

# 型付きAPIリクエストを実現する いくつかの手法とその選択

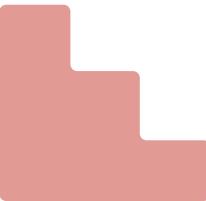
Presented by ユーン  
@TSKaigi Kansai 2024

# 自己紹介

ユーン (@euxn23)

- ドワンゴ教育事業のエンジニア
- ZEN ID という認証基盤を作っています
  - Web フロント
  - 前段バックエンド (TypeScript)
- RABBIT 小隊が好き





2025年4月 ZEN大学 開学



## PICK UP



12日（火）Web出願の受付を開始！

出願開始から1ヶ月以内の出願者を対象として、  
希望者に2つの特典！

簡単申込！ 無料資料請求はこちら



11月12日（火）13時～「ZEN大学発表会～2025年4月開学～2024年10月31日設置認可」



大学職員 &amp; システム開発エンジニア

# 最先端の学びを すべての人に ZEN大学。

ZEN大学は、最先端のIT技術を活用し、すべての人に大学進学の機会を提供します。ZEN大学唯一の学部である「知能情報社会学部」では、特定の学問領域に偏らない学びを修めることで、激変するAI時代に対応して活躍するために必要なリテラシーを身につけることができます。



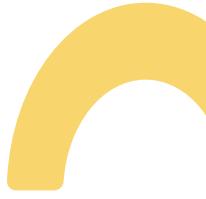
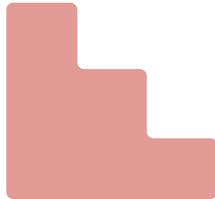
Web出願

説明会・相談会

資料請求



このセッションでは、  
安全な API リクエストを実現する  
いくつかの手法を紹介します



# API 連携の安全性確保のための手法

## コードファーストでない

- API 仕様書 (NOT OpenAPI) を書く
- 結合テストを増やす
- 監視で異常系を発見する

## コードファーストな

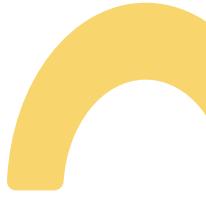
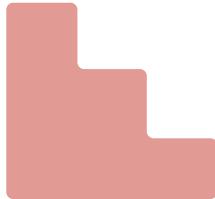
- 実装や定義を共有する
- OpenAPI を実装と結びつける
- フレームワークの機能に乗る

# コードファーストでない API 連携の課題

- API 仕様書
  - 仕様と実装の乖離
- 結合テスト
  - テストケースが多い
- 監視
  - 発見までにエラーは発生してしまう

# コードファーストにするとどう解決されるか

- API仕様書
  - 仕様（OpenAPI）と実装が関連付く、型により守られる
- 結合テスト
  - 型で表現できるレベルのテストケースは省略可能になる
- 監視
  - 型によりコーディングタイムでバグを発見しやすくなる



## コードファーストなアプローチを考える

# コードファーストなアプローチ

## TypeScript ファーストな手法 言語に依存しない手法

- 型定義の共有
- Zod スキーマの共有
- フレームワークの機能の利用
  - Hono RPC
  - tRPC
- サーバコードから OpenAPI を自動生成
- OpenAPI からクライアントコードを自動生成

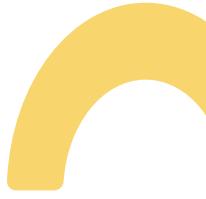
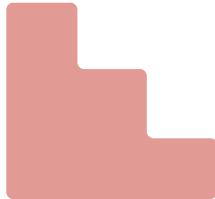
# コードファーストなアプローチ

## TypeScript ファーストな手法

- 型定義の共有
- Zod スキーマの共有
- フレームワークの機能の利用
  - Hono RPC
  - tRPC

## 言語に依存しない手法

- サーバコードから OpenAPI を自動生成
- OpenAPI からクライアントコードを自動生成



注意: ほとんどのケースで monorepo を前提とします

# 1. 型定義の共有

## サーバ側の型定義

```
export type GetPetRequest = {
  param: {
    id: string
  }
};
export type GetPetResponse = Pet;
export type PostPetRequest = {
  body: Pet
};
export type PostPetRequest = never;
```

# 1. 型定義の共有

## クライアント側の実装

```
import type { GetPetRequest, GetPetResponse, PostPetRequest, PostPetResponse } from '@/server';

export async function requestGetPet(req: GetPetRequest): Promise<GetPetResponse> {
    return fetch(`#/api/pets/${req.param.id}`).then(res => res.json());
}

export async function requestPostPet(req: PostPetRequest): Promise<PostPetResponse> {
    return fetch(`/api/pets`, { method: 'POST', body: JSON.stringify(req.body) })
        .then(res => res.json());
}

const pet = await requestGetPet({ param: { id: '1' } });
const newPet = {
    // ...
};
await requestPostPet({ body: newPet });
```

# 1. 型定義の共有

## Pros

- 外部ライブラリや特別な設計に依存しないので簡単に始めやすい

## Cons

- fetch の中は型がわからないまま書くしかない
- API client のレイヤが別れている必要がある

## 2. Zod スキーマの共有

### 共通コード

```
const PetTagSchema = z.object({
  id: z.number(),
  name: z.string(),
});

const PetSchema = z.object({
  id: z.string(),
  name: z.string(),
  tags: z.array(PetTagSchema),
});
```

<https://github.com/colinthacks/zod>

## 2. Zod スキーマの共有

### クライアント側の実装

```
async function postPet(pet: z.infer<typeof PetSchema>): Promise<void> {
  const result = PetSchema.safeParse(pet);
  if (!result.success) {
    throw new Error('Invalid pet');
  }
  await fetch('/api/pets', { method: 'POST', body: JSON.stringify(pet) });
}
```

## 2. Zod スキーマの共有

サーバ側の実装(例: Express)

```
app.post('/pets', async (req, res) => {
  const result = PetSchema.safeParse(req.body);
  if (!result.success) {
    res.status(400).json(result.error);
    return;
  }
  const pet = result.data;
  const inserted = await insertPet(pet)
  res.json(inserted);
  return;
});
```

## 2. Zod スキーマの共有

### Pros

- バリデーションと型定義が一致する
- バリデーションロジックごと  
サーバ・クライアントで共有できる

### Cons

- schema のプロパティ変更は infer  
した型の変数には波及しないため、  
type で引き回す方が良い
- 型定義と Zod の Schema の二重管理  
が発生するが、乖離を防ぐため次の  
ような工夫が必要

## 工夫の例

```
export type PetTag = {
  id: number;
  name: string;
};

export type Pet = {
  id: string;
  name: string;
  tags: PetTag[];
};

const PetTagSchema = z.object({
  id: z.number(),
  name: z.string(),
}) satisfies ZodType<PetTag>;

const PetSchema = z.object({
  id: z.string(),
  name: z.string(),
  tags: z.array(PetTagSchema),
}) satisfies ZodType<Pet>;
```

### 3. フレームワークの機能の利用 - Hono RPC

```
const route = app.post(
  '/posts',
  zValidator(
    'form',
    z.object({
      title: z.string(),
      body: z.string(),
    })
  ),
  (c) => {
    // ...
    return c.json({
      ok: true,
      message: 'Created!',
    }, 201)
  }
)

export type AppType = typeof route
```

### 3. フレームワークの機能の利用 - Hono RPC

```
import { AppType } from '.'
import { hc } from 'hono/client'

const client = hc<AppType>('http://localhost:8787/')

const res = await client.posts.$post({
  form: {
    title: 'Hello',
    body: 'Hono is a cool project',
  },
})

if (res.ok) {
  const data = await res.json()
  console.log(data.message)
}
```

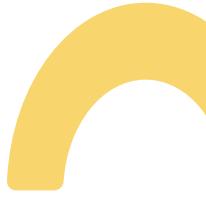
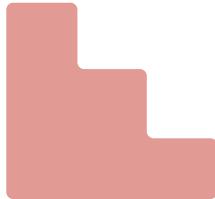
# TypeScript ファーストな手法

## Pros

- コードの共有から小さく始めやすい
- フレームワークの機能に乗れると  
高速に開発ができる

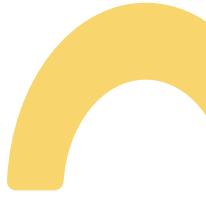
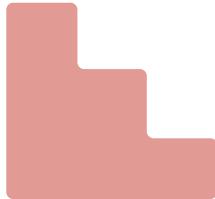
## Cons

- TypeScript であることに強く依存  
→片方を別の言語で書き換えると、  
安全性は失われてしまう

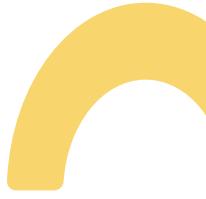
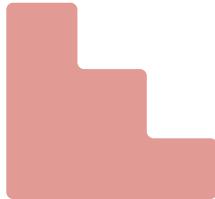


TypeScript に依存してしまっては、  
柔軟さか安全さのどちらかが将来失われてしまう

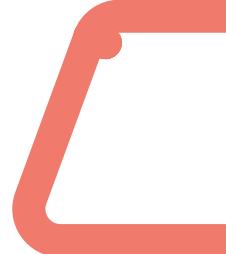
アバンパート終了



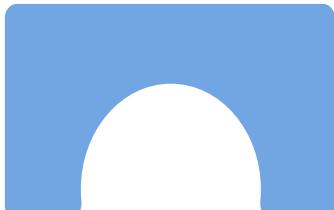
どうする！？ TypeScript と心中するか！？



突如そこに OpenAPI が！



OpenAPI ベースな  
型付き API リクエスト



# コードファーストなアプローチ

## TypeScript ファーストな手法

- 型定義の共有
- Zod スキーマの共有
- フレームワークの機能の利用
  - Home RPC
  - gRPC

## 言語に依存しない手法

- サーバコードから OpenAPI を自動生成
- OpenAPI からクライアントコードを自動生成

# OpenAPI のよいところ

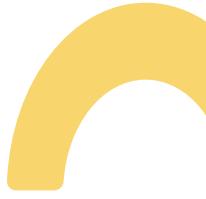
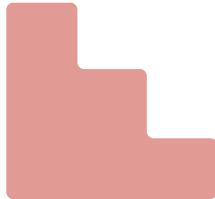
- 言語非依存
- 仕様が標準化されている
- エコシステムが成熟しており、各言語にツールがある
- 既存の実装を損なわずに採用できるケースがままある
- etc…

# なぜ gRPC や GraphQL ではないか

- gRPC
  - HTTP/2 が必要
  - プロトコルバッファの表現力の問題
  - Go まわり以外、エコシステムがあまり成熟していない
- GraphQL
  - そもそもが根本的に難しい
  - バックエンドの実装負担が大きい
  - 既存の実装を GraphQL 化するのは難しい

## OpenAPIと実装を繋ぐ手法

- サーバコードから OpenAPI を生成
- OpenAPI から API クライアントを生成
- OpenAPI からサーバコードを生成



サーバコードから OpenAPI を自動生成

# サーバコードから OpenAPI を生成

サーバコードから OpenAPI を  
生成する機能を持つフレームワーク

- NestJS
- Hono (@hono/zod-openapi)
- FastAPI (Python)
- etc…

# NestJS のサンプルコード

```
export class Cat {  
    /**  
     * The name of the Cat  
     * @example Kitty  
     */  
    name: string;  
  
    @ApiProperty({ example: 1, description: 'The age of the Cat' })  
    age: number;  
  
    @ApiProperty({  
        example: 'Maine Coon',  
        description: 'The breed of the Cat',  
    })  
    breed: string;  
}
```

<https://github.com/nestjs/nest/tree/master/sample/11-swagger>

```
@ApiBearerAuth()
@ApiTags('cats')
@Controller('cats')
export class CatsController {
    constructor(private readonly catsService: CatsService) {}

    @Post()
    @ApiOperation({ summary: 'Create cat' })
    @ApiResponse({ status: 403, description: 'Forbidden.' })
    async create(@Body() createCatDto: CreateCatDto): Promise<Cat> {
        return this.catsService.create(createCatDto);
    }

    @Get(':id')
    @ApiResponse({
        status: 200,
        description: 'The found record',
        type: Cat,
    })
    findOne(@Param('id') id: string): Cat {
        return this.catsService.findOne(+id);
    }
}
```

# @hono/zod-openapi のサンプルコード

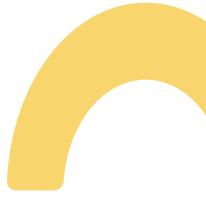
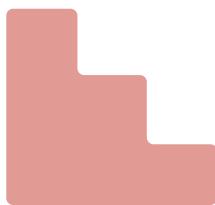
```
const route = createRoute({
  method: 'get',
  path: '/users/{id}',
  request: {
    params: ParamsSchema,
  },
  responses: {
    200: {
      content: {
        'application/json': {
          schema: UserSchema,
        },
      },
      description: 'Retrieve the user',
    },
  },
})
```

<https://hono.dev/examples/zod-openapi>

```
const app = new OpenAPIHono()

app.openapi(route, (c) => {
  const { id } = c.req.valid('param')
  return c.json({
    id,
    age: 20,
    name: 'Ultra-man',
  })
})

// The OpenAPI documentation will be available at /doc
app.doc('/doc', {
  openapi: '3.0.0',
  info: {
    version: '1.0.0',
    title: 'My API',
  },
})
```



FastAPI の例は省略 (Pythonなので)

# サーバ実装から OpenAPI を生成することの是非

## Pros

- 仕様と実装が一致する
- 常に最新になる
- OpenAPI を手書きしなくてよい

## Cons

- 実装を変更しないと OpenAPI を変更できない
- まだ実装されていないが変更予定のものを OpenAPI にしにくい
- OpenAPIを中心として議論しにくい

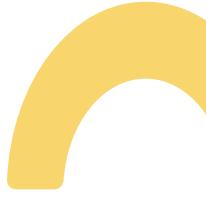
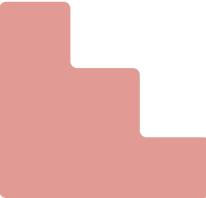


誰がOpenAPIを書くか (=API仕様を決めるか)

に応じて選択しましょう

→ みんなで書くなら実装ベースじゃない方が

議論もしやすい



OpenAPI から API クライアントを生成

# OpenAPI から API クライアントを 生成するライブラリ

- openapi-fetch
- orval
- etc…

# openapi-typescript / openapi-fetch

- openapi-typescript は OpenAPI から TS コードを生成
- openapi-fetch はこれを元に API Client コードを生成

<https://openapi-ts.dev/>

<https://openapi-ts.dev/openapi-fetch/>

# openapi-typescript のサンプルコード

```
import { paths, components } from "./path/to/my/schema"; // ← generated by openapi-typescript

// Schema Obj
type MyType = components["schemas"]["MyType"];

// Path params
type EndpointParams = paths["/my/endpoint"]["parameters"];

// Response obj
type SuccessResponse = paths["/my/endpoint"]["get"]["responses"][200]["content"]["application/json"]
type ErrorResponse = paths["/my/endpoint"]["get"]["responses"][500]["content"]["application/json"][]
```

# openapi-fetch のサンプルコード

```
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" },
  },
});

await client.PUT("/blogposts", {
  body: {
    title: "My New Post",
  },
});
```

<https://openapi-ts.dev/openapi-fetch/>

# orval

- Axios ベースの API Client コードを生成
- TanStack Query や SWR、Zod との連携もある

<https://orval.dev/>

# orval のサンプルコード

```
import type { CreatePetsBody } from '../model';
import { customInstance } from '../mutator/custom-instance';

export const createPets = (
    createPetsBody: CreatePetsBody,
    version: number = 1,
) => {
    return customInstance<void>({
        url: `/v${version}/pets`,
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        data: createPetsBody,
    });
};
```

<https://github.com/anymaniac/orval/blob/master/samples/react-app>

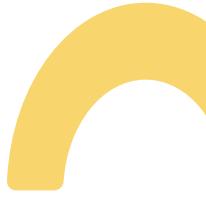
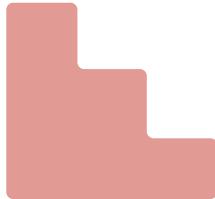
# OpenAPI から API クライアントを生成

## Pros

- 実装が変わった際に型エラーで検知できる
- エンドポイントのパスやパスパラメータも補完・チェックできる

## Cons

- OpenAPI を実装から生成していない場合、実装との乖離は起こりうる



OpenAPI からサーバコードを生成

# OpenAPI からサーバコードを生成

残念ながら、TS ではメジャーな(十分枯れた)ものはない

- 例えば golang なら oapi-codegen や ogen などがある
- ogen は interface を生成、それに合わせて実装する
- レイヤが別れるので後入れは難しい、開発初期なら検討の余地あり



Q. ところで OpenAPI は yaml を手書きするの？

A. OpenAPI を書くための次世代 DSL、TypeSpec がある



# TypeSpec

- TypeScript / C# 風味の DSL で OpenAPI Schema を書けるライブラリ
- LanguageServer を提供されており、VSCode 向けにプラグインを提供
- コンパイル時に valid な記法か確認されるので安心

<https://typespec.io/>

```
import "@typespec/http";

using TypeSpec.Http;

model PetTag {
    id: number;
    name: string;
};

model Pet {
    id: string;
    name: string;
    tags: PetTag[];
}

@route("/pets/{id}")
interface Stores {
    @operationId("get-pet")
    @summary("Get a pet by ID")
    @get
    get(@path id: string): {
        @statusCode
        statusCode: 200;
        @body Pet;
    };
}
```

## yaml 手書きと比べて

- 複雑でない module システムを備え、ファイルの分割や再利用が容易
- デフォルト値が設定されているものは省略可能であり、文量が少ない
- monorepoにも対応、別パッケージの定義を参照することも可能

# まとめ

- サーバとクライアントの TS コード共有は楽だが、結合レイヤの TS 依存はリスクである
- 結合レイヤはエコシステムが充実しており、言語非依存の OpenAPI を用いるのが良いのではないか
- OpenAPI により仕様と実装を近づけることでコード面の安全性や開発効率の向上が期待できる

## 想定質問

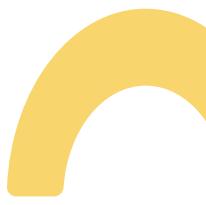
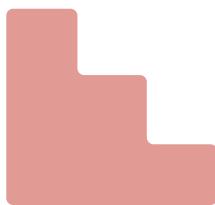
Q. OpenAPI とバックエンドの乖離はどうするの？

A. テストケースを書いて頑張ろう(ここは課題です)

結局バックエンドの実装と仕様を一致させるのが一番大変ですよね

でも、乖離しうる箇所を減らせているだけでも大きな進歩です

テスト用のクライアントコードを OpenAPI から生成するなどして頑張りましょう



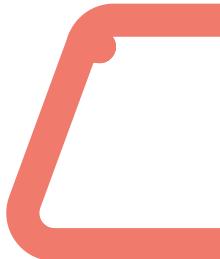
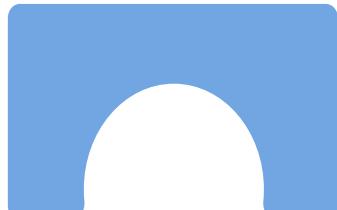
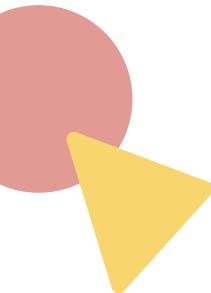
Any question?

ドワンゴのスポンサーブースにいます、遊びに来てね

質問やディスカッションも大歓迎



ありがとうございました



## Appendix: 過去に話した関連する登壇資料

- NestJS アプリケーションから Swagger を自動生成する
- Powerfully Typed TypeScript
- TypeSpec を使い倒して



## Appendix: 関連する有益な参考資料

- WebフロントエンドにおけるGraphQL（あるいはバックエンドのAPI）との向き合い方
- 見よ、これがHonoのRPCだ
- Hono × Zod-OpenAPIで快適API開発
- 最近のGoのOpenAPI Generatorの推しはogen

