GraphQL のエラーレスポンス設計における意思決定

2025 年 3 月 13 日（木）Roppongi.rb #28

自己紹介

栃川 晃佑

- X: @Web_TochiTech
- Ruby 歴: 4 年
    - 人材紹介会社で Rails エンジニアとして働いています！

「コミュニティ活動」への参加をこれから積極的にしようと思い、飛び込みでLT応募しました🙌
もしかしたら今後ちょくちょく顔を出すかもしれないです🙏

はじめに

ごめんなさい 🤦‍♂️

このスライドは、初めて「Cursor + Marp」で作ったので、
時折違和感のあるスライドやコード記述があります。

何卒ご容赦いただけますと幸いです。。。。

今日お話しすること

- GraphQL Spec で推奨されないエラーレスポンスの方法で実装した意思決定の理由を紹介します
- **ぜひフィードバックいただきたいです！**

アジェンダ

- エラーレスポンス設計の選択肢
- 自分がした意思決定とその理由
- 実装例
- まとめ

エラーレスポンス設計の選択肢

そもそもエラーレスポンスの設計ってどんなのがあるの？

エラーレスポンス設計の選択肢

前提として、GraphQL のエラー設計には、大きく以下の 3 つのアプローチがある

1. **標準仕様**
2. **Errors as data パターン**
3. **Union / Result Types パターン**

それぞれの方法にはメリットとデメリットがあり、組織の状況やプロジェクトの要件によって適切な選択が求められる。

※ このあと説明します

1. 標準仕様に従う

GraphQL の標準仕様では、エラーは`data`フィールドと同階層の `errors` フィールドに格納される。

```
{
  "errors": [
    {
      "message": "指定のユーザが見つかりませんでした",
      .
      .

      "path": ["users"]
    }
  ],
  "data": null //  データはNULL
}
```

1. 標準仕様に従う

メリット

- 実装が最も簡単で手軽
- クライアント側で特別なクエリを書く必要がない

デメリット

- エラーの型や構造がスキーマで定義されないため予測が困難
- すべてのエラーが `errors` フィールドに混在するため、クライアント側での判別が難しい
  - GraphQL Spec の作者も、エラーは**例外的な状況を表すべき**であり、ユーザーデータをエラーとして表現すべきではないと主張している。

2. Errors as data

エラーも「**データ**」としてクライアント側に返してあげる方法。
エラー内容をデータと一緒に返してあげる。

```
type SignUpPayload {
  userErrors: [UserError!]! # エラーがあればサーバーサイドでメッセージを格納
  account: Account # 取得しようとするデータ
}

type UserError {
  message: String!
  field: [String!]
  code: UserErrorCode
}
```

2. Errors as data

▽ レスポンスの返却例

```
{
  "data": {
    "signUp": {
      "userErrors": [
        {
          "message": "パスワードは8文字以上である必要があります",
          "field": ["password"],
          "code": "INVALID_PASSWORD"
        },
        {
          "message": "このメールアドレスは既に使用されています",
          "field": ["email"],
          "code": "EMAIL_TAKEN"
        }
      ],
      "account": null
    }
  }
}
```

2. Errors as data

メリット

- スキーマでエラーを管理できるため予測しやすい
- 独自のエラーレスポンスのデータ構造を定義可能

デメリット

- クライアントが `userErrors` フィールドを明示的にクエリする必要がある
- 開発者が常に `userErrors` を意識する必要があり認知負荷が高まる

3. Union / Result Types

Union 型を利用して成功時とエラー時の異なる結果を表現する方法

```
type Mutation {
  signUp(email: String!, password: String!): SignUpPayload
}

# Union型としてそれぞれレスポンスを表現
union SignUpPayload = SignUpSuccess | UserNameTaken | PasswordTooWeak

type SignUpSuccess {
  account: Account
}

type UserNameTaken {
  message: String!
  suggestedUsername: String
}

# 省略
```

3. Union / Result Types（続き）

メリット

- 成功パターンとエラーパターンを型として明確に分離、各エラーシナリオごとにカスタムフィールドを追加可能
- 強い型付けによる安全性

デメリット

- スキーマが肥大化しやすい
- クエリが煩雑化しやすい
- クライアントがすべての可能な型に対してフラグメントを定義する必要がある

3. Union / Result Types

返ってくる型に合わせてクライアントも用意をしておく必要がある。

▽ クライアント側のクエリ例

```
mutation {
  signUp(email: "user@example.com", password: "P@ssword") {
    ... on SignUpSuccess {
      account {
        id
      }
    }
    ... on UserNameTaken {
      message
      suggestedUsername
    }
    ... on PasswordTooWeak {
      message
      passwordRules
    }
  }
}
```

各設計手法の比較

手法	メリット	デメリット
標準仕様に従う	・手軽に実装できる	・予測しにくい ・スキーマ外で管理
Errors as Data	・独自のエラーデータ構造 ・スキーマで管理	・クライアント側での認知負荷が高い
Union / Result Types	・成功とエラーを型として明確に分離 ・強い型付け	・スキーマの肥大化 ・クエリの煩雑化

で、結局何がいいん？

結論、正解はない

- システムエラー以外のエラーはデータとして返すべきというのが GraphQL Spec の作者の主張。
- 標準仕様の errors フィールドに全てのエラーメッセージを格納するのはあまり推奨されてない

自分がした意思決定とその理由

結論、私はあらゆるエラーを標準仕様の errors フィールドに格納するという意思決定をした

- 標準仕様に則ることが一番恩恵を受けられるから

意思決定するにあたってのポイント

- 意思決定するにあたり 3 つのポイント
  - アーキテクチャとプロダクト特性の関係
  - 組織体制とメンバー
  - 直近の技術動向

意思決定ポイント 1: アーキテクチャとプロダクト特性の関係

- アーキテクチャ
  - マイクロフロントエンド（Next.js） + 単一 API の構成で、**複数サービス**を運営している
- プロダクトの特性
  - **各サービスごとに求められる基準値が大きく異なる**
  - 例
    - プロダクト A では、ユーザの CV 数が事業の要になるので、エラーの内容を詳細に返す必要がある
    - プロダクト B では、社内向けである程度リテラシーがあるので、エラーの内容が詳細を返す必要はない

💡 エラーメッセージそのものをAPI側で管理するより、各サービスのフロントでした方が柔軟性がある

意思決定ポイント 2: 組織体制とメンバー

- 事業の急拡大によって、やりたいこと・やるべきことが山積みで、まだまだ**開発エンジニアが足りてない**。
  - 新たにメンバーの加入があれば、 **すぐに立ち上がれるようにしておきたい**
- 所属メンバーは全員フルスタックエンジニアで、**フロントとサーバーサイドの両方の知識を有している**

💡 新規メンバーができるだけ実装や学習コストを下げることが、事業の売り上げに直結する

サーバーサイドのコードを理解できるので、 標準仕様のデメリットであるスキーマ外のエラー管理を許容できる

意思決定ポイント 3: 直近の技術動向

- RSC の登場で、**今後も GraphQL を採用しつづけるとは限らない**
  - データフェッチは ServerComponents で行うことがベストプラクティス
    - 参考: Data Fetching and Caching
  - RSC に GraphQL を組み合わせることはメリットよりデメリットの方が多くなる可能性がある
    - 参考: Next.js の考え方

💡 今GraphQLスキーマを作り込むことにメリットがあるかわからない。（頑張ったのに、技術スタックの変更がありうる）

意思決定ポイントまとめ

- まめるとこんな感じ

要因	状況	ポイント
アーキテクチャ と プロダクト特性	・プロダクトごとに要件が異なる	**・エラーメッセージはフロントで管理して柔軟性担保**
組織体制と メンバー	・リソース不足 ・メンバーの守備範囲の広さ	**・実装・学習コストを抑える** **・デメリットの許容**
技術動向	・RSC の台頭 ・技術スタックの変更可能性	**・作り込むメリットが見えなかった**

うん、標準仕様に乗っかるので良さそうだ！

実装の紹介

GraphQL Ruby での実装例（サーバーサイド）

- ユーザ作成のための mutation で考えてみる

```ruby
module Mutations
  class CreateUser < BaseMutation
    argument :name, String, required: true

    field :user, Types::UserType, null: true

    def resolve(name:)
      user = User.new(name:)

      if user.save
        { user: }
      else
        raise_errors(user) # 後述
      end
    end
  end
end
```

GraphQL Ruby での実装例（サーバーサイド）

- `raise_errors`メソッドで、errors フィールドにエラーコードを格納する処理

```
private

  def raise_errors(user)
    # userオブジェクトからエラーオブジェクトを取り出す
    user.errors.each do |error|
      # エラーの種類に応じて、エラーコードを選択する
      extension_code = case error.type
                       when :taken
                         'NAME_DUPLICATED'
                       else
                         'USER_INPUT_ERROR'
                       end

      # errorsフィールドを追加
      context.add_error(GraphQL::ExecutionError.new(error.message, extensions: { code: extension_code }))
    end
  end
```

- サーバーサイドではこれだけ。

GraphQL Ruby での実装例（フロント）

- エラーコードを元に、メッセージを出し分けるだけで良い

```
createUser({ variables: { input: { name: data.name } } })
  .then(() => {
    console.log("作成成功！");
  })
  .catch((error) => {
    const errors = error.graphQLErrors;

    errors.forEach((error) => {
      const extensionCode = error.extensions.code;

      if (extensionCode === "NAME_DUPLICATED") {
        console.log("すでに登録済みの名前です");
      }

      if (extensionCode === "USER_INPUT_ERROR") {
        console.log("名前が不正です");
      }
    });
  });
```

実際に採用してみて

実際に採用してみて

- よかった点
  - 楽かつシンプルな実装できる点はやっぱりよかった
  - メンバーのキャッチアップは楽にできる
  - （個人的には）フロント側でエラーを意識せずにクエリを書かなくていいので認知負荷は低い
- 少し残念な点
  - extensions の自由度が高いのでチーム内でルールをちゃんと決める必要がありそう
    - エラーコードはなんでもいいので、「USER_INVALID」でも「INVALID_USER」でも同じ意味なのに複数のエラーコードができてしまう可能性がある
  - 属性ごとの細かいエラーを返す場合は、コードがすぐに複雑になる
    - 例: name のエラー、address のエラーそれぞれを分けて表示したいなど

まとめ

まとめ

- GraphQL のエラーレスポンスには複数の設計アプローチがあってそれぞれにメリデリがある
  - エラーレスポンス設計をする際は、技術的な面だけでなく**プロダクトの性質やチーム状況に合わせて設計方針を固める**と良さそう
- 個人的には標準実装はおすすめです
  - 小規模チームでかつ、フルスタックエンジニアのチームであれば、エラーをスキーマ外で管理するデメリットがそんなに大きいとは感じてない
  - 手軽に実装できるのが本当に楽
  - ただし、extensions属性は自由度が高いので、**チーム内でルールは決めた方が良さそう**です

ご清聴ありがとうございました

ぜひ意思決定の過程や、エラーレスポンス設計周りについてのフィードバックもらえると嬉しいです！

参考文献

- GraphQL Spec
- Production Ready GraphQL
- GraphQL Server とエラー処理
- Shopify GraphQL Design Tutorial