

安定した基盤システムのための ライブラリ選定

2025/07/15

実践！バックエンドTypeScript～現場から学ぶTSの可能性～

株式会社力ケハシ
岩佐 幸翠 (@kosui_me)



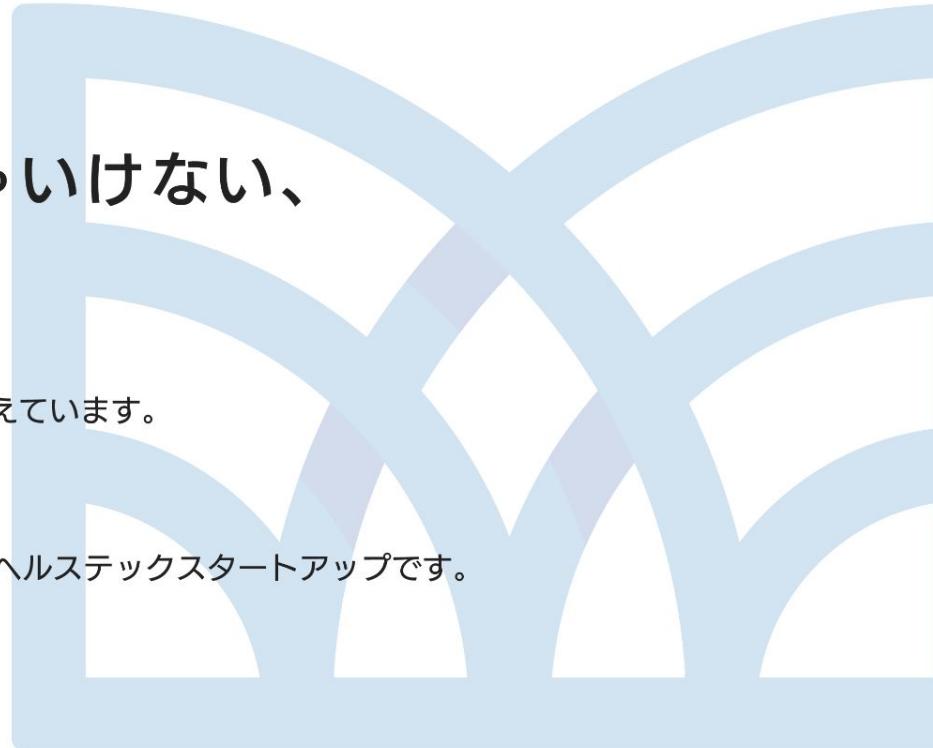
難しい、でも誰かがやらなきゃいけない、 医療という社会課題の解決。

少子高齢化などを背景に、日本の医療制度は危機的な状況を迎えてます。

次の時代を支える、サステイナブルな医療のあり方とは？

カケハシは、その問題と正面から向き合い、

調剤薬局DXを入り口に日本の医療システムの再構築を目指すヘルステックスタートアップです。



#1

医療プロダクトを支える 基盤システム

機能要求

- 認証・認可
複雑な状態遷移を伴う
- ディレクトリサービス
ユーザー・店舗・テナント・グループ・組織階層...
複雑なリレーションシップが絡み合う
- ライセンス
契約とコンプライアンスに直結する

非機能要求

- セキュリティ
- 可用性
- 移植性
- 性能・拡張性

#2

TypeScriptで 基盤システムを構築する

TypeScriptで基盤システムを構築する

基盤システムを構築するなら

静的型付け言語で堅牢かつ効率的な基盤システムを構築したいなら
選択肢は数多くある

- **Java**
パターンマッチ・仮想スレッド・レコードパターン... 今なお進化し続ける言語
- **Go**
ゴルーチンで高パフォーマンスなアプリケーションを容易に構築できる
- **Rust**
メモリ安全で表現力の高い言語

TypeScriptで基盤システムを構築する

なぜTypeScript?

基盤システムでは複雑な機能要件を表現したい

認証・認可、ディレクトリサービス、ライセンスなど

複雑な状態遷移とリレーションシップがコード上で表現されてほしい

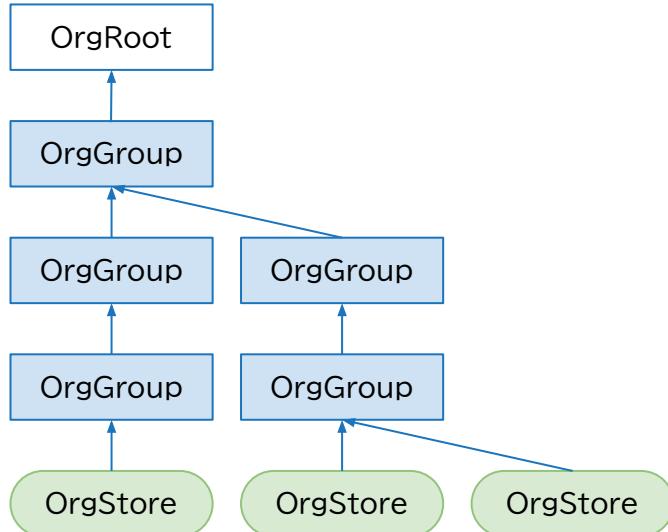
TypeScriptの型の表現力でモデリングする

値やエンティティ、そして状態を型で表現する

コードや仕様の誤りを型検査時に発見できる

TypeScriptで基盤システムを構築する

例) ディレクトリサービス

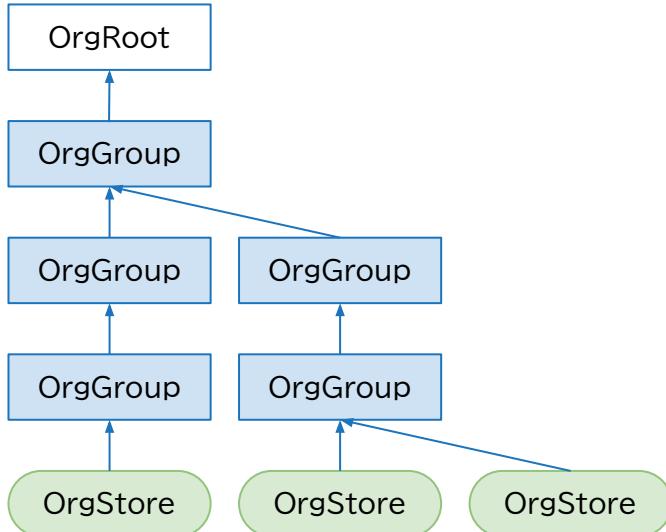


```
1 import { Tagged } from 'type-fest'
2
3 type OrgUnitId = Tagged<string, 'OrgUnitId'>
4
5 type OrgRootId = Tagged<OrgUnitId, 'OrgRootId'>
6 type OrgRoot = Readonly<{
7   kind: 'OrgRoot'
8   orgUnitId: OrgRootId
9 }>
10
11 type OrgGroupId = Tagged<OrgUnitId, 'OrgGroupId'>
12 type OrgGroup = Readonly<{
13   kind: 'OrgGroup'
14   orgUnitId: OrgGroupId
15   parent: OrgRootId | OrgGroupId
16 }>
17
18 type OrgstoreId = Tagged<OrgUnitId, 'OrgstoreId'>
19 type OrgStore = Readonly<{
20   kind: 'OrgStore'
21   orgUnitId: OrgstoreId
22   parent: OrgGroupId
23 }>
```

ルートは親を持たない
グループの親はグループかルート
店舗の親はグループのみ

TypeScriptで基盤システムを構築する

例) ディレクトリサービス



```
1 import { ok, err, Result } from 'neverthrow'
2 type Org = Readonly<{
3   root: OrgRoot
4   units: ReadonlyArray<OrgUnit>
5 }>
6
7 declare const genOrgStoreId: () => OrgStoreId
8 const eq = (x: OrgUnit) => (y: OrgUnit) =>
9   x.kind === y.kind && x.orgUnitId === y.orgUnitId
10
11 type OrgGroupNotFound = Readonly<{ kind: 'OrgGroupNotFound' }>
12 const addStore = (org: Org, parent: OrgGroup): Result<Org, OrgGroupNotFound> => {
13
14   if (!org.units.some(eq)) {
15     return err({ kind: 'OrgGroupNotFound' })
16   }
17
18   const store = {
19     kind: 'OrgStore',
20     parent: parent.orgUnitId,
21     orgUnitId: genOrgStoreId(),
22   } as const
23
24   return ok({
25     root: org.root,
26     units: [...org.units, store]
27   })
28 }
```

シグニチャを見れば発生するエラーがわかる

TypeScriptを採用する場合の注意点

- クラスや例外機構における型推論



過去のJavaScriptコードへ漸近的に型付けできるように

クラスに対する型推論はゆるめに設計されている



try-catch文ではすべての例外がunknown型になるため

ランタイムでの型の検証が必須

- CPU boundな処理が苦手

Node.jsはシングルスレッド



I/O boundな処理（非同期処理）をイベントループで高速に捌ける



CPU boundな処理（同期処理）を掴まされると他イベントの処理が詰まる

TypeScriptで基盤システムを構築する

指針

設計方針

- 型とスキーマを活用し、自己文書化されたコードを目指す
- 移植性を高く保つ
- クラスや例外機構に頼らない

ライブラリ選定方針

- スキーマ駆動である
- 標準に基づいている
- クラスや例外機構への依存度が低い

#3

型とスキーマを活用した 基盤システム

型とスキーマを活用した基盤システム

スキーマ検証ライブラリ

TypeScriptにおけるスキーマ検証ライブラリは
単なる外界とのI/Fの文書化以上の意味を持つ

Standard Schema

Zod、Valibot、Arktype、Effect Schemaなど
多くのスキーマライブラリがサポートする標準インターフェース

型とスキーマを活用した基盤システム

Webフレームワーク

Hono 段階的に採用を進行中

- Web標準に基づいたデザイン
`fetch`、`Requests`、`Responses`、`URL`、`URLSearchParams`など
- Standard Schemaをサポート
リクエストやレスポンスをStandard Schemaで検証できる

NestJS 新規では採用しない予定

Spring BootのようなデコレータベースのDIコンテナを提供する

- 型検査時ではなくランタイム時まで依存性の検証ができない
- 明らかに型と一致しないサービスを注入できてしまう

型とスキーマを活用した基盤システム

データベースとの接続

Kysely 繼続利用中

事前に用意した型定義に従ってクエリ結果に型を付けてくれるクエリビルダー

Drizzle 注視

同じくクエリ結果に型を付けてくれるクエリビルダー

Kyselyよりもリッチなマイグレーション機能を有する

Prisma ORM 弊チームでは撤退済み（2023年頃）

様々なデータベースをサポートするType-safeを謳うORM

発行されるクエリが予想しづらく断念

2024年2月にJOINが扱いやすくなつたらしい

補足 TypeScriptの 注意点を乗り越える

補足: TypeScriptの注意点を乗り越える

CPU boundな処理との向き合い方

コンピューティングリソースを分離する

最もシンプルな解決策

- 他のAWS ECSサービス/Cloud Runへ
- 他のAWS Lambda/Cloud Run functionsへ

Worker threadsを利用する

CPU Boundな処理を他のスレッドへ逃がす方法

適切にWorker threadをハンドリングする能力と気合が必要