

爆速でプロダクトをリリースしようと思ったらマイクロフロントエンドを選んでいた

自己紹介

- 株式会社カケハシ
- ソフトウェアエンジニア
- Nokogiri (@nkgrnkgr)

今日話すこと

- 生成AIの価値を早く届けるために、アーキテクチャ的な正しさよりもスピードと現実解を優先した。
- 既存のAngular製基幹システムにReact製の新機能を組み込み、チームとシステムの境界を保ちながら連携する構成とした。
- 本発表では、その背景、実装上のポイント、実装時の課題、得られた知見を共有する。

プロダクトの歩みと現在地

創業からPMFまで

- カケハシは医療系SaaSを展開し、主に薬局向けの業務システムを提供。
- 創業時のプロダクトが **Musubi**。
- 薬剤師は患者とのやりとりを **薬歴** として記録（医師のカルテに相当）。
- 手書きの **薬歴** をシステム化し、Musubi はPMFを達成。

市場の状況

- Musubi は今では市場の **20%** の薬局で利用されている。
- ただしリリースしてから、後発の競合のプロダクトがリリースされており、以前ほど市場での優位性を失っている。
- 生成AI関連機能では、他社に後れを取っている状態。
- カケハシとしてはユーザーにとって価値のある生成AIプロダクトを市場に投入して優位性を取り戻したい。

開発の制約

- Musubi はリリースして既に10年近く経っており、プロダクトとして安定はしているが、技術的負債が貯まっていたり、素早く機能開発ができる状態ではなくなっている。
- 特にリリースサイクルは月に1回で、薬局の基幹システムでもあるため障害発生時の影響も大きい。
- ユーザー操作の慣れもあるため、UIを大きく変えることも簡単ではない。
- PharmacyAI(新しい生成AIプロダクト) は実験的な機能であり、素早くユーザーに提供してフィードバックサイクルを高速に回したい。
- MusubiはAngular製のプロダクトで、PharmacyAIチームのメンバーはReactでの開発に慣れている。

リリースサイクルの速度差が最大の課題。

- 素早く価値を提供し高速にイテレーションを回したいPharmacyAI
- ゆっくりでも確実にリリースをしていきたいMusubi

このため、両者が独立して開発・リリースできる体制・仕組みを採用。

マイクロフロントエンドという技術・組織的戦略

アプリケーションのフロントエンドを、独立して開発、デプロイ、テスト、保守できる小さな部分に分割するアーキテクチャ

戦略

- **体制:** Musubiの開発チームとは別で、PharmacyAIの開発チームを作る。
- **デリバリー:** 機能開発のリリースもMusubiとは別で行い、PharmacyAI単体でリリース可能にする。
- **技術:** AngularではなくReactに長けたメンバーがReactで開発する。
- **UI:** Musubiの上に配置する独立したUIにすることで、Musubi側の変更の影響を受けづらく、またPharmacyAI側もMusubiに影響を与えずらい見た目にする。

コンウェイの法則に帰結

システムを設計する組織は、その組織のコミュニケーション構造をそのまま反映した設計を生み出す

想定される疑問

- Musubi内にAngularで生成AI機能を実装する選択肢は？
 - 選択肢はあった。ただし、実験的な生成AIをリリースサイクルの長いMusubi内に入れると価値検証が難しくなる。
 - 独立して作るなら、Angularにこだわる必要はないと判断。
- 別プロダクトとして独立提供する選択肢は？
 - 日常的にMusubiを使うユーザーにシームレスに届けたかった。

マイクロフロントエンドの具体的な技術

AngularとReactアプリを共存させるアーキテクチャ

- vite でビルドした Reactアプリの JS と CSS を事前にホスティング
- Angular のアプリにあらかじめ `<div id="react-component" />` を用意し、JS、CSS ファイルをロードしてレンダリング

iframe も選択肢にあったか？

- ユーザーから見てMusubiとPharmacyAIは一つのアプリ
- 認証機能を別で作る必要はなく、むしろMusubi側に任せたい
- ロードされるJSファイルにバージョン管理も不要だった

独立したUI

- Musubiの画面の上に配置する独立したチャットのような見た目
- Musubi本体からUI的にも独立しているため、お互いに影響を受けづらい

CustomEvent を利用したアプリケーション間通信

具体的な実装

```
// 送信側
const event = new CustomEvent("contextChanged", {
  detail: {
    payload: {
      pharmacyId: "pharmacy-123",
      patientId: "patient-456",
      idToken: "token-789"
    }
  }
});
window.dispatchEvent(event);

// 受信側
window.addEventListener("contextChanged", (event) => {
  const { pharmacyId, patientId, idToken } = event.detail.payload;
  // ここで受け取ったデータをReact側の状態管理などに反映する
  console.log("受信:", pharmacyId, patientId, idToken);
  // 例: setContext({ pharmacyId, patientId, idToken });
});
```

相互に通信する方法

...TDB

コンテキストの同期（ページ遷移の追従）

...TDB

認証トークンの有効期限切れチェック、通信時の再認証など

...TDB

実際にマイクロフロントエンドを採用して出た課題

このアーキテクチャを採用したことで出てきた課題や、泥臭い対応などを紹介

開発・デバッグ環境

- CustomEvent のやり取りのデバッグが難しく、イベントの流れを可視化するために専用のChrome拡張機能を作成
- Musubiのローカル開発環境を準備しなくても開発ができるように読み込むURLを差し替える専用のChrome拡張機能を作成した。

実装の泥臭い対応

- PharmacyAIが生成した薬歴をMusubi側に自動挿入する際、Musubi側の画面遷移に追いつけず、**3回までリトライする**という泥臭い実装になった。
- コミュニケーションコストを減らすために、PharmacyAIチームが自らMusubi側のコードを修正し、PRを投げた経験を共有する。

UIの複雑性

- MusubiのグローバルなCSSがPharmacyAIに影響を与えたり、`z-index` の管理が複雑になったりした。

まとめ

- **この選択の成果:**

- 多くの苦労はあったものの、この選択によって**開発から4か月でリリース**できた。
- 継続してユーザーからのフィードバックを受けてイテレーションしている。(毎週リリース)
- 異なる技術スタックを持つチームが独立して開発を継続できる体制が整った。

- **結論:**

- ビジネスとして達成したいゴールがあるから、そこから逆算してアーキテクチャを選ぶのは大事
- フロントエンドエンジニアは、ビジネスの要求に応えるために、このような選択肢もあるということを理解してほしい。