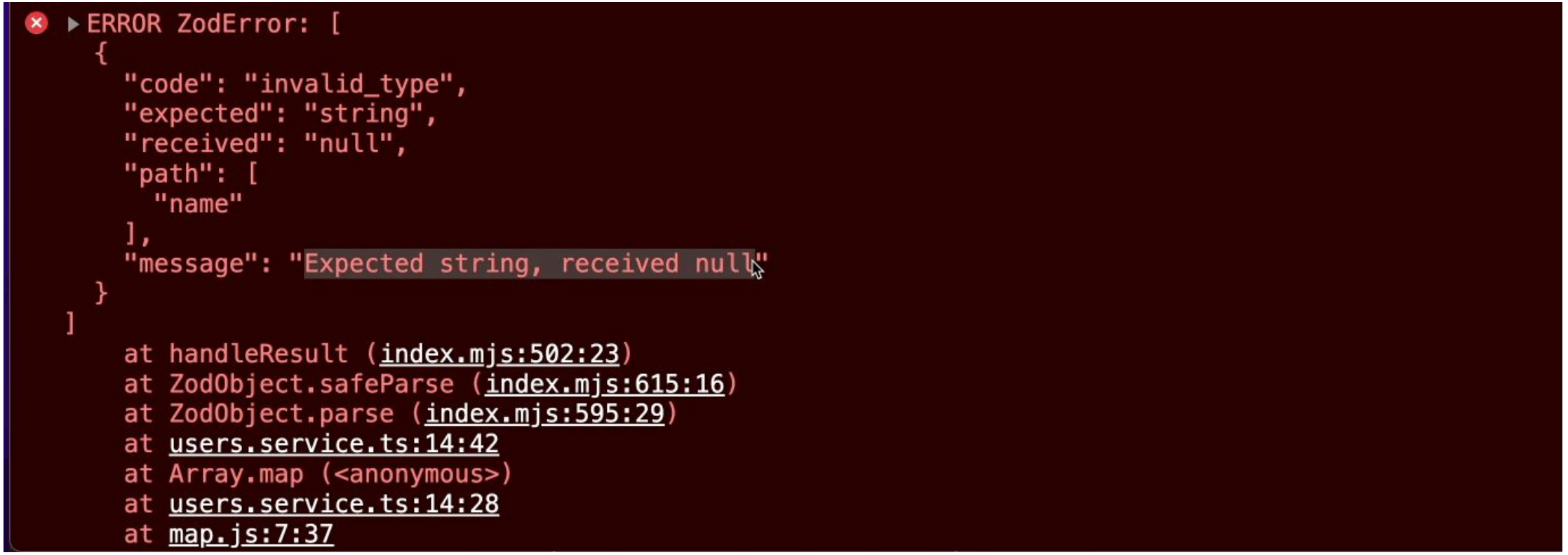
(property) VeryImportantDomainModel.score: number

Фронт — довольно надежный

```ts
const VeryImportantDomainModelSchema = z.object({
  user_name: z.string().min(1).max(100),
  ...
  when: z.string(),  // с датами тут кек...
});

type VeryImportantDomainModel = z.infer<typeof VeryImportantDomainModelSchema>;

async function fetchData(): Promise<VeryImportantDomainModel> {
  const response = await fetch('/rest/simple/');
  const data = await response.json();
  return VeryImportantDomainModelSchema.parse(data);  // Рантайм проверка данных
}

fetchData()
  .then((data) => console.log(data))  // Данные проверены в рантайме + автокомплит
```

Вроде всё хорошо, но

В рантайме мы получаем такое —



Это не плохо

— Мы не сделали неправильных действий в нашей системе

— Zod* предотвратил серьезные ошибки

— Zod == «Pydantic на фронтенде»

Но всё же

— Если мы меняем pydantic* схему на сервере, на клиенте она не обновляется сама
— Менять руками и там и там — рабочий вариант, но люди ошибаются

Давайте поищем пути решения проблемы

GraphQL

DGTL

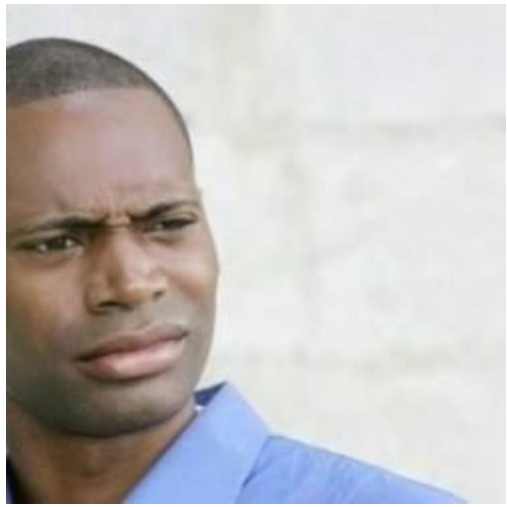
Проходит под темой
«зачем нам этот rest?» и «rest is so 2010»

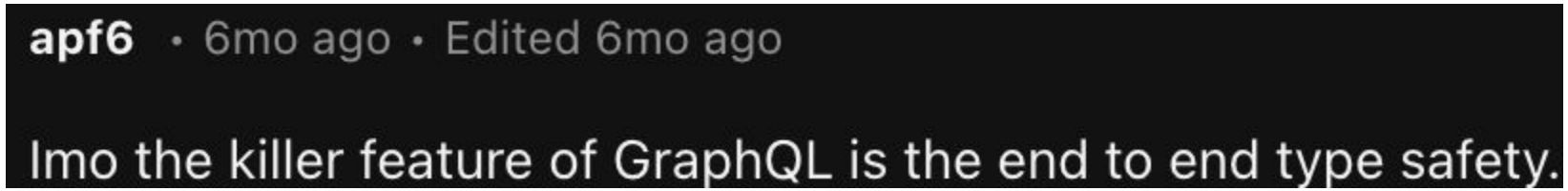
Мои мысли

— Я не большой поклонник GraphQL
— Да и большого опыта с GraphQL нет
— Текущий доклад всё ещё про REST, но посмотрим что в мире GraphQL (не будем делать большое сравнение)

Как оно там с graphql

```
const client = new GraphQLClient('/graphql');
async function fetchData(): Promise {
  const data = await client.request(gql`
  query GetModel {
    get_model {
      user_name
      ...
      when
    }
  }
`);
  return data.get_model;
}

fetchData()
  .then((data) => console.log(data))
```





By using a next-generation ORM to fetch data in your endpoints, you get most of the value of GraphQL — end-to-end typesafety, easy nested fetching, field selection, etc — in a TypeScript-native way.

И так далее…

Спойлеры таковы

— GraphQL не typesafe по дефолту
— Мы так же пишем схемы и нам надо как-то их выводить…
— Способы решения — более 34234324123 штук (от кодогенерации, до вывода типов аля codefirst)
— Много сырых решений :(

gRPC + web

DGTL

Это второе, что приходит на ум

gRPC + web

— Все вопросы со схемами и кодгеном из коробки решены
— Эффективно и быстро
— Но… часто ли GRPC встречается в наших краях?
— А ещё нам придется отказаться от REST
— Но как референс и пример перед глазами стоит вспомнить
— https://github.com/grpc/grpc-web/tree/master/net/grpc/gateway/examples/helloworld

API first

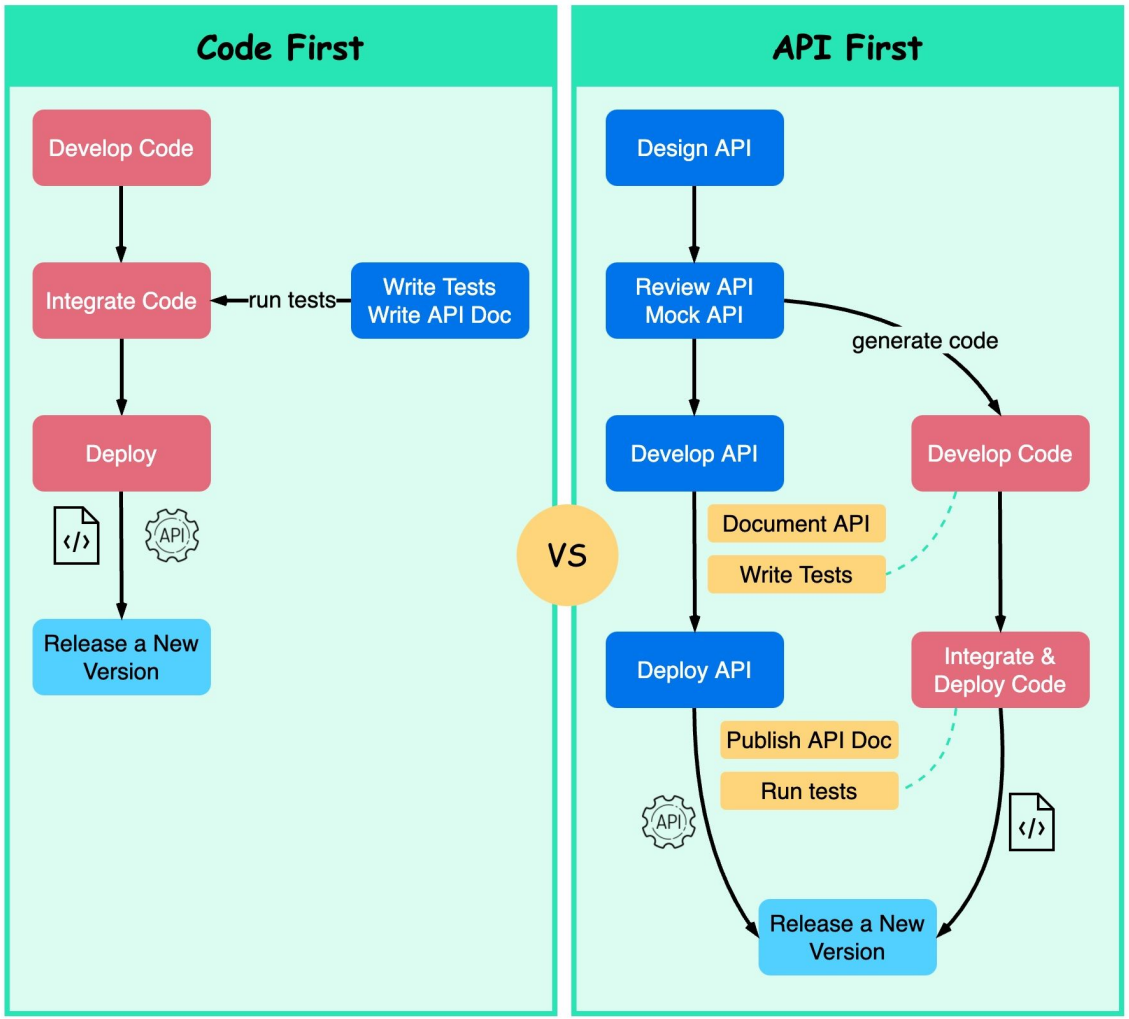
DGTL

Один из вариантов решения

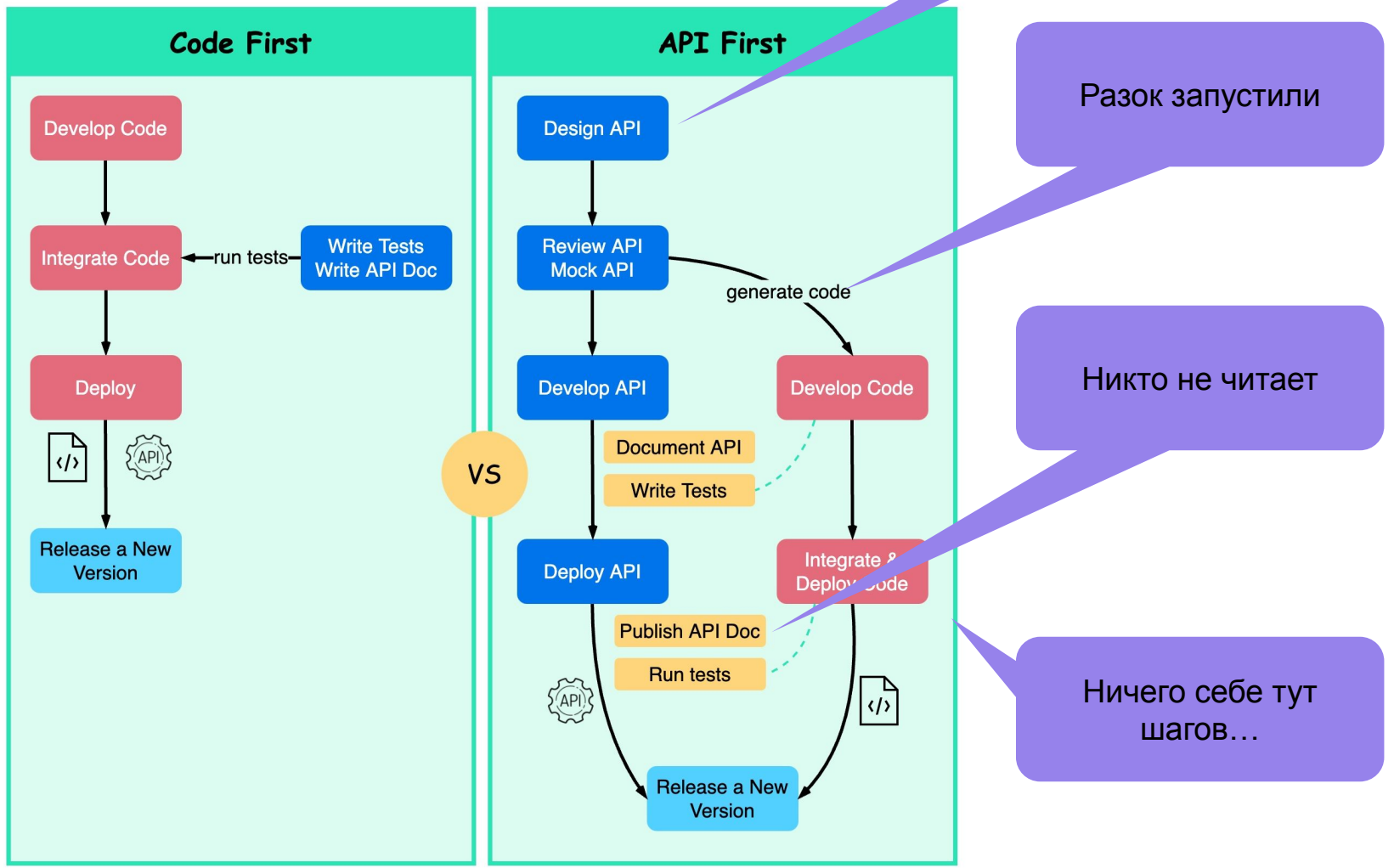
API first «как в книге»

Code First v.s API First Development

blog.bytebytego.com



API first в жизни



Никто не обновляет

Разок запустили

Никто не читает

Ничего себе тут шагов…

Где API first раскрывается?

— Где у вас много разных языковых окружений

— Например, android + backend + ios + web + ещё что-то

— Мы разрабатываем все одновременно

Пример

```yaml
openapi: 3.0.0
...
paths:
  /important-data:
    get:
      ...
components:
  schemas:
    VeryImportantDomainModel:
      type: object
      properties:
        user_name: { type: string, minLength: 1, maxLength: 100 }
        sound_volume: { type: integer, minimum: 1, example: 10 }
        score: { type: integer }
        when: { type: string, format: date-time }
      required: [user_name, sound_volume, score, when]
```

Что дальше?

— Мы берем схему и кладём её в общий для всех репозиторий
— Делаем кодогенераторы в проектах
— Формально — единый источник правды

Всё хорошо

наверное…

Как обычно

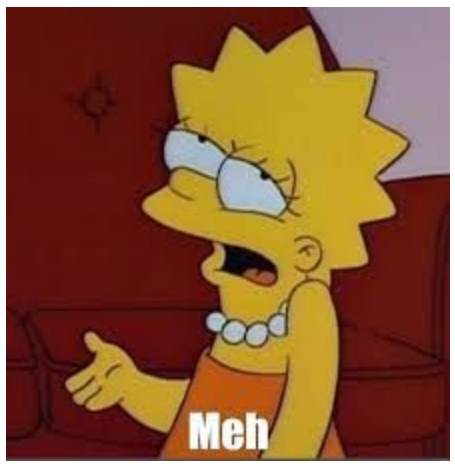
— Кодогенерацию для каждого репозитория кто-то должен запускать
— ОК. Автоматизируем?
— Для каждого языкового окружения нужен скрипт в CI или git хук, а так же пакет, который умеет генерировать схемы для нужного валидатора
— Мы будем постоянно иметь «замусирование» коммитов или коммитами со схемами

Однако

— Вы можете организовать +- всё автоматом

— Вы можете автогенерированные схемы не держать в репе

— Можно сделать генерацию +- удобной, делая кодген дважды


Meh

А теперь поговорим о решениях, которые ближе мне

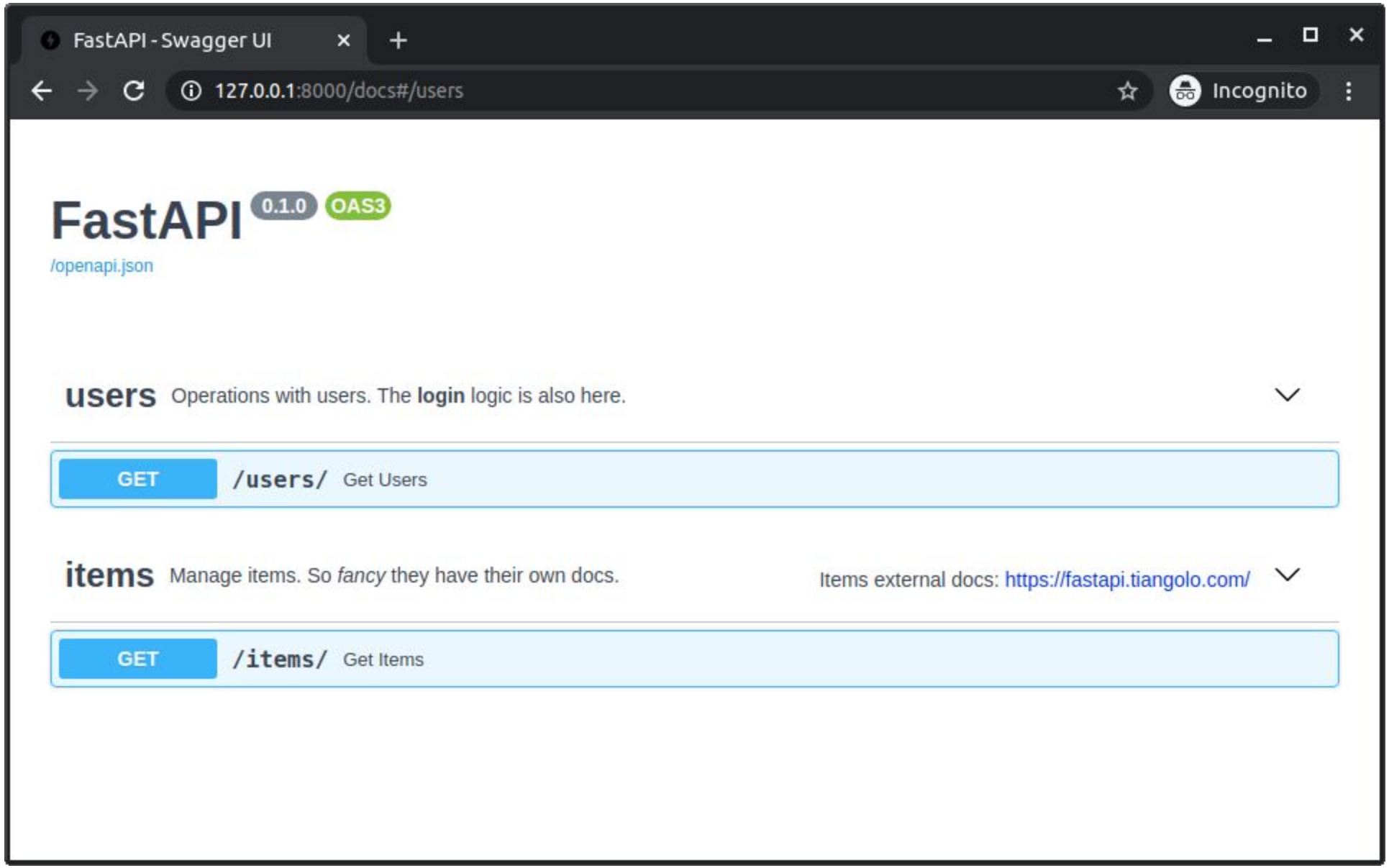
Code first

DGTL

Генерируем спецификацию из кода

Пример из жизни

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

app = FastAPI()
items_db = {}

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

@app.get("/items/")
def get_items(...):
    ...
```


FastAPI - Swagger UI
127.0.0.1:8000/docs#/users
Incognito
FastAPI
0.1.0
OAS3
/openapi.json
users
Operations with users. The login logic is also here.
GET
/users/
Get Users
items
Manage items. So fancy they have their own docs.
Items external docs: https://fastapi.tiangolo.com/
GET
/items/
Get Items

Почему

— Никакого кодгена
— Сидишь, пишешь, дальше всё само :)
— Документация генерируется автоматом — это идеально (т.к. документация «проклята»: её не пишут, не актуализируют, пишут плохо, но всем стыдно, что этого не делают)

tRPC

DGTL

Modern way?

Что такое trpc

— end-to-end typesafety

— никаких кодогенераций

— никаких схем

— статический автокомплит

— всё автоматом

— из минусов: только typescript…

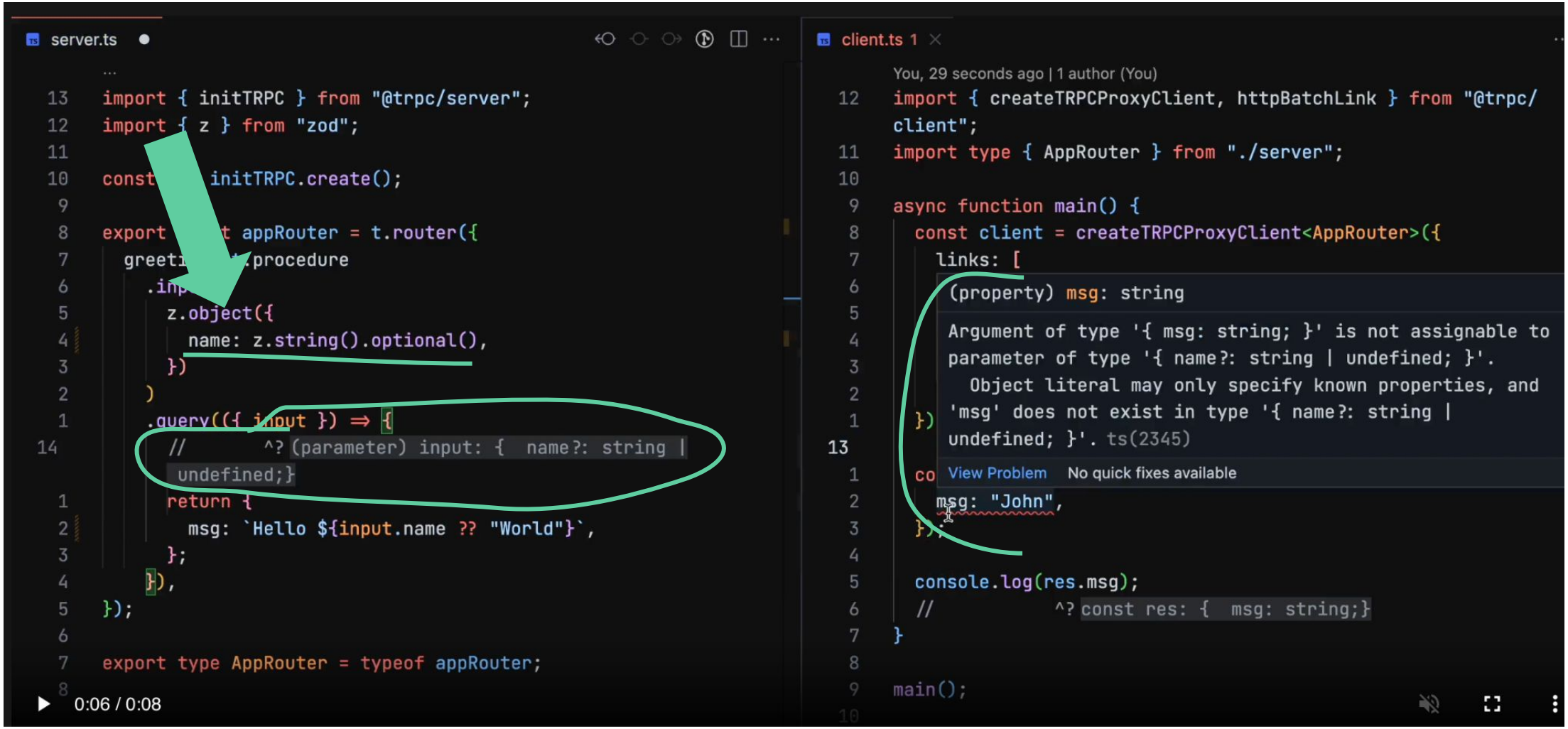
tRPC сервер

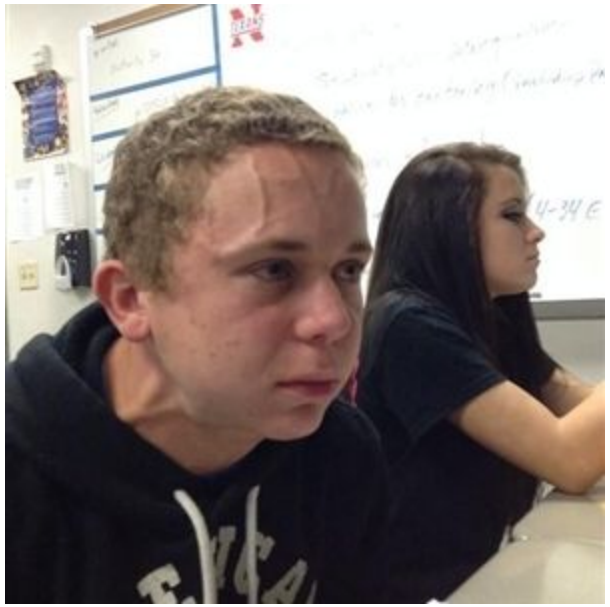
```ts
const t = initTRPC.create();

const router = t.router;
const publicProcedure = t.procedure;

const appRouter = router({
  greeting: publicProcedure
    .input(z.object({ name: z.string() }))
    .query((opts) => {
      const { input } = opts;
      return `Hello ${input.name}` as const;
  }),
});

createHTTPServer({router: appRouter,}).listen(3000);
```

tRPC клиент

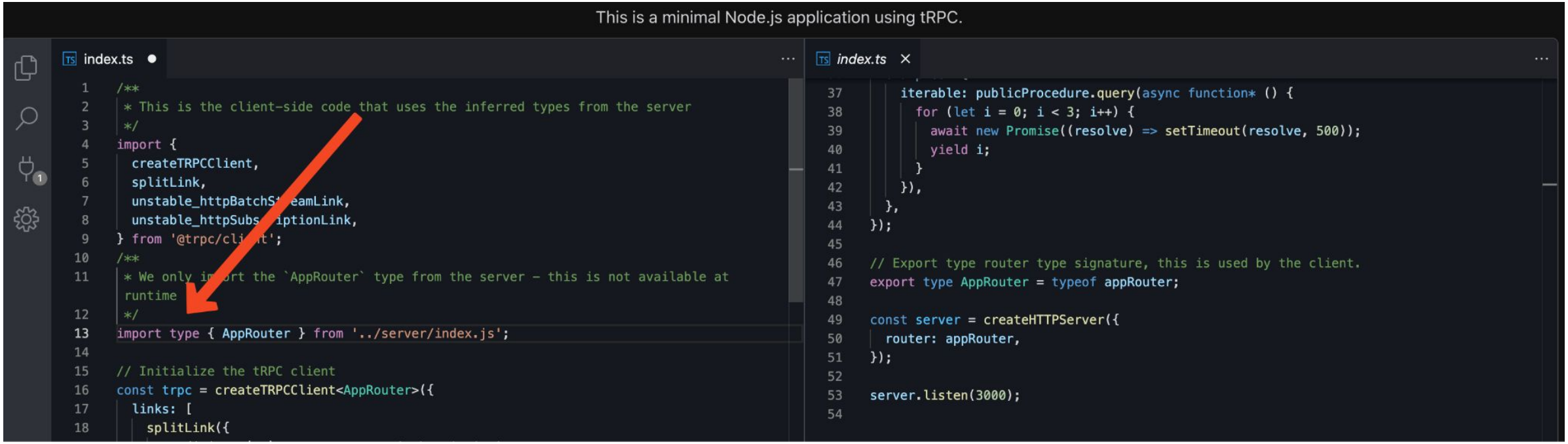
```ts
const trpc = createTRPCClient<AppRouter>({
  links: [
    httpBatchLink({
      url: 'http://localhost:3000',
    }),
  ],
});

const res = await trpc.greeting.query({ name: 'John' });
```

Радости


server.ts
import { initTRPC } from "@trpc/server";
import { z } from "zod";
initTRPC.create();
appRouter = t.router({
.procedure
z.object({
name: z.string().optional(),
})
.query(({ input }) => {
// ^? (parameter) input: { name?: string | undefined;}
return {
msg: `Hello ${input.name ?? "World"}`,
};
}),
});
export type AppRouter = typeof appRouter;
0:06 / 0:08
client.ts 1
You, 29 seconds ago | 1 author (You)
import { createTRPCProxyClient, httpBatchLink } from "@trpc/client";
import type { AppRouter } from "./server";
async function main() {
const client = createTRPCProxyClient<AppRouter>({
links: [
(property) msg: string
Argument of type '{ msg: string; }' is not assignable to parameter of type '{ name?: string | undefined; }'.
Object literal may only specify known properties, and 'msg' does not exist in type '{ name?: string | undefined; }'. ts(2345)
View Problem
No quick fixes available
msg: "John",
console.log(res.msg);
// ^? const res: { msg: string;}
}
main();



А как оно работает?



Type-Only Imports and Export

This feature is something most users may never have to think about;

ElysiaJS

DGTL

Modern way (2)?

elysia: server

```ts
import { Elysia, t } from 'elysia'
import { swagger } from '@elysiajs/swagger'

const app = new Elysia()
    .use(swagger())
    .get('/user/:id', ({ params: { id } }) => id, {
        params: t.Object({
            id: t.Numeric()
        })
    })
    .listen(3000)

export type App = typeof app
```

elysia: client

```ts
import { treaty } from '@elysiajs/eden'
import type { App } from './server'

const app = treaty<App>('localhost:3000')
const { data } = await app.user({ id: 617 }).get()

console.log(data)
```

```ts
const { data } = await app.user({ id: 617 }).get()

const data: number | null
```

Мы ближе к Граалю?

Но на typescript, а хотелось-то бы на python!

DGTL

Давайте поближе к python уже… :)

Code first 2!

DGTL

Генерируем спецификацию из кода вновь

Собираем по деталям

— Возьмём на backend fastapi или litestar. С pydantic

— Почему не msgspec? ~~По-качану~~Просто самый распостраненный

— Фронтенд — react + typescript

— Кодогенерация (sad part)

(1) Kubb

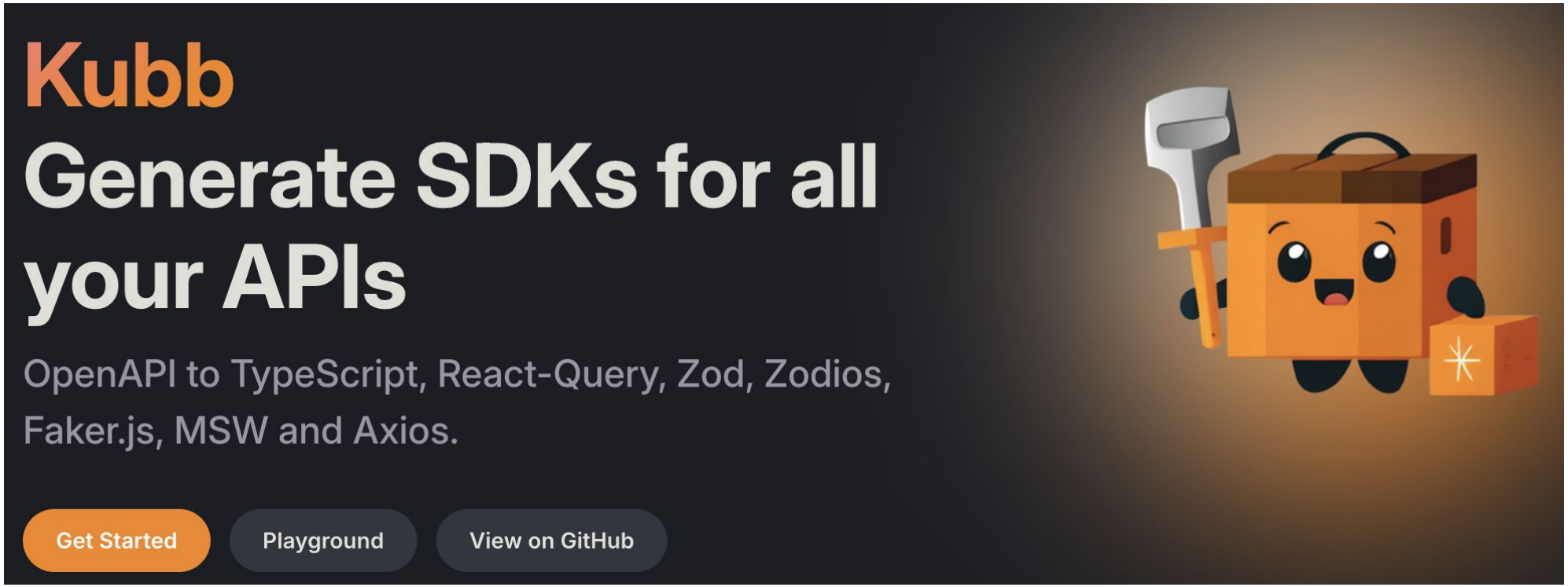
DGTL

Что это такое?

— Кодогенератор… (

— Много всякого …

Сайт


Kubb
Generate SDKs for all your APIs
OpenAPI to TypeScript, React-Query, Zod, Zodios, Faker.js, MSW and Axios.
Get Started
Playground
View on GitHub

Releases 3,457

unplugin-kubb@0.1.87 Latest

5 hours ago

+ 3,456 releases

Опять не грааль

но…

Генерация

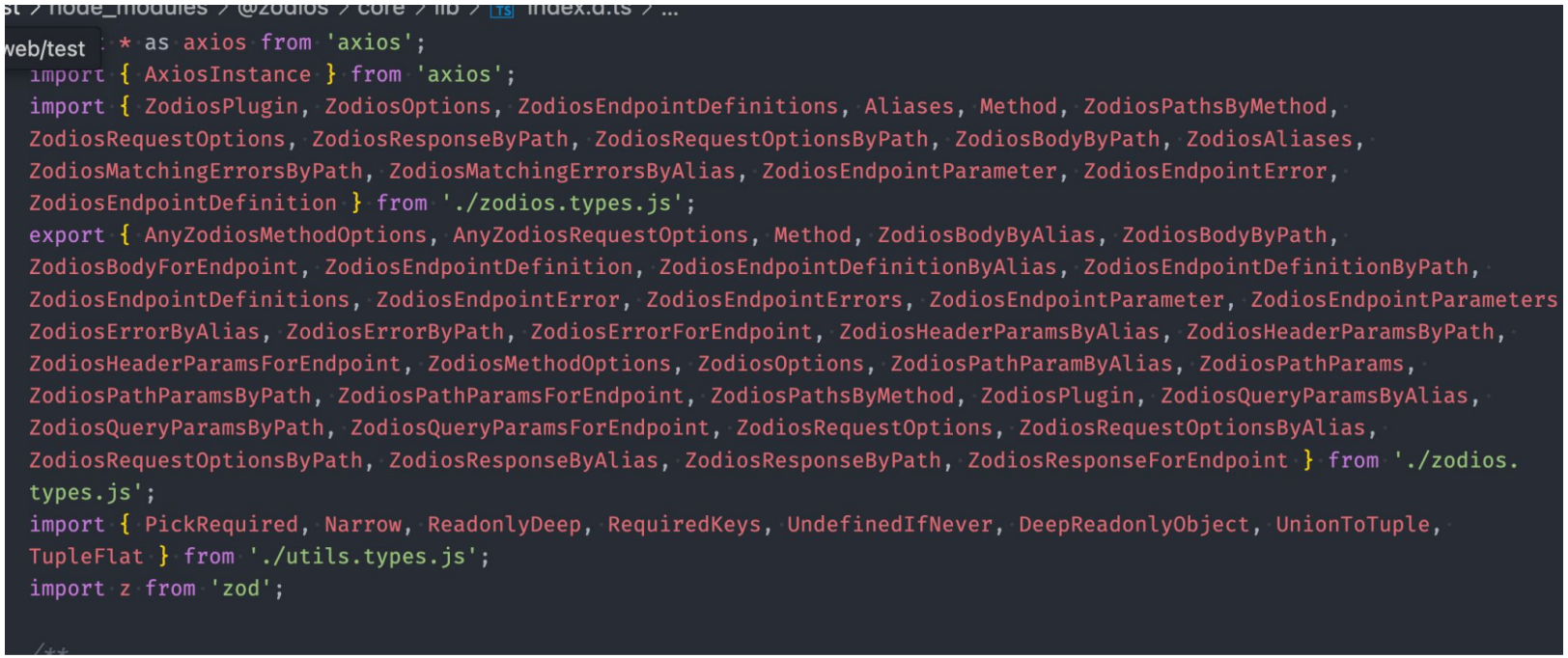
```
npx kubb
```

Конфиг: ну…

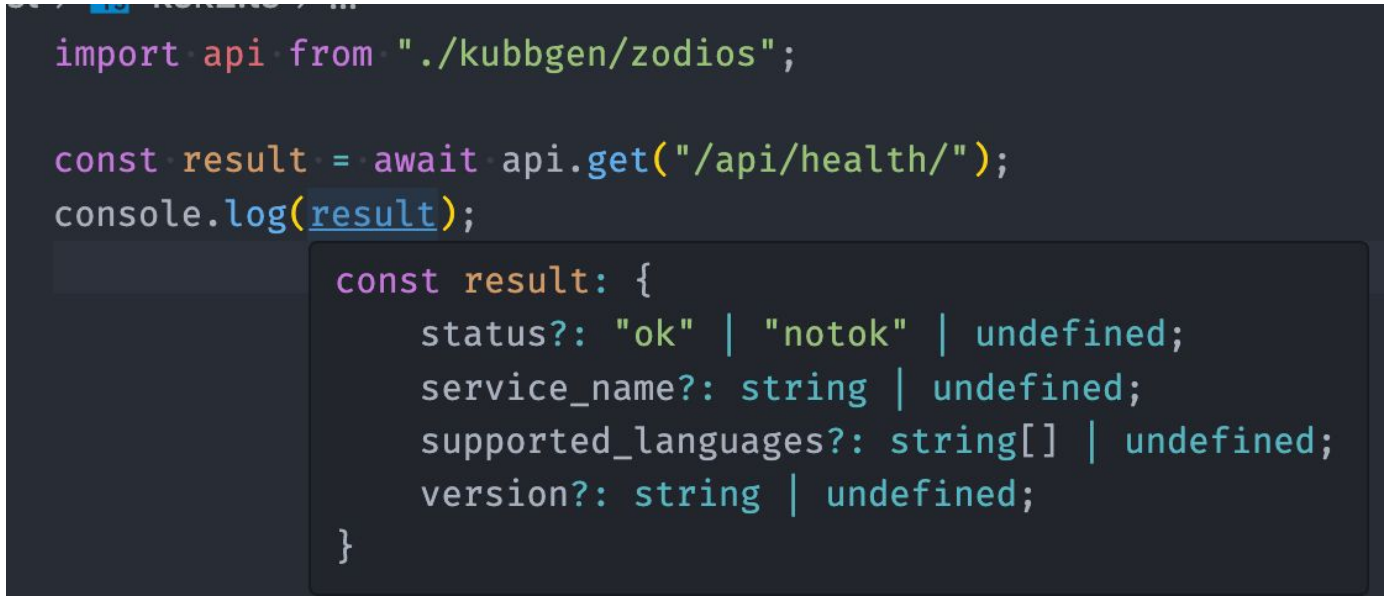
```ts
import { defineConfig } from "@kubb/core";
import { pluginZodios } from "@kubb/swagger-zodios";
import { pluginOas } from "@kubb/plugin-oas";
import { pluginZod } from "@kubb/swagger-zod";

export default defineConfig({
  input: {
    path: "http://127.0.0.1:8000/api/openapi.json",
  },
  output: {
    path: "./kubbgen",
  },
  plugins: [
    pluginOas(),
    pluginZod(),
    pluginZodios({
      output: {
        path: "./zodios.ts",
      },
    }),
  ],
});
```

Кодген красив

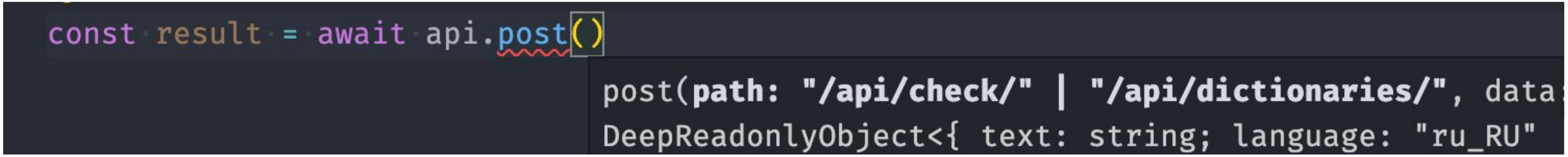
очень




```
    */
    get<Path extends ZodiosPathsByMethod<Api, "get">, TConfig extends ZodiosRequestOptionsByPath<Api, "get", Path>>
    (path: Path, ...[config]: RequiredKeys<TConfig> extends never ? [config?: ReadonlyDeep<TConfig>] : [config:
    ReadonlyDeep<TConfig>]): Promise<ZodiosResponseByPath<Api, "get", Path>>;
    /**
```


import api from "./kubbgen/zodios";
const result = await api.get("/api/health/");
console.log(result);
const result: {
status?: "ok" | "notok" | undefined;
service_name?: string | undefined;
supported_languages?: string[] | undefined;
version?: string | undefined;
}


const result = await api.get("
/api/health/


const result = await api.post()
post(path: "/api/check/" | "/api/dictionaries/", data
DeepReadonlyObject<{ text: string; language: "ru_RU"

Что в итоге

— Возиться с конфигами и плагинами

— Кодген пугает

— Пристойный результат

— Хороший автокомплит

(2) Openapi to ts

DGTL

openapi to typescript

```
npx openapi-typescript http://127.0.0.1:8000/api/openapi.json --output ./generated-types.ts
```

Openapi fetch

— Берем эти типы

— И получаем типо-безопасный fetch

— Без рантайма…

— Без рантайма? :(

openapi fetch

```ts
import createClient from "openapi-fetch";
import type { paths } from "./my-openapi-3-schema"; // generated by openapi-typescript

const client = createClient<paths>({ baseUrl: "https://myapi.dev/v1/" });

const {
  data, // only present if 2XX response
  error, // only present if 4XX or 5XX response
} = await client.GET("/blogposts/{post_id}", {
  params: {
    path: { post_id: "123" },
  },
});
```

Что в итоге

— Быстро

— Кое-что подсказывает

```
await client.POST("/api/check/")
await client.GET("/")
```

/api/health/

```
const { data, errors } = await client.POST("/api/check/", {
  params: {}, body: {})
await client.GET
```

Type '{}' is missing the following properties from type '{ text: string; language: "ru_RU" | "en_US" | "es_ES" | "fr_FR" | "de_DE" | "pt_PT"; user_name?: string | null | undefined; exclude_urls: boolean; }': text, language, exclude_urls ts(2739)

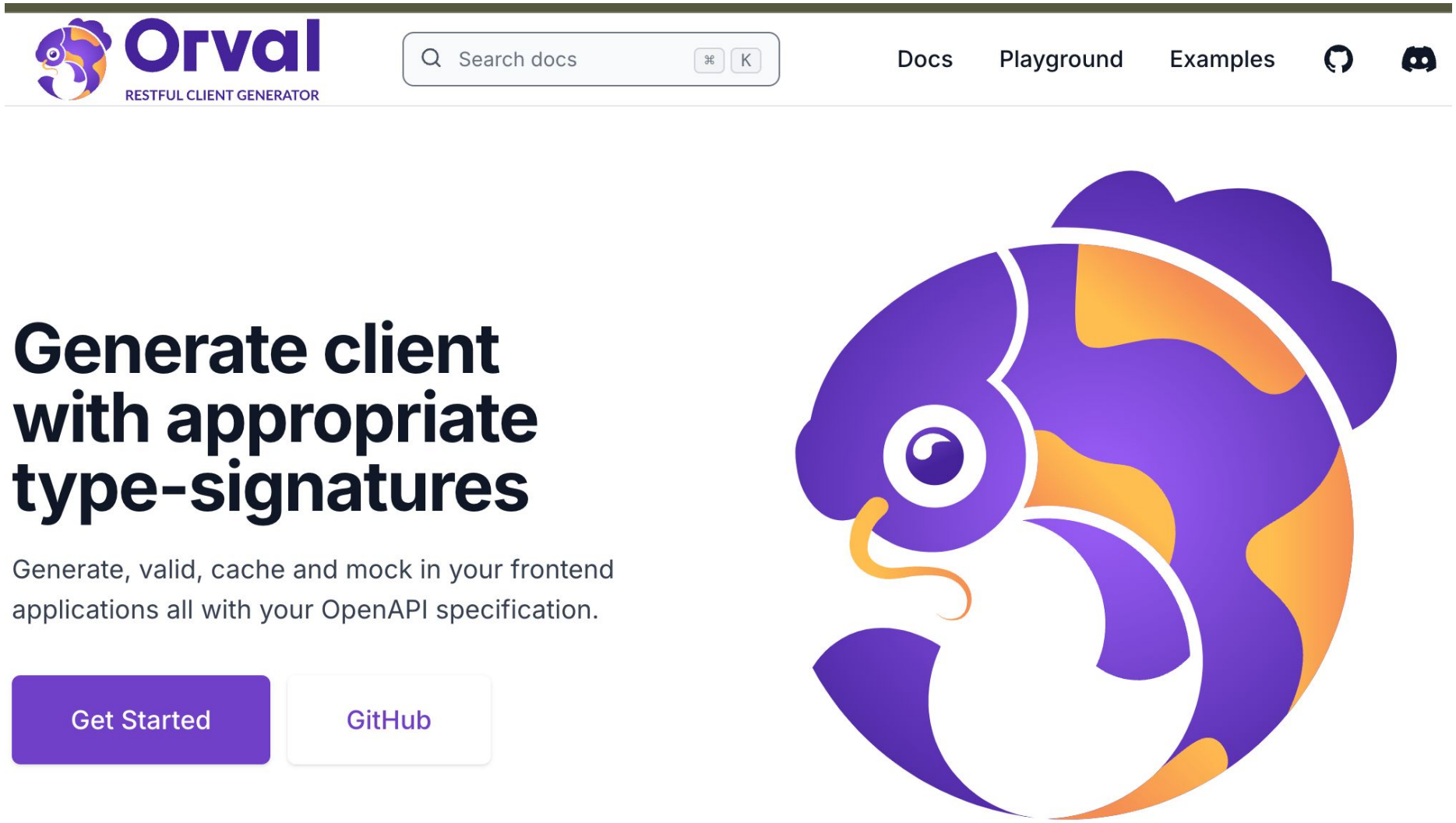
```
const { data, errors } = await client.POST("/api/check/", {
  params: {},
  body: { text: "medved", language: "ru_RU", exclude_urls: false },
});
console.log(data);
console.log(errors);
```

Что в итоге 2

— Не всю статическую интроспекцию смотреть удобно
— Автокомплита в атрибутах нет
— Нет рантайм проверок
— В остальном, рабочий вариант (прикручиваем в CI и локально, радуемся жизни)

(3) Orval

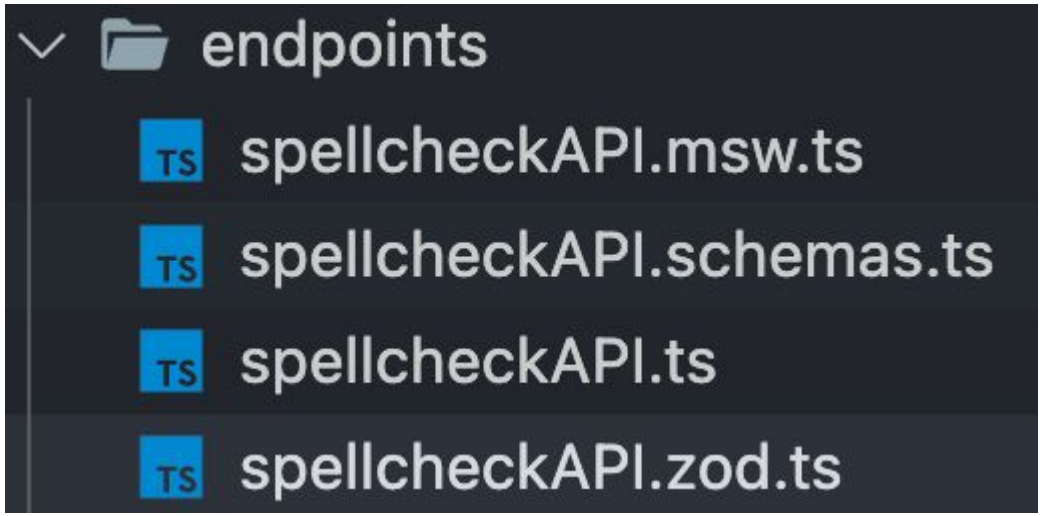
DGTL


Orval
RESTFUL CLIENT GENERATOR
Search docs
⌘
K
Docs
Playground
Examples
Generate client with appropriate type-signatures
Generate, valid, cache and mock in your frontend applications all with your OpenAPI specification.
Get Started
GitHub

orval.config.js

```
module.exports = {
  petstore: {
    input: {
      target: "http://127.0.0.1:8000/api/openapi.json",
    },
    output: {
      mode: "split",
      client: "swr",
      target: "./endpoints",
      mock: true,
    },
  },
  petstoreZod: {
    input: "http://127.0.0.1:8000/api/openapi.json",
    output: {
      target: "./endpoints",
      fileExtension: ".zod.ts",
      client: "zod",
    },
  },
};
```

Что генерируется



Как использовать

```
import { deleteWordApiDictionariesDelete } from "./endpoints/spellcheckAPI";
import { UserDictionaryRequestWithWord } from "./endpoints/spellcheckAPI.schemas";
import {
  deleteWordApiDictionariesDeleteResponse,
  deleteWordApiDictionariesDeleteBody,
} from "./endpoints/spellcheckAPI.zod";

const params: UserDictionaryRequestWithWord = {
  exception_word: "",
  user_name: "",
};

const myFancyAnswer = await deleteWordApiDictionariesDelete(
  deleteWordApiDictionariesDeleteBody.parse(params)
);
deleteWordApiDictionariesDeleteResponse.parse(myFancyAnswer);
```

Всё, что нужно?

Вроде да, но не очень удобно

(4) Zodios

DGTL

Опять правильные слова!

Что это такое?

— апи клиент для рестов

— typesafe! (мы рядом с end-to-end)

— так заявлено ведь?

— end-to-end вместе с typescript бекендом (и опять и опять)

— zod валидация в рантайме

— статические схемы

zodios

```
const api = makeApi([
  {
    method: "get",
    path: "/items/:id",
    alias: "getItem",
    parameters: [
      { name: "id", type: "Path", schema: z.string() },
    ],
    response: z.object({
      id: z.string(),
      name: z.string(),
      description: z.string().optional(),
    }),
  },
]);

const client = new Zodios("http://localhost:3000", api);
client.getItem({ id: "123" }).then((response) => console.log(response));
```

Пристойно

если не хочется идти дальше (+ с fastapi легко подружить)

Но

увы, полностью, в «наши слоны» записать мы не можем

(5) Openapi zod client

DGTL

Удобно

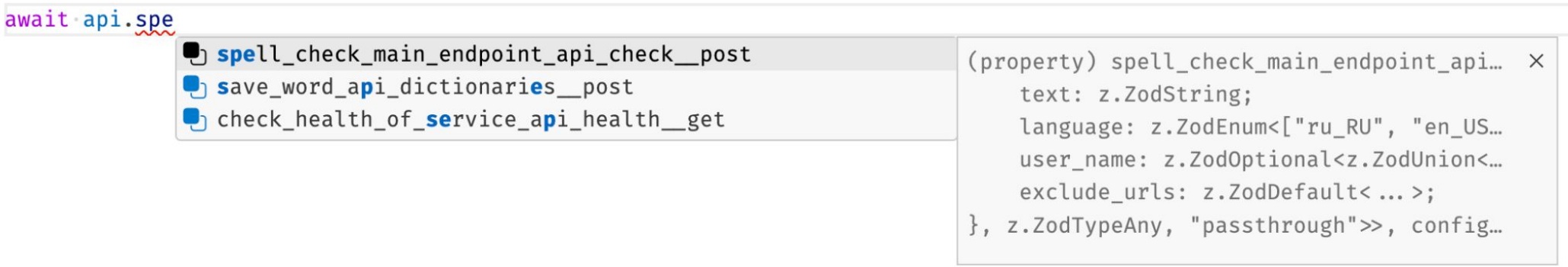
```
npx openapi-zod-client http://127.0.0.1:8000/api/openapi.json -o ./generated-client.ts
```

Но

— Вывод типов и статические подсказки — вообще норм

— Алиасы апишек…

```
  },
  {
    method: "get",
    path: "/api/health/",
    alias: "check_health_of_service_api_health__get",
    description: `Check health of service.`,
    requestFormat: "json",
    response: HealthCheckResponse,
  },
]);
```


await api.spe
spell_check_main_endpoint_api_check__post
save_word_api_dictionaries__post
check_health_of_service_api_health__get
(property) spell_check_main_endpoint_api…
text: z.ZodString;
language: z.ZodEnum<["ru_RU", "en_US…
user_name: z.ZodOptional<z.ZodUnion<…
exclude_urls: z.ZodDefault< ... >;
}, z.ZodTypeAny, "passthrough">>, config…

```
await api.spell_check_main_endpoint_api_check__post({
  text: "Привет",
  language: "ru_RU",
});
```

В целом

— Почти мечта
— Алиасы исправляемы небольшим скриптом (вот он)
— Рантайм валидация
— end-to-end typesafety (по сути)
— Из неудобного (всё ещё кодген) — имеет смысл прикрутить в CI/запускать локально

(6) Tanstack query + openapi-rq

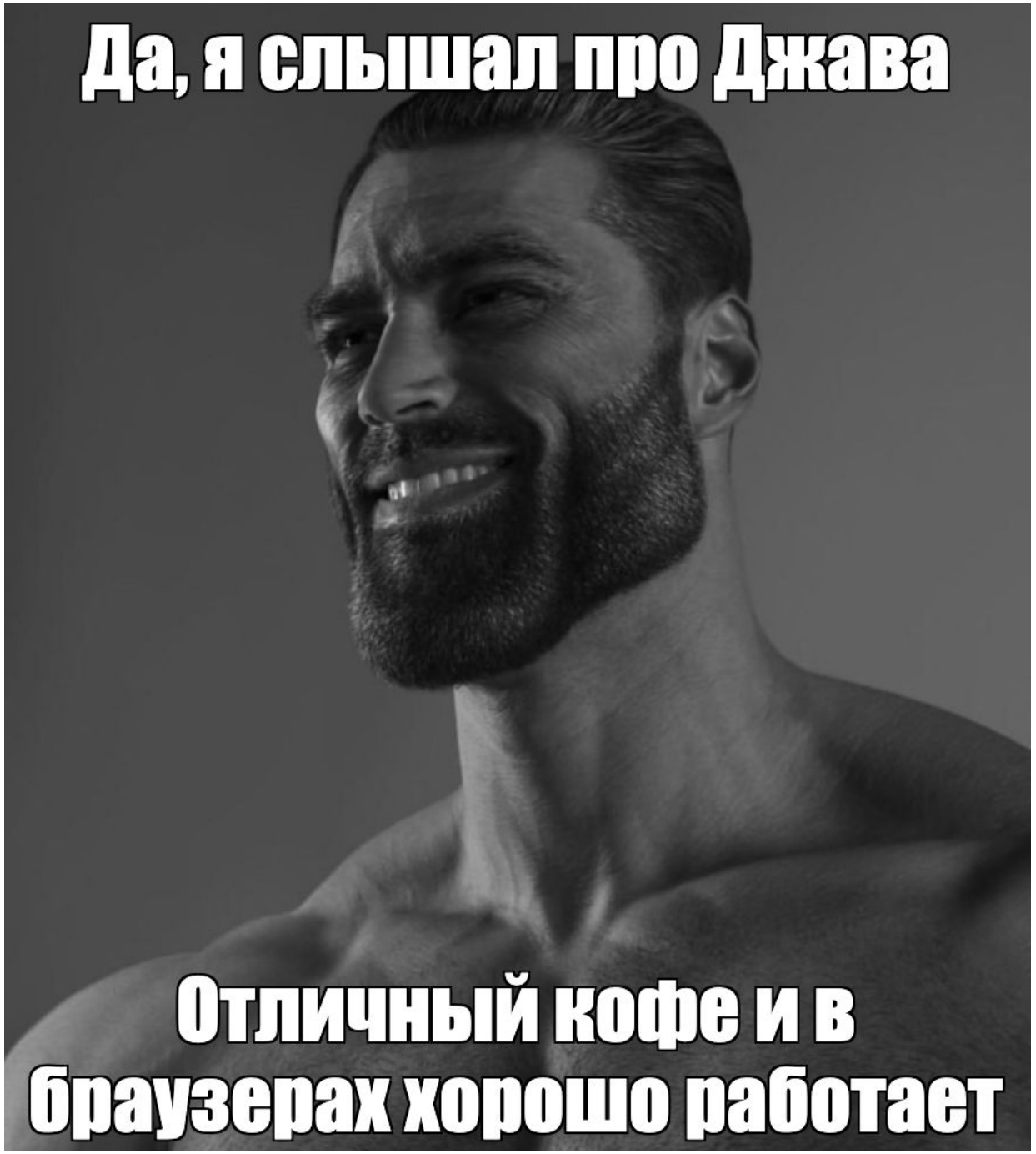
DGTL

Tanstack query + openapi-rq

```
npx --package @7nohe/openapi-react-query-codegen openapi-rq -i http://127.0.0.1:8000/api/openapi.json -c axios
```

Живой код интересный

```
import { useDefaultServiceSpellCheckMainEndpointApiCheckPost } from "./openapi/queries";

function App() {
  const { data } = useDefaultServiceSpellCheckMainEndpointApiCheckPost({
    body: {
      text: "test",
    }
  });

  return <div className="App"></div>;
}

export default App;
```


Да, я слышал про Джава
Отличный кофе и в браузерах хорошо работает

Как мы видим

— Здесь решился вопрос с snake_case ⇔ camelCase

— Но вот с return типами пока проблемка

— А так — ну весело, одна команда и сразу результат

DGTL

Подведем некоторые выводы

Выбираем из всех

— kubb сыроват, но даёт много разного ✅

— openapi to ts не для всех кейсов, нет рантайма

— orval — пристойно, но многословно ✅

— zodios — не очень, если только руками писать

— openapi zod client — вообще неплохо (мой выбор) ✅

— tanstack query + openapi-rq — тоже ничего так ✅

Минусы нашего выбора

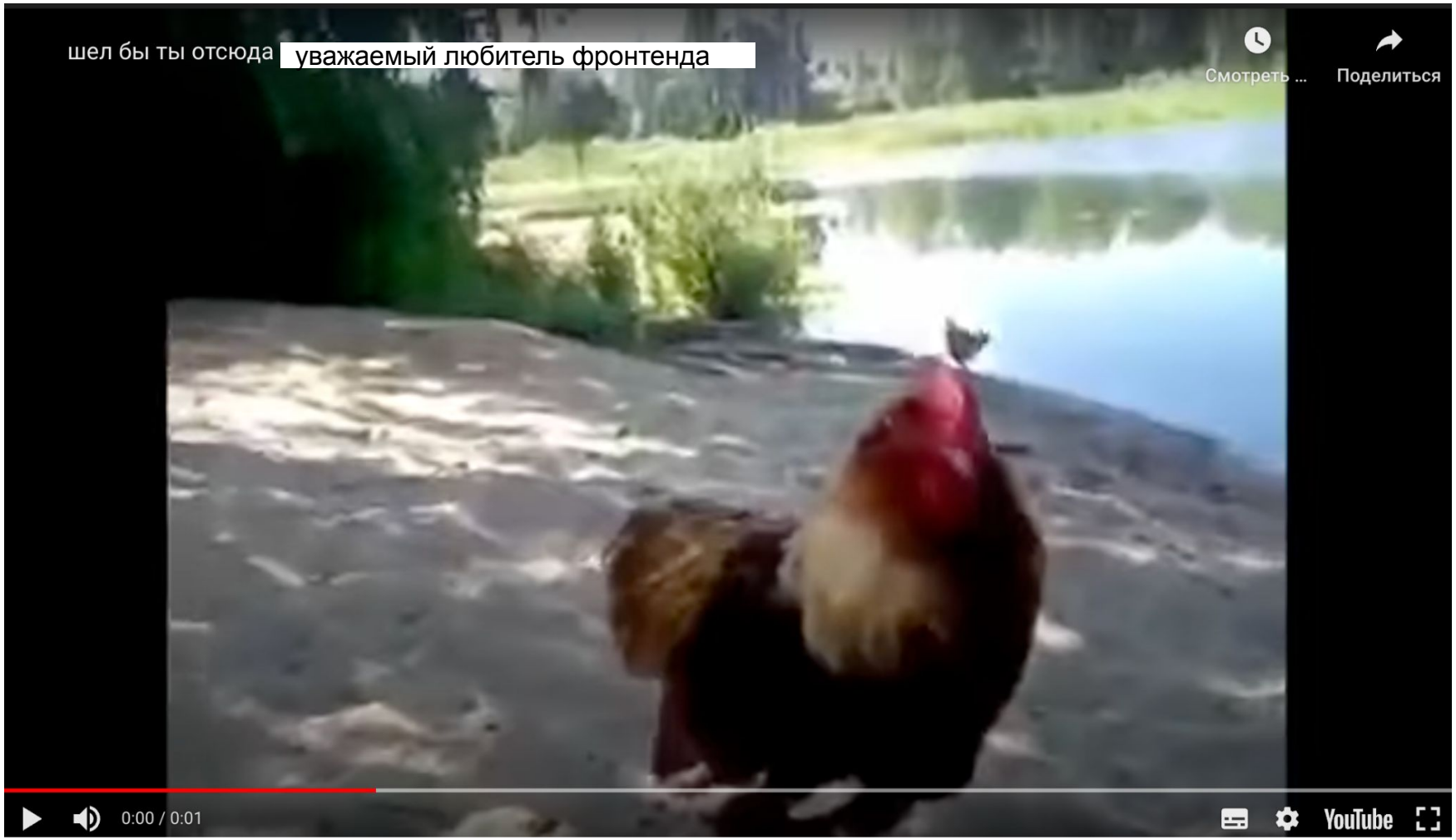
— Кодо-генерация
— == Необходимость гонять в CI или на гит хуках
— Косяки промежуточных пакетов
— Солянка решений
— Ощущение, что мы начинаем мешать что-то из мира grpc, что-то из мира graphql и мир rest, теряем простоту

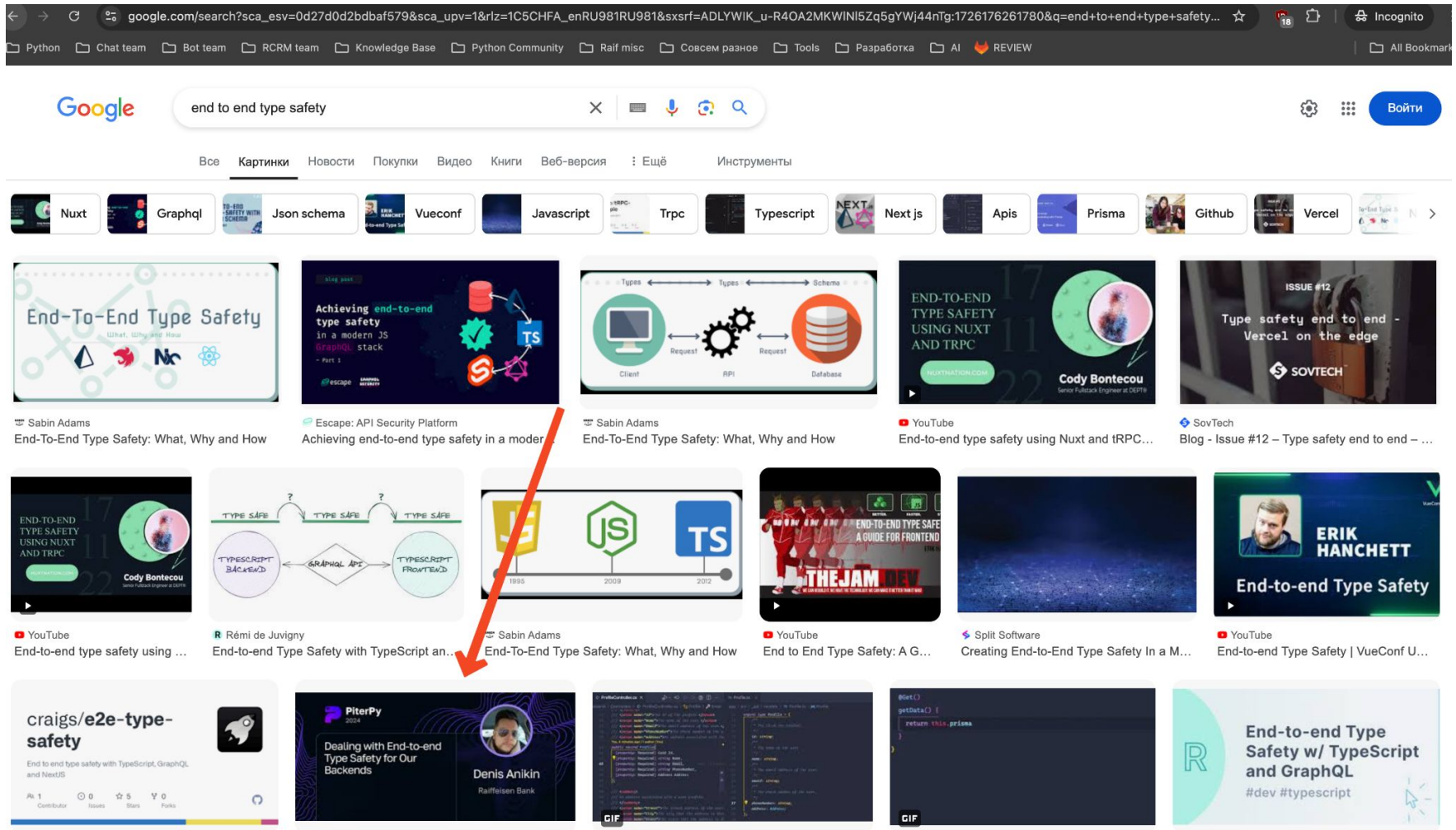
Плюсы

— end-to-end typesafety
— автокомплит
— нет рассинхрона
— нет человеческих ошибок
— идеальная совместимость с fastapi (и litestar) за счёт codefirst подхода этих фреймворков
— бонус ✨✨✨: orval и kubb умеют генерировать msw, что даёт вам «фейковые» апишки для полноценных интеграционных тестов
— крутейший фронт + крутейший бек (всё самое актуальное)

Мне грустно, что я не знал раньше

Справедливый комментарий




google.com/search?sca_esv=0d27d0d2bdbaf579&sca_upv=1&rlz=1C5CHFA_enRU981RU981&sxsrf=ADLYWIK_u-R4OA2MKWINI5Zq5gYWj44nTg:1726176261780&q=end+to+end+type+safety...
Incognito
Python
Chat team
Bot team
RCRM team
Knowledge Base
Python Community
Raif misc
Совсем разное
Tools
Разработка
AI
REVIEW
All Bookmark
Google
end to end type safety
Войти
Все
Картинки
Новости
Покупки
Видео
Книги
Веб-версия
Ещё
Инструменты
Nuxt
Graphql
Json schema
Vueconf
Javascript
Trpc
Typescript
Next js
Apis
Prisma
Github
Vercel
End-To-End Type Safety
What, Why and How
Sabin Adams
End-To-End Type Safety: What, Why and How
Achieving end-to-end type safety in a modern JS GraphQL stack
Escape: API Security Platform
Achieving end-to-end type safety in a moder...
Types
Types
Schema
Request
Request
Client
API
Database
Sabin Adams
End-To-End Type Safety: What, Why and How
END-TO-END TYPE SAFETY USING NUXT AND TRPC
Cody Bontecou
YouTube
End-to-end type safety using Nuxt and tRPC...
ISSUE #12
Type safety end to end - Vercel on the edge
SOVTECH
SovTech
Blog - Issue #12 – Type safety end to end – ...
END-TO-END TYPE SAFETY USING NUXT AND TRPC
Cody Bontecou
YouTube
End-to-end type safety using ...
TYPE SAFE
TYPE SAFE
TYPE SAFE
TYPESCRIPT BACKEND
GRAPHQL API
TYPESCRIPT FRONTEND
Rémi de Juvigny
End-to-end Type Safety with TypeScript an...
1995
2009
2012
Sabin Adams
End-To-End Type Safety: What, Why and How
END-TO-END TYPE SAFE
A GUIDE FOR FRONTEND
THEJAM.DEV
YouTube
End to End Type Safety: A G...
Split Software
Creating End-to-End Type Safety In a M...
ERIK HANCHETT
End-to-end Type Safety
YouTube
End-to-end Type Safety | VueConf U...
craigs/e2e-type-safety
End to end type safety with TypeScript, GraphQL and NextJS
PiterPy
2024
Dealing with End-to-end Type Safety for Our Backends
Denis Anikin
Raiffeisen Bank
GIF
GIF
End-to-end Type Safety w/ TypeScript and GraphQL
#dev #typescript

DGTL

Спасибо, что послушали