GraphQL を利用したアーキテクチャの勘所

@qsona

2021-04-21 iCARE Dev Meetup #20

whoami

- @qsona
- Web Engineer at Quipper Ltd
- Microservices / Rails / Node.js / GraphQL

※ 本資料に登場するリンクは Speaker Deck の概要欄に
すべて載せておきます

モチベーション (1)

- GraphQL は比較的新しい技術で、最近より注目されている
- 大規模な開発に取り入れたり、既存のプロダクトに後から導入される事例なども増えてきている
- GraphQL を利用した設計・アーキテクチャの重要性が増している

モチベーション (2)

- 既存の設計ノウハウをうまく取り入れたい
- 逆に、GraphQL と取り合わせの悪い設計パターンは
  なるべく使いたくない
- GraphQL の設計とうまく付き合うために
  個人的に考えている整理手法やパターンについて話したい

Agenda

- 1. GraphQL とアーキテクチャ
  - Better Web API としての GraphQL
  - GraphQL as a Service (Hasura / AppSync)
- 2. GraphQL 注意すべきパターン集

1. GraphQL とアーキテクチャ

GraphQL の特徴

- Web API として提供する GraphQL Schema
- 欲しいデータを取得する Query
- 上の2つが分離されている

A query language for your API

GraphQL is a query language for APIs and a runtime for fulfilling those queries with your existing data. GraphQL provides a complete and understandable description of the data in your API, gives clients the power to ask for exactly what they need and nothing more, makes it easier to evolve APIs over time, and enables powerful developer tools.

Ask for what you need, get exactly that

Send a GraphQL query to your API and get exactly what you need, nothing more and nothing less. GraphQL queries always return predictable results. Apps using GraphQL are fast and stable because they control the data they get, not the server.

Query と Schema

```
{
  hero {
    name
    friends {
      name
      homeWorld {
        name
        climate
      }
      species {
        name
        lifespan
        origin {
          name
        }
      }
    }
  }
}
```

```
type Query {
  hero: Character
}

type Character {
  name: String
  friends: [Character]
  homeWorld: Planet
  species: Species
}

type Planet {
  name: String
  climate: String
}

type Species {
  name: String
  lifespan: Int
  origin: Planet
}
```

Describe what's possible with a type system

GraphQL APIs are organized in terms of types and fields, not endpoints. Access the full capabilities of your data from a single endpoint. GraphQL uses types to ensure Apps only ask for what's possible and provide clear and helpful errors. Apps can use types to avoid writing manual parsing code.

アーキテクチャ的側面から見る

- GraphQL についてアーキテクチャ的な考察をしていくため、
  これらに対応する概念を既存の語彙で割り当ててみると...

- query <=> Usecase
- Schema, Types <=> Resource
- この考え方をベースに、既存のアーキテクチャとの組み合わせなどの考察をしていきたい

※ Usecaseとは? Resourceとは?

類似の概念

(似たような方法で大きく二分している例)

(GraphQL Query)	(GraphQL Schema and Types)
Usecase	Resource
Presentation	Domain
System of Engagement	System of Resource
Frontend	Backend

a) Better Web API としての GraphQL 設計手法

Web API の設計手法

- 大きく2通りあると考えている
  - 1. Usecase ベース
  - 2. Resource ベース

Usecase-based Web API

- クライアントのユースケースに合わせて Web API を作る
  - ユースケース ≒ 画面

Usecase-based Web API の問題点

- 似たようなAPIが乱立しやすい
  - 毎度API作るのが面倒 =>
    複数のユースケースに対応する **神API** ができる傾向
- Usecase は変わりやすいが、それに応じて毎度APIを修正する必要がある
  - 後方互換性を保つために、修正ではなく新規API追加が必要になることも多い

Resource-based Web API

- アプリケーションそのものの仕様に合わせて Web API を作る
  - (基本的には API 提供側の都合で作る)
- 例: (注意深く設計された) RESTful API

Resource-based Web API

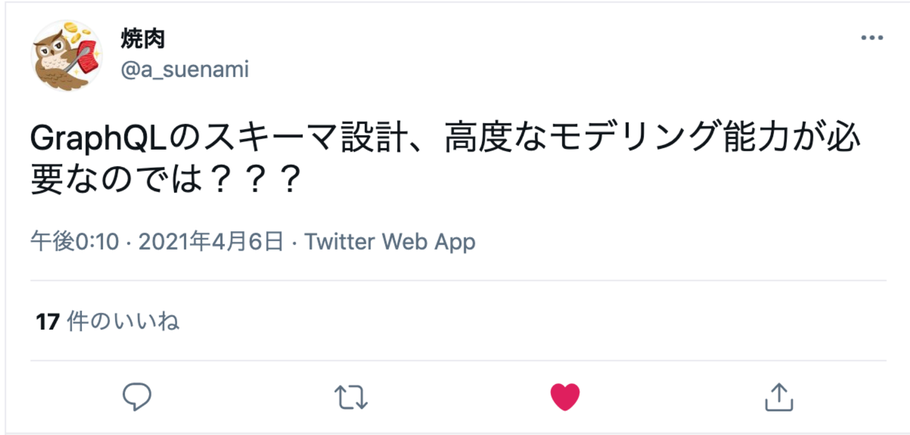
- 問題点: クライアントのユースケースに合わない時がある
  - 1画面で複数のAPIリクエストが必要になったり
  - 逆に1リクエスト中で不要なフィールドも含めて取ってしまうことがある
  - (=> パフォーマンスの悪化)
  - Resource が細かくなりすぎて (テーブル単位など)
    クライアントから使いにくいことがある

GraphQL Web API 設計の勘所

- 基本的に Resource-based API として作る (GraphQL の良さを活かすために)
- GraphQL Schema / Types が Resource を表す
- query が Usecase を表す
- Query Type の field が endpoint である
- つまり、Resource ベースのAPIと Usecase ベースのAPIの
  良いところを両立できる!

GraphQL Web API 設計の勘所

- 注意深く Resource (≒ GraphQL Schema, Types) を定義していく必要がある
  - 常に高いレベルのモデリング力が求められる
  - Type の粒度を整える (テーブルと1:1とは限らない)
- (注意深く) RESTful API を作るときと、基本的には変わらない



注意深い Resource 設計

(※ 個人的なプラクティスです)

- まずは Usecase をきちんと理解する
- 同時に、データ設計をイメージする
- 以下を満たすスキーマを考える
  - ユースケースを自然に満たせる (必要以上に複雑な query や client 実装にならない)
  - データ設計から自然に resolvers の実装ができる
  - ビジネス的な共通理解・共通言語を正しく反映している

GraphQL Web API 実装の悩み所 (概要)

- (RESTful API に比べて) クエリの自由度が高すぎることにより
  パフォーマンス低下を引き起こす可能性がある
  - 典型的には N+1 問題
- 対処法
  - Dataloader Pattern
  - ユースケースの管理と妥協

例) 一覧画面と詳細画面

- 一覧画面 (各項目の情報量: 少)
- => 詳細画面 (情報量: 多)

RESTful API の場合

よくある設計手法

- 一覧画面: GET /posts
  - 配列を返す
  - 1件ずつの情報量は少ない
- 詳細画面: GET /posts/:id
  - 情報量を多くする

```
// GET /posts
[
  { "id": 1, "title": "投稿1" },
  { "id": 2, "title": "投稿2" }
]

// GET /posts/1
{
  "id": 1,
  "title": "投稿1",
  "body": "...長文です",
  "author": {
    "id": 1,
    "name": "qsona"
  }
}
```

GraphQL API の場合

- Schema は 1つで両方に対応する
  - Usecase を基本的に意識しない
- 一覧画面、詳細画面それぞれで
  必要なデータを取得する query を書く

GraphQL API の場合

```
type Query {
  posts: [Post!]!
  post(id: ID!): Post
}

type Post {
  id: ID!
  title: String!
  body: String!
  author: User!
}

type User {
  id: ID!
  name: String!
}
```

```
# 一覧画面
query PostList {
  posts {
    id
    title
  }
}

# 詳細画面
query Post($id: ID!) {
  post(id: $id) {
    id
    title
    body
    author {
      id
      name
    }
  }
}
```

GraphQL API 特有の課題

- posts (配列) を、中身の詳細まで取得するクエリも書けてしまう
- サーバー側で適切に対応していない場合、N+1 問題が起きたりする

```
query PostList {
  posts {
    id
    title
    body
    author {
      id
      name
    }
  }
}
```

Dataloader パターン

- 典型的には Dataloader パターンを適用すべき
  - resolver を遅延評価し、バッチで実行する仕組み
- が、そう簡単にいかない場合もある
  - 例: author ではなく authors (複数) で、
    ソースとなるテーブルが複数あり、さらにソートが必要...
    のような複雑なケースだと Dataloader を適用するのが難しい場合がある

ユースケースの管理と妥協

- 原理的に右のようなクエリが実行可能ということと、実際に投げられることがあるかどうかは別問題
- 実際に投げられることがないなら、問題も起きないので対処不要
- => そのようなユースケースが存在しない、ことが分かっていればよい
  - 出てきたら対処する or そもそも禁止する

```
query PostList {
  posts {
    id
    title
    body
    author {
      id
      name
    }
  }
}
```

Persisted Queries

(以下は運用方法の一例)

- クライアントが投げうるクエリを、事前に GraphQL API サーバーに登録する手法
- クライアントのビルド・デプロイ時に Pull Request を投げる
  - Backend エンジニアが確認し、問題なければマージ
  - N+1 問題不可避のようなクエリの場合、Frontend エンジニアと相談し、どうしても必要ならサーバー側で対応する
- 登録されたクエリ以外は、弾くようにしておく

Persisted Queries を利用するメリット

- ユースケースをサーバーサイドで管理できる
  - サーバーサイドで最適化すべき resolvers がどれかわかる
  - API の破壊的変更/廃止がスムーズに行える
- リクエストのペイロードを減らせる
  - Query ごとに id を振ったり hash 値を計算しておき、
    実際のリクエストでは Query そのものではなく id / hash値を送る
  - POST ではなく GET リクエストも使えるようになる => キャッシュ可能に

Better Web API としての GraphQL 設計手法 まとめ

- Resource-based API として設計する
- 一方、Usecase (query) のことも
  ある程度理解しながら設計するとよい
  - 使いやすい API になる
  - 最適化を後回しにできる

b) GraphQL as a Service

(Hasura / AppSync)

GraphQL as a Service

- もともと GraphQL は Web API を想定して作られた
- その後、Datastore に対して GraphQL としてクエリできる
  ライブラリ/製品が登場した
  - Graphcool (終了), Prisma 1 (新規開発なし), PostGraphile
- Web API を自分で実装しなくても、データストアを用意するだけで
  簡単に GraphQL Web API を提供できるサービスが登場した
  - Hasura, AppSync (これらを一旦勝手に **GraphQL as a Service** と呼ぶことにします)

もはや SQL と何が違うのか...?

Qiita キーワードを入力

73

34

@yancya が2019年02月05日に更新

SQLQL

Rails, PostgreSQL

この記事は最終更新日から1年以上が経過しています。

概要

- GraphQL で柔軟なリクエストができるようになれば、サーバー側の実装をせずとも新しい機能を効率よく作れて便利みたいな話を見かけた
- GraphQL では独自のクエリ言語でサーバーに対してデータを要求するが、クエリ言語と言えば SQL というものがあるので、どうせなら SQL をサーバーに投げつけるようにすれば、似たようなことが出来るのではないかと考えたのでやってみた

Takafumi ONAKA
@onk

GraphQL と SQL の比較、そもそも GraphQL と RESTful API の比較が終わった上で、「本当に新しい Query Language が必要なのか」という話なので (結論としては新たに存在した方が便利そうな感じ)

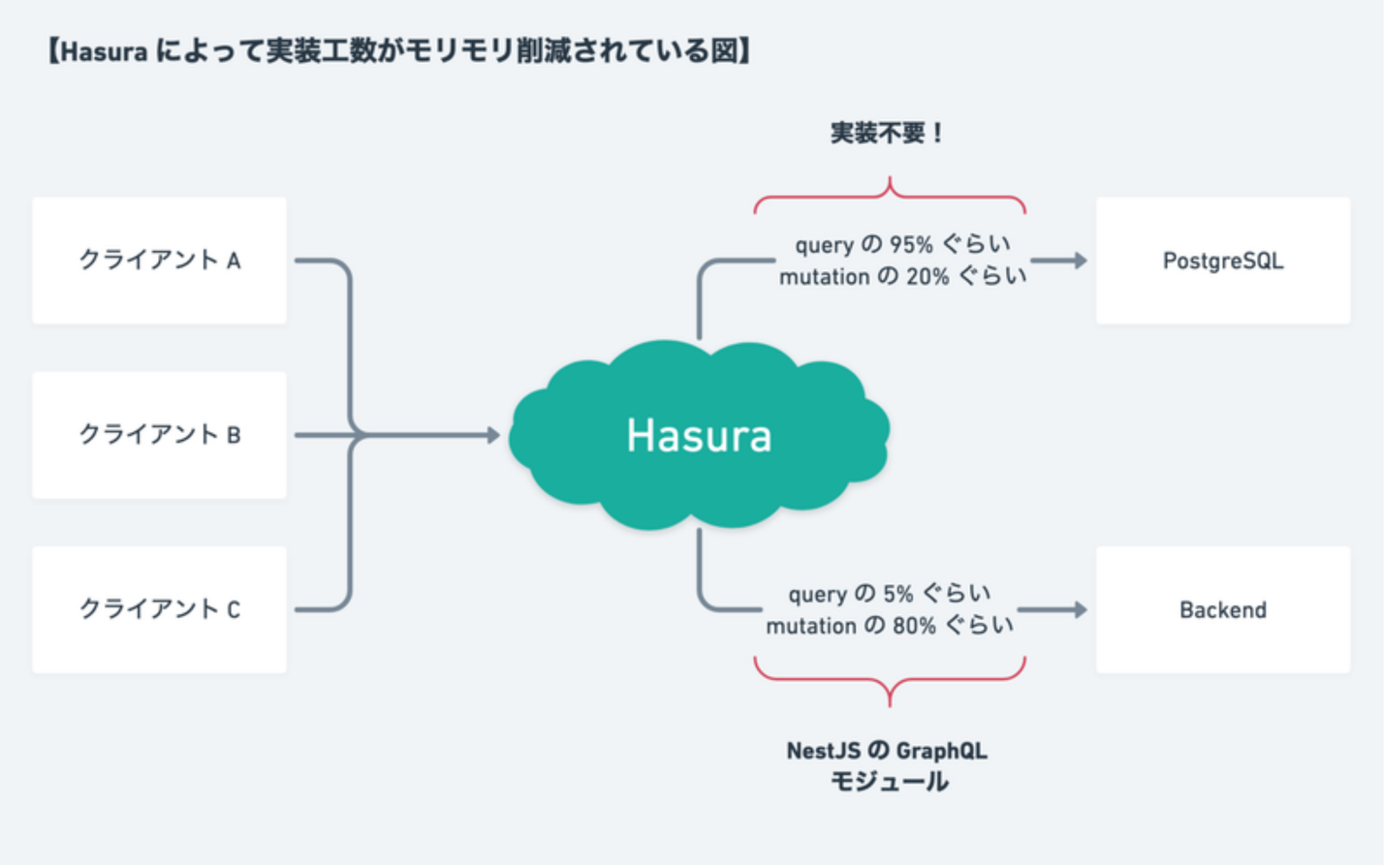
午前10:36 · 2017年9月26日 · Twitter Web Client

2 件のリツイート **13** 件のいいね

拡張性の高い GraphQL as a Service

- Resolvers として、データストア呼び出し以外も選択できる
  - AppSync: AWS Lambda, HTTP resolvers
  - Hasura: Remote Schema
- 特に Hasura の場合、最終的には単なる GraphQL Gateway として存在できる

事例: dinii 社さんの Hasura の使い方



- 少数 (5%) の Query と
  多数 (80%) の Mutation を
  別立てのGraphQL サーバーに流
  している

- 開発が複雑化するにつれ、この
  割合は増えていくのではないか
  (qsona の見解です)

引用元: 【エンジニアブログ】ダイニーのエンジニアリング3カ条｜dinii（ダイニー）公式｜note
https://note.com/dinii/n/n9be778bd7da3

Command - Query Separation

- Query のほとんどで Hasura を直接使えている、便利
- 一方で Mutation は基本的に特有のドメインロジックが必要になってくることが多い
- => CQS パターンに近い

初期実装コスト低 + 将来的な拡張◎

Hasura の良さそうなところ

- 初期は Hasura の API をそのまま使うことで、実装コストを下げられる
- 実装が複雑になってきたら Remote Schema を活用し、
  ロジックをバックエンドに寄せていく。
  将来的な拡張性も担保できる
- (個人的には Firebase よりも有望視していますが、どちらもまだ使ったことない...)

2. GraphQL 注意すべきパターン集

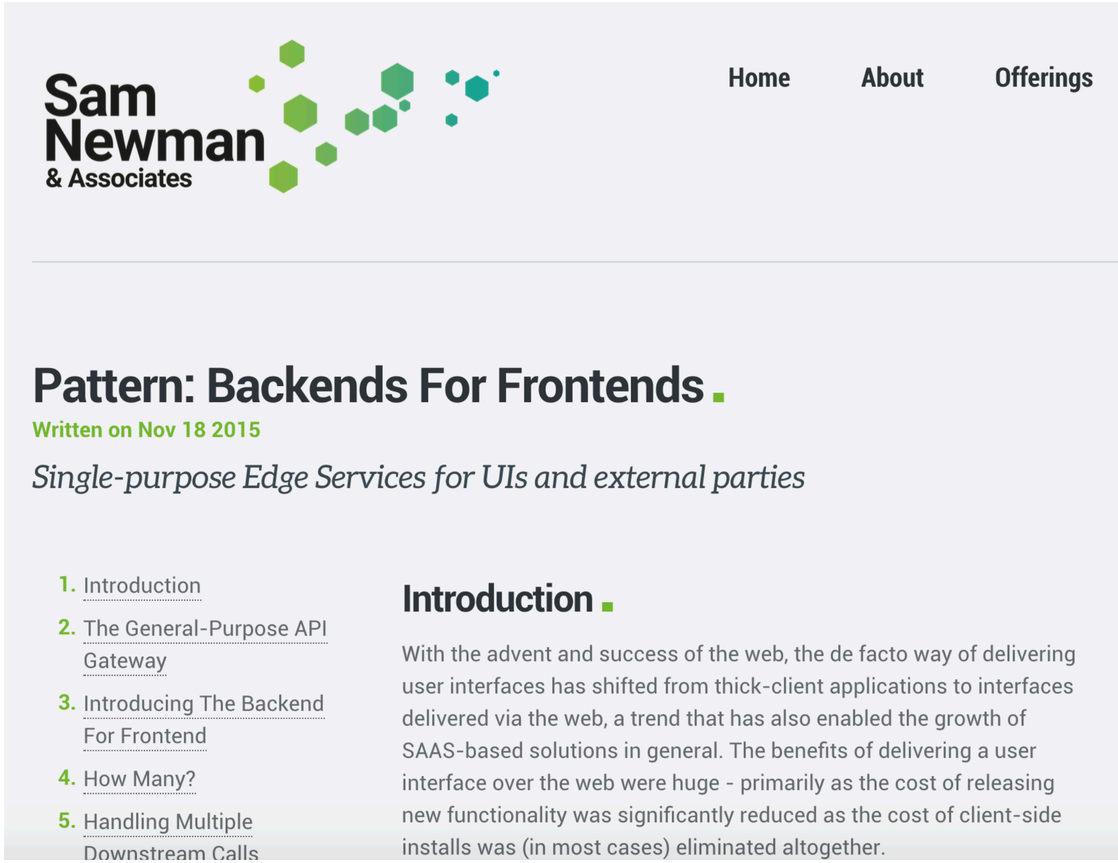
"アンチパターン" ではないが...

- これから紹介するパターンは「アンチパターン」ではない
- 状況によっては有用
- 特に、一定以上の大規模開発においては必要になりそうだが、
  初手から使うと GraphQL の良さを殺す可能性が高く、
  気をつけたほうがよいパターン

パターン1: GraphQL as BFF

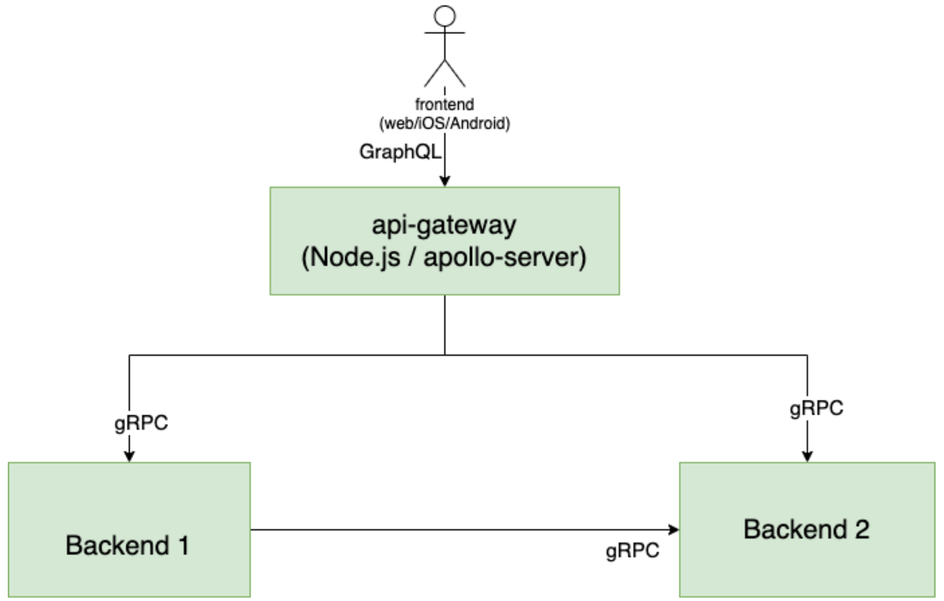
BFF (Backends for Frontends)

- Microservices 文脈でよく登場する
- Frontend のためにサーバーを立て、Backend Services を集約して Frontend 向けの API を提供するパターン
- Frontend エンジニアが開発・運用することが推奨される



GraphQL as BFF

- Backend APIs は典型的には REST や gRPC などで提供されているとする
- BFF として GraphQL API サーバーを作るパターン



TECHNOLOGY RADAR

Download Subscribe Search Build your Radar About

Techniques | Tools | Platforms | Languages & Frameworks

Techniques

GraphQL for server-side resource aggregation

Published: May 15, 2018 Last Updated: May 19, 2020

MAY 2020

TRIAL

We see more and more tools such as **Apollo Federation** that can aggregate multiple GraphQL endpoints into a single graph. However, we caution against misusing **GraphQL**, especially when turning it into a server-to-server protocol. Our practice is to use **GraphQL for server-side resource aggregation** only. When

NOT ON THE CURRENT EDITION

This blip is not on the current edition of the Radar. If it was on one of the last few editions it is likely that it is still relevant. If the blip is older it might no longer be relevant and our assessment might be different today.

GraphQL for server-side resource aggregation | Technology Radar | ThoughtWorks
https://www.thoughtworks.com/radar/techniques/graphql-for-server-side-resource-aggregation

TECHNOLOGY RADAR

Download Subscribe Search Build your Radar About

Techniques Tools Platforms Languages & Frameworks

Techniques

GraphQL for server-side resource aggregation

Published: May 15, 2018 Last Updated: May 19, 2020

MAY 2020

~~TRIAL~~ ? **HOLD** (qsona の見解です)

We see more and more tools such as **Apollo Federation** that can aggregate multiple GraphQL endpoints into a single graph. However, we caution against misusing **GraphQL**, especially when turning it into a server-to-server protocol. Our practice is to use **GraphQL for server-side resource aggregation** only. When

NOT ON THE CURRENT EDITION

This blip is not on the current edition of the Radar. If it was on one of the last few editions it is likely that it is still relevant. If the blip is older it might no longer be relevant and our assessment might be different today.

GraphQL for server-side resource aggregation | Technology Radar | ThoughtWorks
https://www.thoughtworks.com/radar/techniques/graphql-for-server-side-resource-aggregation

GraphQL as BFF の問題点

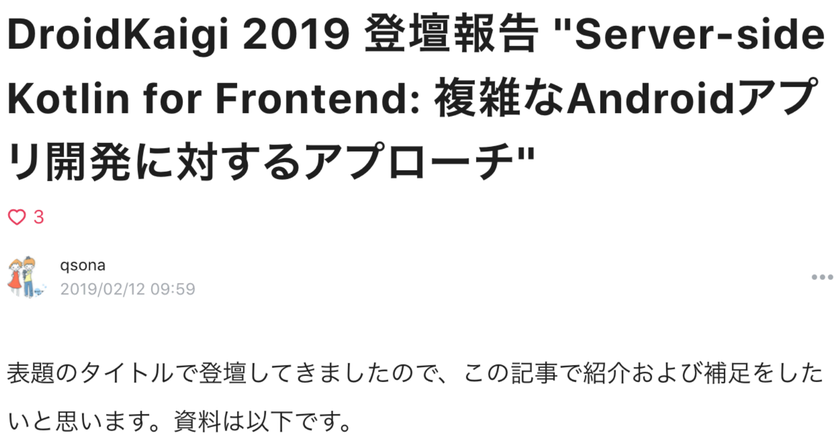
- BFF とはそもそも Frontend のための Usecase を定義する層
- GraphQL を利用すると、query が Usecase になる
  - => Usecase が2層になり、役割が被る

GraphQL as BFF の具体的な問題点

- Backend APIs (REST/RPC) を GraphQL に変換するコストが高い
  - 単なるプロトコルの変換だけでなく、 GraphQL resolvers に変換する上で本質的なインピーダンスミスマッチが存在する
  - さまざまな Usecase に対応しうる Schema をメンテナンスしつづけるコスト
- 一方で、使われる Usecase は限られている
  - 使うのは自分たちだけ (全世界に公開するわけではない)

代替手法の提案 (1)

- BFF を Usecase-based API として作る
  - プロトコルは JSON over HTTP でもいいし、
    RPC 系の何かを使ってもよい
  - GraphQL は利用しない
- (この路線で BFF を作る場合、
  良ければ右の資料を参考にしてみてください)



代替手法の提案 (2)

- Backend Services GraphQL API として実装する
- GraphQL Gateway サービスを置く
  - 技術としては Schema Stitching や Apollo Federation を検討
    - => プロトコルの変換を手でやる必要がなくなる
- GraphQL Gateway は Backend 寄りのエンジニアが管理する
  - 基本的に Backend のものを使いやすくするだけで、ロジックは基本的に置かない

PR:
Quipper における
GraphQL 活用事例紹介

国内向けサービス

スタディサプリ

スタディサプリ

小学生から受験生や大人まで、学習したい全ての人が学べる月額制のオンライン学習サービス。
約4万本の録画授業動画が見られるベーシックプランのほか、オンラインコーチングプランや生配信で授業を受けられるライブプランなど、一人一人が自由に学習できるよう、様々なプランを展開しています。

スタディサプリ
for TEACHERS

先生方が生徒個々人のレベルに合った最適な学習を提供できる校内インフラサービス。クラス全員に特定の講義や確認テスト、宿題を配信することができるほか、アクティブラーニングに使える教材も提供。
生徒が夢中になって学び、希望する進路を実現することを支援しています。

スタディサプリ
ENGLISH

隙間時間に3分で学習できる英語サービス。リスニングと発話を鍛えられる「新日常英会話コース」、短期間でのスコアアップを狙う「TOEIC®L&R TEST対策コース」、「ビジネス英語コース」があり、業界初オンライン完結型コーチングも提供しています。

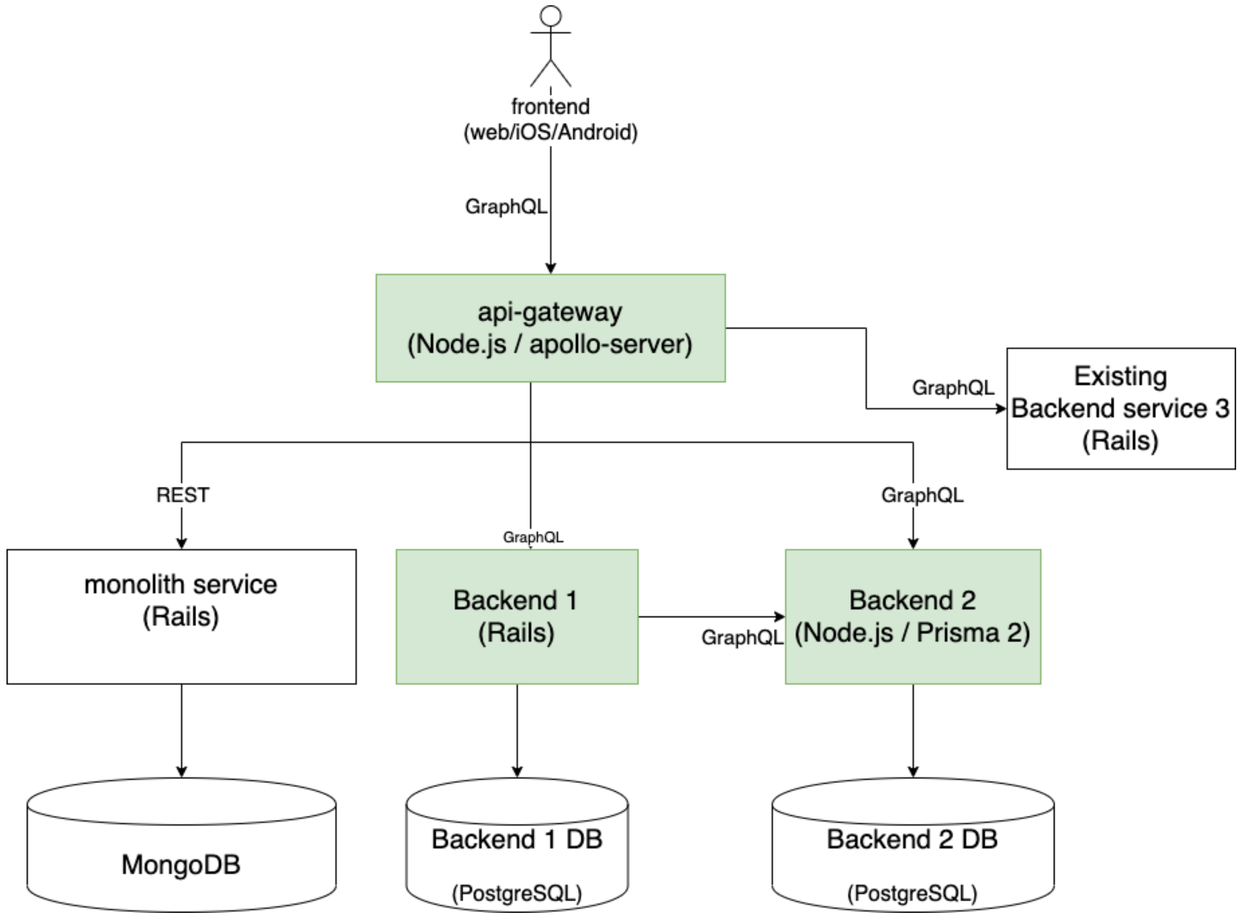
※Business Insider/2020年1月20日配信にて記事掲載

新規開発プロジェクトについて

- 開発するもの
  - iOS / Android / PC Web Apps
  - Backend APIs
  - コンテンツを管理するシステム
- 開発期間: > 1年
- 開発者の人数: > 10人

新規開発に GraphQL を全面採用

- client => gateway => backends
  全体で GraphQL を採用
- gateway は schema stitching を利用している
- その他利用技術
  - Ruby on Rails
  - Node.js + TypeScript + Apollo Server + Prisma 2
  - React, Apollo Client



GraphQL 採用の動機

- native devs (iOS/Android) と web devs (Backend + Web Frontend) でうまく協働できるようにしたかった
- => Schema-Driven Development を採用したい
- (一方で BFF パターンを回避したかった)
- さらに、できるだけ全体の開発工数も減らしたい

GraphQL を全面採用してよかったこと

- Schema-Driven Development により
  コミュニケーションのロスが少なく、
  安定して開発を進められた
- 繋ぎこみが超スムーズ
  - backend / frontend それぞれ ~9ヶ月開発したものを
    1週間で全部繋ぎこみ完了した
- GraphQL Gateway がとても便利で、かつ
  開発工数がほぼかからない

Schema-Driven Development とアジャイルなチーム

♡ 27

qsona
2020/07/22 23:27

最近は新規サービス開発で GraphQL を使っている。同じチームでAndroidエンジニアの geckour 氏が記事を書いてくれていたので、触発されてちょっと書く。

GraphQL + Apollo の世界 ~Android 編~ - Quipper Product Team Blog

こんにちは。Android アプリ開発者の geckour です。 今回は、Android における GraphQL と

quipper.hatenablog.com

Hatena Blog

GraphQL + Apollo の世界 ~Android 編~

Engineering　Engineering-Native-A…

Quipper Product Team Blog

上の記事の中で、この新規開発以前にあったAPI開発の問題点となぜ
Schema-drivenな開発にしたいのかについて触れられているので、ぜひそち

We're hiring!

- ひきつづき、絶賛新規開発中です!
- 運用フェーズに向けて、さらなる技術チャレンジをしていきたい
  - 例) Persisted Queries のフロー整備, Apollo Federation の検討, etc...
- GraphQL, Microservices はもちろん
  フロントエンドからバックエンドまで
  広い技術に触れられる機会あり
- まずはカジュアル面談から、ぜひご応募ください 😄

閑話休題

パターン2:

Repository pattern on Client-side GraphQL

Repository パターン

- API 呼び出し(など)をラップし、
  Domain Model を返すレイヤー
- Usecase から使われる
- API 以下の具体的な詳細を隠蔽することができる
  - "腐敗防止層"

Repository パターンと GraphQL

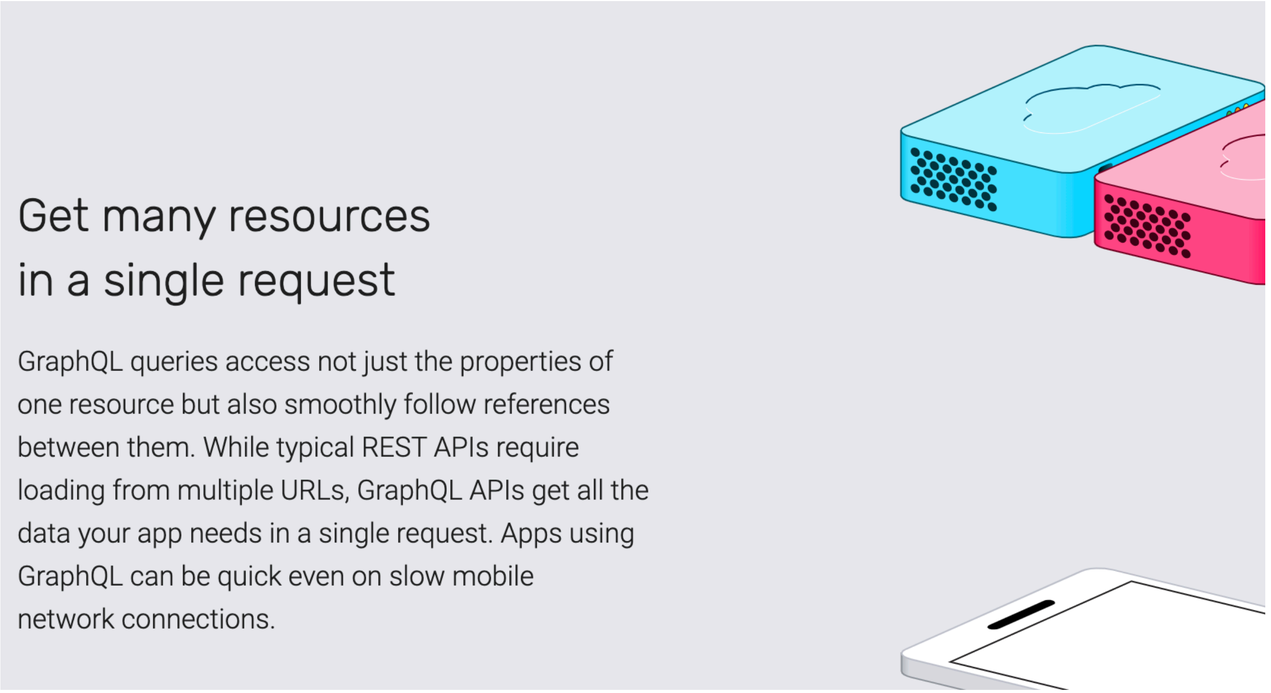
- あまり相性がよくない
- Usecase ==> Repository ==[query]==> GraphQL API
  - query は Usecase であると捉えると
    Domain の層が Usecase を知っていることになり、おかしい
- => query が Usecase を表せなくなり、GraphQL の良さが生きにくい
  - over-fetching などの問題が起きやすくなる

代替手法の提案

- GraphQL Schema をドメイン知識豊富なものとして、
  そもそも腐敗しないようにする
  - そのドメイン知識をクライアントとも共有する
- サーバーサイドまで含めたアーキテクチャと捉えると良い
  - クライアントエンジニアも GraphQL Schema の設計に積極的に関わるのが理想

Fragment Colocation

- GraphQL の重要な特徴として、1リクエストで同時に複数のリソースを取得できる
- その特徴を活用しつつも、Usecase を適切に分割したい



Fragment Colocation

- 例: 1画面内に複数の Usecase / Component があり、
  それらのデータ取得を1リクエストにまとめたい

  - Usecase (Component) は
    多段になっている (ネスト) こともある

- 各 Usecase / Component ごとに fragment をもつ (colocation)

- 上位の Usecase で fragment をまとめて1つの query にする

まとめ

GraphQL とアーキテクチャまとめ

- query が Usecase

  Schema / Types が Resource である、と捉えよう

- その上で、既存の設計手法と組み合わせて考えよう

- Client / Server の局所的な視点だけでなく、

  大局的に見て設計しよう

Thank you for listening!

- 登場したリンクは Speaker Deck の概要欄に
  すべて載せておきます