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

- 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。
- 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。
- 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。
- ただしリリース後、後発の競合プロダクトが登場し、以前ほどの優位性はない。
- 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、プロダクトとしては安定しているが、技術的負債が蓄積しており素早い機能開発が難しくなっている。
- 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。
- ユーザーの操作慣れもあるため、UIを大きく変えることは簡単ではない。
- 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。

	PharmacyAI	Musubi
技術スタック	React	Angular
リリースサイクル	毎週リリース・高速イテレーション	月1回リリース・安定重視
開発チーム	新規チーム（生成AIに特化）	既存チーム（基幹システム担当）
UI/UX	実験的・新しいUIも導入しやすい	既存ユーザーの慣れを重視
機能の性質	生成AIなど新規・実験的な機能	基幹業務・安定運用が求められる機能

両者が独立して開発・リリースできる体制を採用。

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に、PharmacyAIの開発チームを作る
- **デリバリー**: 機能リリースもMusubiとは切り離し、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とし、相互に影響を受け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` を利用した疎結合な通信

```
// 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

型定義で通信仕様を固める

- どのCustomEventをどのキー名とペイロードで扱うか、チーム間で合意する必要がある
- TypeScriptの `Mapped Type` を使って型安全にCustomEventを管理する

```ts
export const CustomEventName = {
  MusubiContextChanged: "musubi_context_changed",
} as const;

export type MusubiContextChangedPayload = {
  patientId: string | null;
}

export type CustomEventMap = {
  [CustomEventName.MusubiContextChanged]: MusubiContextChangedPayload;
};
```

```ts
export const sendCustomEvent = <K extends keyof CustomEventMap>(
  name: K,
  payload: CustomEventMap[K],
) => {
  const id = crypto.randomUUID();
  window.dispatchEvent(new CustomEvent(name, { detail: { id, kind: name, payload } }));
  return id;
};
```

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（例えばリトライしたいとき）。

- **送信側**: 一意なid付きでCustomEventを発火する
- **受信側**: イベントを受け取り、処理結果を作る
- **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
- **送信側**: 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（未応答はタイムアウト）

ポイント: 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

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

const emitPosting = (payload: PostingPayload, timeout = 3000) =>
  new Promise((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;
  // payload に応じた処理

  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 } }));
});
```

ページ遷移の相互追従

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}`); // 事前に自分が画面遷移
  window.dispatchEvent(
    new CustomEvent("pharmacy_ai_navigation_request", { detail: { patientId } })
  );
}
```

CustomEventのデバッグ

CustomEventのやり取りは、Chromeのネットワークタブのようにどのようなリクエスト/レスポンスが発生したかを確認しづらい。
CustomEventの流れを可視化する**Chrome拡張機能**を作成して、イベントの流れを見える化した。

ローカル開発環境①：Musubiスタブ画面

- Musubiのスタブ画面を準備し、PharmacyAIの `JS`、`CSS` ファイルを読み込む
- スタブ画面からCustomEventを発火してPharmacyAIの動作確認をする

本物のMusubiの画面ではないためある程度開発はできるが、厳密な動作確認のためには一度デプロイし、Musubiと結合して動作する環境で確認する必要があった。

ローカル開発環境②：Musubiが読み込むJS/CSSをChrome拡張で差し替える

- `chrome.declarativeNetRequest` というAPIを使い、Musubiが読み込むJS/CSSファイルの `URL` をランタイムで動的に差し替える
- ローカルで起動したURLに差し替えることで、Musubiはdev環境、PharmacyAIはローカル環境のファイルという動かし方が可能になった

その他の泥臭い工夫

- 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**最大3回までリトライ**する。二つのアプリケーションの状態の完全な同期は難しく、一部でネットワーク通信のような振る舞いが必要
- コミュニケーションコストを減らすため、**PharmacyAIチームが自らMusubi側のコードを修正し、PRを投げる**
- MusubiのグローバルCSSがPharmacyAIに影響を与えることがあり、`z-index` も考慮して実装する必要があった

さらに詳しく知りたい方へ

CustomEventやローカル開発環境の詳細は、以下のブログにも詳しく記載しています。併せてご確認ください。

- CustomEventの設計・型・テスト: 型とテストで守るカスタムイベント通信 - 実プロダクトでの実装事例
- 開発環境（Chrome拡張でのJS/CSS差し替え ほか）: 生成AIで内部ツール開発のジレンマを解決する

まとめ

- **この選択の成果**:
  - 多くの苦労はあったものの、**開発から4か月でリリース**できた
  - ユーザーからのフィードバックを継続的に受け、**毎週リリースでイテレーション**している
  - 異なる技術スタックのチームが、独立して開発を継続できる体制が整った
- **結論**:
  - まずビジネスとして達成したいゴールを定め、そこから逆算してアーキテクチャを選ぶことが重要
  - 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ことを知っておいてほしい

宣伝

株式会社カケハシではエンジニアを募集しています！