

Flutter × GraphQL

宣言的モバイルアプリ開発

2025/2/7

吉田 航己

吉田 航己

Yoshida Koki

株式会社サイバーエージェント
ト
モバイルアプリエンジニア

X : [@koki8442](https://twitter.com/koki8442)

GitHub : [llttt06](https://github.com/llttt06)



目次

1. GraphQL の概要
2. ジャンプTOON の GraphQL 活用
3. Flutter で GraphQL を使う利点
4. Flutter × GraphQL 開発の工夫
5. おわりに

GraphQL とは？

REST の問題を解決するための API クエリ言語およびランタイム

- データを Node と Edge のグラフで構造化し、必要に応じてそのグラフの一部を利用する
- 単一のエンドポイントから必要なデータだけを Query するため、オーバー(アンダー)フェッチを防げる

使用パッケージ

GraphQL クライアント : [graphql_flutter](#) ([graphql](#))

GraphQL 自動生成関連 : [graphql_codegen](#)

SettingScreen の例

データの取得を Screen 内の `useQuery$Setting` で行う

```
// setting_screen.dart
class SettingScreen extends HookConsumerWidget {
  const SettingScreen({super.key});
  @override
  Widget build(BuildContext context, WidgetRef ref) {
    final isGuestUser = useIsGuestUser();
    final query = useQuery$Setting(options);
    return Scaffold(
      // ...
      body: GraphQLQueryContainer(
        query: query,
        onLoadingWidget: SkeletonSettingScreen(isGuestUser: isGuestUser),
        onErrorWidget: (error, stackTrace) => ErrorContainer(
          error: error,
          stackTrace: stackTrace,
          onAction: query.refetch,
        ),
        child: (data) {
          return SingleChildScrollView(
            // ...
          );
        },
      ),
    );
  }
}
```

```
// graphql_query_container.dart
class GraphQLQueryContainer extends HookWidget {
  const GraphQLQueryContainer({
    required this.query,
    required this.child,
    this.onLoadingWidget,
    this.onEmptyWidget,
    this.onErrorWidget,
    super.key,
  });

  final QueryHookResult query;
  final Widget Function(TParsed data) child;
  final Widget? onLoadingWidget;
  final Widget? onEmptyWidget;
  final Widget Function(GraphQLError error, StackTrace? stackTrace)?
    onErrorWidget;

  @override
  Widget build(BuildContext context) {
    if (result.hasException && onErrorWidget != null) {
      // エラーが発生した場合
      return onErrorWidget!(
        GraphQLException.fromOperationException(result.exception),
        StackTrace.current,
      );
    } else if (result.data == null && result.isNotLoading) {
      // データがない場合
      return onEmptyWidget ?? const SizedBox();
    } else if (result.data == null && result.isLoading) {
      // ローディング状態
      return onLoadingWidget ?? const SizedBox();
    }
  }
}
```

query の定義と自動生成ファイル

query を `.graphql` に記述し、`.dart` のコードを自動生成する

```
# setting_screen_query.graphql
query Setting {
  me {
    id
  }
}
```

```
// setting_screen_query.graphql.dart
class Query$Setting {
  Query$Setting({
    required this.me,
    this.$__typename = 'Query',
  });

  factory Query$Setting.fromJson(Map<String, dynamic> json) {
    final l$me = json['me'];
    final l$__$typename = json['__typename'];
    return Query$Setting(
      me: Query$Setting$me.fromJson((l$me as Map<String, dynamic>)),
      $__typename: (l$__$typename as String),
    );
  }

  final Query$Setting$me me;
```

Fragment Colocation

コンポーネントとそこで使用するデータ郡(fragment)を1:1対応させ、近くに配置 ([参考資料](#))

```
ui/  
├── component/  
│   └── text/  
│       ├── user_name_text_fragment.graphql  
│       ├── user_name_text_fragment.graphql.dart  
│       └── user_name_text.dart  
└── screen/  
    ├── root/  
    │   └── my_page/  
    │       ├── my_page_query.graphql  
    │       ├── my_page_query.graphql.dart  
    │       └── mv_name_screen.dart
```


Flutter × GraphQL の利点

1. スキーマファースト開発
2. クライアントキャッシュ
3. 宣言的 UI との親和性

スキーマファースト開発

最初に GraphQL Schema を定義し、Schema 定義に合うようにコード(API など)を実装する手法

- クライアントーサーバー間のコミュニケーションコスト削減
- API 実装完了前でも Schema からデータをモック可能
- [file-sync-action](#) でサーバーでマージした Schema を同期

スキーマファースト開発

The screenshot shows a GitHub pull request interface. At the top, there are tabs for Conversation (4), Commits (1), Checks (14), and Files changed (2). A bot user, 'app', has commented 3 days ago. The comment states: 'This PR contains the following updates:'. Below this, there is a table with three columns: Change, Synchronizing Repository, and Workflow. The 'Change' column shows '2 files'. The 'Synchronizing Repository' column shows 'schema'. The 'Workflow' column shows 'Sync Schema Files#240'. Below the table, there is a section titled 'Changed Files' which lists two file changes: 'schema/graphql/userbff/enums_gen.graphql to packages/app/lib/data/schema/graphql/userbff/enums_gen.graphql' and 'schema/graphql/userbff/series.graphql to packages/app/lib/data/schema/graphql/userbff/series.graphql'. At the bottom right, there is a note: 'Generated by wadackel/files-sync-action.'.

Change	Synchronizing Repository	Workflow
2 files	schema	Sync Schema Files#240

Changed Files

- `schema/graphql/userbff/enums_gen.graphql` to `packages/app/lib/data/schema/graphql/userbff/enums_gen.graphql`
- `schema/graphql/userbff/series.graphql` to `packages/app/lib/data/schema/graphql/userbff/series.graphql`

Generated by [wadackel/files-sync-action](#).

クライアントキャッシュ

モバイルアプリにとって個人的に GraphQL を採用する一番のメリット

GraphQL クライアントでは、Query, Mutation などの Operation の結果を**正規化**してキャッシュ。正規化は以下の 3 ステップで行われる。

1. Operation 結果を分割して個別のオブジェクトにする
2. 分割したオブジェクトに一意的 key をつける
3. それぞれのオブジェクトをフラットに保存する

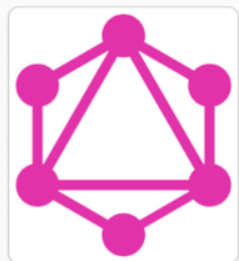
クライアントキャッシング

```
# リクエスト
query {
  comics {
    id
    title
    episodes {
      id
      title
    }
  }
}
```

```
# レスponse
[
  Comics: [
    {
      id: 1,
      __typename: "Comics",
      title: "WebToon Title",
      episodes: [
        {
          id: 1,
          __typename: "Episodes",
          title: "Revenge",
        },
        {
          id: 2.
```

3. Flutter × GraphQL の利点

```
query {  
  comics {  
    id  
    title  
    episodes {  
      id  
      title  
    }  
  }  
}
```



```
[  
  Comics: [  
    {  
      id: 1,  
      __typename: "Commics",  
      title: "WebToon Title",  
      episodes: [  
        {  
          id: 1,  
          __typename: "Episodes",  
          title: "Revenge",  
        },  
        {  
          id: 2,  
          __typename: "Episodes",  
          title: "Imagination",  
        },  
      ],  
    },  
  ],  
]
```

Normalization

```
Comics:1  
{  
  id: 1,  
  __typename: "Commics",  
  title: "Webtoon Title",  
  episodes: {  
    [0]: {__ref: "episodes:1"},  
    [1]: {__ref: "episodes:2"},  
  },  
}
```

```
Episodes:1  
{  
  id: 1,  
  __typename: "Episodes",  
  title: "Revenge",  
},
```

```
Episodes:2  
{  
  id: 2,  
  __typename: "Episodes",  
  title: "Imagination",  
},
```

クライアントキャッシュ

キャッシュの更新、保存を自動で GraphQL クライアントが行う

`useQuery` を使用している Widget は別の Query や Mutation でのキャッシュの更新に伴って UI も自動で更新される

宣言的 UI との親和性

データを元に UI を構築する宣言的 UI は、GraphQL クライアントのキャッシュ機構との親和性が高い。

ある Operation で正規化されたキャッシュデータが更新されると、それを参照しているすべての UI が更新されるため。

REST では同じことは出来ないの??

宣言的 UI との親和性

- Redux の公式ドキュメント [Store データの正規化](#)
 - 正規化を手動で行う必要がある。GraphQL はデータの構造が Schema に **Graph** として表現されているため機械的にこれができる。
- React の [TanStack Query](#) や [SWR](#)
 - 基本的に URL を key としてキャッシュするため、個々のデータに分割してキャッシュされない。
 - Flutter にはこれらにインスパイアされた [fQuery](#) がある。

おわりに

詳細は [こちら](#)



ジャンプTOON app

Flutter×GraphQL

宣言的なアプリ開発の工夫

JUMPTOON DEVELOPERS

参考文献

- [GraphQL](#)
- [Caching in Apollo Client](#)
- [Demystifying Cache Normalization](#)
- [Normalizing State Shape](#)
- [GraphQL Client Architecture Recommendation 社外版](#)
- [宣言的UIの状態管理とアーキテクチャSwiftUIとGraphQLによる実践/swiftui-graphql](#)