JSConf JP 2021

GraphQL 導入の反省と再挑戦

Sponsor LT

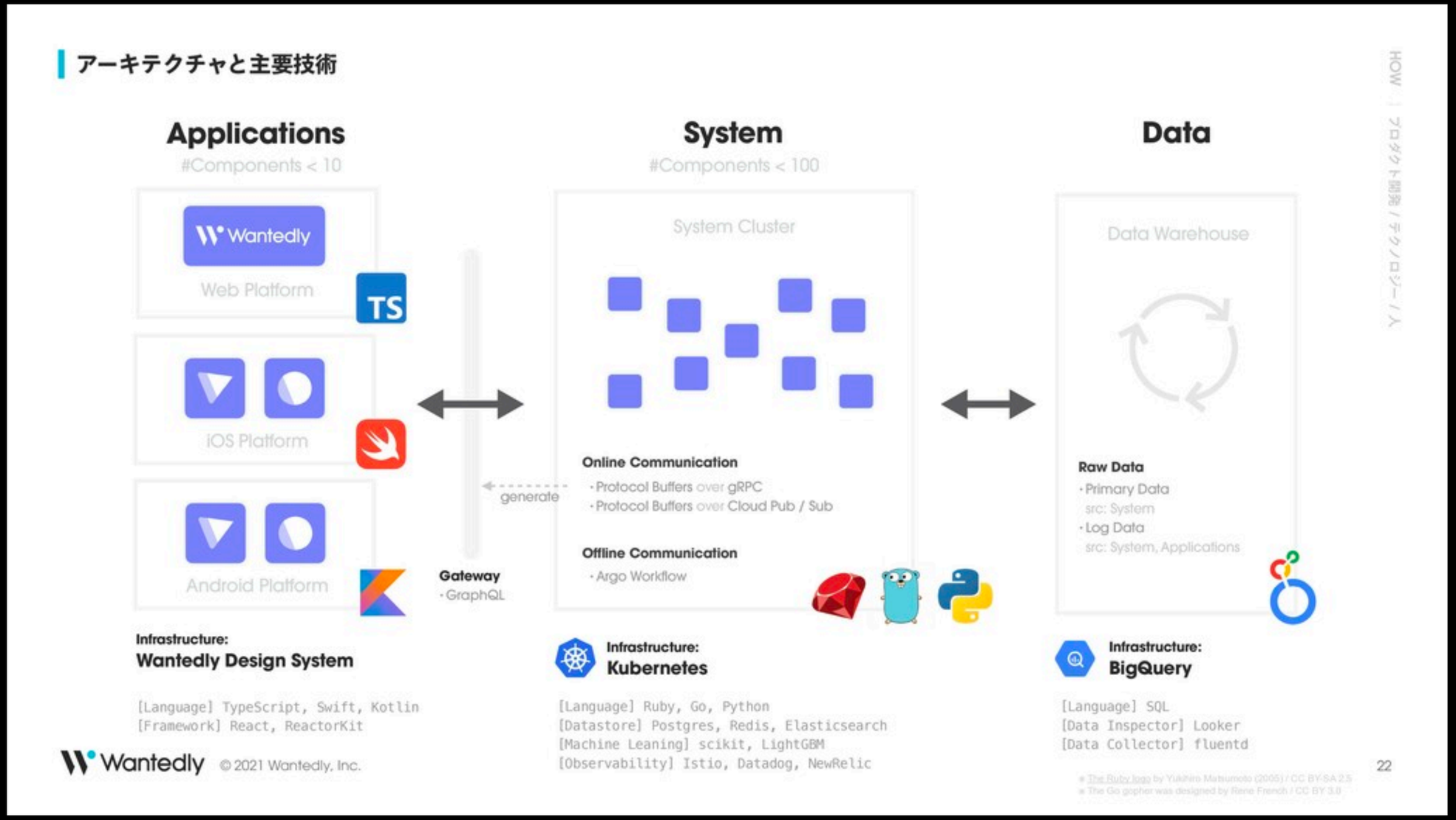
2021-11-27 @izumin5210

Wantedly

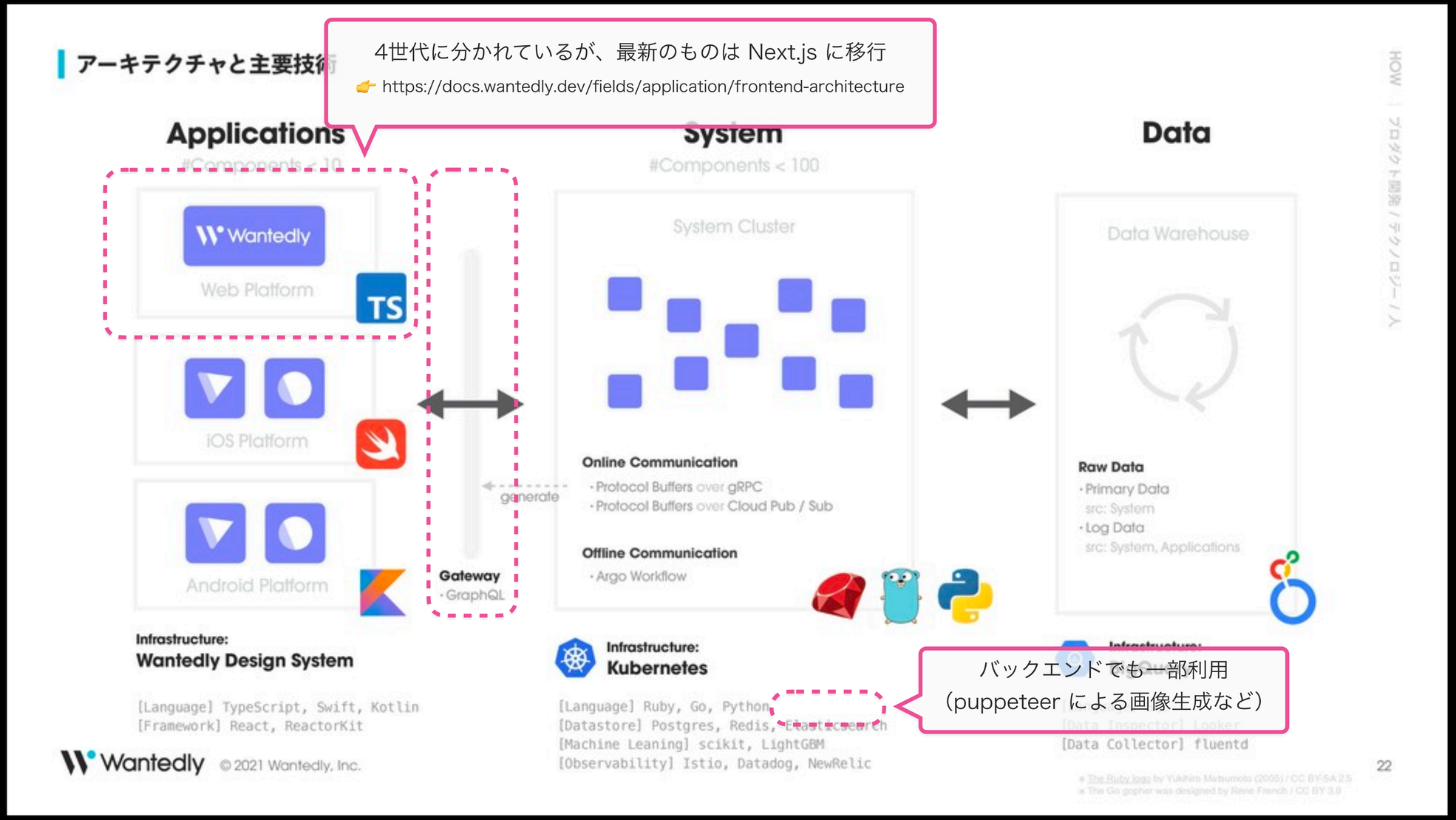
@izumin5210

- Arch squad / Wantedly, Inc.
  - Backend, Web Frontend の Tech Lead 的なこともしています
  - 代表作
    - grapi: Go の gRPC server 用マイクロフレームワーク
    - UI デザインシステムの React 実装の設計
    - GraphQL Gateway の設計・実装 (←今日はこの話)
- 好きな TC39 proposal は Error Cause
  - https://github.com/tc39/proposal-error-cause
  - いつのまにか Stage 4 になってた

JavaScript and Wantedly



JavaScript and Wantedly



Webフロントエンドアプリのアーキテクチャ

本章では、生産性の高い状態で継続的に開発していくために、今後のWebフロントエンドのアーキテクチャについてFrontend Chapter と Arch Chapter のメンバーで議論した内容について紹介します。

前提知識

ここ数年のリニューアルやリノベによりプロダクトのUIがアップデートされ、それに伴いフロントエンドの技術スタックもアップデートされてきました。これまでの技術スタックは大きく4世代存在し、v1〜v4と呼ばれています。現時点でも v4 以前の古い世代の技術スタックで書かれたコードは存在します。

- v1: Haml / SCSS / CoffeeScript + AngularJS & jQuery & Backbone.js
- v2: JavaScript + React + Redux + redux-thunk / SCSS + CSS Modules / webpack
- v3: TypeScript + React + Redux + redux-thunk / styled-components / webpack
- v4: TypeScript + React(Next.js) + GraphQL / styled-components

全体のアーキテクチャに関しての方針

- 新しいページは v4 で作る
- 企業ユーザー向け管理画面は wantedly/visit-admin-frontend
- ユーザー向け画面は wantedly/wantedly-frontend
- GraphQLサーバーは wantedly/wantedly-graphql-gateway

新規開発やリニューアルのタイミングでは可能な限り v4 で開発することを推奨します。つまり visit-admin-frontend もしくは wantedly-frontend のどちらかの上で開発を行い、GraphQL サーバーを経由して各マイクロサービスと通信を行うようにします。例えば、採用担当者向けのダッシュボード画面は visit-admin-frontend、ユーザープロフィール画面は wantedly-frontend で開発するという棲み分けです。

GraphQL サーバーについては wantedly-graphql-gateway を利用していく方針です。2021年7月時点では検証段階のため、visit-api-node で代用します。visit-api-node はすぐになくなることはありませんが、大きな新規開発で Query/Mutation を作る際にはどちらを採用するか議論と検討を行ってください。

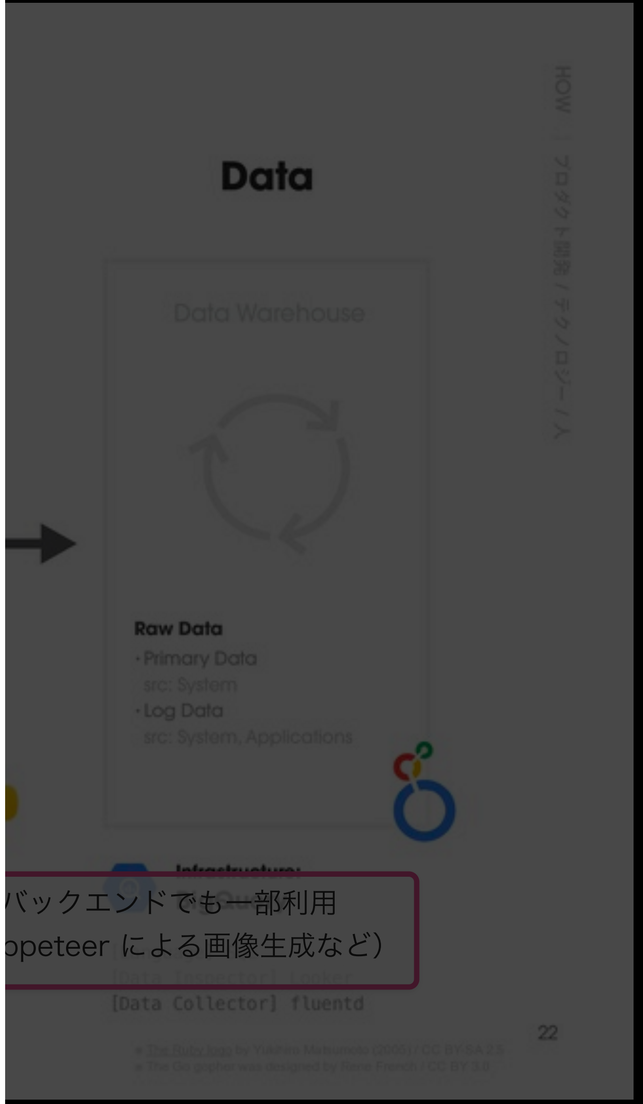
個別アプリに関して

Next.js への移行の背景

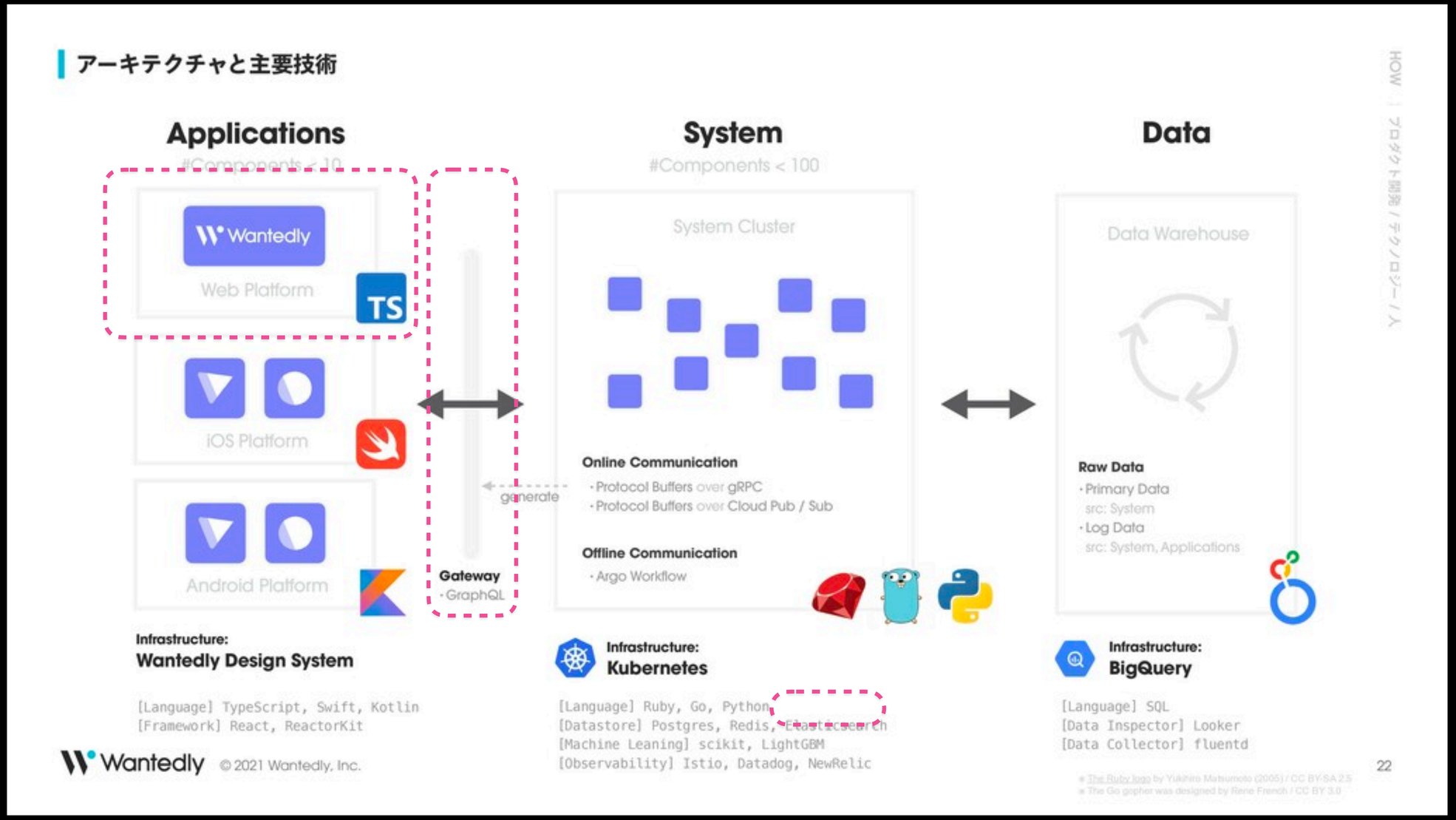
2021年10月にフロントエンドの主要レポジトリである visit-admin-frontend, wantedly-frontend ともに Create React App ベースから Next.js ベースに移行しました。以下がその背景です。

- HTML 配信における課題の解消
  - SSR の基盤を世の中的にデファクトといえるフレームワークの上に乗せ、インタフェース・実装ともにより洗練されたものに追従する
  - SSR におけるデータ取得の最適化や、SSG, ISR など他の HTML 配信戦略の検討を可能にする
- 開発体験上の課題の解消
  - フレームワーク統一により、一貫した開発体験を提供する
    - 特にルーティングフレームワーク, ビルドシステムなど

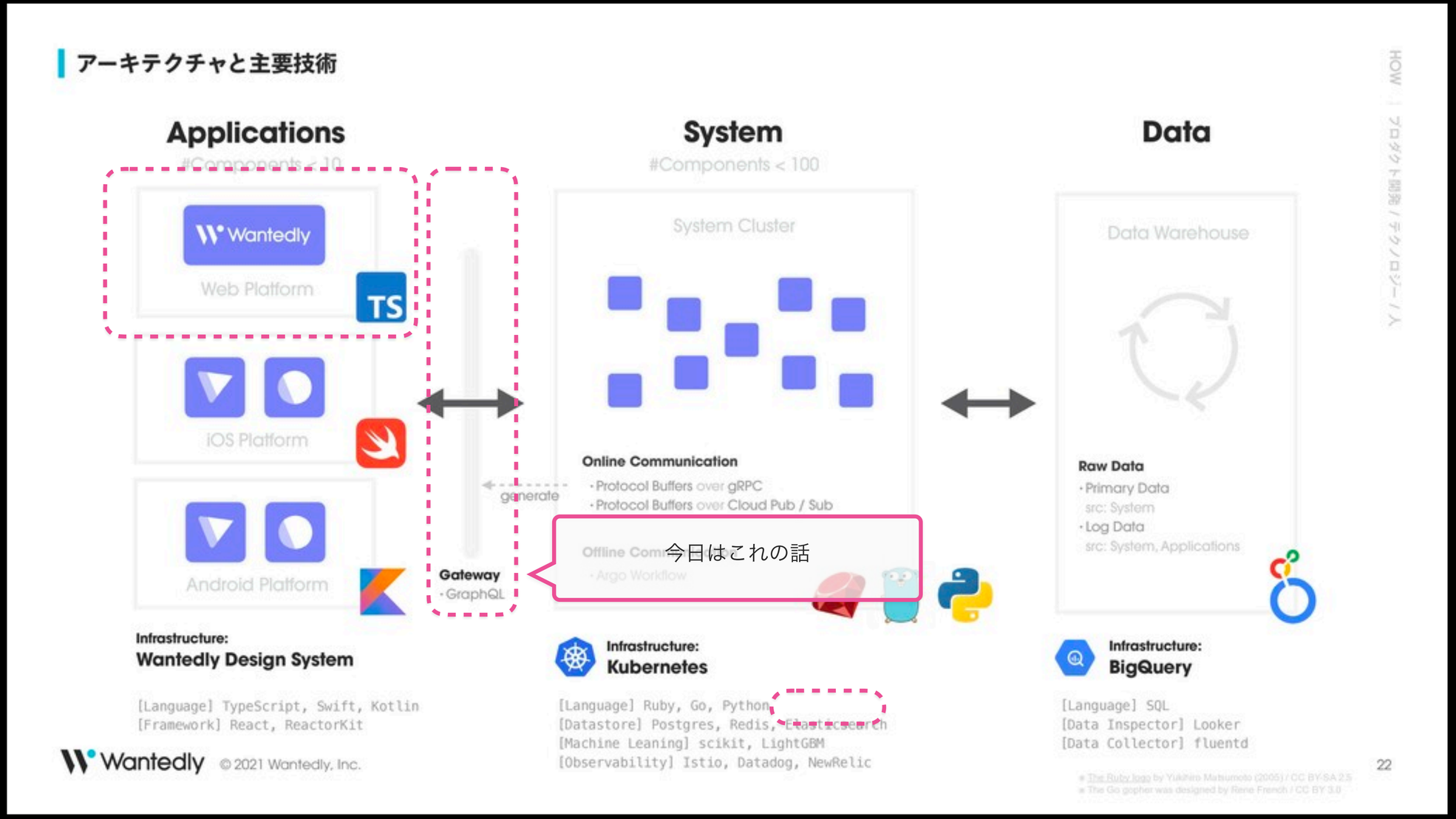
個別アプリのアーキテクチャ



JavaScript and Wantedly



JavaScript and Wantedly



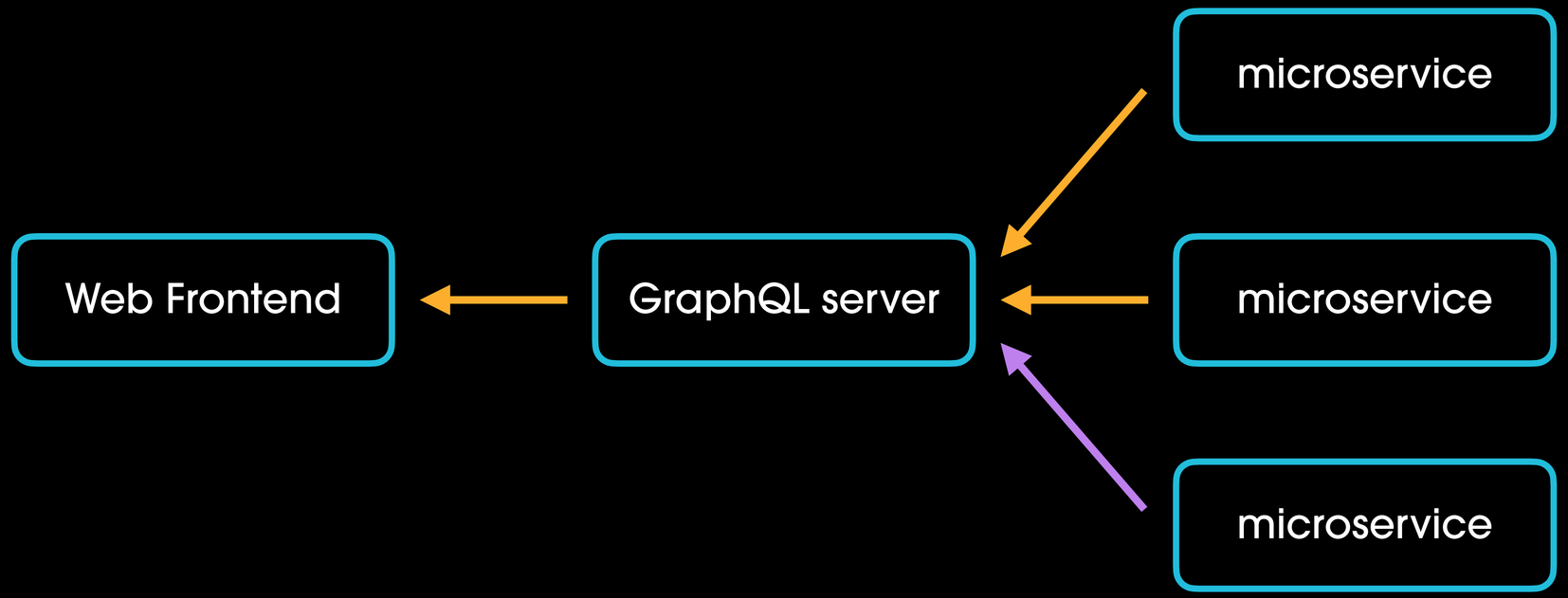
最初の GraphQL 導入

- 2019年はじめごろで、Wantedly Visit の企業側画面に導入したのが最初

- Web Frontend における課題を解決するために導入
  - Web Frontend で型付きのスキーマで、生産性高く開発したい
  - 複数マイクロサービスからのデータを集約し、フロントエンドで使いやすい形に変換したい

- いわゆる “BFF” 的な使われ方
  - フロントエンド寄りの開発者がスキーマ設計・実装をおこなう
  - （モバイルで使ってなかったのは、単純に検証できてなかったのもある）

GraphQL and Wantedly



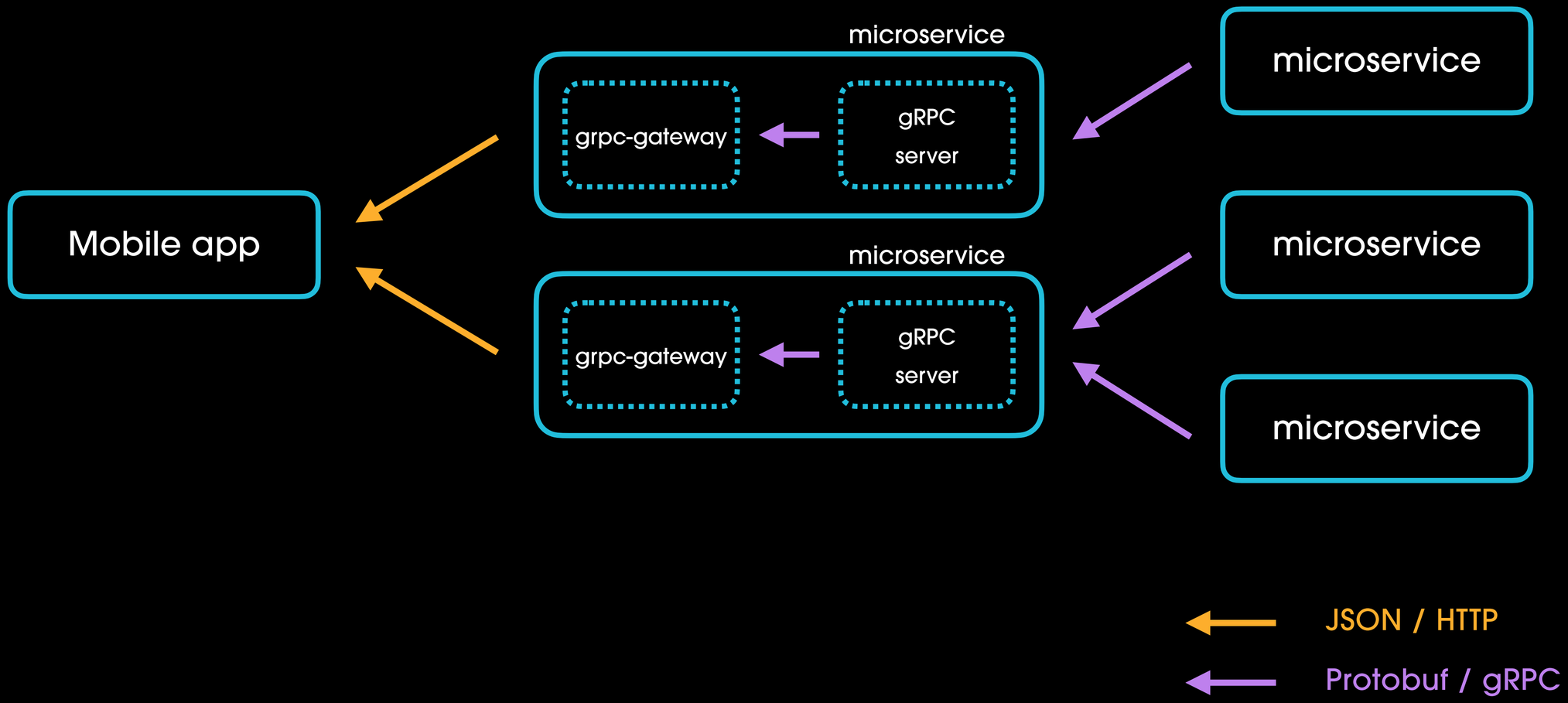
Wantedly

振り返り - 最初の GraphQL 導入

- Web Frontend における生産性は確かに向上した
  - スキーマから正しい型が生成されて便利
  - 状態管理も Apollo Client に任せることができた

- 一方で、アーキテクチャ全体として最適な形になったかは別
  - Wantedly People(mobile app), Profile 側のチームが BFF を利用するときに問題が顕在化した

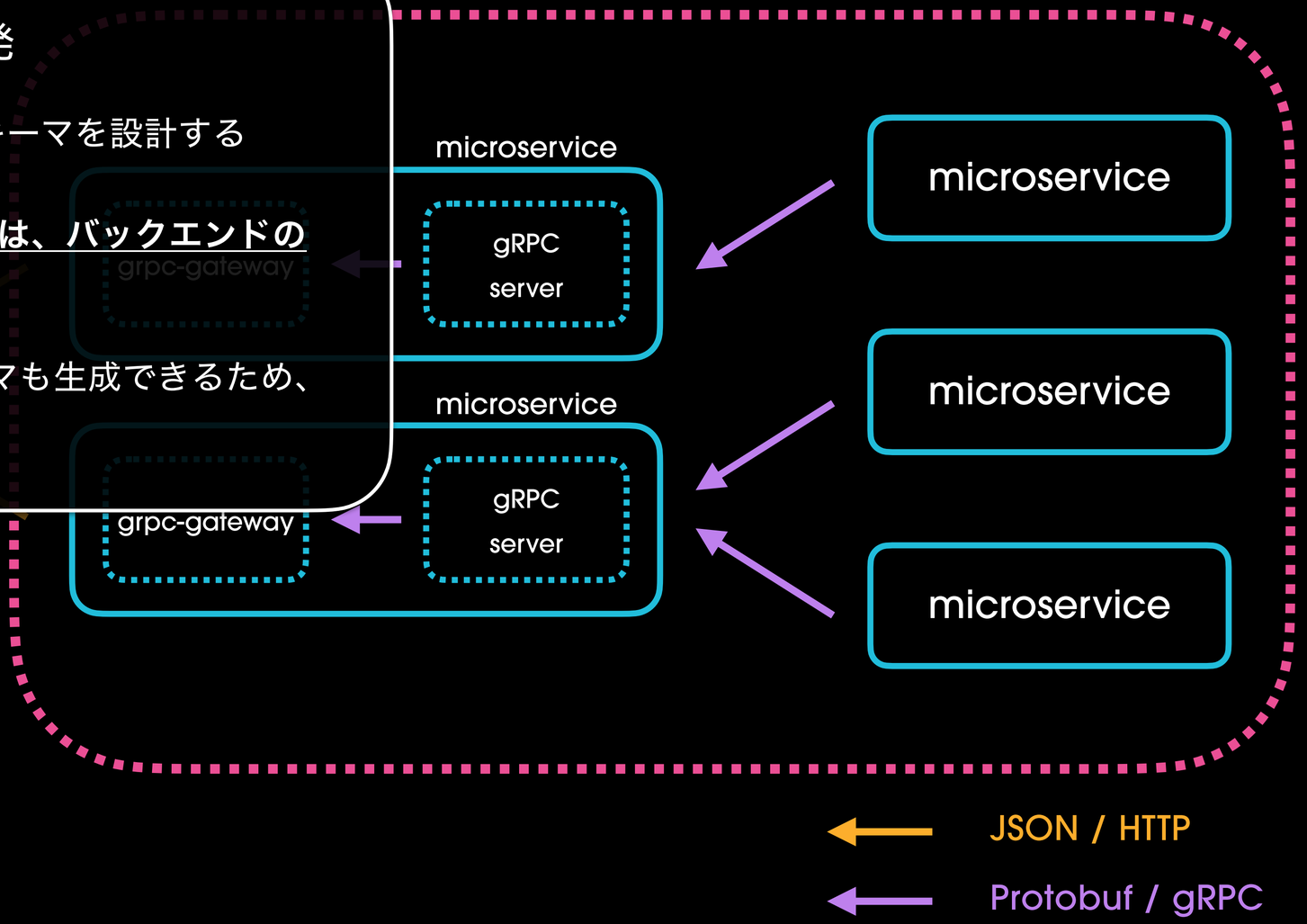
Wantedly People(mobile app) Architecture



Wantedly People(mobile app) Architecture

Protobuf によるスキーマ駆動開発

- バックエンドとモバイルが協働してスキーマを設計する
  - 主導はバックエンドになることも多い
  - **マイクロサービス・アーキテクチャでは、バックエンドのシステム間通信もスキーマが重要**
- Protobuf IDL から OpenAPI のスキーマも生成できるため、モバイル側でもスキーマを活用可能



Wantedly, Inc.

フォロー中 https://wantedlyinc.com 東京都

ホーム 私たちについて メンバー ストーリー 募集

Protocol Buffers によるプロダクト開発のススメ - API 開発の今昔 -

竹野 創平
開発チーム アーキテクト

on 2021/02/12

26 6 B! 110

こんにちは、Wantedly People アプリの開発をしている竹野（Altech）です。今回は、Protocol Buffers についての記事になります。

Wantedly People では、2018年に Protocol Buffers （以下、Protobuf と呼ぶ）がとあるマイクロサービスに入って以降、何度か大規模に Protobuf を使った開発をしてきました。またその経験を通じて、Protobuf には単に「型がついて嬉しい」というだけではないパラダイム的な変化があることが分かってきました。

その知見を全社に展開するため、去年「Protobuf によるプロダクト開発速習会」という会を行いました。この記事の内容は、そこで話したことの前半「Protobuf を使うと開発がどう変わるのか？」になります。

なお、Protobuf にはバイナリフォーマットとしての役割とインターフェイス定義言語としての役割がありますが、ここでは後者にフォーカスして取り上げています。したがって gRPC gateway などを通して実際には HTTP を話す場合にもそのまま適用できる内容となっています。

Protobuf

- バックエ
  - 主導は
  - マイクロ
    システム

- Protobu
  モバイル

microservice

microservice

microservice

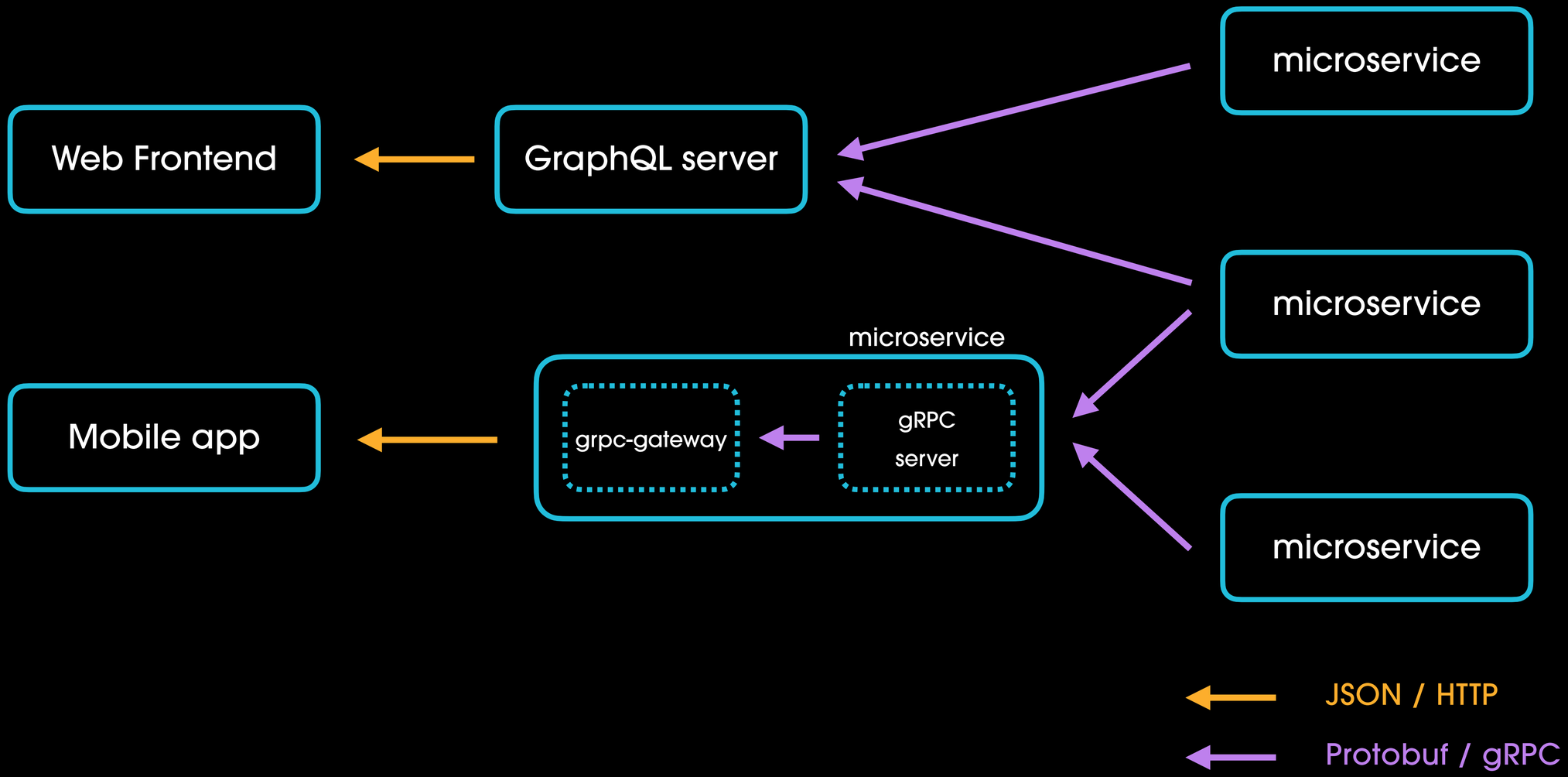
JSON / HTTP

Protobuf / gRPC

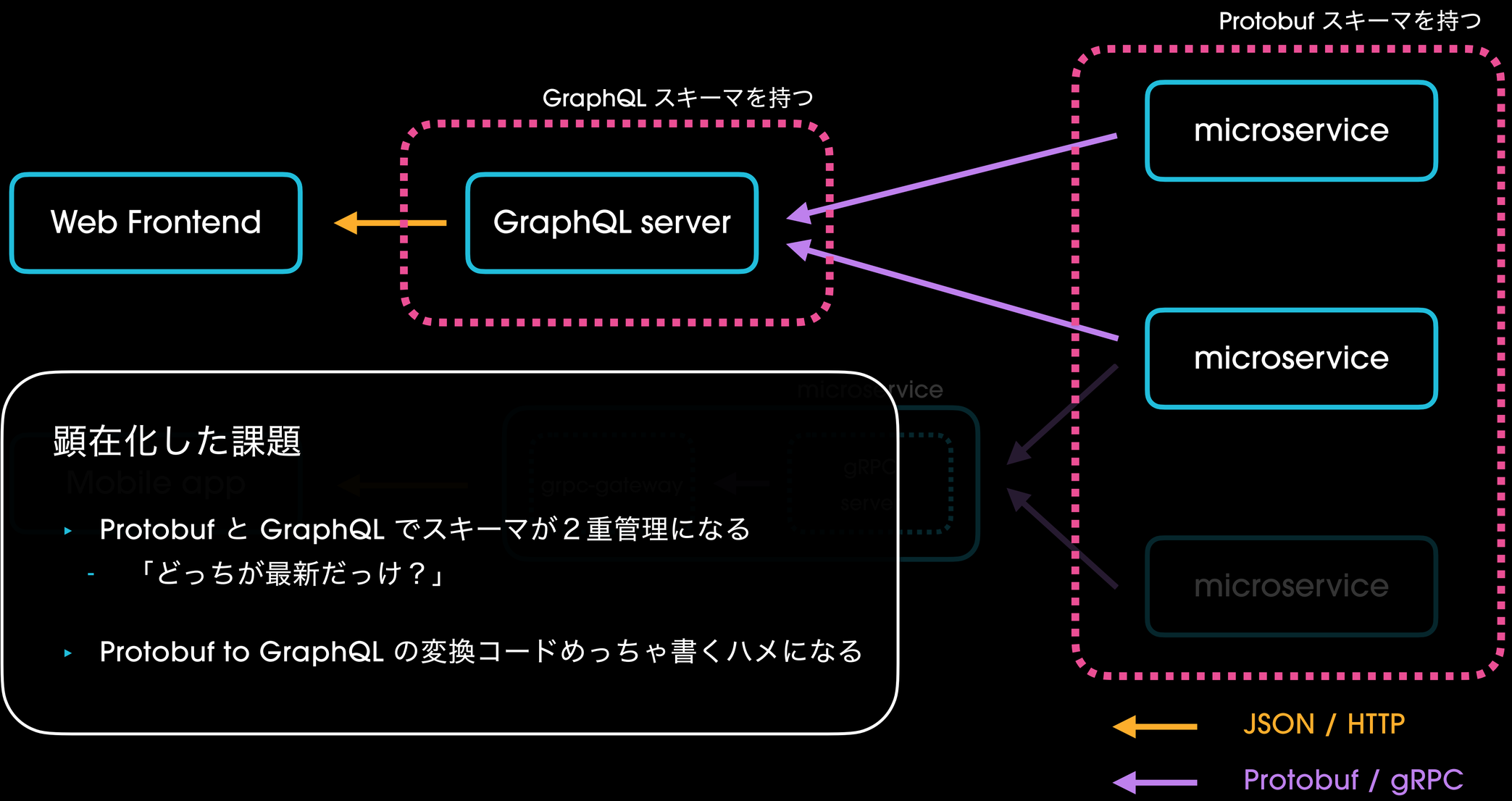
https://www.wantedly.com/companies/wantedly/post_articles/309513

Wantedly

Wantedly People(mobile app) Architecture



Multi-application and Microservices and GraphQL

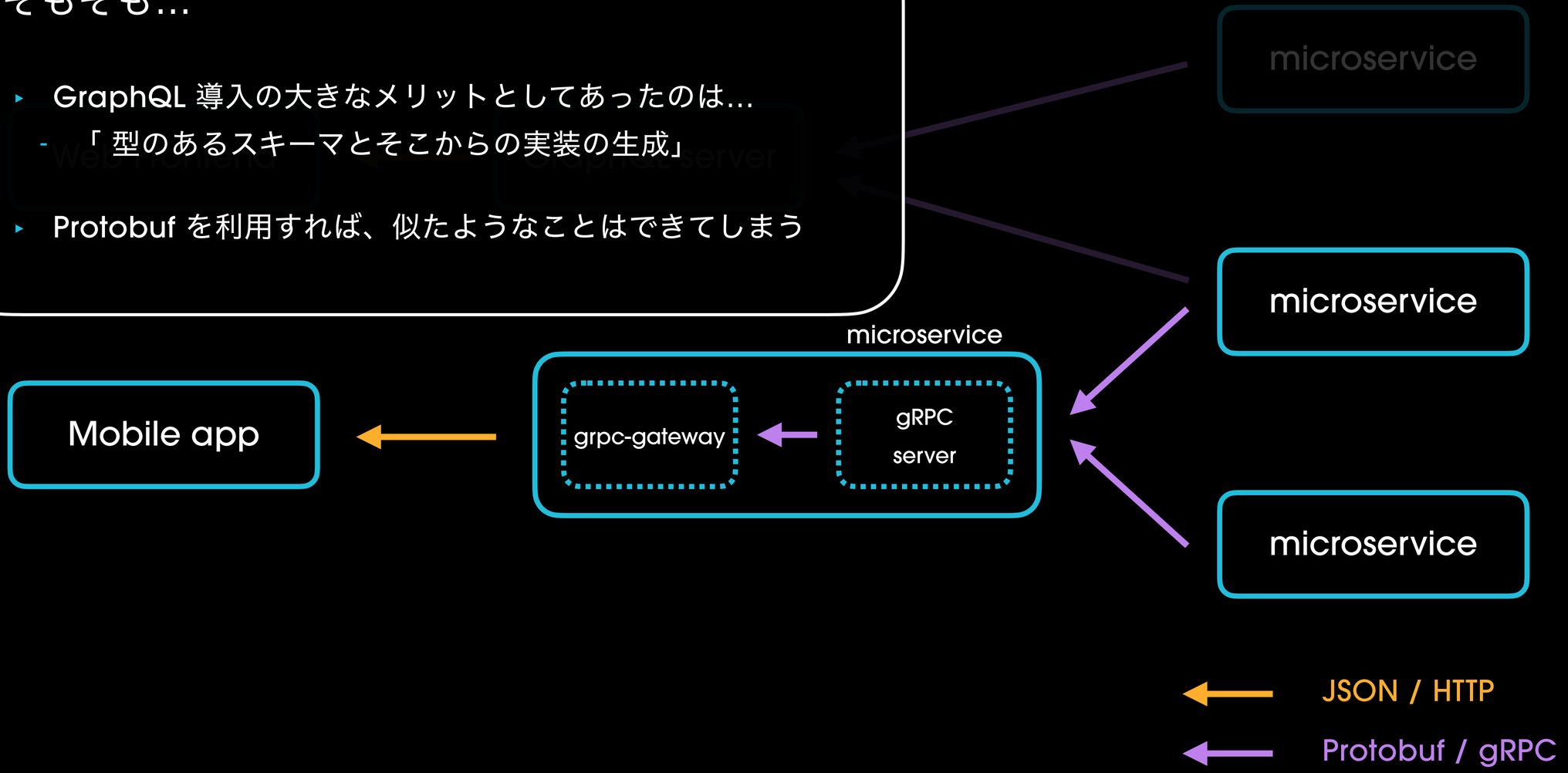


顕在化した課題

- Protobuf と GraphQL でスキーマが２重管理になる
  - 「どっちが最新だっけ？」
- Protobuf to GraphQL の変換コードめっちゃ書くハメになる

そもそも…

- GraphQL 導入の大きなメリットとしてあったのは…
  - 「型のあるスキーマとそこからの実装の生成」
- Protobuf を利用すれば、似たようなことはできてしまう



GraphQL を入れる目的を再考する

- GraphQL のメリット再考
  - 型のあるスキーマと、そこからのコード生成
  - フィールド・関連オブジェクトを選択的に・柔軟に取得できるインタフェース
  - データ形式・通信方式ともに一般的なフォーマット（JSON over HTTP）なので、どこでも動く

- Wantedly で守りたいところ
  - バックエンドのシステム間通信でもスキーマ駆動開発したい（既存の資産を使いたいので、gRPC 使いたい）
  - 1つのスキーマを絶対的なソースとして扱いたい

Reconsider “GraphQL”

Protobuf IDL で記述されたスキーマにはドメイン知識が反映されている
ドキュメントが自然言語でも記述されている
wantedly/apis というリポジトリで中央管理されており、
そこから各言語実装が自動生成される

<u>このリポジトリにあらゆる知識が集積され、プロダクト開発を加速させている</u>
（詳しくは「Protocol Buffers によるプロダクト開発のススメ - API 開発の今昔 - | Wantedly Engineer Blog」を参照）

（...ので、スキーマ２重管理により信頼性が既存されるのは困る）

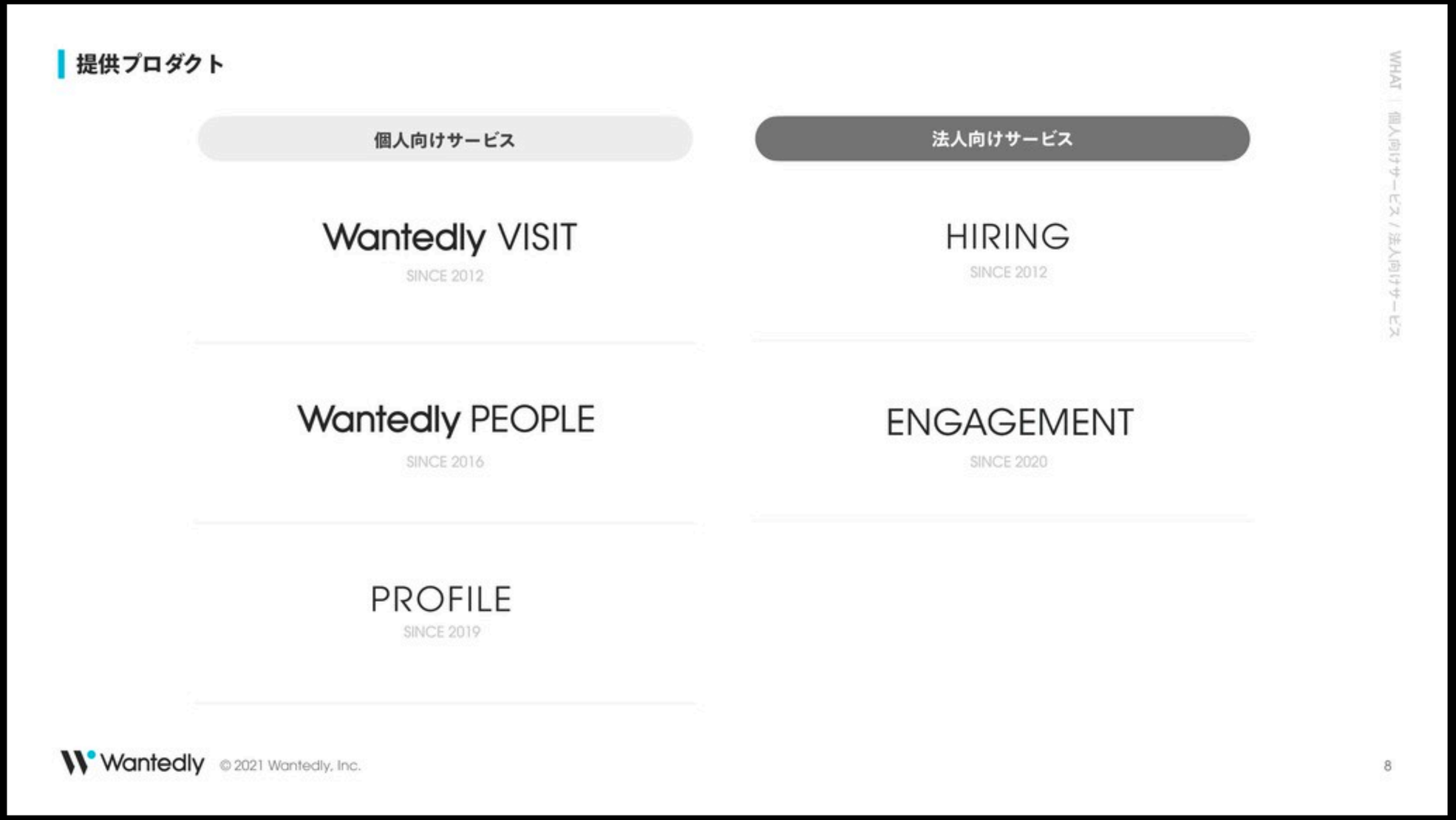
```
// Wantedly ユーザを表すオブジェクト。
// ID 以外の値を直接保持することはなく、他ドメインオブジェクトへの関連のみを持つ。
// ユーザドメインに属するオブジェクトは「他のユーザに見せるか」「当人が変更可能な情報か」の2軸で分類される。
//
// |          | 変更できない      | 変更できる    |
// | -------- | ---------------- | ------------ |
// | 見せる    | `UserSystemInfo `| `Profile`    |
// | 見せない | `UserSystemInfo `| `Preference` |
//
// GraphQL 上ではユーザ以外のドメインへの関連を持つこともある。
// たとえば「共通のつながり（つながりドメイン）」や「ブックマークした募集（募集ドメイン）」など。
message User {
  // Required. Immutable. ユーザID
  uint64 id = 1;
  // Required. Output only.
  Profile profile = 2 [(graphql.field).skip_resolver = true];
  // Optional. Output only.
  UserPreference preference = 3 [(graphql.field).skip_resolver = true];
  // Required. Output only.
  UserSystemInfo system_info = 4 [(graphql.field).skip_resolver = true];
}

// ユーザが所有しているユーザ個人の情報と、それに付随する設定。
//
// 責務は以下の通り:
//
// * ユーザー個人の情報
//     * ユーザーが所有している
//     * ユーザーが所有している個人情報なのでユーザーが公開範囲を制御できる
//     * 制御するための設定がある
// * ユーザーが所有しているドメインオブジェクトとその配下
//
// 内部的にはメイン DB の `profiles`, `user_names` テーブルなどに相当。
// 未登録のユーザや名刺 virtual_user であっても名前や avatar は割り当てられているため、すべてのユーザに存在する。
message Profile {
  // Required. Immutable. ユーザID
  uint64 user_id = 1;
  // Required. Profile URL v2 https://www.wantedly.com/id/sohei_takeno の`sohei_takeno`の部分。
  // 登録ユーザーでない場合は空文字が入る。
  string slug = 2;
  // Required. Output only. 名前（現地語表記）
  string name = 3;
  // Required. Output only. 名前（英語表記）
  string name_en = 4;

  // Required. Output only. アバター画像
  // ユーザ設定値がない場合、プレースホルダ画像が返る
  string avatar_url = 11;
  // Optional. アバター画像（ユーザ設定値）
  google.protobuf.StringValue raw_avatar_url = 12;

  // Required. Output only. カバー画像
  // ユーザ設定値がない場合、プレースホルダ画像が返る
  string cover_image_url = 13;
  // Optional. カバー画像（ユーザ設定値）
  google.protobuf.StringValue raw_cover_image_url = 14;

  message NameParts {
    // Required. 名前
    string first_name = 1;
    // Required. ミドルネーム
```

GraphQL を入れる目的を再考する

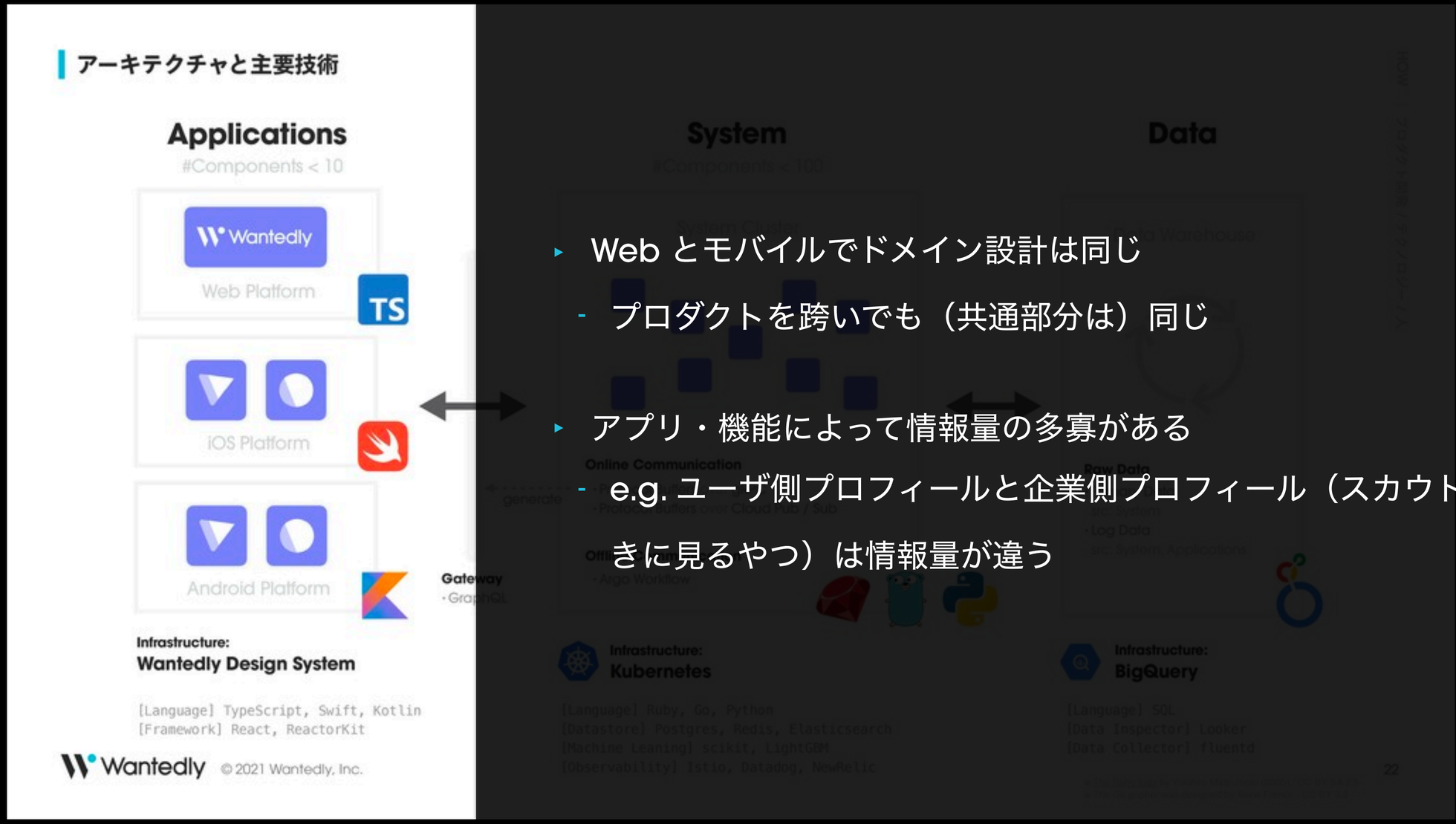
- GraphQL のメリット再考
  - 型のあるスキーマと、そこからのコード生成
  - フィールド・関連オブジェクトを選択的に・柔軟に取得できるインタフェース
  - データ形式・通信方式ともに一般的なフォーマット（JSON over HTTP）なので、どこでも動く

- Wantedly で守りたいところ
  - バックエンドのシステム間通信でもスキーマ駆動開発したい（既存の資産を使いたいので、gRPC 使いたい）
  - 1つのスキーマを絶対的なソースとして扱いたい

Wantedly のプロダクト性質



Wantedly のプロダクト性質



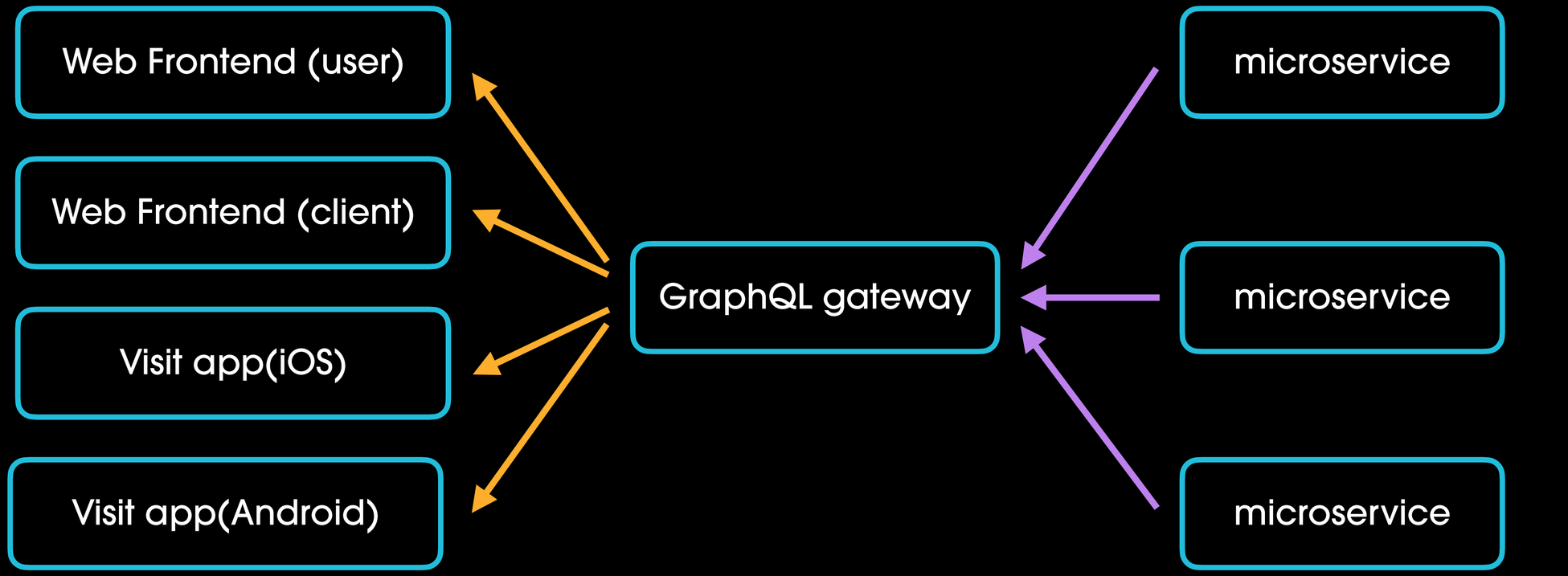
- Web とモバイルでドメイン設計は同じ
  - プロダクトを跨いでも（共通部分は）同じ
- アプリ・機能によって情報量の多寡がある
  - e.g. ユーザ側プロフィールと企業側プロフィール（スカウト送るときに見るやつ）は情報量が違う

GraphQL を入れる目的を再考する

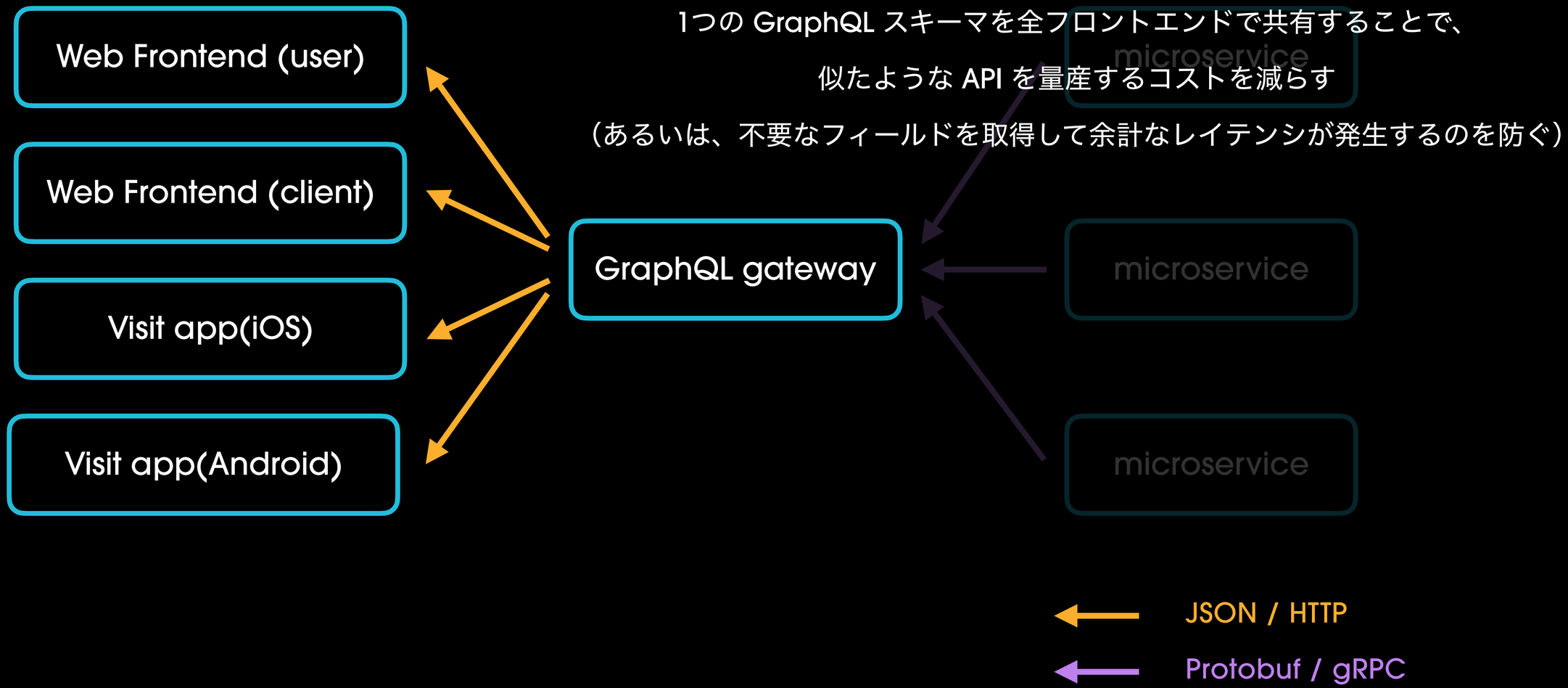
- GraphQL のメリット再考
  - 型のあるスキーマと、そこからのコード生成
  - フィールド・関連オブジェクトを選択的に・柔軟に取得できるインタフェース
  - データ形式・通信方式ともに一般的なフォーマット（JSON over HTTP）なので、どこでも動く

- Wantedly で守りたいところ
  - バックエンドのシステム間通信でもスキーマ駆動開発したい（既存の資産を使いたいので、gRPC 使いたい）
  - 1つのスキーマを絶対的なソースとして扱いたい

GraphQL Gateway 構想



GraphQL Gateway 構想



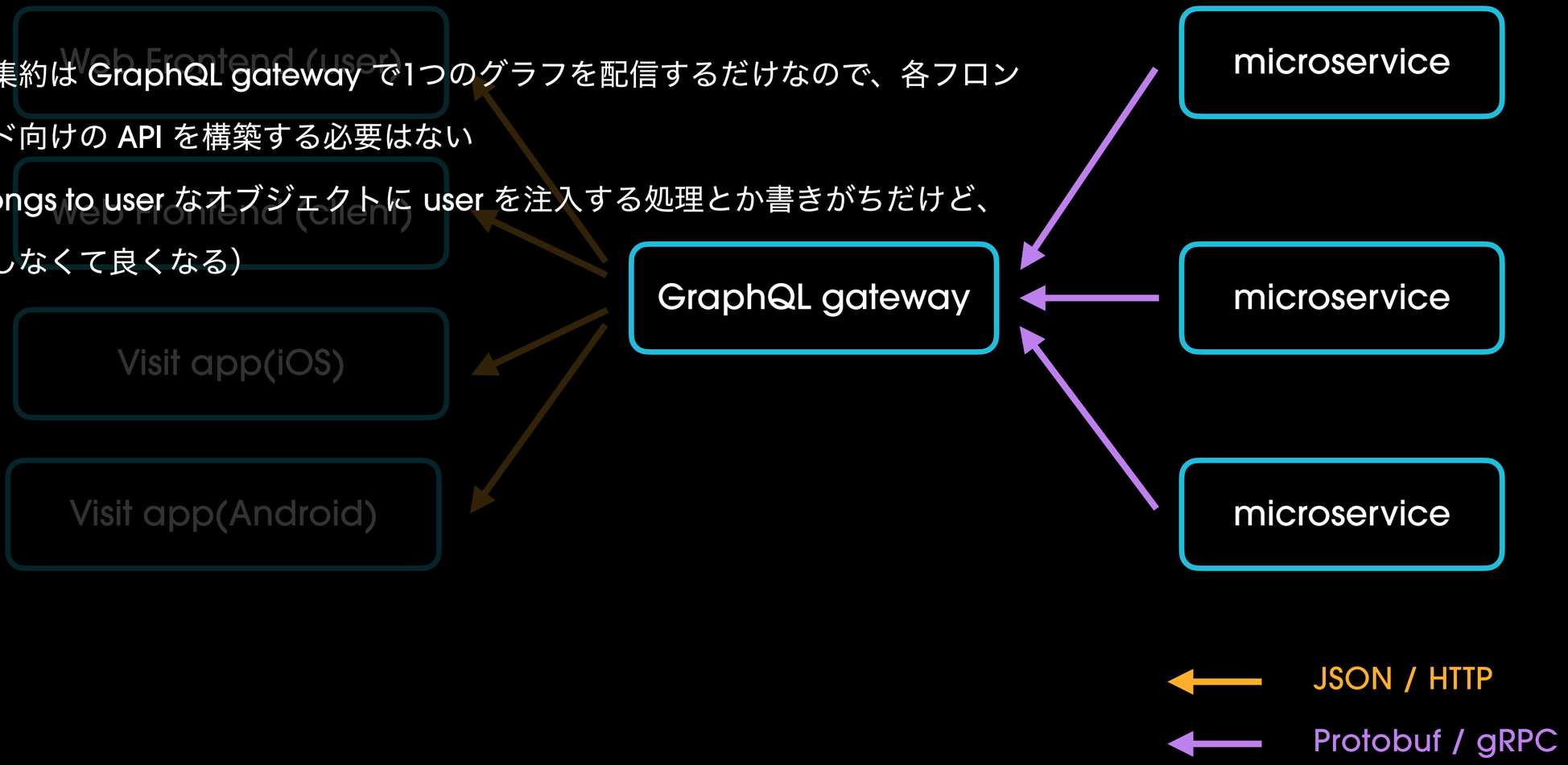
GraphQL Gateway 構想

ドメインロジックはバックエンドに実装される

（複数のフロントエンドで同じロジックを実装するのを防ぐ）

情報の集約は GraphQL gateway で1つのグラフを配信するだけなので、各フロントエンド向けの API を構築する必要はない

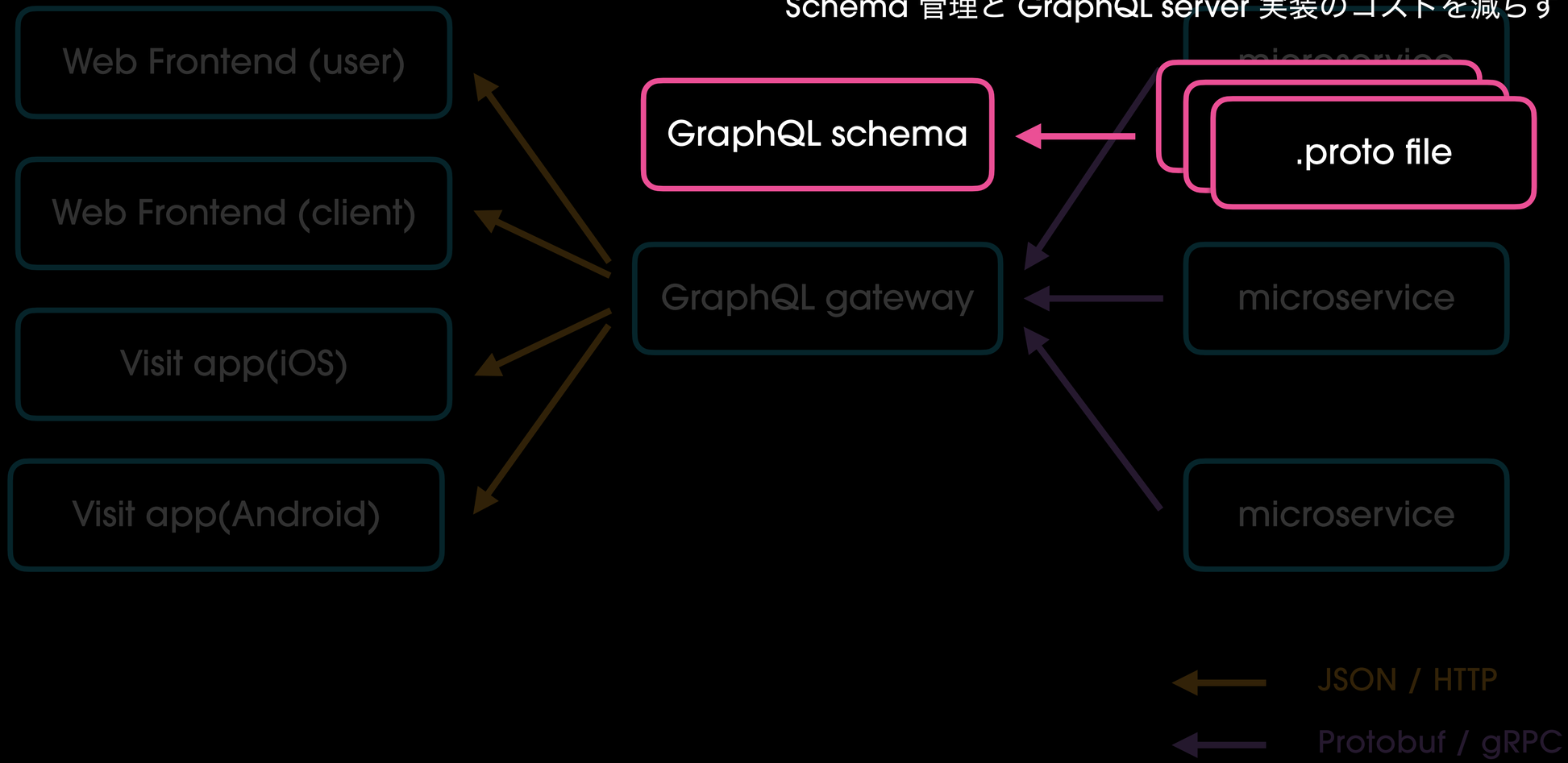
（belongs to user なオブジェクトに user を注入する処理とか書きがちだけど、それをしなくて良くなる）



GraphQL Gateway 構想

.proto から GraphQL schema と実装を生成するツールを自作し、

Schema 管理と GraphQL server 実装のコストを減らす



graphql-gateway

- Protobuf にある情報はなるべく GraphQL に引き継ぐ
  - コメントももちろん残す（GraphiQL やクライアント側の VSCode 上でも見れて便利）
  - Nullability
    - (gRPC には non-null を宣言する仕組みは標準では存在しないが、コメントや extension にそれを記述しよう！という規約を導入している → https://google.aip.dev/203)
  - Abstraction: oneof -> union
- オブジェクトは Protobuf から生成し、オブジェクトとオブジェクトの接続は TypeScript で記述
- 将来的には開発のスケーラビリティ担保のために GraphQL Federation とかしたくなりそう
- とりあえず1箇所に入れてみて検証
  - アーキテクチャに大きな変更を入れるときは撤退可能にしておくのが大切
- Node.js (TypeScript + GraphQL Nexus)
  - Go や Rust も選択肢としてはあったが、リファレンス実装である graphql-js を使うことを優先
    - あとで変えたくなっても、GraphQL のスキーマが守れていればなんとかなる
  - Apollo Server などはとりあえず使わない（必要になりそうなタイミングで考える）
  - Envelop は使いたい https://www.envelop.dev/
- Resolver 内で gRPC を叩く
  - 取得系は基本的に DataLoader を経由し、いわゆる N+1 問題を防ぐ
  - バックエンドが FieldMask pattern に対応している場合、GraphQL クエリから賢く FieldMask を組み立てる

graphql-gateway

- Protobuf にある情報はなるべく GraphQL に引き継ぐ
  - コメントももちろん残す（GraphiQL やクライアント側の VSCode 上でも見れて便利）
  - Nullability
    - (gRPC には non-null を宣言する仕組みは標準では存在しないが、コメントや extension にそれを記述しよう！という規約を導入している → https://google.aip.dev/203)
  - Abstraction: oneof -> union
- オブジェクトは Protobuf から生成し、オブジェクトとオブジェクトの接続は TypeScript で記述
- 将来的には開発のスケールを担保するために GraphQL Federation したかったが断念
- とりあえず1箇所に入れてみて検証
  - アーキテクチャに大きな変更を入れるときは撤退可能にしておくのが大切
- Node.js (TypeScript + GraphQL Nexus)
  - Go や Rust も選択肢としてはあったが、リファレンス実装である graphql-js を使うことを優先
    - あとで変えたくなっても、GraphQL のスキーマが守れていればなんとかなる
  - Apollo Server などはとりあえず使わない（必要になりそうなタイミングで考える）
  - Envelop は使いたい https://www.envelop.dev/
- Resolver 内で gRPC を叩く
  - 取得系は基本的に DataLoader を経由し、いわゆる N+1 問題を防ぐ
  - バックエンドが FieldMask pattern に対応している場合、GraphQL クエリから賢く FieldMask を組み立てる

~~自慢したい~~話したいことは無限にあるが時間が足りない

~~バトルしたい~~話を聞きたい・議論したいことがある人は話を聞きに来てください！

（Spatial Chat にもいます）

https://www.wantedly.com/projects/748998

まとめ・まなび

- 技術の導入は、何の問題を解決するのが目的なのかを意識しよう
  - 「悪くなった！やめる！」ではなく、結局「何を解決したくて」「何とコンフリクトしているのか」を考える
- とはいえ撤退する勇気も大事
  - 大きな技術導入は、撤退しやすいように小さくすすめるのもポイント
  - 実環境で動かさないとわからないこともある
- GraphQL (というか API 技術全般）について色んな人と議論したいです！ぜひ来てね！