

GraphQL ハンズオン

やること

ハンズオンとなっておりますが動作するコードは用意してあるので説明 70%、手を動かすの 30% くらいです 😊

流れ

- サンプルアプリについて
- GraphQL の基本的なところ
- Apollo Studio で Query や Mutation を実行する
- Resolver について
 - これがメイン、残りは時間があまれば
- フロントエンドのうれしさ
- PostGraphile

ゴール

このハンズオンのゴール

- GraphQL がなんとなくわかる
- Resolver の仕組みがわかる
- （時間があれば）
 - フロントエンドのうれしさがわかる
 - PostGraphile の便利さがわかる

事前準備

ハンズオンの実施前に事前準備を済ませておいてください！

<https://github.com/adwd/graphql-sample-app> のリポジトリを使います。Node.js (14 以上), yarn, docker が必要です。

```
> node -v
v14.18.1
> yarn -v
1.22.17
> docker -v
Docker version 20.10.8, build 3967b7d

> git clone https://github.com/adwd/graphql-sample-app
> cd graphql-sample-app

> yarn # 依存ライブラリのインストール
> docker compose up -d # PostgreSQL サーバーの起動
```

5432 番ポートに PostgreSQL サーバーが立ち上がったたり、4000 とかも GraphQL サーバーが立ち上がるのでそのへんを開けておいてください。

サンプルアプリについて

GraphQL のサーバーが 2 つ、React クライアントアプリが 1 つ、それらを動かす docker-compose.yaml が入っています。

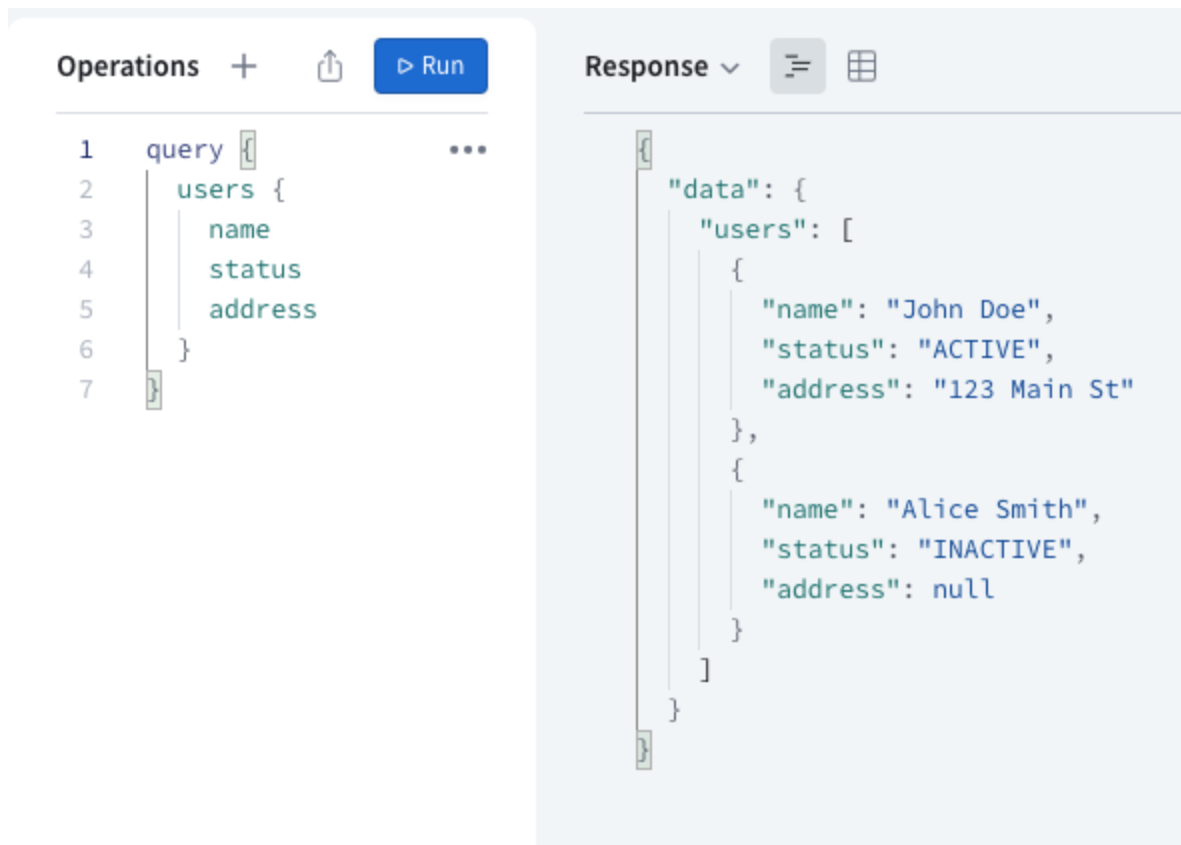
yarn workspace というのを使っていて、プロジェクトのルートディレクトリから

- `yarn server1`
 - Apollo Server
- `yarn server2`
 - Hasura Server
- `yarn client-app`
 - React/Apollo Client

のコマンドでそれぞれ起動できます。またサーバー・クライアントともに起動中にコードをいじると随時反映されるようになっているので、何か試してみるとすぐ動作確認できます。

GraphQL の基本的なところ

GraphQL は GraphQL Foundation が定める仕様で、主にクライアントが要求したクエリ通りのデータが返ってくるための仕組みを定義してあります。



こんな感じで、クライアントが欲しいデータの形を要求するとそのまま返ってくるのが特徴です。

GraphQL ではクライアントからサーバーに対して 3 つのオペレーションがあります。

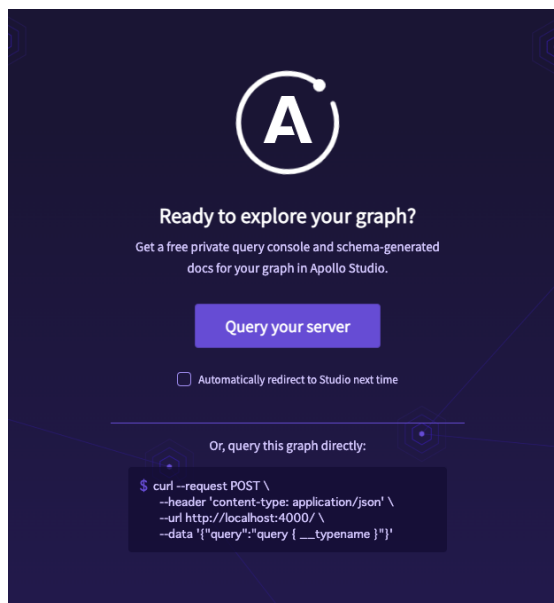
- Query
 - データを取ってくる
- Mutation
 - データを変更する
- Subscription
 - リアルタイムにデータをサーバーから受け取る

Apollo Studio を立ち上げて触ってみる

```
› yarn server1
```

```
Server ready at http://localhost:4000/
```

<http://localhost:4000/> を開くとこんな画面が表示されるので、「Query your server」ボタンを押します。



Apollo Studio が起動します。Apollo Studio は GraphQL を使った開発に必要な多くの機能を提供するサービスですが、今回は Query や Mutation を実行できるプレイグラウンドとスキーマのドキュメント機能を使います。似た用途の OSS としては [GraphiQL](#) や [GraphQL Playground](#) があります。

The screenshot displays the Apollo Studio web interface. The top navigation bar includes the Apollo logo, a 'SANDBOX' tab, the URL 'http://localhost:4000/', and a 'Log in' button. The left sidebar contains a 'Documentation' section with a breadcrumb 'Root > Query' and a list of fields: 'users: [User]' (selected) and 'libraries: [Library]'. The main area is divided into three panels. The 'Operations' panel on the left shows a GraphQL query:

```
1 query {
2   users {
3     address
4     name
5   }
6 }
7
```

 with a 'Run' button. The bottom panel shows 'Variables' with a single entry '1' in 'JSON' format. The 'Response' panel on the right shows the JSON output:

```
{
  "data": {
    "users": [
      {
        "address": "123 Main St",
        "name": "John Doe"
      },
      {
        "address": null,
        "name": "Alice Smith"
      }
    ]
  }
}
```

 The status bar at the top right of the response panel indicates 'STATUS 200', '16.1ms', and '103B'.

Query を実行してみる

Operation と書いてあるテキストエリアに

```
query {  
  users {  
    name  
    status  
    address  
  }  
}
```

と入力して、Run ボタンを押すと結果が右側に表示されます。

いろんな Query、Mutation を実行してみる

ハンズオンタイム

一度 users Query を実行して結果を確認したあと、addUser Mutation を実行してユーザーを追加して、もう一度 users Query でユーザーが増えていることを確認しましょう。

左側の Documentation と書いてあるところのチェックボックスを付けたり、アイテムを選択すると Operations のところに反映されて、スキーマの構造を見つつ試しに Query を発行できます。

Ctrl+Space でサジェスト、Cmd+Enter でオペレーション実行などのショートカットがあります。

細かい話

この Playground の仕組みは、GraphQL の [Introspection](#) という仕組みを使っていて、`__schema` などの Query が用意されていて Apollo Studio はこれを起動時に実行することでスキーマ情報を取得しています。Chrome Devtools の Network タブから見れます。

これは開発用途の仕組みのため、プロダクション環境では無効化することが推奨されています。

```
query IntrospectionQuery {  
  __schema {  
    types {  
      name  
      kind  
      description  
    }  
  }  
}
```

GraphQL のスキーマを試してみる

Apollo Studio の <https://studio.apollographql.com/sandbox/schema/reference> を開くと、GraphQL スキーマを Apollo Studio が見やすくドキュメントにしたものが表示されます。

コードとしては <https://github.com/adwd/graphql-sample-app/blob/main/packages/apollo-prisma-server/src/index.ts> に定義してあります。

type Query, type Mutation をルートとして、そこで使われている type がそれぞれ定義してある感じです。

Resolver とは

Resolver は GraphQL サーバーを実装する上で重要な要素で、GraphQL の特徴であるクライアントが指定したデータだけを返す仕組みを実現しているものです。

<https://graphql.org/learn/execution/>

今回のサンプルコードはほとんど <https://www.apollographql.com/docs/apollo-server/data/resolvers/> からの流用です。

雑に言うと、Query されたツリーの形に従って Resolver を順番に呼び出してデータを取得する仕組みです。

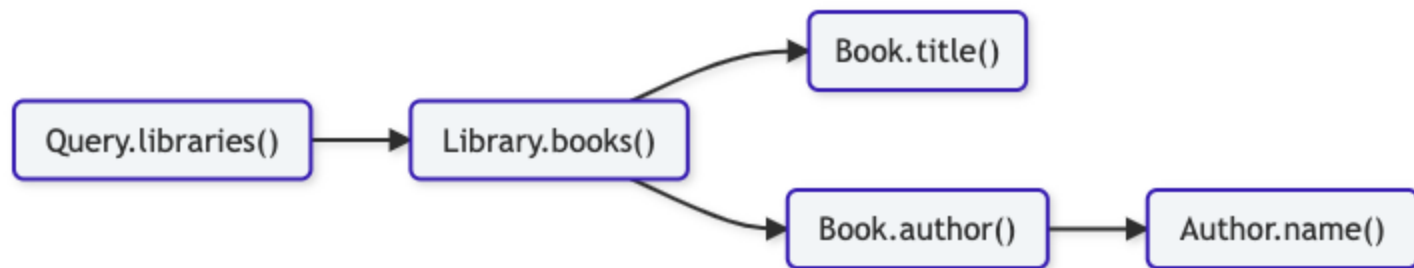
サンプルアプリの Resolver のコードはこんな感じです。これがどう動くかイメージしにくいと思うので次のページから説明します。

```
// https://github.com/adwd/graphql-sample-app/blob/main/packages/apollo-prisma-server/src/index.ts
const resolvers = {
  // 前段で実行された Resolver の結果が引数の parent に入って実行されます。
  Query: {
    libraries() {
      return libraries;
    },
  },
  Library: {
    books(parent, args, context, info) {
      return books.filter((book) => book.branch === parent.branch);
    },
  },
  Book: {
    author(parent) {
      return {
        name: parent.author,
      };
    },
  },
};
```

Resolver 完全理解タイム

例えばこの Query に対する GraphQL サーバーの Resolver の実行は下の図のようになります。Query のツリーの形に沿って Resolver が実行されます。前のページのコードと見比べると動作のイメージがわかってきますか...？

```
query {  
  libraries {  
    books {  
      title  
      author {  
        name  
      }  
    }  
  }  
}
```



サンプルコードだと、ハードコードしてあるデータの形と GraphQL スキーマの型が微妙に違っています。Resolver の実装を追うとその違いがあってもちゃんと GraphQL スキーマの型にあったデータを返せてる様子が見えます。

ここで `Book.title` と `Author.name` の Resolver が定義されていません。その 2 つは `parent[field]` を返すだけのこういう Resolver で、

```
Book: {  
  title(parent) {  
    return parent.title;  
  },  
},
```


その場合は Default Resolver として Apollo が勝手に値を返してくれます。 **重要**

Default Resolver

GraphQL らしい Resolver を書こうとすると最初は難しいですが、Query の段階で完全にデータが得られるようにしておけばその後はすべて Default Resolver で `parent[key]` が返されるだけです。そう考えると多少強引ですが REST API を書くのとは大差なく書けると思います。

```
const resolvers = {
  Query: {
    libraries() {
      return repository.libraries(); // RESTと同じくDBからデータ取って返してるだけ
    },
    books(parent, args, context, info) {
      // Queryでどのフィールドを書いているかはinfoのSelectionSetからわかる
      if (shouldFetchAuthor(info.selectionSet)) {
        return repository.booksWithAuthor(); // booksテーブルとauthorsテーブルをJOINする
      }

      return repository.books(); // authorsテーブルとJOINしない
    },
  },
};
```

 ここで Resolver まで終わったので、このあとのフロントエンドの話とか Hasura の話は時間があればやる

フロントエンドは何がうれしいのか

サンプルアプリのフロントエンドアプリは Server2 と通信します。
下記コマンドで起動します。

```
› yarn server2  
# 別のターミナルから  
› yarn client-app
```

上記コマンドで <http://localhost:3000/> にクライアントアプリが立ち上がります。server2 は レンタル DVD を題材にしたサンプルの DB なので、フロントエンドもレンタル DVD の Web サイトみたいなのを雑に作っています。

DVDレンタルストアのWebサイト

人気の俳優

()の中は出演本数

- Gina Degeneres (42 本)
- Walter Torn (41 本)
- Mary Keitel (40 本)

レビュー評価の高い映画

レンタルできる映画の総数: 1000, 表示件数 50

<div>4.99 Horror 2006 • 貸出し 3 日</div> <div>Ace Goldfinger</div> <div>A Astounding Epistle of a Database Administrator And a Explorer who must Find a Car in Ancient China</div> <div>Bob Minnie Sean Fawcett Zellweger Guinness</div>	<div>4.99 Comedy 2006 • 貸出し 6 日</div> <div>Airplane Sierra</div> <div>A Touching Saga of a Hunter And a Butler who must Discover a Butler in A Jet Boat</div> <div>Jim Richard Oprah Mostel Penn Kilmer</div>	<div>4.99 Horror 2006 • 貸出し 6 日</div> <div>Airport Pollock</div> <div>A Epic Tale of a Moose And a Girl who must Confront a Monkey in Ancient India</div> <div>Fay Kilmer Gene Willis Susan Davis</div>
<div>4.99 Sports 2006 • 貸出し 6 日</div> <div>Aladdin Calendar</div> <div>A Action-Packed Tale of a Man And a Lumberjack who must Reach a Feminist in Ancient China</div> <div>Alec Wayne Judy Dean Val Bolger</div>	<div>4.99 Horror 2006 • 貸出し 4 日</div> <div>Ali Forever</div> <div>A Action-Packed Drama of a Dentist And a Crocodile who must Battle a Feminist in The Canadian Rockies</div> <div>Cary Christopher Kenneth Mcconaughey Berry Torn</div>	<div>4.99 Music 2006 • 貸出し 4 日</div> <div>Amelie Hellfighters</div> <div>A Boring Drama of a Woman And a Squirrel who must Conquer a Student in A Baloon</div> <div>Carmen Walter Ed Hunt Torn Mansfield</div>

このページでは

- actor のうち出演本数が多い 3 人の名前とその出演本数
- film のうちレビュー評価の高い 50 本

を表示しています。

REST だといろんなエンドポイントを叩いて集める必要があるのを GraphQL だと一つの Query で済む

ダッシュボード的なページを作ろうとするといろんなデータが必要で、REST だとその分フロントエンドが何回も API を叩くか、もしくは BFF でそのページに合ったデータを取得する GET /dashboard みたいなのを実装する必要がありますが、GraphQL だと一つの Query に必要なデータを並べれば1度のリクエストで取得できます。

Fragment を使うと子コンポーネントが必要なデータを親コンポーネントが知らなくて済む

レビュー評価の高い映画リストで、カード状に表示しているところは Fragment という仕組みを使って、親コンポーネントはカードにどういうデータが必要かを知らなくて済むようになっていきます。

```

2 import { Box, Flex, Heading, ListItem, UnorderedList } from "@chakra-ui/react";
3 import { useHighestRatedFilmsPageQuery } from "../generated/graphql";
4 import { FilmCard } from "../FilmCard";
5
6 const query = gql`
7   query HighestRatedFilmsPage($filmCount: Int!) {
8     Execute Query
9     actors(orderBy: FILM_ACTORS_COUNT_DESC, first: 3) {
10       nodes {
11         firstName
12         lastName
13         id
14         filmActors {
15           totalCount
16         }
17       }
18
19       films(first: $filmCount, orderBy: RENTAL_RATE_DESC) {
20         totalCount
21         nodes {
22           ...FilmCard
23         }
24       }
25     }
26   };

```

```

27
28 const filmCount = 50;
29 export const HighestRatedFilmsPage = () => {
30   const { data, loading, error } = useHighestRatedFilmsPageQuery({
31     variables: { filmCount },
32   });
33
34   if (loading || !data) {
35     return <loading/>;
36   }
37   if (error) {
38     return <{error.message}/>;
39   }
40
41   return (
42     <div style={{ margin: "16px" }}>
43       <Heading as="h1" size="2xl">
44         DVDレンタルストアのWebサイト
45       </Heading>
46       <Heading mt="4" as="h2" size="xl">

```

```

2 id
3 title
4 description
5 releaseYear
6 rentalDuration
7 rentalRate
8 filmCategories {
9   nodes {
10     id
11     category {
12       name
13     }
14   }
15 }
16 filmActors(first: 3, orderBy: NATURAL) {
17   nodes {
18     id
19     actor {
20       ...ActorList
21     }
22   }
23 }
24 }
25

```

FilmCard.tsx ×

packages > react-apollo-client > src > dvd-rental > FilmCard.tsx > FilmCard > fil

```

masahiro.nishida, a week ago | 1 author (masahiro.nishida)
1 import { Box, Badge } from "@chakra-ui/react";
2 import { FC } from "react";
3 import { FilmCardFragment } from "../generated/graphql";
4 import { ActorList } from "../ActorList";
5
6 export const FilmCard: FC<{ film: FilmCardFragment }> = ({ film }) =>
7   return (
8     <Box
9       maxW="sm"
10       borderWidth="1px"
11       borderRadius="lg"
12       overflow="hidden"
13       w={300}
14     >
15       <Box p="6">
16         <Box display="flex" alignItems="baseline">
17           <Badge borderRadius="full" px="2" colorScheme="teal">

```


今回はネストが2段階しか無いですが、もっと深くなると子・孫コンポーネントの表示に必要なデータをルートコンポーネントの Query で管理するのはかなり面倒です。

コンポーネントが必要なデータを自分で定義することができるので凝集性が高まります。

PostGraphile

PostGraphile は、PostgreSQL から GraphQL サーバーを生成するライブラリです。Node.js サーバーに組み込んで実行できる他、`npx postgraphile -c postgres:///mydb -s public -a -j` のように CLI から起動できます。サービスで PostgreSQL を使っている人はローカルや Dev のサーバーに繋げる人はつなぐと面白いかもしれません 😎

サンプルコードは以下にあります。

<https://github.com/adwd/graphql-sample-app/blob/main/packages/postgraphile-server/src/index.ts>

以下のコマンドで PostGraphile サーバーが立ち上がって、<http://localhost:5000/graphql> から触れます。

```
› yarn server2
```

PostGraphile はそのままだと DB をそのまま公開するけしからんライブラリですが、下記の機能を使ってけしからなさをなくすることができます。

- PostgreSQL の role などと連携して、PostGraphile サーバーへのリクエストに付随する JWT を使って認証・認可ができる
 - <https://www.graphile.org/postgraphile/postgresql-schema-design/>
- JavaScript/TypeScript で Query/Mutation に任意のロジックを追加できる
 - <https://github.com/adwd/graphql-sample-app/blob/main/packages/postgraphile-server/src/my-plugin.ts>

PostGraphile を使うとサーバーの実装コストをすごく削減できます（特に参照系）