3月中旬までに決めること

* バックエンドアーキテクチャ
* バックエンドフレームワーク

3月末までに決めること

* インフラ

**いづメモ：バックエンドAPI方式のADR３回目なので、忘れつつあるかもしれないのでここで改めて目的を再確認（なぜここ）**

* いろんな状況（顧客価値、納期、チーム、自分たちの意欲、今の技術力などなど）をかいまみて、顧客と自分たちにおいて今取りうるベストな選択を考える
* いきなり一極集中せずに、一度選択の幅をひろげてから考える。
* 気をつけること：チャレンジ要素を増やしすぎると自分たちが苦しくなるかもなので、引き算も大事

3/13 細かい要素技術を仮決めする

* 有料でないと使えないものは事前に決めておかないと、許可取得に時間がかかる
  + 予算取り
  + セキュリティ確認
  + などに時間がかかる

* 項目
  + CI/CD
    - GitHub Actions(Enterpriseで契約している)
    - [料金プラン](https://docs.github.com/ja/enterprise-cloud@latest/billing/managing-billing-for-your-products/managing-billing-for-github-actions/about-billing-for-github-actions#included-storage-and-minutes)
  + dbt
    - [dbtとは](https://zenn.dev/dbt_tokyo/books/537de43829f3a0/viewer/what_dbt)
    - **有料ライセンス必要**([プラン](https://www.getdbt.com/pricing))
    - 市東さん（&はまさきさん）が1番詳しいと思うので、相談する
    - 開発者が1人なら無料でいけそうだが…（1人はよくなそう）
  + テスト計画
    - AIを使って加速させたい領域
  + モックをどうするか
    - モック・スタブ
    - SCOPEやTimetrackerの動きを再現する箱が必要
  + IaC
    - terraform ([open tofu](https://opentofu.org/))またはCDK
    - hashicorpがIBMに買収されたので、terraformが将来的に有償化するリスクあり
    - CDKはtypescriptでかける
    - nimmtチームはterraformが多い
    - 言語自体はあまり大きな壁にならない
      * 生成AIでパッと意図通りの構成で出力できる
  + ネットワーク
    - DNネットワークに外部（AWS）からアクセスするためにリバプロが必要（**ALADIN有料が必要**）
    - 米岡さんに要件を決めてもらうのがよさそう
  + 認証
    - 本番環境ではMicrosoft EntraIDを使用することが決まっているが、開発環境ではMicrosoft EntraIDに依存しないで開発できるようにしたい
    - フロントとバックで、何を使うのか、調査が必要
  + 開発環境
    - フロントエンドとバックエンドともにコンテナ化したいが、DNさんの社内からだとproxyを通さないと使えないなど環境依存となり、問題が発生した場合に手間取る可能性あり  
      →　基本、リモートワークとすれば問題は回避可能

[**3/13 サーバーorコンテナ or サーバーレスどうするかのADR**](https://docs.google.com/document/d/1E-k2HyC3dGtilCwvUZ5IG5YLqOqRRSwT6TFtE23YqEQ/edit?tab=t.0#heading=h.686vsegl1e7h)

**ステータス: Status**

* accepted 受理

**状況 : Situation**

* 言語やフレームワークは大まかに決まった状態
* 決まったこと
  + レンダリング方式はCSR = SPA
  + フロントエンドreact router v7(SPAモード)
  + バックエンドのAPI方式はWebAPI
  + DBはPostgreSQL
  + DWH(Data Ware House)はSnowflake
  + バックエンドフレームワークはNestJS(TypeScript)
  + 本番環境の認証はMicrosoft EntraID
* 決まっていないこと
  + BIはPowerBI

**背景 : Context(なぜ今この意思決定をするのか)**

* こういう構成でいこうと思うんですがどうでしょう？くらいの土台を固める
* CI/CD（ライセンスまわりも含めて、早く決めたい）
* IaC(何使う？)
* 開発環境（ライセンスまわりも含めて、早く決めたい）
  + cursor?
  + bolt.new?
  + snowflakeの開発環境 (DBT)
* 認証
* ネットワーク
* サーバーorコンテナ or サーバーレス （決めた）
* AWSアーキテクチャ、

**決定のための材料 : Material for decision**

* フロント
  + S3 + CloudFrontとか
* バックエンド
  + アプリケーション
  + バックグラウンド処理(SCOPE連携とか)
    - 蔵内さんの今までの話だとAWS Batchをイメージしていそうだった
* [ChatGPT Deep Research](https://chatgpt.com/share/67ca79be-e118-8009-8496-baef0312b7ea)

バックエンドアプリケーションをサーバー/コンテナ/サーバレスにするか

| 分類 | コンテナ ++++ | サーバレス |
| --- | --- | --- |
| 選択肢 |  |  |
| 概要 |  |  |
| 顧客への推しポイント |  |  |
| メリット | ・開発時の環境構築が容易  ・FargateとNestJSのMicroservicesの相性がよい  ・運用を考慮した時に自由度が高い  ・メンバーが変わっても引き継ぎしやすい  ・fargateは情報が多くつぶしがきく  ・コンテナをフル活用して開発環境と本番環境を合わせると認知負荷が下げられる | ・安い  ・インフラの管理が不要 |
| デメリット |  | ・開発環境でのテストなどがコンテナと比較すると大変  ・運用の負担が大きくなる可能性があるので、あとから運用周りを充実させる必要があるサーバレスの場合そこがやりにくい ・NestJSと相性悪い。lambdaに載せるのが大変。Beenstackは知らない  ・コールドスタート |
| 学習コスト | 中 | 少 |
| 設計コスト |  | 高い。聞いているイメージだとマイクロサービスっぽい感じなので |
| 運用コスト | 中程度 | 低い |
| 具体的な実現手段 | fargate,ECS | Lambda, AppRunner |
| 懸念 | デンソーさんメンバーが不慣れだけど、CLが経験あるのでなんとかなりそう |  |
| 経験者 | CLメンバー |  |

**表明**

* 烈さん
  + コンテナ(Fargate)
    - LambdaにNestJSはなしかな
    - Fargateは事例が多い
* 岡﨑
  + コンテナ
    - MicroServicesとの噛み合わせが良さそう
    - 将来的な開発メンバーの入れ替わりにも対応しやすそう
* 佐藤
  + コンテナ
    - 開発環境と本番環境の差分を少なくしたい（認知負荷が少ない）
* 市東
  + コンテナ
    - 開発の主体はCLなので、DNとしての経験の少なさは気にしなくて良いか

**結論 : Decision**

コンテナ

**結果 : Consequences**

[**3/11 バックエンド技術選定（フレームワーク）のADR**](https://docs.google.com/document/d/1E-k2HyC3dGtilCwvUZ5IG5YLqOqRRSwT6TFtE23YqEQ/edit?tab=t.0#heading=h.79wxsek6mwch)

**ステータス: Status**

* accepted 受理

**状況 : Situation**

* 言語とフレームワークと、API方式、サーバレスにするかどうかなどなどについて広く決まっていない状態
* 決まったこと
  + レンダリング方式はCSR = SPA
  + フロントエンドreact router v7
  + バックエンドのAPI方式はWebAPI
  + DBはPostgreSQL
  + DWH(Data Ware House)はSnowflake
* メンバーのスキルセット
  + デンソーさん3名
    - 市東さん
      * TSもPythonも書ける。
      * MVCのフレームワークはJavaのSpringを研修で経験済
    - 西村さん
      * Javaのみ
        + 別のプロジェクトでRAG触っているのでPythonやってるかも？
      * 市東さんの見立てではすぐにこなせるスキルがありそう
      * 開発メインではなかったので慣れるのは少し時間かかりそう
      * TS勉強中。JavaのSpring経験済
    - 濵﨑さん
      * PythonもTSも市東さんよりかなりできる！
      * なんでも対応できそう
      * Reactは勉強中。
      * JavaのSpringは未経験
      * AWSバリバリ
  + CL新メンバー
    - 陳さん
      * TSが得意
      * NestなどのTSバックエンドの経験が豊富
    - ひとり
      * NestやRails経験あり

**背景 : Context(なぜ今この意思決定をするのか)**

**決定のための材料 : Material for decision**

* [ChatGPT Deep Researchの調査結果](https://chatgpt.com/share/67ca79be-e118-8009-8496-baef0312b7ea)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **項目** | **Express (TS)** | **Hono (TS)** | **NestJS (TS) +佐藤+山本+市東+蔵内** | **Flask (Py)** | **FastAPI (Py)** | **Django REST (Py)** |
| **特徴 / 概要** | - Node.jsの老舗フレームワーク  - シンプルで柔軟性高い  - ミドルウェアの充実度◎ | - 超軽量＆高速ルーティング  - Web標準APIに近い設計でマルチランタイム対応（Cloudflare, Deno等）  - TypeScriptとの親和性が高い | - モジュールアーキテクチャ / DIを備えたフルスタックフレームワーク  - Angularライクな構造  - 大規模APIに向く | - シンプルなマイクロフレームワーク  - 必要最小限のみ提供  - 拡張ライブラリ豊富 | - 非同期処理対応のモダンAPIフレームワーク  - Pydanticで入力バリデーション / OpenAPI自動生成  - 高速・少ないコード量で開発しやすい | - Django上に構築されたREST専用フレームワーク  - 認証や管理画面、ORMなどフル装備  - 大規模開発や管理系機能向き |
| **学習コスト / DX** | - JS経験者なら入りやすい  - 設計ルールは自分で決める必要  - 大規模化で混乱しやすい (自由度高) | - 非常にシンプルなAPI & ミドルウェア  - 設計自由度が高く軽量  - ドキュメント・コミュニティは成長中 | - デコレータ＆モジュール構造を学ぶ必要あり  - CLIや公式ドキュメントで学習しやすい  - 大規模プロジェクトでの効率◎ | - Pythonで最もシンプル  - 初期学習は容易だが、拡張は自前設計次第  - 大規模化すると構造管理が課題 | - 非同期I/O + Pydanticで作業効率高い  - 自動ドキュメント機能充実  - Pythonが初めてでも比較的習得しやすい | - Djangoの概念(プロジェクト構造, ORM, 管理UI等)を習得する必要  - 大規模サイト想定の設計がデフォルト  - 設定や概念が多めで小規模にはオーバースペックになりやすい |
| **認証・認可 (JWT, RBAC, ABAC)** | - Passportや各種JWTミドルウェアで実装  - RBAC/ABACは自前かCasbin等を組合せ | - JWTミドルウェア標準提供  - RBAC/ABACは自前で組む必要  - 軽量ゆえ自由度は高い | - Passport統合、公式ドキュメントでJWT認証例あり  - Guards + DecoratorsでRBACを記述しやすい  - Casbin/CASLとの連携事例も | - Flask-JWT-Extended等でJWT実装  - RBAC/ABACはFlask-Principalや自前実装 | - OAuth2/JWTを公式チュートリアルでも使用  - Dependencyで権限チェック挟める  - RBAC/ABACは自前ロジック or Casbinなど | - django-rest-framework-simplejwtでJWT容易  - Django本体のPermission/Group機能強力 (RBAC可)  - ABAC的な細かい制御はカスタムPermissionクラス |
| **パフォーマンス / スケーラビリティ** | - Node.js特有のノンブロッキングI/Oで高並行処理得意  - Express自体は高速だが競合に比べやや古い設計 | - ベンチで非常に高速 (Expressより上)と報告あり  - 軽量ルーター/Trieで高RPS  - HTTP/3対応などモダン機能 | - 多少のオーバーヘッド（DIやReflect）あり  - 主要アダプタはExpress or Fastify  - 大規模/マイクロサービス化の公式サポートあり | - 1リクエスト1スレッド同期処理が基本  - Gunicorn+複数ワーカーでスケール  - PythonのGIL制約下でI/O中心なら十分性能 | - 非同期I/O (Uvicorn/uvloop)により高並行処理可能  - Expressに匹敵するかやや劣る程度 (ケースバイケース)  - 大規模トラフィックにも対応しやすい | - Djangoは同期処理が基本 (ASGI対応もあり)  - 大規模サイト実績あり  - 中/大規模でもORMやキャッシュレイヤで対応しやすい |
| **REST API実装 / バリデーション** | - ルーティング & ボディパースは自分で設定 (router, body-parser等)  - joi, zod 等のバリデーションを併用 | - ルーティング宣言的にシンプル  - zodやValidatorでバリデーション可  - OpenAPI自動生成はサードパーティ | - デコレータでルーティング定義  - DTO + class-validatorでリクエスト検証  - @nestjs/swaggerでOpenAPI自動生成 | - デコレータでルート定義  - Marshmallowなど外部ライブラリでバリデーション  - REST構造は自分で設計 | - 関数引数にPydanticモデルを置くだけで自動バリデ  - OpenAPI/Swaggerが標準装備  - コード量少なく迅速にAPI作成可能 | - Viewセット/SerializerでCRUDを半自動化  - Django ORM連携、Serializerがバリデ実行  - drf-yasg等でOpenAPI生成 (公式にもSchema機能追加) |
| **エコシステム & 周辺ツール連携** | - ミドルウェア/プラグイン多数  - Passport, Sequelize, TypeORM, Prisma 等豊富  - npmでほぼ何でも揃う | - 新興フレームワークだが公式ミドルウェア増加中  - Cloudflare Workers対応など先進事例あり  - npmモジュール併用可能 | - NestJS公式モジュール(認証, GraphQL, etc)豊富  - TypeORM, Prismaとの統合テンプレ多  - CLIやスキャフォールドが充実 | - Flask拡張ライブラリが多数 (Flask-Login, Flask-CORS etc)  - SQLAlchemy, AlembicでDB管理  - 自分で組み合わせが必要 | - Starlette/SQLAlchemy連携など最新トレンド  - Pydanticとの相性◎  - AI/ML等Python資産と組み合わせやすい | - Django本体の巨大エコシステムが利用可  - Admin機能、認証基盤、ORMなど一通り揃う  - 広大なコミュニティとパッケージ |
| **フロントエンドとの相性 (TS/React)** | - 同じ言語(JS/TS)で思考切替不要  - コード共有はやや工夫必要(型生成など) | - 全TS環境のため学習コスト低・認知負荷小  - 超軽量高速API→フロントとの相性良い | - TSで統一、デコレータ型定義とOpenAPIから型生成可  - 大規模でもモジュール構造で管理しやすい | - 異なる言語(Python)で切替が発生  - フロントエンドから見るとAPI検証時に別言語リポジトリを読む | - PythonとTSで言語が分かれるがFastAPI自動ドキュメントあり  - Pydanticモデル→OpenAPI→TS型生成すれば型整合性担保 | - Django & TypeScriptは言語が分かれる  - DRFスキーマからTS型生成も可能  - 管理画面等フロントと並行開発しやすい |
| **保守性・拡張性** | - シンプルなので自由設計だが大規模向け指針少ない  - マイクロサービス化は手動設計 | - 軽量で機能少なめ -> 大規模化で工夫必要  - シンプルなのでサービス分割もしやすい | - フレームワークが構造を強制(モジュール/DI)  - 大規模開発やマイクロサービス対応が容易  - 長期保守にも向く | - 小規模～中規模向けの設計  - 大規模化にはFlask-Blueprint分割や拡張工夫  - Celery等で非同期タスク連携可能 | - 高い生産性と自動ドキュメントで小～中規模を素早く  - 非同期I/Oで拡張性もOK  - 大規模化にはサービス分割や設計工夫が必要 | - 大規模サイトに最適化 (認証, 管理UI, ORM, 国際化など含む)  - 「バッテリー同梱」思想で拡張容易  - 小規模にはオーバースペックだが長期保守に強い |
| **BI用途 (分析/グラフ生成)** | - Node.jsで重い計算は得意でない(非同期I/Oが強み)  - 大量データ分析はDBやサーバレス連携が一般的 | - 高速ルーティングでI/O部分は良好  - 計算処理は普通のNode環境と同じ | - サーバー内部で大きな分析をする場合はWorker活用  - Pythonに比べMLライブラリは少ない | - Pythonライブラリ (NumPy, Pandas, Matplotlib)が強い  - APIから直接分析結果返せる  - CPU負荷時はワーカー数でスケール | - Pythonの科学技術ライブラリを活用  - リアルタイム分析にも対応しやすい(非同期+Pandas)  - Node.jsより大規模データ分析は得意 | - 同上。DjangoでのビジネスロジックにPandas/NumPy利用可  - 大規模分析ならCeleryやJupyter連携など大きな生態系活かせる |
| **生成AIとの親和性** | 高？  枯れた技術なので情報は多い。 | 中〜低？  新興フレームワークなのでどのくらい補完が効くか不明。 | 高？ | 高？ | 高？ | 高？ |
| 経験者 |  |  |  |  |  |  |

**3/12 お気持ち**

* 佐藤
  + NestJS
  + 理由：今回のシステムだと認証周りが複雑になるので、NestJSならその辺りも対応できそう。
* 山本
  + NestJS
  + フロントの親和性を考えるとTS系のフレームワークが良い。
  + 初心者が多いならHonoやExpressでも良いかと考えたが、できる人が多いのでNest JS
* 市東
  + NestJS（個人的） or Express（会社的）
  + 社内事情的にはExpressを利用している人が多い。
  + SandthrewのExpressはフロントまで使っているのでそのまま活かせるかは不明。
* 蔵内
  + NestJS
  + 機能的に充足していること
  + 経験者が多い
  + 枯れた技術なのでAIとの親和性も高い。

**結論 : Decision**

**結果 : Consequences**

Nest.jsに決定

**2/17 バックエンド技術選定（API方式）のADR**

**いづメモ：バックエンドAPI方式のADR３回目なので、忘れつつあるかもしれないのでここで改めて目的を再確認（なぜここ）**

* いろんな状況（顧客価値、納期、チーム、自分たちの意欲、今の技術力などなど）をかいまみて、顧客と自分たちにおいて今取りうるベストな選択を考える
* いきなり一極集中せずに、一度選択の幅をひろげてから考える。
* 気をつけること：チャレンジ要素を増やしすぎると自分たちが苦しくなるかもなので、引き算も大事

**ステータス: Status**

accepted 受理

**状況 : Situation**

* 言語とフレームワークと、API方式、サーバレスにするかどうかなどなどについて広く決まっていない状態
* レンダリング方式はCSR = SPA
* フロントエンドreact router v7
* DBはPostgreSQLに決定している
* DWH(Data Ware House)はSnowflakeに決定している
* BIはPowerBIに決定している

**背景 : Context(なぜ今この意思決定をするのか)**

**決定のための材料 : Material for decision**

tRPCのDeep Research: <https://chatgpt.com/share/67b2a1a5-8ee0-8009-b430-0636e68cc981>

AWSのREST GraphQL比較記事　[GraphQL と REST API の比較 - API デザインアーキテクチャの違い - AWS](https://aws.amazon.com/jp/compare/the-difference-between-graphql-and-rest/)

バックエンド技術選定のDeep Research: <https://chatgpt.com/share/67b2a705-2ff8-8009-a08b-0d5ae353ef17>

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 選択肢 | RESTful API | Web API | GraphQL | tRPC | gRPC |
| 概要 | REST の原則を厳密に守っているので、より統一したインターフェースを提供できる。  ステートレス  プロトコルは基本はHTTP | 最も広く使用されているAPI方式です。  プロトコルは基本はHTTP以外でも使用可能 | Facebookが開発  クエリベースのAPIで、クライアントが必要なデータだけを指定して取得できます | server側で定義したInterfaceをそのままclient側で取り込んで繋ぎ込みができるという代物です。 | Googleが開発した、高性能なリモートプロシージャコール（RPC）システム |
| 顧客への推しポイント | 昔からあるシンプルな設計思想なのでわかりやすい。 | 広く普及しているため、システム間の相互運用性が高い | 必要なデータだけリクエストができるので、効率よくデータ転送ができる | REST APIとGraphQLの中間。型安全でフロントとバックエンドのスキーマ齟齬を防げる→バグが起こりにくい |  |
| メリット | ・慣れている。  ・標準的  ・クライアント（フロントエンド）から扱いやすい  ・長期的なメンテナンスはやりやすい | ・慣れている。  ・このなかで一番知見が多い。  ・広く知られているので情報が多い | オーバーフェッチ、アンダーフェッチを減らせる。  フロント側で、レスポンスの構造を決定できる。  サーバーからデータの変更をプッシュできる。  型定義がしっかりしている。  複数のデータソースを一つのエンドポイントにまとめられる。 | フロントとバックエンドで型を共有できる  [Concepts | tRPC](https://trpc.io/docs/concepts) | データ通信量やパフォーマンスが重要なシステムに向いている |
| バックエンド言語 | ほとんどのバックエンド言語OK | ほとんどのバックエンド言語OK | ほとんどのバックエンド言語OK。  リゾルバはVTL or JS(TS) | Node.jsのみ | ほとんどのバックエンド言語OK |
| デメリット | 1APIが一つのリソースに対して一つ動作（CURD）を行うので、複数のリソースが絡んだ複合的なデータを取り出すには何度もAPIを打つ必要がある | 型指定が弱い。  開発者間の認識齟齬により結合時に問題が発生しやすい。  フロントエンドとバックエンドの型整合を取るのにサードパーティのツールが必要。  どういったAPIにするかの設計が必要。（URLの設計も含め） | ・REST APIと比較して、GraphQLはサーバーサイドの実装がやや複雑  ・認証と権限管理がRESTより難しい  ・新しい技術であり、特に初心者にとっては学習コストが高いことがあります。 | モノレポに固定？ バックエンドのスキーマファイル駆動になるので、フロントとバックエンドで分かれて開発はしづらい  [Define Procedures | tRPC](https://trpc.io/docs/server/procedures#writing-procedures)  実装ベースになりそう | バイナリでの通信なので、REST APIに比べて、テストやデバッグが複雑になる。 |
| 学習コスト | 中？ | 低 | 中 | 高？ | 高？ |
| 設計コスト | 低 （RESTに従うだけ。知見あり） | 高 | 中（GraphQLの作法に従える。知見なしのため中） |  |  |
| 運用コスト |  |  |  |  |  |
| 具体的な実現手段 | ECS+django, API Gateway+Lambda,  Nestjsなど |  |  |  |  |
| 懸念 | 厳密なRESTful APIの設計経験なし |  | ガシガシ使った経験はないから不安 |  |  |
| 経験者 | ちょびっと（蔵内、佐藤） | 全員 | ちょびっと（蔵内、佐藤） | なし | なし |

* REST推し、必要になってきたらGraphQLを使うでいいんじゃないかなー（岡崎さん、ぽんさん）
* 合理性でいうとREST推し、でもちょっと攻めたいので違うのを選びたい気もする（烈くん）
* ちゃんと書かないと読みにくくなる（奥が深い）
* せっかく自分たちで技術を選定できるので、なにかではちょっと攻めていきたい
* RESTか、外部システムとの連携があるっていみではGraphQLかな
* これまでだと結合テストで問題起きることが多いけど、GraphQLだとそういった問題が起きにくいのでは。ガシガシ使った経験はないから不安
* チャレンジはどこでする？
  + 技術選定ではなく、開発生産性とかデプロイの効率を上げることでチャレンジするのでもよさそう
  + react routerも初めての技術だからそういう意味ではチャレンジ
* 思いつき（いづ）
  + チームでチャレンジしたいことを１つ決めてもいいかも
    - 新しい技術にチャレンジして技術習得しちゃうぞ！
    - 開発生産性を高めて、納期に寄与するぞ！
    - チームの軸が決まる
* 選択肢
  + RESTful API　only +
  + WebAPI only ++
  + RESTful + GraphQL(重そう)
  + WebAPI + GraphQL(重そう)
  + GraphQL only ++

現段階での意見

* 烈くん
  + 完全なRESTful APIは難しいと思うので、WebAPIかGraphQLかな
* 岡崎さん
  + WebAPIonly
    - GraphQLは難易度が高いのと使い所がよくわからない。やるとしても後から追加する感じ、スピードを考えるとWebAPI
* 佐藤さん
  + GraphQLは開発難易度が難しい、ResolverでのJoinが必要なのかどうかとかももっと調べたほうがいい
  + 画面に合わせたRESTfulっぽいAPIが良いかな
* 蔵内さん
  + スピードを考えるとGraphQL
    - テーブルのジョインとかはGraphQLがやってくれる
    - n+1問題とかの回避は工夫は必要(data loaderを利用)
    - バッチの使いまわしができる
    - GraphQL特有の部分の学習コストはかかるけど、トータルで見ると早くなるのではないか？
    - ビジネスモデルは複雑にならないと思っている
    - 新メンバーが入ってくるのでGraphQLに関する調査はやってもらえそう
* GraphQL
  + 学習コストはかかる
    - Amplifyの学習コストと運用コストが高い
    - 烈くんと新メンバーが主にやっていく
  + 運用コスト
    - Amplifyを使うと運用の難易度があがる
  + 不明点、懸念点が多い
  + 使い所のイメージがついていない
  + 結合テストは楽になりそう
  + トータルでは早くなりそう
  + n+1問題
    - →解決策はある
  + data APIを使うとlambdaを使わなくてよくなる
  + ツリー階層のアクセス権のあるプロジェクトの取得にSQLのCTEクエリが必要になりそうなので、data APIでできるのか？確認が必要
  + App Syncを使うときはAmplify gen2に乗っかるのがよい
    - 開発者毎に分離されたクラウドサンドボックス環境を用意できる
    - 学習コストが高い
    - Amplify gen2でAppSyncだけ切り離して構築できるのか？
* WebAPI
  + みんな知っている
  + 烈くんがフロントとバック両方やるなら、フロントでも覚えることが多いからWebAPIのほうが
  + スキーマ駆動のためのCI/CDを早期に構築するのが良い
* RESTful
* Amplify gen2
  + スキーマ作成
  + AppSyncのIaC
    - ここでオーロラのDBも作られるとややこしいことに
  + resolverの定義
  + バックエンドの人数が多いとぶつかる
* Amplify gen2の課題
  + バックエンドの平行開発が可能か？
  + AppSyncのみを管理できるか？
  + デンソーのEntraIDでの認証の検証
  + ツリー階層のアクセス権のあるプロジェクトの取得にSQLのCTEクエリが必要になりそうなので、GraphQL（Data API）でできるのか？確認が必要
  + AWSにロックインされる可能性

⅗ スタート時点での意見

蔵内

* Web APIが良いと思う（コンテナマイクロサービスの構成）
* 理由
  + 運用より開発の工数削減を意識すると以下の理由からWeb APIがいいと思う。
  + 情報が多い（AIの生成が効きやすい）
  + コンテナマイクロサービスの構成にすることで開発も進めやすい
  + Amplify Gen2は情報が少ない（AI駆動しにくい）
    - GraphQLの良さを生かそうとするとAmplify Gen2を使うことになるが、Amplify Gen2がリリースされたばかりで、情報が少ない
  + Amplify Gen2開発環境の分離ができるが、作成されたリソースが追いにくい
    - AmplifyでSandboxを作成できるが、生成されたリソースがどれなのかCloudFormationを追わないとわからない。
* 岡﨑
  + 結論：webAPIでマイクロサービス化（連携システムが多いので）
    - appsync単体の利便性が低いのであればWebAPIがいいと思う
    - 少なくともAmplifyは現実的ではなさそう
      * 思想としてフルスタックにスピード感を持って開発する点はマッチしているが、シンプルなインフラやある程度Amplifyにロックされる前提になる
      * 後でカスタマイズしようとした時にとても複雑になる懸念
      * 福井さんから後から機能追加していく前提との共有があったので、それも考慮すると少ともamplifyだとやりにくい面が増えそう
* 山本
  + 結論：webAPIかな...
  + GraphQL
    - GraphQLの感触はよかった
    - インフラ側でビジネスロジックや型などが分かれているのがわかりやすいというのはあった。これなら開発は早そうではある
    - デプロイやテストの懸念は引き続きあると思う
    - スキーマだけ決めれば平行で開発できるんじゃないかな？現段階ではそこまで大きく平行には走らないと思うし、そこまで考えるならマイクロサービスとかになる気がする
  + webAPI
    - ソフトウェア設計でビジネスロジックやドメイン層の部分をうまく書けるようにカバーする必要がある
      * 理想ではあるがここがぐちゃぐちゃになりやすいよねは正直ある
    - やるならソフトウェアアーキテクチャしっかりさせていきたい
    - インフラ的なわかりやすさはあるからデンソーさんを巻き込むのを考えるとこっち？
* Web API or RESTful APIはどう？
  + RESTfulのルールに厳格に従うとなるとかえってやりにくい面がでてくるので、WebAPIのほうが融通がきくのでは

**結論 : Decision**

* Web APIに決定（GraphQL+Amplifyは調査の結果懸念点を払拭できなかったため）

**結果 : Consequences**

**2/14 フロントエンド技術選定のADR**

**前回のおさらい**

* レンダリング方式は**CSR = SPA**に決定

**今日のゴール**

* レンダリング方式を踏まえつつフロントエンドの技術を選定する
* (できればインフラも決めたい)

**ステータス: Status**

* accepted 受理

**状況 : Situation**

* 決定時のビジネスにおける制約や会社、チームの状況、前提、制約
* レンダリング方式はCSR = SPAに決定
* 今日はソフトウェアアーキテクチャの中のフロントエンド技術を決定する
* ソフトウェアアーキテクチャに関して、顧客へ提案する前のチーム内合意をする段階
* DENSOビジネス基盤開発課の標準スタックがReactである
* 動的なページの生成ができるアーキテクチャであること
* 開発期間は半年
* 大人数開発を見越して考えて、フロントとバックエンドを分けておきたい
* BIでグラフ表示もある

**背景 : Context(なぜ今この意思決定をするのか)**

* ADRを作成するに至った経緯
* ソフトウェアアーキテクチャ選定についてこれから顧客へ提案を行いたい
* どのアーキテクチャを選ぶかによって費用や難易度もかわるため、「CLとしてこれがいいと思う」という提案を、理由を添えた上で顧客へ示したい
* 1年くらいの長いプロジェクトになるのでここはしっかり考えたい
* （局所的な議論ではなく）一旦広い選択肢を把握してからベストな選択を決定していきたい
* チーム内の知識量がバラバラだと思うので議論を通じてある程度近づけたい
* PoCフェーズが進んできて、ソフトウェアのアーキテクチャによって提案内容が変わる可能性が出てきたため。
* PoCのウェブアプリはNext.jsのSSG作成しているが、実際のウェブアプリでは使用できない方式のため
  + サーバーサイドレンダリングの知見tがなかったため、クライアントサイドでの生成を簡単にできる方法として一時的にSSGを選択した
* バックエンド部分のアーキテクチャを検討していく上でも決めておく必要があるため。

**決定のための材料 : Material for decision**

* どんな選択肢があるか
* その選択肢の情報
* その選択肢のメリット・デメリットなど
* ChatGPT o3-mini-high Deep Search: <https://chatgpt.com/share/67aef1d5-12c8-8009-beca-64c0469f815c>

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 選択肢 | react router v7++ | react (単体) | next.js(CSR) | remix | Gatsby |
| 概要 | フレームワーク（v7よりも前はルーティングライブラリだった） | ライブラリ | フルスタックフレームワーク | remixの新しいバージョンはreact-router v7に統合された（ていく）ので、react-routerの旧バージョンのイメージ | Reactをベースにした静的サイトジェネレーター（とgptが言っている） |
| 顧客への推しポイント | nextよりもシンプルなので学習コストが低く、キャッチアップがしやすい。  SSRやキャッシュ周りでハマる可能性を低減できる。 |  | app routerが使いやすい。  情報が多い |  |  |
| 動的ページ作成可/不可 | 動的ページ作成できる | React単体ではできないので他のライブラリ（何？）と組み合わせる必要があり | 動的ページ作成できる |  | 静的サイト生成に特化したフレームワークですが、外部データを使って動的なページ（静的に生成された動的ページ）を作成することができます。  めんどくさそう |
| メリット | SPAがメイン。  キャッシュが自動で有効になったりしないので、理解しやすそう。  ファイルベースのルーティングとコンフィグベースのルーティングを選択できる。 | 軽い。シンプル | 認証の仕組みをフレームワークに則って実現できるならそっちのほうが楽  CSRとも相性が良い  みんなやったことがある  バックエンドの使い分けをすればフロントエンドが得意なエンジニアでも対応できそう | [SPA](https://remix.run/docs/en/main/guides/spa-mode)モードはある  react-routerの古いバージョンに当たる（？）ので、情報は多そう |  |
| デメリット |  | **React自体はライブラリ**であり、完全なフレームワークではないため、動的ページを作成するためには、**追加のツールやライブラリとの組み合わせが必要になる** | 純粋なSPAを作ろうとするとSSRの機能が開発時に邪魔で辛い [Next.jsでSPAを構築する辛み](https://speakerdeck.com/hayatow/next-dot-js-de-spa-wogou-zhu-suruji-noxin-mi?slide=10) | 知見がない |  |
| 開発コスト |  | 中くらい？  わからん | メリットのある使い方をできていないので逆に学習コストが高いかもしれない  next.js未経験者が途中から参加する人にはハードルが高い |  | 高そう  （経験者なし） |
| 費用 |  |  |  |  |  |
| 具体的な実現手段 |  |  |  |  |  |
| 懸念 | 出たばかりで情報が少ない。 | 情報は多い | 情報は多い。  Next.jsを使った方が学習コストが高いのではないか。  （SSRがメインの使い方なので、SPAを使おうとすると情報も少なく、問題が起こる可能性が高い。） |  |  |
| 経験者 | **なし** | なし | **みんな、部分的に利用** | **なし** | **なし** |

**結論 : Decision**

* 全員合意したので、react router v7に決定

**結果 : Consequences**

**12/25 レンダリング方式のADR**

**今日のゴール**

* レンダリング方式について、顧客へ提案する前に「CLとしてはこれでいこうと思うんです」というチーム内合意をする
* ADRという新しい「チームでの合意の仕方」にチャレンジしてみて、よさそうだったら継続する
* 意思決定が前よりもちょっとうまくなっている

**ADR（Architecture Decision Record）とは**

ADRとは重要なアーキテクチャの意志決定を、背景、結果と共に記録したドキュメント

* 何がいいのか？
  + なぜその選択をしたのかを当時の制約や背景と共に思い出すことができる
  + プロジェクトに新しく参加した人が、意思決定の背景から理解できる
  + 変えるべきタイミングが来た時に、どう変えればいいのか判断する材料になる（やみくもに変更することを防ぐ）
  + チームの意思決定がうまくなる
* ポイント
  + 意思決定の背景・動機が残っている状態となること
  + ドキュメントは小さな粒度で残していくこと
  + ADRは仕様書ではなく、決定のログとして残していくこと

・・・を目指します。最新に更新しておくことを目指しません。

* 参照：<https://techblog.zozo.com/entry/adr-culture-of-zozofit>

**ステータス: Status**

* この項目があることで、ドキュメントの状態が一目で分かるようになる
* ADRの意思決定がどの状態かを記載（以下の3種類から選択）
  + proposed 提案中 :同意がなされていない状態
  + accepted 受理 :プロジェクト内で合意がなされている状態
  + deprecated 廃止 :本ADRの意思決定を変更あるいは取り消されている状態

**状況: Situation**

* 決定時のビジネスにおける制約や会社、チームの状況、前提、制約
* 重要な決定ほど外的要因に左右されるため、決定に作用した状況を記載します
* 意思決定の状況があることでテクノロジー以外の部分で決定に作用した内容を把握できるようになります

**背景 : Context**

* ADRを作成するに至った経緯
* 決定内容は記録が残っていなかったとしても最新のプロダクトからある程度把握が可能ですが、決定の背後にある動機や理由は記録がなければ読み取ることは非常に困難です
* 読み手に対して何故この意思決定が必要となったのかを伝える上でとても重要な項目です

**決定のための材料 : Material for decision**

* 決定の材料を記載します。意思決定をスムーズに行うために必要な項目です
* どんな選択肢があるか
* その選択肢の情報
* その選択肢のメリット・デメリットなど

**結論 : Decision**

* 最終的な決定内容と理由
* 最終的な決定を簡潔に記述します
* 決定に至った背景・理由も記述すると後追いで見た時により価値が高いドキュメントになる

**結果 : Consequences**

* 最終的にどうなったか（物事が最終的にどういきついたか）
* やってみてチームへの影響とか、あらたにできて制約と影響とか
* 決定時に結果を記入できるものは記録し、振り返りが必要なものに対しては後追いで記述します

**レンダリング方式のADR**

**ステータス: Status**

* accepted 受理 🎉

**状況 : Situation**

* ソフトウェアアーキテクチャに関して、顧客へ提案する前のチーム内合意をする段階
* 前回同じテーマを議論したけど決めきれなかった
* ソフトウェアアーキテクチャの中でも、まず「レンダリング方式」を決めたい
* チームはレンダリング方式についての理解度はバラつきがある状態
* チームは新たに編成したチームで、フロントエンドで採用するReactもプロダクトレベルでの使用は初めて。
  + Vue.jsは一部のメンバー経験あり。
* SPAの経験はあるが、それ以外のアーキテクチャでのプロダクト経験はない。
* 開発期間がきっちりと決まっており、プロジェクト規模から行くと短い。
* 4ヶ月後に2名エンジニアが増える。（採用が必要）
* 本開発では、蔵内、岡崎が抜ける。

**背景 : Context(なぜ今この意思決定をするのか)**

* ソフトウェアアーキテクチャ選定についてこれから顧客へ提案を行いたい
* どのアーキテクチャを選ぶかによって費用や難易度もかわるため、「CLとしてこれがいいと思う」という提案を、理由を添えた上で顧客へ示したい
* 1年くらいの長いプロジェクトになるのでここはしっかり考えたい
* （局所的な議論ではなく）一旦広い選択肢を把握してからベストな選択を決定していきたい
* チーム内の知識量がバラバラだと思うので議論を通じてある程度近づけたい
* PoCフェーズが進んできて、ソフトウェアのアーキテクチャによって提案内容が変わる可能性が出てきたため。
* PoCのウェブアプリはNext.jsのSSG作成しているが、実際のウェブアプリでは使用できない方式のため
  + サーバーサイドレンダリングの知見tがなかったため、クライアントサイドでの生成を簡単にできる方法として一時的にSSGを選択した
* バックエンド部分のアーキテクチャを検討していく上でも決めておく必要があるため。

**決定のための材料 : Material for decision**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 選択肢 | SSR | CSR = SPA | SSR + CSR | SSG | ISR |
| 概要 | ・サーバーサイドでレンダリング  ・ユーザーのリクエストがあるたびに、サーバー側でウェブページをすべて完成させてから、その状態をブラウザに送る | ・クライアントサイドレンダリング  ・動的にページを生成できる | SSRとCSRのいいとこどり。 | 静的なページの生成 | SSGの拡張手法  SSG＋動的更新ができる |
| 顧客への推しポイント | ページの表示速度が速い | 初期読み込みは遅いが、ページ遷移が早く、ユーザー体験がよいと思われる  JSの分割ロードで初期読み込みは改善しやすくなっている | 初回ページの表示速度がそこそこ早く、インタラクティブなUIを実現可能。 | 読み込みが早い。インフラ面での管理がとても簡単になるので運用費が安くなる | SSG並みに高速だが、自動的な更新も可能。 |
| メリット | ・ページがリクエストされるたびに最新の状態にレンダリングされる。  ・SEO対策として有利  ・ReactやVue.jsのようなフレームワークでもSSRがサポートされている | ・初回のロード後はクライアント側で必要な部分だけを更新するため、ページ遷移が非常に速い。  ・動的にデータを取り込むのでページの部分的な更新ができ、ユーザ体験が良い。  ・S3にデプロイ可能。  フロントとバックエンドで責任が明確に分かれているので、理解しやすい。 | ・サーバー側でフェッチできる  ・バックエンドも実装できるので、少人数のフルスタックチームに適している（認知不可は大きい）  ・フロントエンドとバックエンドのアプリを統合できる | ・圧倒的な高速表示  ・サーバー負荷ゼロ、SSGでは、コンテンツは事前に全て生成されるため、リクエスト時にサーバーで動的処理が必要ありません | ・初回リクエスト時に即座に静的なページを返す  ・サーバー負荷の軽減とスケーラビリティ |
| デメリット | ・サーバーへの負荷が大きくなることがある  ・リアクティブなページの作成にはCSRと組み合わせる必要がある | ・初回ロードが遅くなる可能性あり | 予期せぬコードが公開されてしまい、セキュリティリスクになる可能性がある。 | 反映の即時性がない（プロジェクトを作るたびにビルドが必要） | プロジェクトを作るたびにビルドが必要  リアルタイム性が求められるケースに不向き（株価の表示や、オンラインゲームのスコアボード） |
| 開発コスト | サーバ側のコードも混ぜて書けるので、別途APIの定義等をする必要がなく、開発コストが低い。 | 別途バックエンドの構築・実装が必要。（コスト高） | SSRのコードとCSRのコードが混ざるので、認知負荷が高い。 | 動的にページを生成できないので設計がしづらい | 中身をちゃんと理解している人がチームにいない（コスト高） |
| 費用 | フロントの提供にサーバーが必要。  （コスト高） | S3で提供可能。（コスト低）  バックエンドは別途必要（コスト高） | インフラの選択肢がCSRほど多くない | フロントの提供にサーバーが必要。  （コスト高） | フロントの提供にサーバーが必要。  （コスト高） |
| 具体的な実現手段 | next.js | react router v7 | next.js |  |  |
| 懸念 | ・サーバーサイドでの開発経験のあるメンバーが少ない  ・インフラコストがCSRより高そう | ・APIを別途構築しないといけないのでどのくらい開発にかかるか？ | ・サーバーサイドでの開発経験のあるメンバーが少ない  ・インフラコストがCSRより高そう | 動的なページの生成ができない。プロジェクト生成などのユースケースに対応できない | 動的なページの生成ができない。プロジェクト生成などのユースケースに対応できない |
| 経験者 | 岡﨑　**少し** | 蔵内、佐藤 | 岡﨑　**少し** |  |  |

メモ：

* 観点：
  + 今回のプロジェクトは開発スピード、ユーザビリティ重視
  + 費用はそこまでうるさくない、許容できそう
  + 4月からメンバー2人増えるから、そのへんのキャッチアップしやすさも考えたい
  + 本開発では、蔵内、岡崎が抜ける
  + DENSOさん（デジ本）が横展開しやすいように理解しやすさ（CL内だけではなくDENSOさんも絡んでくる）
    - Pythonでのウェブアプリ開発の経験者は少ない
  + 「SSR + CSR」とCSRの2択かな
  + 大量データ突っ込まれた時のことを考えると、SSRも選択肢に入ってくる
  + SPAだとJSが肥大化して初回ロードが遅くなりがち。
    - 今は分割ロードの機能を持っているフレームワークが多いので回避できそう。
    - ↑起こってから考える？
    - 改善するための参考情報はたくさんありそうだし
  + 将来起こりそうなことと、今の開発スピードを天秤に考えると今の開発スピードの方をメインに考えたい
  + SSR + CSRは経験者が少ないのでハマりそう

**結論 : Decision**

* **CSR = SPAにする！**
  + 開発スピード、ユーザビリティ重視を考えた時に、経験がある人が多いのでリスクが低い
  + バックエンドも同じ言語で統一できるので、バックエンドの学習コストはそこまで上がらないはず！（SSR+CSRでもそんなに変わらないが複合機能がない分負担が少ない）
  + 「SSR + CSR」とCSRの2択で考えた時に、どちらも顧客に提供できる価値はほぼ似ていて決め手にかけたので、開発スピードの観点でメリットがあるCSRを選択した
  + SSRとCSRの使い分けでつまづくリスクを回避する（見積もりずれを防ぎたい）
    - SSR + CSRの知見のある開発メンバーがいない
  + 新しく入るメンバー（4月から＋2名）もキャッチアップの負担が少ない可能性が高い
  + 2025年12月のフェーズ1開発が終わったタイミングで再度妥当性を判断できるタイミングがあるので、そこでバンドルサイズを鑑みてSSRとの併用パターンを検討すれば良い

**結果 : Consequences**

* レンダリング方式についてはチーム内合意までいけた
* 開発スピード、ユーザビリティを重視して進めたいことがチームで認識があった
* 途中入場の人の学習コストを考慮した
* バックエンドの技術（フレームワーク）選定が必要
* インフラの検討も必要

テンプレ

**ステータス: Status**

**状況 : Situation**

**背景 : Context(なぜ今この意思決定をするのか)**

**背景 : Context(なぜ今この意思決定をするのか)**

**決定のための材料 : Material for decision**

**結論 : Decision**

**結果 : Consequences**