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

- アーキテクチャの正しさよりもユーザーへの価値提供スピードとフィードバックループを早く回せることを優先した。
- 既存のAngular製のシステムに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。
- 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Musubiに搭載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に影響を与えづらい見た目にする。

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

設計編：マイクロフロントエンドを支える技術

Angular×Reactの共存 / Musubi非依存UI / アプリ間通信

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

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

アプリケーション間通信

`CustomEvent` を利用した具体的な実装

```
// 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;
  console.log("受信:", pharmacyId, patientId, idToken);
});
```

実践編：導入して見えた課題と工夫

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

CustomEventを「受領通知」で双方向化する

CustomEventは基本「投げっぱなし」。しかし送信側が「受信側が受け付けたか」を知りたい場面がある。例えばリトライするとか

- 送信: **送信側**が一意なid付きでCustomEventを発火する。
- 受信/処理: **受信側**がそのイベントを受け取り、転記を実行して結果を作る。
- 応答: **受信側**同じidを含む応答イベント（受領通知）をmusubi_ack_${id}で返す。
- 待機/取得: **送信側**はそのackを待ち受け、結果（OK/NGとメッセージ）を受け取ってUIやログに反映する（未応答はタイムアウト）。
- ポイント: 相関IDでリクエストと応答を対応づけ、投げっぱなしのCustomEventを「受領通知」で双方向化している。

送信側：実装例

```
const EVENT = "pharmacy_ai_counseling_record_posting";
const ACK = (id: string) => `musubi_ack_${id}`;

const emitPosting = (payload: any, timeout = 3000) =>
  new Promise<any>((resolve, reject) => {
    const id = crypto.randomUUID();
    const onAck = (e: Event) => resolve((e as CustomEvent).detail.payload);
    const timer = setTimeout(() => {
      window.removeEventListener(ACK(id), onAck);
      reject(new Error("timeout"));
    }, timeout);

    window.addEventListener(ACK(id), (e) => {
      clearTimeout(timer);
      onAck(e);
    }, { once: true });

    window.dispatchEvent(new CustomEvent(EVENT, { detail: { id, payload } }));
  });

const result = await emitPosting({ prescriptionId: "1", soaps: [] });
console.log(result); // => { status: "OK", message: "薬歴の転記が完了しました" }
```

受信側：実装例

```
const EVENT = "pharmacy_ai_counseling_record_posting";
window.addEventListener(EVENT, (e) => {
  const { id, payload } = (e as CustomEvent).detail;
  const result = { status: "OK", message: "薬歴の転記が完了しました" };
  window.dispatchEvent(new CustomEvent(`musubi_ack_${id}`, { detail: { payload: result } }));
});
```

ページ遷移を相互追従

Musubi 側からコンテキスト（患者情報）の変更のCustomEventが送信される。逆にPharmacyAI側でページ遷移を要求したい場合もある。

MusubiからPharmacyAIへ

- Musubiの画面遷移や患者切替の通知を受け取り、PharmacyAI側も状態・画面を合わせる。
- ただし「すでに同じ患者なら何もしない」ことで、PharmacyAI発の遷移を上書きしない。

PharmacyAIからMusubiへ

- PharmacyAIが遷移したいときは「遷移リクエスト」を送る。同時にPharmacyAI側も先に自分の画面を目的地へ移動しておく。

実装サンプル

Musubi→PharmacyAI（通知を受けて追従。自分が先に同じ患者へ遷移していたら無視）

```
window.addEventListener("musubi_context_changed", (e) => {
  const { patientId } = (e as CustomEvent<{ patientId: string | null }>).detail;
  if (getCurrentPatientId() === patientId) return; // 自分発の遷移と競合しないように無視
  navigateTo(patientId ? `/patients/${patientId}` : "/");
});
```

PharmacyAI→Musubi（通知からの遷移。先に自分も遷移しておき、後続のMusubi通知は一致で無視される）

```
const navigateAndRequest = (patientId: string) => {
  navigateTo(`/patients/${patientId}`); // 先に自分を目的地へ
  window.dispatchEvent(
    new CustomEvent("pharmacy_ai_navigation_request", { detail: { patientId } })
  );
}
```

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
```

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。
  - 継続してユーザーに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。