Webアプリケーション設計の第一歩は
ディレクトリの整理から

Mar 24, 2023 / Encraft #1 フロントエンド × 設計

@okunokentaro

誰

- 奥野 賢太郎 @okunokentaro
- クレスウェア株式会社
- TypeScript歴10年

設計と私


大型フロントエンド開発における
TypeScriptとDDD
FRONTEND CONFERENCE 2016 / March 5, 2016
chatwork


Angular 中〜大規模アプリ設計
Oct 10, 2017 ng-japan meetup 2017 Autumn
@okunokentaro


大規模開発に打ち勝つための
マルチパラダイム
Jun 16, 2018 / ng-japan 2018
@okunokentaro


大規模は開発環境保守が
大変なので改善する
Oct 6, 2018 / ng-kyoto Angular Meetup #8
@okunokentaro


500日のトライエラーから生まれた
大規模設計ノウハウ
Dec 8, 2018 / Frontend Conference Fukuoka 2018
@okunokentaro


ディレクトリ構成ベストプラクティス
Angularアプリを作り続けてわかったこと
Nov 2, 2019 / FRONTEND CONFERENCE 2019
@okunokentaro

設計とは

- 辞書で設計を引くと？（小学館 精選版 日本国語大辞典）
- せっ-けい【設計】
  - 工事・工作などで、工費・材料・敷地・形式などの計画を立て、図面その他によって具体的に示すこと。みつもり。もくろみ。
  - （比喩的に）人生・生活などの計画を立てること。
  - 明治20年代後半からdesignの訳語として登場、中国洋学書の影響

設計とは

- 辞書で設計を引くと？（小学館 精選版 日本国語大辞典）
- せっ-けい【設計】
  - 工事・工作などで、工費・材料・敷地・形式などの計画を立て、図面その他によって具体的に示すこと。みつもり。もくろみ。
  - （比喩的に）人生・生活などの計画を立てること。
  - 明治20年代後半からdesignの訳語として登場、中国洋学書の影響

建造物を連想する人が多い

- 「アプリケーション設計」は比喩的な用法
- ビルやマンションなどの建造物設計と同じではない
- 建造物設計のつもりでアプリケーション設計に臨んだり
  他者に説明したりするのは適切とは言い難い

建造物とアプリケーションの相違点

- 建造物設計とアプリケーション設計を混同してはいけない点
  - 設計寿命
    - 建造物は数十年
      （とくにWeb）アプリケーションで、数十年を想定することは事実上不可能
  - 建築・開発のプロセス
    - 建築は完全なウォーターフォール、開発は昨今はアジャイルが一般的
  - 竣工後・リリース後の扱い
    - ビルメンテナンスの文脈は別文脈、アプリケーションは「機能追加」や「負荷対策」の概念がある

アプリケーション「維持」である

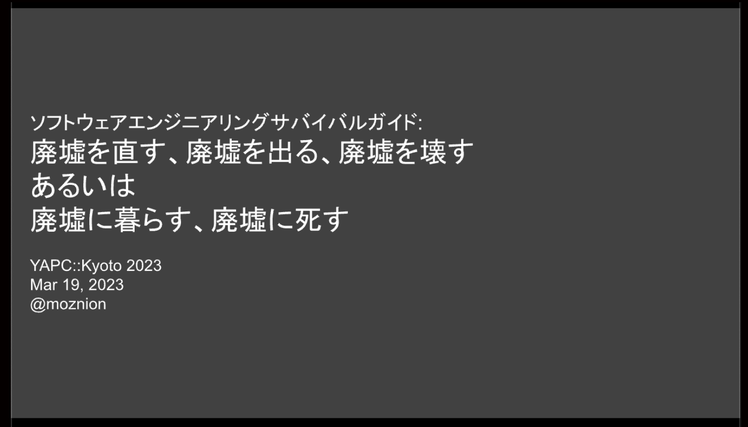
- 我々がアプリケーション設計と述べる文脈の大半は
  実はアプリケーション「維持」を目的としていることが多い
- 「0→1」より「1→10、10→100」が機会として圧倒的に多い
  - 1→Nの段階において「アプリケーション設計」とは
    「このアプリケーションを今後どうしていきたいか」を考えること
- データベース・スキーマなどの検討と、アプリケーションコードの設計は別
  「データベース設計」については今回論じていない

何を維持すべきか

- 理解のしやすさ

理解しやすくあれ

- 機能追加しやすい、修正しやすい、説明しやすい、チームに加入しやすい
  - すべて「理解しやすい」ことが前提で成り立つ
- 理解しがたい状態を継続させるな
  - 廃墟にするな



https://docs.google.com/presentation/d/1hDY2pb-nYVSLr0HrtQ4EVyrDU4QGgwp4-VRG-Rf26DA

なんたらアーキテクチャにこだわるな

- 「なんたらアーキテクチャ」の採用と厳守にこだわる人をしばしばみかける
- すべて暗記しなくていい
  - クリーン・アーキテクチャ
  - ヘキサゴナル・アーキテクチャ
  - オニオン・アーキテクチャ
  - レイヤード・アーキテクチャ
- 歴史的に「どういうことが求められて生まれたのか」の背景を学ぶ
  - 何をすべきなのか考える

整理され理解しやすければよい

- アーキテクチャの大半は「依存管理」と「理解」を助けるための整理のルール
- 整理されている状態
  → 理解しやすさが維持されている状態
  → アプリケーション設計に成功している状態
- 「アプリケーション設計に失敗してしまった」とよく聞く
  - リアーキテクチャどうこうを考えるより、まず整理の再開を
  - 整理して、俯瞰して理解してから「次に改善すべきこと」が浮かび上がってくる

整ったディレクトリ構造を維持する

理解しやすいディレクトリ構造

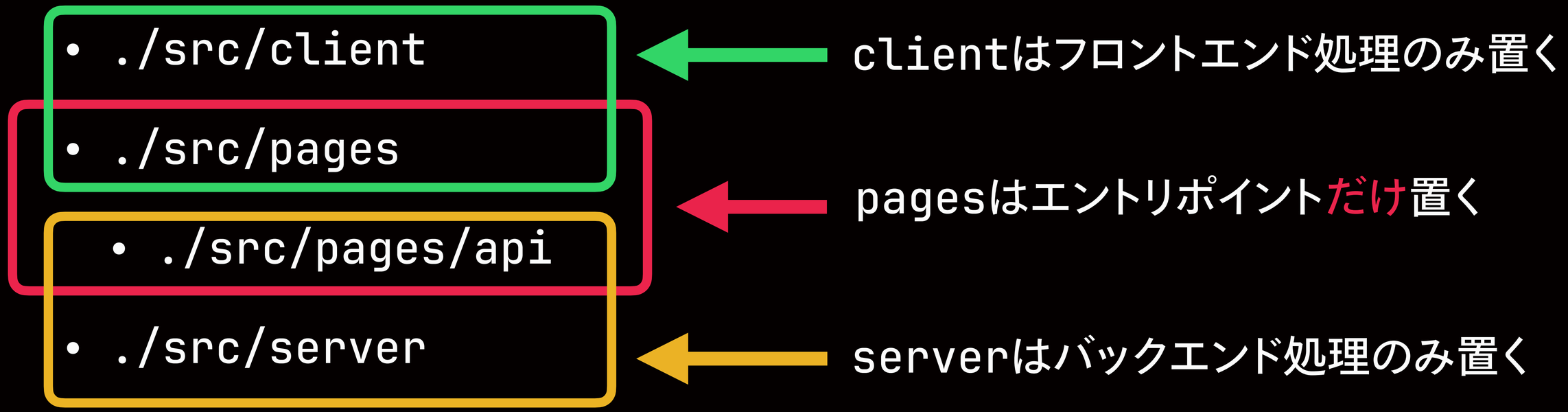
- ディレクトリ構造とアーキテクチャは密接
  - ディレクトリ構造が整っているとその前提で「新機能の設計」や「修正方針の決定」が行いやすい
  - 「アプリケーションの現在」を表す地図となる
- なんたらアーキテクチャの完全模倣を目指すよりもまずディレクトリを整理する

Next.js案件での例

- ./src/client
- ./src/pages
  - ./src/pages/api
- ./src/server

※ ./はpackage.json置き場

/pagesになんでも置かない



コンテキストごとのディレクトリ

- ./src/client
  - ./src/client/user-page
  - ./src/client/todo-page
- ./src/server
  - ./src/server/handlers/users/get
  - ./src/server/handlers/todos/get
  - ./src/server/handlers/todos/post

boundaryを設ける

- ./src/client
  - ./src/client/boundary
  - ./src/client/user-page
  - ./src/client/todo-page
- ./src/server
  - ./src/server/boundary
  - ./src/server/handlers/users/get
  - ./src/server/handlers/todos/get
  - ./src/server/handlers/todos/post

clientから外部環境への依存
例えば、Firebase, Stripe, Sentry

serverから外部環境への依存
例えば、社内のバックエンドの外部システム
Auth0, Redis, CMS, メール送信サービス

共通化したい欲求をどう扱うか

- ./src/client
  - ./src/client/boundary
  - ./src/client/user-page
  - ./src/client/shared
- ./src/models
- ./src/server
  - ./src/server/boundary
  - ./src/server/handlers/users/get
  - ./src/server/shared

- sharedとmodelsという概念を設ける
  - models
    - あらゆる外部ライブラリ（React, Apollo 等）に一切依存してはならない
    - npmインストールなくとも解釈可能な純粋なTypeScriptコードのみここへ
  - shared
    - 外部ライブラリに少しでも依存するコードはここへ
      - ReactのHookやコンポーネントの共通化
      - Apolloのエラーハンドリング層の共通化

clientとserverをまたぐ定義はどこへ?

- ./src/client
  - ./src/client/boundary
  - ./src/client/user-page
  - ./src/client/shared
- ./src/models
- ./src/schema
  - ./src/schema/handlers/users-get
- ./src/server
  - ./src/server/boundary
  - ./src/server/handlers/users/get
  - ./src/server/shared

- modelsとは別にschemaという概念を設ける
  - ajv, json-schema-to-tsへの依存を許可する
  - JSONスキーマ定義とTypeScript型定義は全部ここへ
- /handlers/users-get/request-body-schema.ts
- /handlers/users-get/request-body.ts
- /handlers/users-get/response-body-schema.ts
- /handlers/users-get/response-body.ts
- /handlers/users-get/make-path.ts

扱いに困るファイルは?

- ./src/client
- ./src/env
- ./src/models
- ./src/pages
- ./src/schema
- ./src/server
- ./src/utils

- envとutilsという概念を設ける
  - env
    - あらゆる環境変数はprocess.env.SOMETHINGを
      あちこちに書くより、定数としてまとめておいたほうが
      管理・検索しやすい
    - すべてstringのままにするより
      数値であれば事前にparseInt()しておくなど
    - boundaryやmodelsにバラけさせると、逆に把握が困難
  - utils
    - 自社ドメインに影響しない処理
      例えば、「オブジェクトのDeep Equal」だったり
      Array<T>からTを求める型だったり
    - ここに入れることはかなり慎重になるべき
    - 一番「なんでも箱」になる

分類ルールを守る

守れないルールならいらない

- 「どのディレクトリには何を入れてください」という口頭の説明だけでは
  いずれ守れない人がでてくる
  - 悪意に限らず、新人への周知漏れ、うっかり忘れがある
  - コードレビュー時にレビュアーも見逃すかもしれない
    （GitHubはディレクトリパスやファイル名自体のレビューがやりにくい）
- eslintでしっかり守れるものにする
  - 人にも機械にも理解しやすいルールを維持する

import/no-restricted-paths

- `eslint-plugin-import`に手頃なルールがある

  `https://github.com/import-js/eslint-plugin-import/blob/main/docs/rules/no-restricted-paths.md`

- このルールを採用すると「任意のディレクトリから別のディレクトリへの依存を禁止する」ことができる
  - サブディレクトリ同士の禁止も指定できる
  - 例えば`handlers/users/get`内の処理が`handlers/users/post`に依存してはならないというルールも書ける

維持のための一工夫

- import/no-restricted-pathsの欠点は禁止リストの管理であること
  - 禁止リストの管理だと、サブディレクトリの増減に応じてメンテが必要
  - 許可リストの管理にすれば、原則は禁止で、追加許可したいときだけメンテナンスすればよい
- 許可リストを定数として定義し、任意のスクリプトを実行することで
  .eslintrc.jsを自動生成するように仕組みを整えた
  - 許可リストをもとに、反転させ全禁止リストを自動生成
  - npm iの際に常に実行するように手配する

1ディレクトリを1パッケージと捉える

- もうひとつeslintルールを導入する
  - eslint-plugin-import-access
    https://zenn.dev/uhyo/articles/eslint-plugin-import-access
- @packageディレクティブを付与することで、
  他のディレクトリから参照した際にLintエラーにできる
- 「exportしたくないから1ファイルに大量に書く」という必要がなくなる

ルールを守るとどうなるか

- 依存方向や影響範囲を理解しやすい
  - `schema`ディレクトリに依存している`client`と`server`配下から影響範囲特定
    - どのReact Hookがこのエンドポイントを呼んでる? このエンドポイントはどのエラーを返しうる?
  - バックエンド処理のなかで認証系を担当しているのはどこか?
    - `server/boundary/auth`内のファイルか、そこに依存しているファイルを検索すれば機械的に洗い出せる
  - 取り違えの防止
    - `server/handlers/users/get`で`server/handlers/todos/get`の処理に依存していたら取り違え
- ファイル先頭のimport文と、exportされる変数名・関数名を眺めるだけで
  「このファイルは何がしたいか」が一目瞭然となる

コードレビュアーに優しい

- 変更ファイル一覧が要約を満たすため、レビュアーはまず変更されたファイル一覧を眺めると理解しやすい
- `client/**, client/shared, models`の変更
  - 「画面側に新たな概念を表示させるため、新しいコンポーネント実装をしたんだな」
- `server/handlers/**, server/shared, server/boundary`の変更
  - 「認可系の修正のために、バックエンドを横断的に変更したんだな」
- `client/**, client/shared, models, server/**, server/shared`の変更
  - 「範囲が広い! クラス名や汎用関数名の改名か? あるいは何か大掛かりなことをやりすぎていないか?」

テストに優しい

- ディレクトリごとに「どういうテストを書けばよいか」が求まりやすい
  - コンポーネントの見た目をテストしたいのか
  - 複数の外部サービスをつないでいるときに、エラーハンドリングをモックを使った結合テストとして検証したいのか
  - 実際に外部サービスに対してリクエストを投げてEnd to Endなテストを実施したいのか
- どのディレクトリ内にあるファイルをモックにすればよいかが求まりやすい
  - `boundary`への依存をモックする、`env`への依存をモックする、など

Conclusion

- アプリケーションは理解しやすい状態を維持すべきである
- ディレクトリ構成が整理されていると、
  機能の多さや複雑さが視覚的に理解しやすい
- 依存方向にLintによる束縛を加えると、過度な密結合を防ぎやすい
  - テストの作成やリファクタリングの実施が容易
  - 結果的に「なんたらアーキテクチャ」の要点を満たすことができる
- まずはディレクトリを整理しよう