

A collection of various geometric shapes including triangles, squares, circles, and polygons in yellow, red, and blue, scattered around the text.

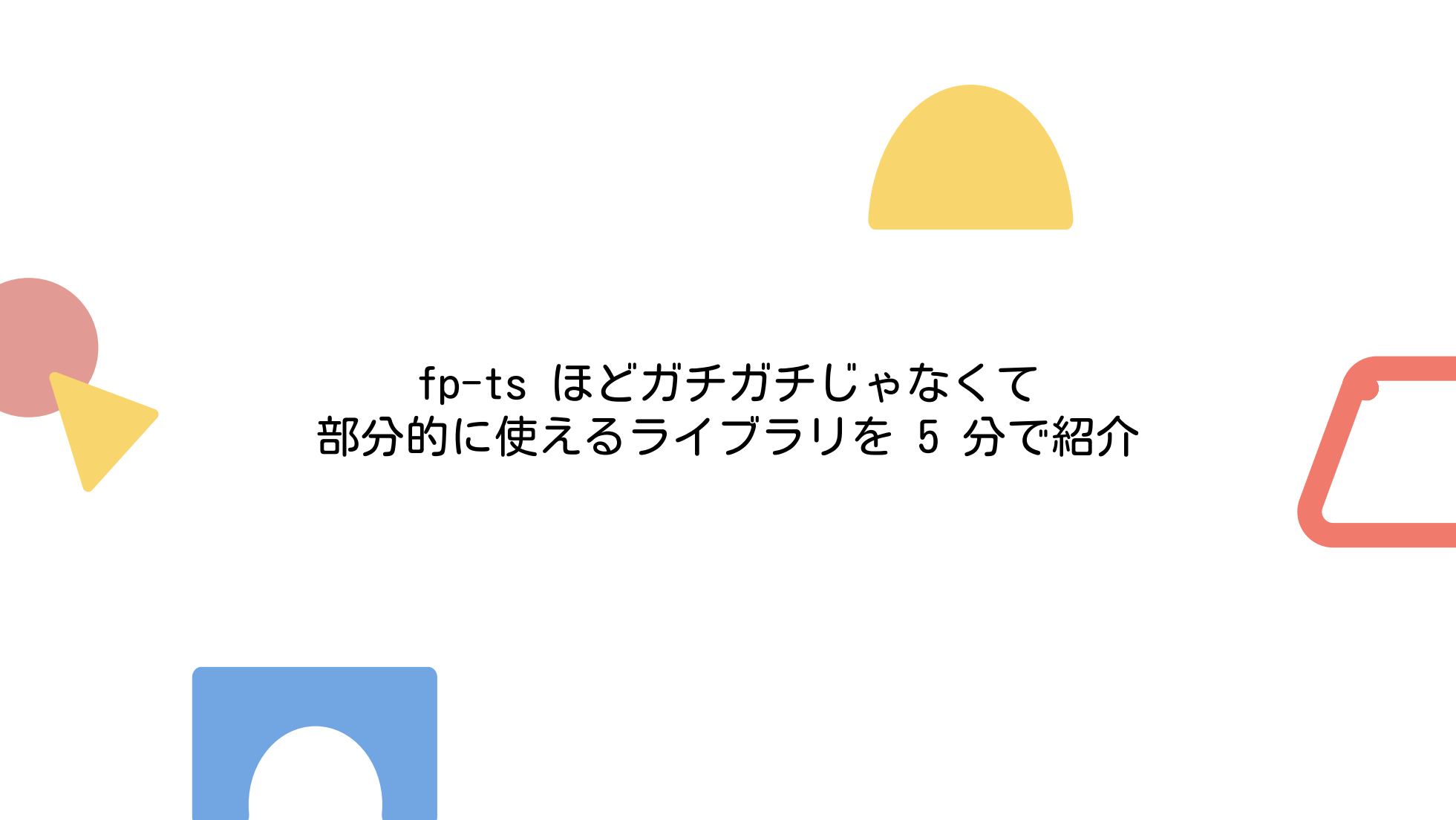
Powerfully Typed TypeScript

自己紹介



ユーン (@euxn23)

- from ドワンゴ教育事業
- フロントだけじゃない
- RABBIT 小隊が好き



fp-ts ほどガチガチじゃなくて
部分的に使えるライブラリを 5 分で紹介

zod

- 言わずと知れたバリデーション用ライブラリ
- 構造の定義からランタイムバリデータと型定義を生成できる
- `infer` が TS Server への負荷が高いため多用に注意

```
import { z } from "zod";

const User = z.object({
  username: z.string(),
});

User.parse({ username: "Ludwig" });

type User = z.infer<typeof User>;
```

<https://github.com/colinhacks/zod#basic-usage>

never throw

- TS に Result 型をもたらしてくれるライブラリ
- throw の代わりに使ってエラー時の処理を追いやすくする
- 0 Dependencies

```
const fetchUser = async (): Promise<Result<User, Error>> => {  
  const response = await fetch(...)  
  if (response.ok) {  
    return ok((await response.json()) as User)  
  } else {  
    return err(new Error(...))  
  }  
}  
  
const fetchResult = await fetchUser()  
  
if (fetchResult.isErr()) {  
  return fetchResult.error // ← Error 型である  
}  
return fetchResult.value // ← User 型である
```

ts-pattern

- TS にパターンマッチをもたらしてくれるライブラリ
- 0 Dependencies

```
import { match, P } from 'ts-pattern';

type Data =
  | { type: 'text'; content: string }
  | { type: 'img'; src: string };

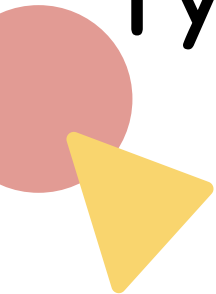
type Result =
  | { type: 'ok'; data: Data }
  | { type: 'error'; error: Error };

const result: Result = ...;


const html = match(result)
  .with({ type: 'error' }, () => <p>Oops! An error occurred</p>)
  .with({ type: 'ok', data: { type: 'text' } }, (res) => <p>{res.data.content}</p>)
  .with({ type: 'ok', data: { type: 'img', src: P.select() } }, (src) => <img src={src} />)
  .exhaustive();
```



TypeScript だけじゃない……！



OpenAPI + TypeScript で
さらに Powerful に



openapi-typescript & openapi-fetch

- openapi-typescript は OpenAPI Schema から TS コードを生成
- openapi-fetch はそれを利用した API Client コードを提供

```
$ npx openapi-typescript ./path/to/my/schema.yaml -o ./path/to/my/schema.d.ts
```

```
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"]["schema"];
type ErrorResponse = paths["/my/endpoint"]["get"]["responses"][500]["content"]["application/json"]["schema"];
```

<https://github.com/drwpow/openapi-typescript/tree/main/packages/openapi-typescript#readme>

openapi-typescript & openapi-fetch

```
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://github.com/drwpow/openapi-typescript/tree/main/packages/openapi-fetch#readme>

orval

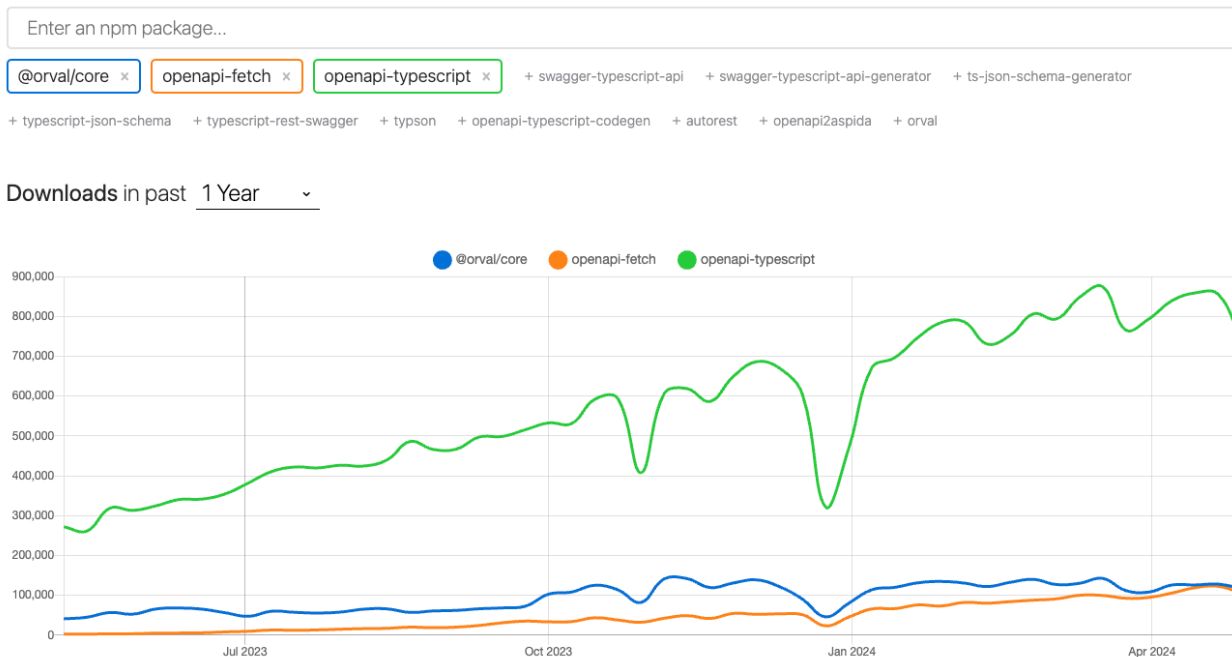
- こちらも OpenAPI Schema から API Client を生成するライブラリ
- Axios を標準として Custom Client を生成できる口がある。TanStack Query や SWR、Zod との連携もある。

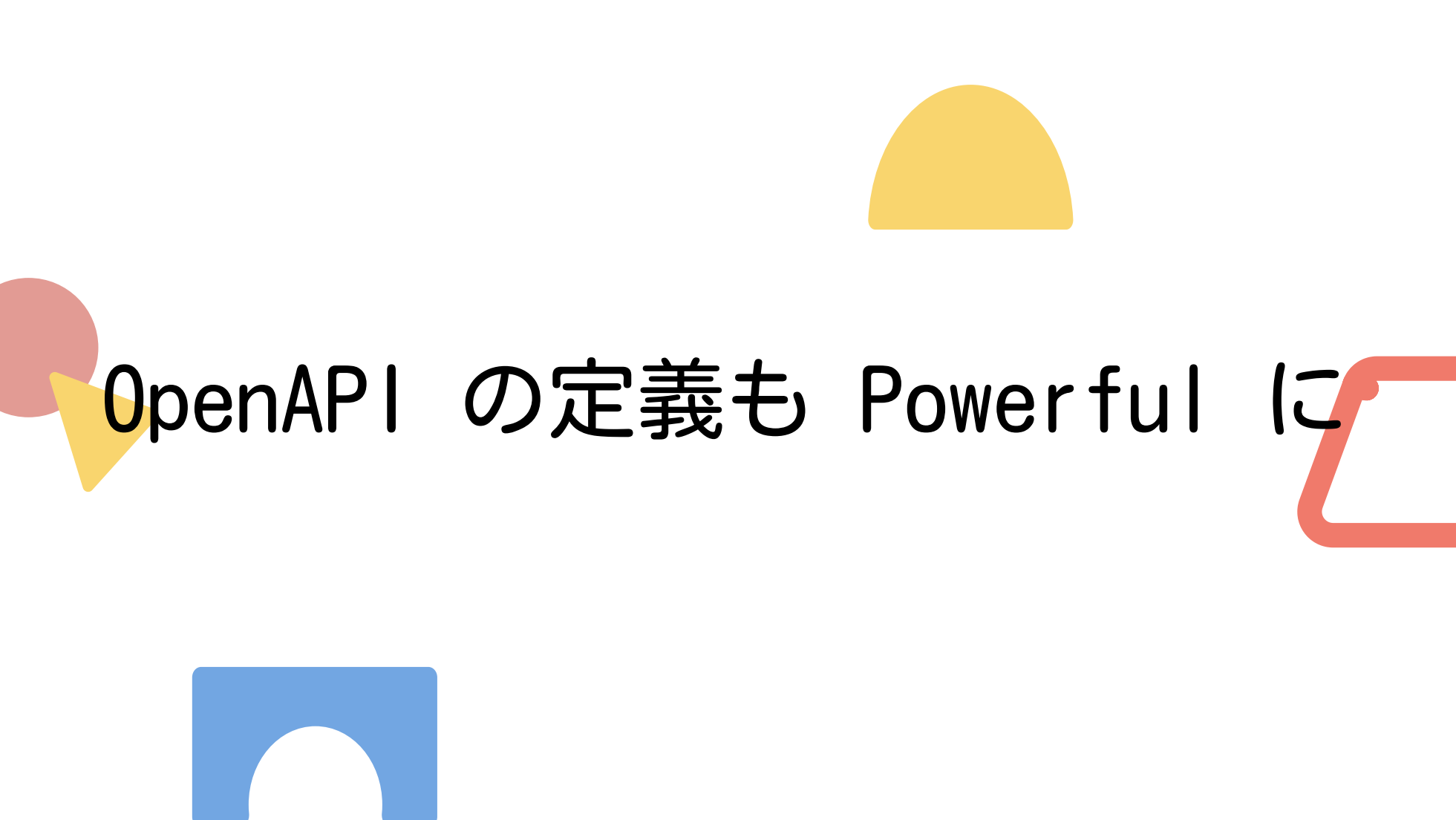
```
import type { CreatePetsBody } from '../model';
import { customInstance } from '../mutator/custom-instance';

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-fetch と orval はおなじくらい



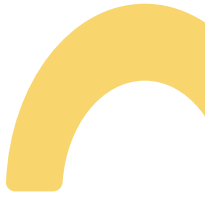




OpenAPI の定義も Powerful に



typespec

- TypeScript / C# 風味の DSL で OpenAPI Schema を書けるライブラリ
 - OpenAPI の yaml を手で書くのは大変だしエディタサポートが貧弱
 - TypeSpec は VSCode 向けの LSP を提供している
 - 複雑でない module システムによりファイルの分割が容易に可能
 - コンパイル時に valid な記法が確認されるので安心
- 
- 
- 



typespec



```
import "@typespec/http";

using TypeSpec.Http;

model Store {
  name: string;
  address: Address;
}

model Address {
  street: string;
  city: string;
}

@route("/stores")
interface Stores {
  list(@query filter: string): Store[];
  read(@path id: Store): Store;
}
```



<https://typespec.io/>



@hono/zod-openapi

- Hono の endpoint と param の定義に zod を使うと OpenAPI も生えてくる

```
import { z } from '@hono/zod-openapi'

const ParamsSchema = z.object({
  id: z
    .string()
    .min(3)
    .openapi({
      param: {
        name: 'id',
        in: 'path',
      },
      example: '1212121',
    }),
})
```

<https://github.com/honojs/middleware/tree/main/packages/zod-openapi>

@hono/zod-openapi

- UserSchema の定義

```
const UserSchema = z
  .object({
    id: z.string().openapi({
      example: '123',
    }),
    name: z.string().openapi({
      example: 'John Doe',
    }),
    age: z.number().openapi({
      example: 42,
    }),
  })
  .openapi('User')
```


@hono/zod-openapi

- route の定義

```
import { createRoute } from '@hono/zod-openapi'

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

@hono/zod-openapi

- app をセットアップすれば完成

```
import { OpenAPIHono } from '@hono/zod-openapi'



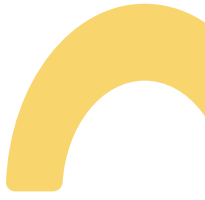
const app = new OpenAPIHono()

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',
  },
})
```



@hono/zod-openapi

- TypeScript の型情報の共有のみに頼り切らず OpenAPI エコシステムを利用
 - Framework Agnostic に成熟している
 - path や param 、 status code など詳細な定義が可能
 - 最悪 Hono をやめても OpenAPI が残る
 - 実装と API 定義が一体化しているため、 OpenAPI Schema が嘘になりにくい
 - 実装から OpenAPI Schema を吐くことの良し悪しはケースバイケースで検討が必要
- 
- 
- 

と書いていたところ、なんと Hono 公式で RPC が出た

```
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
```

<https://hono.dev/guides/rpc#client>

Hono RPC

- client 側のコードは以下

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



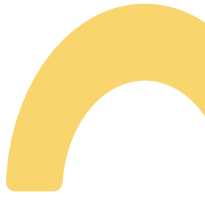
const client = hc<AppType>('http://localhost:8787/')

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)
}
```



Hono RPC

- OpenAPI を介さず Hono のみで完結する仕組み。実体は fetch と REST API である
 - zod 以外でも validator で params の型が定義されていれば OK
 - 前述の @hono/zod-openapi と組み合わせられるので、OpenAPI の出力も可能
 - OpenAPI Client の生成を介さないなので非常にパワフル
 - だが、実は OpenAPI エコシステムを介してもあまり負担ではない
 - プロジェクトが大きくなったときに AppType の推論がどれくらい重くなるかは未知数
 - OpenAPI Schema を介した自動生成では TS 側の負荷をオフロードできるメリットは大きい
- 
- 
- 



まとめ

- ・ 部分的に入れられるライブラリで小さくはじめよう
- ・ TypeScript だけでなく OpenAPI エコシステムも強力
- ・ Hono の OpenAPI や RPC はフル TS で開発する上でパワフル



ありがとうございました

