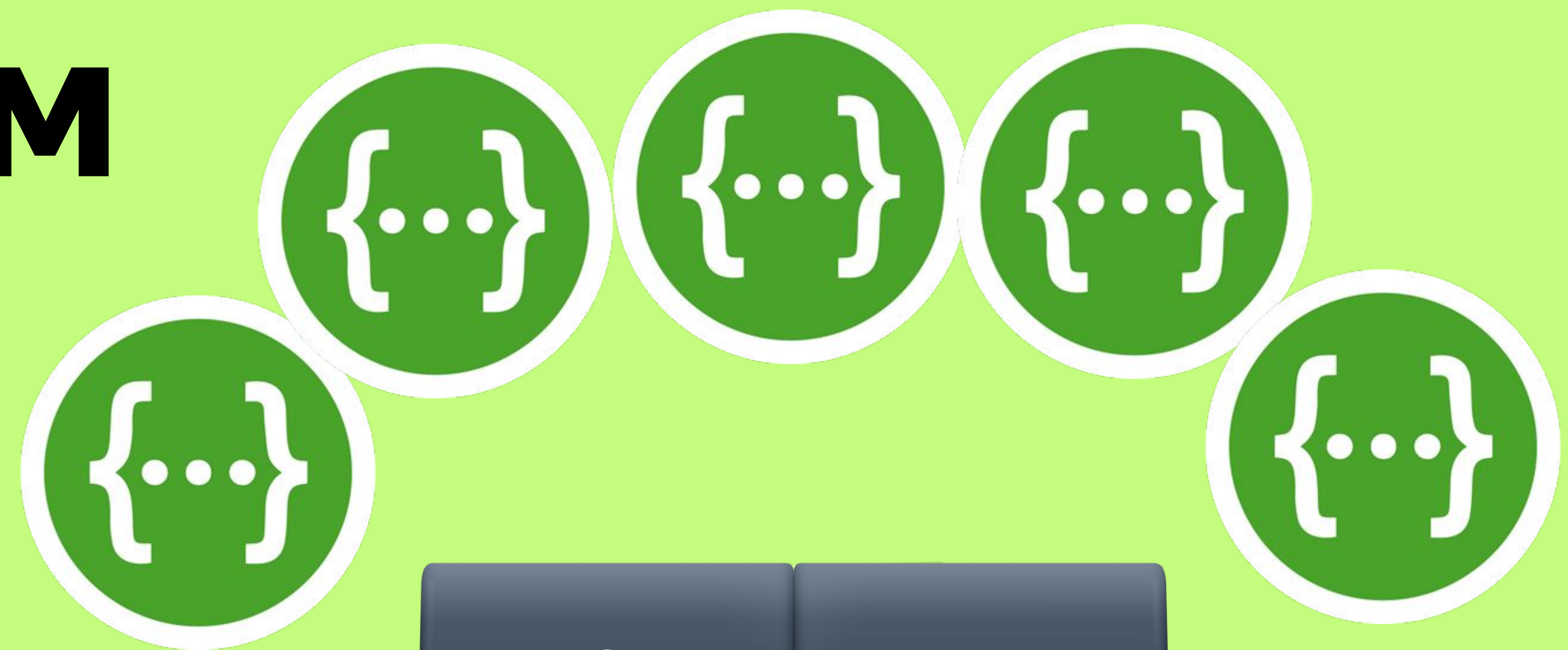


Наедине с тремя сотнями OpenAPI-схем



avito.tech

Данила Фомин

Старший разработчик команды PaaS/dev

Я

- Два года работаю в Авито кодогенератором,
- Год валидирую OpenAPI,
- До этого три года валидировал перловый код глазами.



План:

- Вспомним, что такое OpenAPI;
- Вспомним про то, что мы ошибаемся;
- Посмотрим, как мы в Авито стараемся избегать ошибок;
- Подумаем, как писать OpenAPI-спеки так, чтобы ошибок было меньше.



OpenAPI

Это формат описания HTTP API

Мы в Авито его используем

... и вы, скорее всего, тоже

```
openapi: 3.0.0
info:
  version: 1.0.0
  title: wordcounter

paths:
  /example/path:
    get:
      Try it | Audit
      parameters:
        - name: bar
          in: query
          schema:
            type: integer
      responses:
        200:
          description: OK
          content:
            application/json:
              schema:
                type: object
                properties:
                  foobar:
                    type: string
```

Ситуация, которая случалась с каждым



Барсик 10:42 (Прочитано)

Привет, в общем, фичу сделали

В ответе `/cats/and/dogs` теперь приходит `owner` вместо `ownerID`

Скажи, как выкатите, мы поле `ownerID` удалим



Пушок 16:33

Спасибо

Уже в проде ❤️



Барсик 16:34

Ой, то есть `owner`

Почему же Барсик ошибся?

Потому что:

- Обновления происходят посредством обсуждения в личке или таске,
- Валидация совместимости не автоматизирована,
- Все иногда ошибаются.

А Барсик не виноват!



Дано:

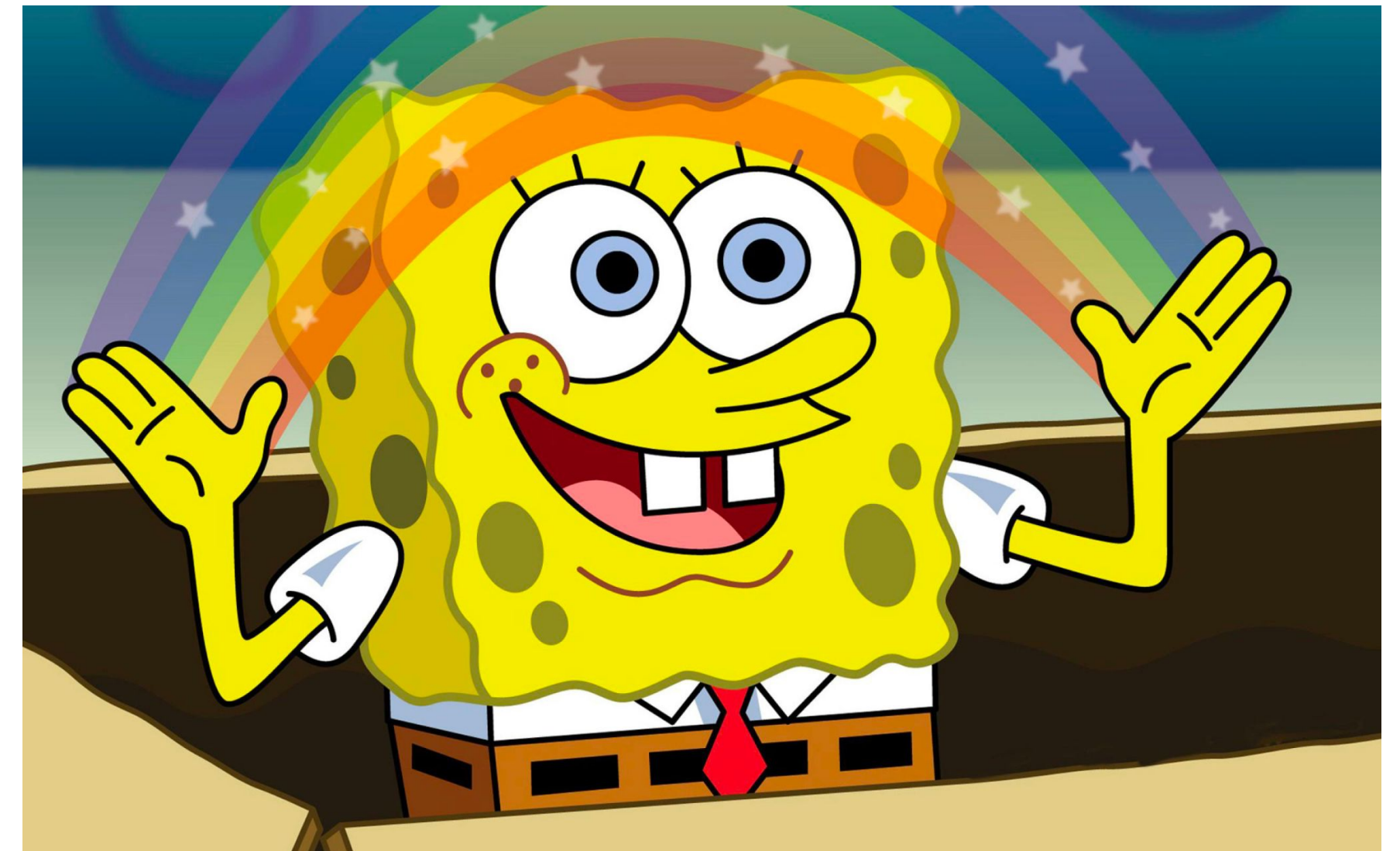
- Самописный **кодогенератор** Go из OpenAPI;
- **Over 5000** сгенерированных им **эндпоинтов** в 300 сервисах;
- Сайт Авито и несколько версий iOS- и Android-приложений, которые их используют;
- **Небывалая решимость помочь Барсику.**



Как помочь Барсику?

Нужна система, которая будет:

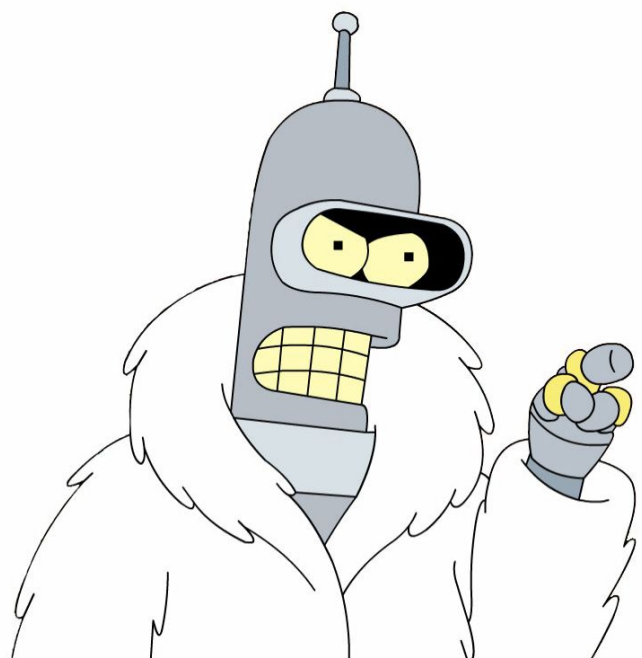
- Блокировать ломающие изменения OpenAPI;
- Гарантировать, что код в проде и OpenAPI консистентны;
- Ограничивать использование фичей OpenAPI.



И за 20 лет ничего подходящего не изобрели...

(на самом деле изобрели, но примерно пару месяцев назад)

... поэтому мы сделали её сами



Приступаем



Модель

You, 1 second ago | 1 author (You)

```
type Model struct {
  AppName string
  Storage *ref.Storage[RefTypeDesc] // map[ref.Ref]RefTypeDesc
  Paths []struct {
    Path string
    Methods map[string]struct {
      Name string
      Description string

      HeaderParams ref.Ref
      PathParams ref.Ref
      QueryParams ref.Ref
      CookieParams ref.Ref

      RequestBody *struct {
        Required bool
        Variants map[string]KindDesc // ref.Ref | Primitive
      }

      Responses map[int]struct {
        Description string
        Headers ref.Ref
        Body *struct {
          MimeType string
          Type KindDesc // ref.Ref | Primitive
        }
      }
    }
  }
}
```

Тут хранятся описания
всех объектов

Тут основная структура
OpenAPI-документа

```
type StructDesc struct {
  Nullable bool
  Description string

  Name string
  IsComponent bool

  Fields []struct {
    Name string
    Required bool
    Type KindDesc
  }
}
```

```
type Kind string

const (
  Reference Kind = "reference"
  Inplace Kind = "inplace"
)
```

```
type InplaceType string

const (
  String InplaceType = "string"
  Int InplaceType = "int"
  Float InplaceType = "float"
  Bool InplaceType = "bool"
  Binary InplaceType = "binary"
  Interface InplaceType = "interface"

  Array InplaceType = "array"
  Map InplaceType = "map"
)
```


Сравнение моделей

```
# example/service/schema.yaml
paths:
  /foo/bar:
    get:
      Try it | Audit
      requestBody:
        required: true
        content:
          application/json:
            schema:
              type: object
              required: [id, token]
              properties:
                id: {type: string}
                token: {type: string}
      responses:
        200: {description: Ok}
        201: {description: Created}
```

```
# example/client/schema.yaml
paths:
  /foo/bar:
    get:
      Try it | Audit
      requestBody:
        required: true
        content:
          application/json:
            schema:
              type: object
              required: [id]
              properties:
                id: {type: integer}
      responses:
        200: {description: Ok}
```

```
func main() {
  serviceModel, _ := ReadOpenAPI("example/service", "schema.yaml")
  clientModel, _ := ReadOpenAPI("example/client", "schema.yaml")

  errors := differ.Diff(clientModel, serviceModel)

  for _, err := range errors {
    fmt.Println("=====")
    fmt.Println(err)
  }

  fmt.Println("=====")
}
```

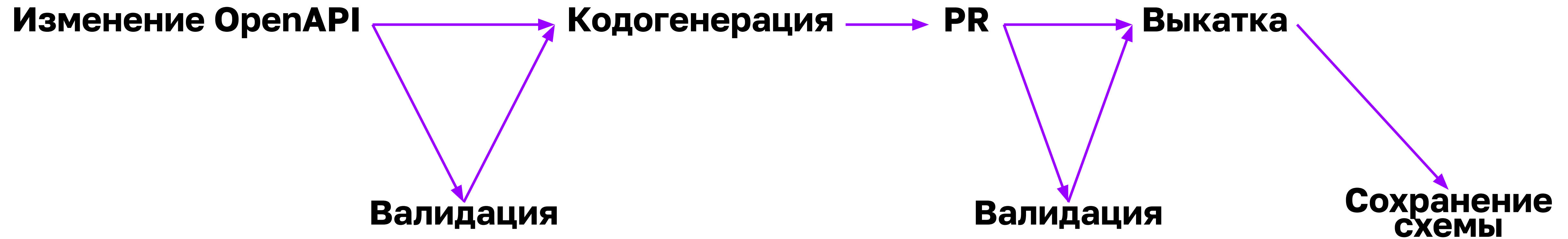
% go run ./example

=====
 типы по пути "/foo/bar.GET.requestBody.[application/json].id"
 не совместимы на клиенте (int) и в сервисе (string)
 =====

для входящего параметра
 "/foo/bar.GET.requestBody.[application/json]" в сервисе
 объявлено required поле "token", о котором не знает клиент
 =====

для ответа "/foo/bar.GET" в сервисе есть код 201, но клиент
 его не учитывает
 =====

Добавляем проверку в цикл разработки



С чем работает Барсик (бэкендер)



- Пока есть несовместимость с сохранённой схемой клиента:
 - код сервиса не сгенерируется,
 - сервис не выкатится.
- После выкатки новая OpenAPI-схема сервиса будет сохранена.

С чем работает Пушок (фронтендер)



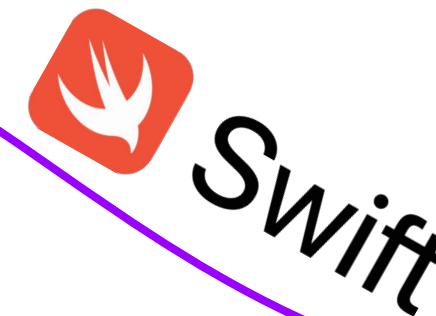
- Пока есть несовместимость с сохранённой схемой сервиса:
 - код клиента не сгенерируется,
 - фронтенд не выкатится.
- После выкатки фронтенда его OpenAPI-схема будет сохранена.

Единая кодогенерация

```
paths:
  /foo/bar:
    get:
      Try it | Audit
      requestBody:
        required: true
        content:
          application/json:
            schema:
              type: object
              required: [id, token]
              properties:
                id: {type: string}
                token: {type: string}
      responses:
        200:
          description: Ok
          content:
            application/json:
              schema:
                type: object
                required: [warnings]
                properties:
                  warnings:
                    type: array
                    items: {type: string}
```



% avito codegen



```
type JsonDto struct {
  Id string `json:"id" form:"id"`
  Token string `json:"token" form:"token"`
}

type OkRespData struct {
  Warnings []string `json:"warnings" form:"warnings"`
}
```

```
export type JsonDto = {
  id: string
  token: string
}

export type OkRespData = {
  warnings: Array<string>
}
```

```
/* DTOs for `GET: /foo/bar` */

data class JsonDto(
  @SerializedName("id") val id: String,
  @SerializedName("token") val token: String,
)

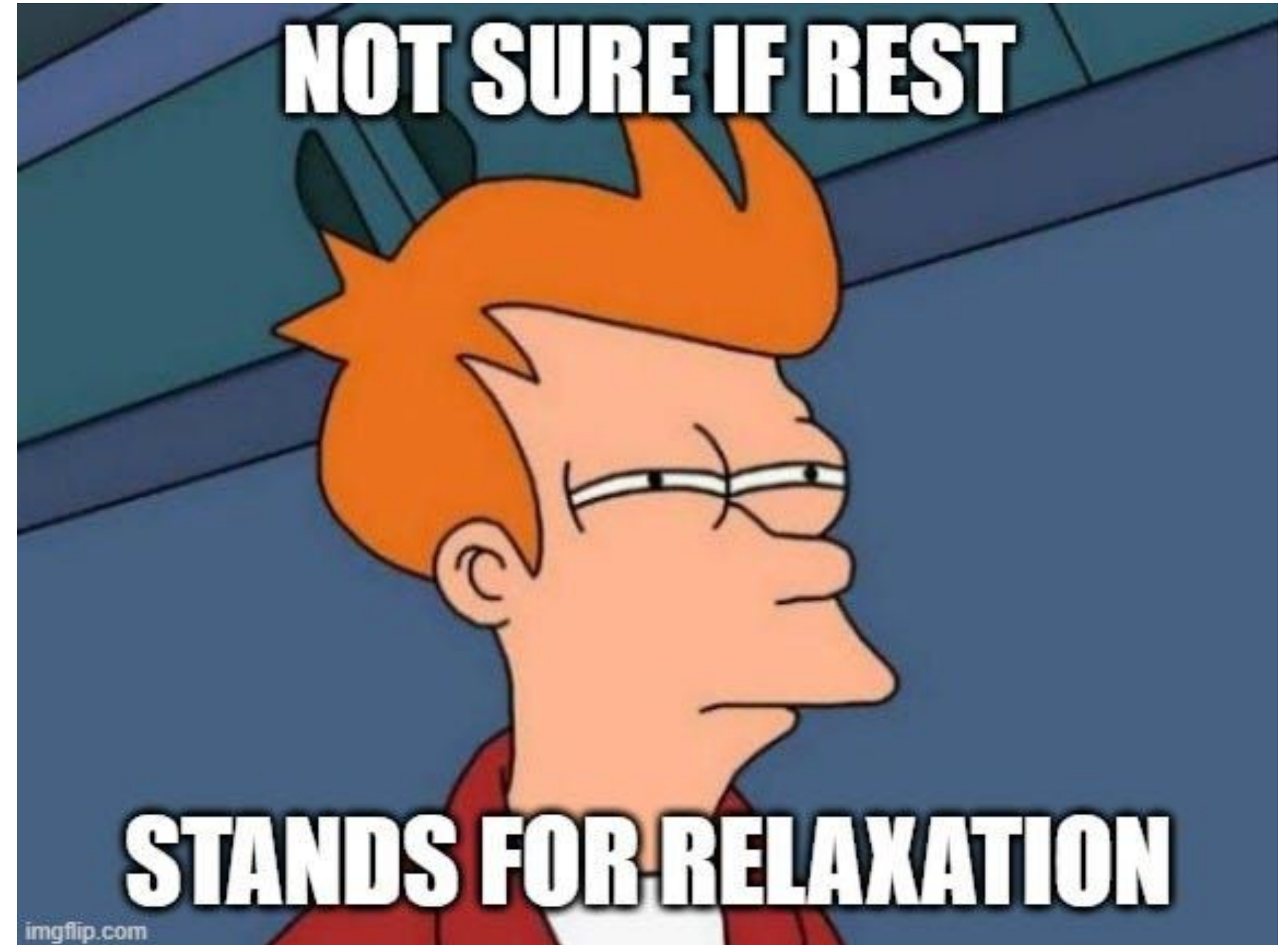
data class OkRespData(
  @SerializedName("warnings") val warnings: List<String>,
)
```

```
/// DTOs for `GET: /foo/bar`
extension GetFooBar {
  struct JsonDto: Codable {
    private enum CodingKeys: String, CodingKey {
      case id = "id"
      case token = "token"
    }
    var id: String
    var token: String
  }

  struct OkRespData: Codable {
    private enum CodingKeys: String, CodingKey {
      case warnings = "warnings"
    }
    var warnings: [String]
  }
}
```



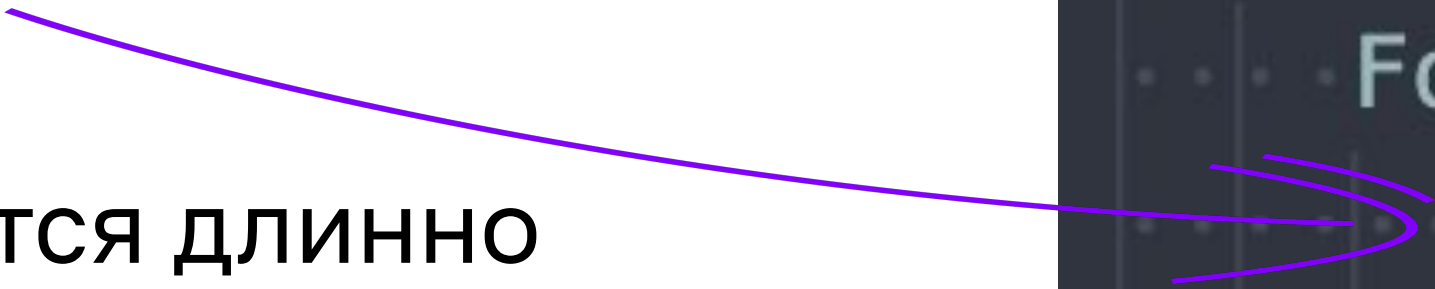

Габбли OpenAPI



Габри OpenAPI: имена типов

Всегда указывайте имена типов

Если их генерировать — получается длинно и непонятно.



```
components:
  schemas:
    FooBar:
      title: FooBar
      type: object
      required: [foobar]
      properties:
        foobar:
          type: string
```


Грaбли OpenAPI: схемы вне компонентов

Все схемы объектов пишите в компонентах

Иначе схема получается дико вложенная и непонятная.

А ещё это помогает избегать коллизии имён.



```
components:
  schemas:
    FooBar:
      title: FooBar
      type: object
      required: [foobar]
      properties:
        foobar:
          type: string
```

Грaбли OpenAPI: опциональность

```
components:
  schemas:
    FooBar:
      title: FooBar
      type: object
      properties:
        foobar:
          # foobar — не required
          type: string
```

```
{ } # валидно
{"foobar": "some string"} # валидно
{"foobar": null} # НЕВАЛИДНО
```


Габри OpenAPI: опциональность

```
components:
  schemas:
    FooBar:
      title: FooBar
      type: object
      required: [foobar] # foobar required
      properties:
        foobar:
          nullable: true # ... и nullable
          type: string
```

```
{ } # НЕВАЛИДНО
{"foobar": "some string"} # валидно
{"foobar": null} # валидно
```

Грaбли OpenAPI: опциональность

```
components:
  schemas:
    FooBar:
      title: FooBar
      type: object
      properties:
        foobar:
          nullable: true # не required и nullable
          type: string
```

```
{ } # валидно
{"foobar": "some string"} # валидно
{"foobar": null} # валидно
```

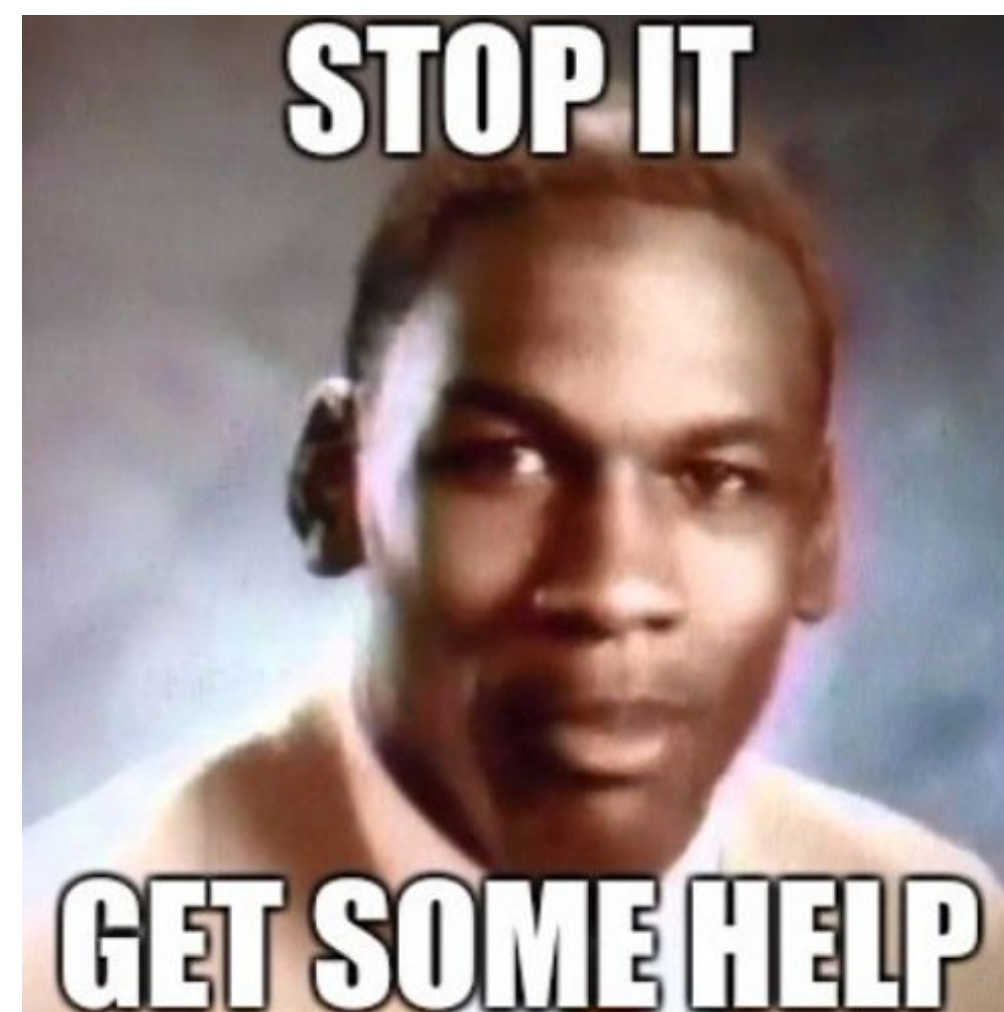
Большинство ЯП со статической типизацией не различают отсутствие поля и null.

Как признак опциональности одновременно используйте `not-required` и `nullable`.

Грабли OpenAPI: объекты где не надо

OpenAPI позволяет беспрепятственно использовать `type: object/array`, где пожелаешь

Используйте `object/array` только с `json` и ваша жизнь станет в разы проще.



```
paths:
  /example/path:
    get:
      Try it | Audit
      parameters:
        - name: bar
          in: query
          schema:
            type: object
            required: [foo, bar]
            properties:
              foo:
                type: object
                required: [boo, far]
                properties:
                  boo: {type: string}
                  far: {type: number}
              bar:
                type: object
                required: [oob, raf]
                properties:
                  oob: {type: string}
                  raf: {type: number}
```


Габри OpenAPI: oneOf

```
components:
  schemas:
    Foo:
      type: object
      required: [userID, foo]
      properties:
        userID: {type: string}
        foo: {type: string}
    Bar:
      type: object
      required: [userID, bar]
      properties:
        userID: {type: string}
        bar: {type: string}
    FooBar:
      title: FooBar
      oneOf:
        - $ref: '#/components/schemas/Foo'
        - $ref: '#/components/schemas/Bar'
        - type: string
```

```
type Foo = { userID: string, foo: string }
type Bar = { userID: string, bar: string }
type FooBar = Foo | Bar | string

function doSomething(fooBar: FooBar) {
  if (typeof fooBar === 'string') {
    // ...
  } else if ('foo' in fooBar) {
    // ...
  } else if ('bar' in fooBar) {
    // ...
  }
}
```



Грабли OpenAPI: oneOf (discriminator)

```
components:
  schemas:
    Foo:
      type: object
      required: [type, userID, foo]
      properties:
        type: {type: string}
        userID: {type: string}
        foo: {type: string}
    Bar:
      type: object
      required: [type, userID, bar]
      properties:
        type: {type: string}
        userID: {type: string}
        bar: {type: string}
    FooBar:
      title: FooBar
      oneOf:
        - $ref: '#/components/schemas/Foo'
        - $ref: '#/components/schemas/Bar'
        # — type: string — это нам придется удалить
      discriminator:
        propertyName: 'type'
        mapping:
          foo: '#/components/schemas/Foo'
          bar: '#/components/schemas/Bar'
```

```
type Foo = { type: 'foo', userID: string, foo: string }
type Bar = { type: 'bar', userID: string, bar: string }
type FooBar = Foo | Bar

function doSomething(fooBar: FooBar) {
  if (fooBar.type == 'foo') {
    // ...
  } else if (fooBar.type == 'bar') {
    // ...
  }
}
```



```
type Foo struct {
  Type string `json:"type"`
  UserID string `json:"userID"`
  Foo string `json:"foo"`
}
type Bar struct {
  Type string `json:"type"`
  UserID string `json:"userID"`
  Bar string `json:"bar"`
}

func UnmarshalFooBar(fooBar []byte) {
  wrapper := struct {
    Type string `json:"type"`
  }{}

  _ = json.Unmarshal(fooBar, &wrapper)

  if wrapper.Type == "foo" {
    foo := Foo{}
    _ = json.Unmarshal(fooBar, &foo)

    // ...
  }

  if wrapper.Type == "bar" {
    bar := Bar{}
    _ = json.Unmarshal(fooBar, &bar)

    // ...
  }
}
```


Грабли OpenAPI: oneOf (резюме)

OneOf в OpenAPI сломан, потому что имя варианта представлено внутри типа.

- Это очень неудобно валидировать в языках со статической типизацией,
- Это очень неудобно сравнивать на совместимость,
- Это имеет неспецифицированные краевые случаи.

```
components:
  schemas:
    Foo:
      type: object
      required: [type, foo]
      properties:
        type: {type: string}
        foo: {type: string}
    Bar:
      type: object
      required: [type, bar]
      properties:
        type: {type: string}
        bar: {type: string}
    FooBar:
      oneOf:
        - $ref: '#/components/schemas/Foo'
        - $ref: '#/components/schemas/Bar'
        - type: string
      discriminator:
        propertyName: type
        mapping:
          foo: '#/components/schemas/Foo'
          bar: '#/components/schemas/Bar'
          # type: string ????
```


Габбли OpenAPI: oneOf (решение)

Лучше используйте **object** у которого **все поля не-required**.

В JSON-представлении присутствует только поле для нужного варианта.

```
{"foo": {"userID": "some_id", "foo": "some_foo"}} # это вариант foo
{"bar": {"userID": "some_id", "bar": "some_bar"}} # это вариант bar
{"string": "some_string"} # это вариант string
```

- Это компактнее,
- Это однозначнее,
- Это проще валидировать,
- Это сгенерируется любым кодогенератором.

Для вашего кодогенератора можно сделать кастомный *format: enum*.

```
components:
  schemas:
    Foo:
      type: object
      required: [userID, foo]
      properties:
        userID: {type: string}
        foo: {type: string}
    Bar:
      type: object
      required: [userID, bar]
      properties:
        userID: {type: string}
        bar: {type: string}
    FooBar:
      title: FooBar
      type: object
      # сторонний кодогенератор
      # проигнорирует формат
      format: enum
      properties:
        foo: {$ref: '#/components/schemas/Foo'}
        bar: {$ref: '#/components/schemas/Bar'}
        string: {type: string}
```

Вывод:

OpenAPI-спецификация щедро сдобрена граблями, а человеческий фактор никто не отменял.

Постоянное катание на этих граблях в Авито привело к тому, что теперь:

- Эндпоинты валидируются на совместимость;
- Мы можем отслеживать и удалять поля, которые не используются;
- Мы можем централизованно добавлять любые валидации;
- У нас есть общая кодогенерация для сервиса и всех клиентов.

И Барсик никогда не виноват!



avito.tech
avito.tech
avito.tech
avito.tech
avito.tech

Данила Фомин



ddf1998@gmail.com



@dedefer

