Red Hat Managed Integration 2

データ同期アプリケーションの開発

Red Hat Managed Integration 2 向け

Last Updated: 2022-06-16

Red Hat Managed Integration 2 データ同期アプリケーションの開発

Red Hat Managed Integration 2 向け

Enter your first name here. Enter your surname here.
Enter your organisation's name here. Enter your organisational division here.
Enter your email address here.

法律上の通知

概要

本書では、データ同期アプリケーションである Red Hat Managed Integration 2 を作成するための包括的な説明と使用方法について説明します。

目次

はじめに

サーバーと API を提供する他のモバイルサービスとは異なり、Data Sync はサービスの開発を可能にするフレームワークです。通常、Data Sync サービスは次のように開発します。

1. GraphQL スキーマを設計します。
2. 必要な機能を備えた Data Sync サーバーと Data Sync クライアントを開発します。
3. Data Sync サーバーをコンテナー化し、OpenShift にデプロイします。
4. Data Sync サーバーを指すようにモバイルアプリケーションを設定します。
5. モバイルアプリケーションの開発を完了します。
6. モバイルアプリケーションをビルドして実行します。

第1章 はじめに

1.1. DATA SYNC の紹介

Data Sync は、開発者がモバイルクライアントと Web クライアントの両方にリアルタイムのデータ同期を追加できるようにする JavaScript フレームワークです。Data Sync フレームワークは、クライアントがオフラインで動作し続けることを可能にするオフライン機能も提供し、接続が再確立されると、クライアントは自動的に同期されます。Data Sync フレームワークを使用して構築されたアプリケーションは、通常、データの永続性のためにデータソースに接続します。ただし、Data Sync フレームワークを使用して構築されたアプリケーションは、データソースがなくても機能します。

Data Sync フレームワークを使用して構築されたアプリケーションは、次の 2 つのコンポーネントで構成されています。

- Data Sync クライアントは、クライアント側のエクステンションとサーバー側の統合を提供する JavaScript クライアントです。Data Sync クライアントは、React や Angular などのフレームワークに統合できます。

- Data Sync サーバーは、Node.js ベースの GraphQL API を構築するためのフレームワークです。Data Sync サーバーは、データのセキュリティー、インテグリティー、およびモニタリングを保証するためのエンタープライズエクステンションを提供します。既存の Node.js アプリケーションに統合できます。

Data Sync フレームワークは、GraphQL 実装として Apollo platform を使用します。

関連情報

- モバイルクライアントと Web クライアント間でのリアルタイムのデータ同期。
    - Websocket を使用すると、複数の Data Sync クライアント間でリアルタイムのデータ同期が可能になります。Data Sync クライアントは、競合検出が Data Sync サーバーによって処理されるため、ローカルデータを明示的にクエリーすることなく、Data Sync サーバーから更新を受信します。

- Data Sync クライアントは、接続状態に関係なく、任意の操作を実行できます。
    - ネットワーク接続が懸念される場合、Data Sync クライアントは、接続状態に関係なく、任意の操作を実行できます。Data Sync クライアントは、オンラインでもオフラインでも同じ操作を実行できます。この機能により、Data Sync を安全に使用してビジネスクリティカルなアプリケーションを作成できます。

- 完全にカスタマイズ可能な競合の検出と解決を開発者に提供します。
    - Data Sync を使用すると、ユーザーは Data Sync サーバー上の競合を検出して解決できるため、さまざまな Data Sync クライアントにデータをシームレスに送信できます。Data Sync では、開発者がこの戦略を採用したい場合に、Data Sync クライアントで競合を解決することもできます。

- インスタント同期クエリーは、開発者にインスタントフィードバックを提供します。
    - Data Sync クライアントがオンラインの場合、インスタントクエリーを使用すると、開発者はエラーにすばやく対応し、操作の実行時に結果をユーザーに表示できます。開発者は Data Sync サーバーから即時の応答またはエラーを取得できますが、Data Sync クライアントは Data Sync サーバーに接続している必要があります。

- 柔軟なデータソース。

- Data Sync は、クラウドストレージ、MongoDB や PostgreSQL などのデータベース、既存のバックエンドデータソースなど、さまざまなデータソースに接続できます。

1.2. DATA SYNC の技術概要

このセクションでは、Data Sync の技術的な側面について説明します。

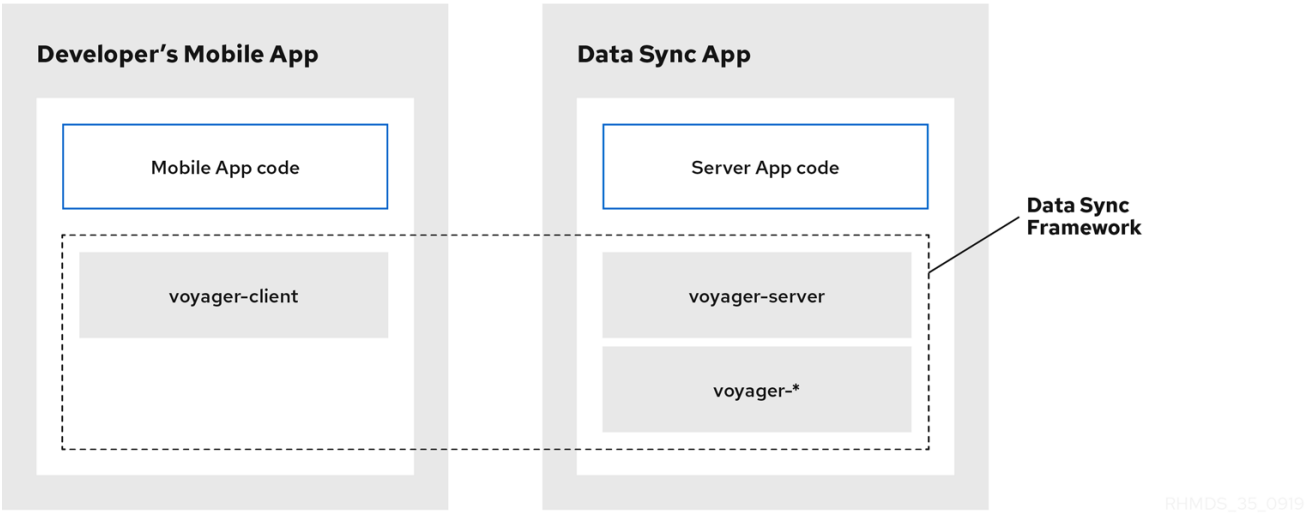


表1.1 Data Sync のケーススタディ

コンポーネント	技術的ロール
同期クライアント	Sync クライアントは、Web およびモバイルアプリケーションケーションの構築に使用されるクライアント側の JavaScript ライブラリーです。Sync サーバーとの連携が簡単にできるようになります。
Sync サーバー	Sync サーバーは Apollo Server フレームワークに基づいており、2 つの主要な機能を実行します。データソースからデータを送受信し、Sync クライアント間でデータを同期します。Sync サーバーは GraphQL を使用してカスタム接続を作成し、それによってさまざまなタイプの Sync クライアントが接続できるようにします。
データソース	データソースはデータを保存します。このデータは通常、Sync クライアント間で同期されるものです。

Apollo Server フレームワークの詳細は、ここから開始して Apollo プラットフォームについて学習してください。

1.3. DATA SYNC の用語

このセクションでは、Data Sync に関連する用語について説明します。

Data Sync の用語

GraphQL

API のクエリー言語、およびタイプシステムを使用するクエリーを実行するためのサーバー側ランタイムです。詳細については、GraphQL を参照してください。

Apollo

Apollo は、最新のデータ駆動型アプリケーションを構築する製品エンジニアリングチームのニーズに合わせて設計された GraphQL の実装です。Apollo には、Apollo Server と Apollo Client の 2 つのオープンソースライブラリーが含まれています。Data Sync フレームワークは、Apollo 機能を活用します。

Sync Server

Sync Server は、Node.js ベースの GraphQL API を構築するためのフレームワークです。

同Sync Client

Sync クライアントは、クライアント側のエクステンションとサーバー側の統合を提供する JavaScript クライアントです。Sync Client は、React や Angular などのフレームワークに統合できます。

データソース

Data Sync フレームワークは通常、データの永続性のためにデータソースと組み合わせて使用されます。ただし、Data Sync フレームワークを使用して構築されたアプリケーションは、データソースがなくても機能します。

Data Sync フレームワーク

Data Sync は、開発者がモバイルクライアントと Web クライアントの両方のデータをリアルタイムで同期する機能を追加できるようにする JavaScript フレームワークです。

関連情報

- Learn GraphQL
- Voyager Server GitHub repository
- Voyager Client GitHub repository
- Apollo Server
- Apollo Client

1.4. HELLO WORLD DATA SYNC の使用を開始する

この例では、Data Sync Server ライブラリーを Express node.js プロジェクトに追加し、**index-1.js** ファイルを作成し、サーバーを実行して、GraphQL にクエリーを実行します。

- Data Sync Server は、Data Sync サーバーの構築に使用できる Node.js ライブラリーのセットです。
- Data Sync Server は、Data Sync アプリケーションを開発するための開始点です。

前提条件

- Node.js と npm がインストールしました。
- **node.js** プロジェクトを作成しました。

手順

1. Node.js アプリケーションにライブラリーを追加します。

```
$ npm install graphql (1)
$ npm install express (2)
$ npm install @aerogear/voyager-server (3)
```

(1) See https://graphql.org/

(2) https://expressjs.com/ を参照します。

(3) Data Sync を可能にする Data Sync Server ライブラリー

2. 次のコンテンツで **index-1.js** ファイルを作成します。

```
const express = require('express')
//Include our server libraries
const { VoyagerServer, gql } = require('@aerogear/voyager-server')

//Provide your graphql schema
const typeDefs = gql`
  type Query {
    hello: String
  }
`

//Create the resolvers for your schema
const resolvers = {
  Query: {
    hello: (obj, args, context, info) => {
      return `Hello world`
    }
  }
}

//Initialize the library with your Graphql information
const apolloServer = VoyagerServer({
  typeDefs,
  resolvers
})

//Connect the server to express
const app = express()
apolloServer.applyMiddleware({ app })

app.listen(4000, () =>
  console.log(`  Server ready at http://localhost:4000/graphql`)
)
```

3. サーバーを実行します。

```
$ node index-1.js

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

4. **http://localhost:4000/graphql** を参照してプレイグラウンドを操作します。以下に例を示します。

```
{
  hello
}
```

5. 出力を確認します。上記の例では、出力は次のようになります。

```
{
  "data": {
    "hello": "Hello world"
  }
}
```

Data Sync フレームワークの使用を開始するには、sample application を参照してください。このアプリでは、より複雑なスキーマを探索できます。

先に進む前に、次の GraphQL の概念を理解していることを確認してください。

- スキーマ設計
- リゾルバー
- サブスクリプション

第2章 DATA SYNC クライアントを使用した DATA SYNC サーバーのクエリー

このセクションでは、Voyager Client を使用して、Voyager サーバーアプリケーションと通信できるモバイルおよび Web アプリケーションを作成する方法について説明します。

Data Sync は、JavaScript アプリケーションを Data Sync も使用するサーバーと統合する JavaScript ライブラリーを提供します。クライアントライブラリーは、Apollo client に基づいています。

ライブラリーをモバイルプロジェクトに追加し、クライアントクラスを設定し、サーバーに接続して、それが機能することを確認します。

前提条件

- Node.js と npm がインストールしました。
- webpack getting started ガイドを使用して、ES6 をサポートする空の Web プロジェクトを作成しました。
- サーバー入門ガイドを完了し、アプリケーションを実行しました。

手順

1. アドレス帳サーバーを作成します。
   a. 次のコンテンツで **index-2.js** ファイルを作成します。

```
const express = require('express')
//Include our server libraries
const { VoyagerServer, gql } = require('@aerogear/voyager-server')

//Provide your graphql schema
const typeDefs = gql`
type Query {
  info: String!
  addressBook: [Person!]!
}

type Mutation {
  post(name: String!, address: String!): Person!
}

type Person {
  id: ID!
  address: String!
  name: String!
}
`

let persons = [{
  id: 'person-0',
  name: 'Alice Roberts',
  address: '1 Red Square, Waterford'
}]
```

```
let idCount = persons.length
const resolvers = {
  Query: {
    info: () => `This is a simple example`,
    addressBook: () => persons,
  },
  Mutation: {

    post: (parent, args) => {
      const person = {
       id: `person-${idCount++}`,
       address: args.address,
       name: args.name,
      }
      persons.push(person)
      return person
    }
  },
}

//Initialize the library with your Graphql information
const apolloServer = VoyagerServer({
  typeDefs,
  resolvers
})

//Connect the server to express
const app = express()
apolloServer.applyMiddleware({ app })

app.listen(4000, () =>
  console.log(`  Server ready at http://localhost:4000/graphql`)
)
```

b. サーバーを実行します。

```
$ node index-2.js

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

c. **http://localhost:4000/graphql** を参照してプレイグラウンドを操作します。以下に例を示します。

```
{
  addressBook {
    name
    address

  }
}
```

d. 出力を確認します。上記の例では、出力は次のようになります。

```
{
  "data": {
```

```
    "addressBook": [
     {
      "name": "Alice Roberts",
      "address": "1 Red Square, Waterford"
     }
    ]
   }
}
```

2. 次のライブラリーを JavaScript クライアントに追加します。

```
npm install @aerogear/voyager-client
npm install graphql
npm install graphql-tag
```

注記

前提条件は、webpack getting started ガイドを使用して、ES6 をサポートする空の Web プロジェクトを作成していることです。

3. **index.js** ファイルを作成して、手順 1 と同じクエリーを作成しますが、JavaScript を使用します。
この例では、設定オブジェクトが作成され、**httpUrl** フィールドが Voyager サーバーアプリケーションの URL に設定されています。クライアントアプリケーションがサブスクリプションを使用する場合は、**wsUrl** フィールドも必要です。

src/index.js

```
// gql is a utility function that handles gql queries
import gql from 'graphql-tag';

import { OfflineClient } from '@aerogear/voyager-client';

// connect to the local service.
let config = {
  httpUrl: "http://localhost:4000/graphql",
  wsUrl: "ws://localhost:4000/graphql",
}

async function queryPeople() {

  // Actually create the client
  let offlineClient = new OfflineClient(config);
  let client = await offlineClient.init();

  // Execute the query
  client.query({
     fetchPolicy: 'network-only',
     query: gql`
     query addressBook{
       addressBook{
       name
       address
       }
```

```
      }
      `
    })
    //Print the response of the query
    .then( ({data}) => {
      console.log(data.addressBook)
    });
}

queryPeople();
```

4. クライアントアプリケーションをビルドして実行します。

5. クライアントアプリケーションを参照し、コンソール出力を確認します。
次のような配列を含める必要があります。

```
address: "1 Red Square, Waterford"
name: "Alice Roberts"
__typename: "Person"
```

第3章 DATA SYNC クライアントにミューテーションを追加する

前提条件

- Node.js と npm がインストールしました。
- Queries section を完了しましたが、サーバーはまだ実行中です。

手順

1. ミューテーションを実行するようにクライアントアプリケーションを変更します。

   src/index.js

```
// gql is a utility function that handles gql queries
import gql from 'graphql-tag';

import { OfflineClient } from '@aerogear/voyager-client';

// connect to the local service.
let config = {
  httpUrl: "http://localhost:4000/graphql",
  wsUrl: "ws://localhost:4000/graphql",
}

async function addPerson() {

  // Actually create the client
  let offlineClient = new OfflineClient(config);
  let client = await offlineClient.init();

  // Execute the mutation
  client.mutate({
      mutation: gql`
       mutation {
         post(name: "John Doe", address: "1 Red Hill") {
           id
         }
       }
       `
    })
    //Print the response of the query
    .then( ({data}) => {
      console.log(data)
    });
}

addPerson();
```

2. クライアントアプリケーションをビルドして実行します。
3. クライアントアプリケーションを参照し、コンソール出力を確認します。
   次のような配列を含める必要があります。

```
{
  "data": {
    "post": {
      "id": "person-1"
    }
  }
}
```

4. http://localhost:4000/graphql を参照し、アドレス帳のプレイグラウンドクエリーを入力します。以下に例を示します。

```
{
  addressBook {
    name
    address

  }
```

結果は以下のようになります。

```
{
  "data": {
    "addressBook": [
      {
        "name": "Alex Smith",
        "address": "1 Square Place, City"
      },
      {
        "name": "John Doe",
        "address": "1 Red Hill"
      }
    ]
  }
}
```

第4章 モバイルアプリケーションでオフライン機能をサポートする

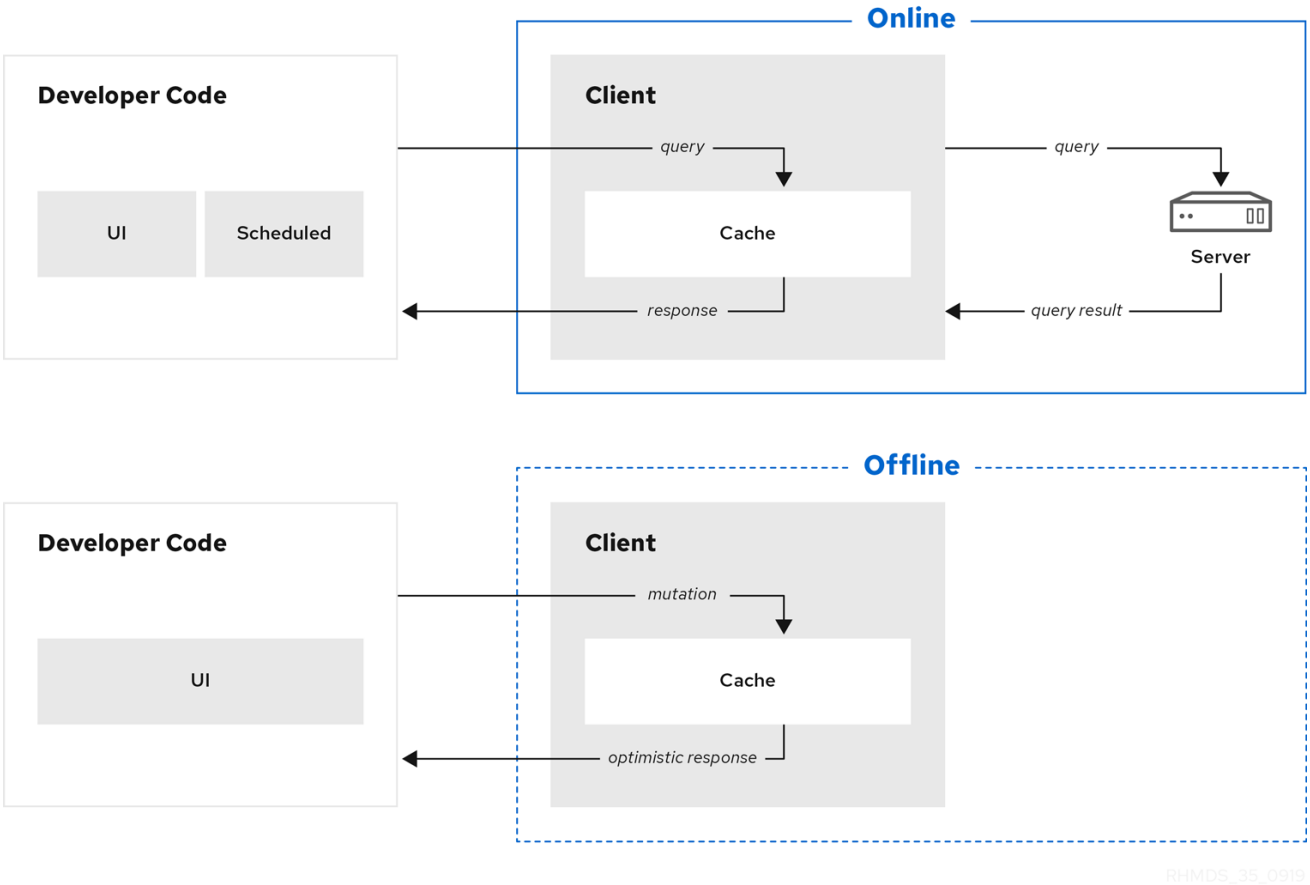
4.1. オフライン機能について

モバイルアプリケーションはオフラインで実行でき、ユーザーは @aerogear/voyager モジュールを使用してミューテーションをクエリーおよび作成できます。

すべてのクエリーはキャッシュに対して実行され、ミューテーションストア (またはオフラインストア) はオフラインミューテーションをサポートします。

クライアントが長期間オフラインになると、ミューテーションストアは競合解決戦略を使用してサーバーとローカル更新をネゴシエートします。

クライアントが再びオンラインになると、ミューテーションがサーバーにレプリケートされます。



開発者はリスナーを接続して、サーバーに適用された更新または失敗した更新に関する通知を受け取り、適切なアクションを実行できます。

ミューテーションとローカルキャッシュ

デフォルトでは、クエリーとミューテーションの結果がキャッシュされます。

ミューテーションはクエリー結果を変更する可能性があります。必ず **refetchQueries** を呼び出すか、**mutate** メソッドのオプションを **update** して、ローカルキャッシュが最新の状態に保たれるようにしてください。

@aerogear/voyager-client モジュールは、「キャッシュ更新ヘルパーの使用」 で説明するように、必要なコードの量を減らすためのキャッシュヘルパー機能も提供します。

mutate と使用可能なオプションの詳細は、Apollo's document about mutations を参照してください。

4.2. オフラインクライアントの作成

@aerogear/voyager-client モジュールは、次の機能を公開する **OfflineClient** クラスを提供します。

- ミューテーションストアへの直接アクセス
- 「イベントを聞く」で説明されているように、複数のオフラインイベントリスナーを登録できます。
- モバイルアプリケーションのローカルキャッシュが最新の状態に保たれることを自動的に保証します。「キャッシュ更新ヘルパーの使用」で説明されているように、このクライアントは **update** メソッドを自動的に生成します。

クライアントを作成するには:

```
import { OfflineClient } from '@aerogear/voyager-client';

let config = {
  httpUrl: "http://localhost:4000/graphql",
  wsUrl: "ws://localhost:4000/graphql",
}

async function setupClient() {

  let offlineClient = new OfflineClient(config);
  let client = await offlineClient.init();
}

setupClient();
```

このクライアントは、同じ機能をサポートしているため、Apollo クライアントを置き換えることができます。

第5章 オフライン中にミューテーションを検出する

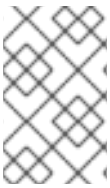
デバイスがオフラインのときにミューテーションが発生した場合、**client.mutate** 関数は次のようになります。

- 即座に返します。
- エラーのある promise を返します

error オブジェクトをチェックして、オフライン状態に関連するエラーを分離できます。**error** オブジェクトで **watchOfflineChange()** メソッドを呼び出すと、オフラインの変更がサーバーと同期されるのを監視し、トリガーされると通知を送信します。

以下に例を示します。

```
client.mutate(...).catch((error)=> {
  // 1. Detect if this was an offline error
 if(error.networkError && error.networkError.offline){
   const offlineError: OfflineError =  error.networkError;
   // 2. We can still track when offline change is going to be replicated.
   offlineError.watchOfflineChange().then(...)
 }
});
```



注記

個々のミューテーションを監視することに加えて、「イベントを聞く」 で説明されているように、クライアントを作成するときにグローバルオフラインリスナーを追加できます。

第6章 オフライン中にミューテーションを実行する

@aerogear/voyager-client モジュールは、Apollo の mutate 関数をいくつかの追加機能で拡張する **offlineMutate** メソッドを提供します。これには、各操作のコンテキストにいくつかのフィールドを自動的に追加することが含まれます。

オフラインクライアントを設定するには、「オフラインクライアントの作成」 を参照してください。

設定が完了すると、**offlineMutate** を使用できるようになります。

注記

offlineMutate メソッドは、**mutate** と同じパラメーターを受け入れますが、いくつかの追加のオプションパラメーターも使用できます。

```
const { CacheOperation } = require('@aerogear/voyager-client');

client.offlineMutate({
  ...
  updateQuery: GET_TASKS, (1)
  operationType: CacheOperation.ADD, (2)
  idField: "id", (3)
  returnType: "Task" (4)
  ...
})
```

1. ミューテーションの結果で更新する必要がある 1 つまたは複数のクエリー。
2. 実行されている操作のタイプ。追加、更新、または削除する必要があります。指定しない場合、デフォルトで追加になります。
3. オブジェクトを識別するために使用されるオブジェクトのフィールド。指定しない場合、デフォルトは id です。
4. 操作対象のオブジェクトのタイプ。

6.1. サポートアプリケーションはオフライン中に再起動します

Apollo クライアントは、すべてのミューテーションパラメーターをメモリーに保持します。オフラインの Apollo クライアントは、ミューテーションパラメーターを保存し続け、オンラインになると、すべてのミューテーションをメモリーに復元します。ミューテーションに提供される更新関数は、Apollo クライアントによってキャッシュできず、再起動後にすべての楽観的な応答が失われます。ミューテーションに提供された **Update functions** は、キャッシュに保存できません。その結果、すべての **optimisticResponses** は再起動後にアプリケーションから消え、Apollo クライアントがオンラインになってサーバーと正常に同期したときにのみ再表示されます。

再起動後にすべての **optimisticResponses** が失われるのを防ぐために、すべての **optimisticResponses** を復元するように **Update Functions** を設定できます。

```
const updateFunctions = {
  // Can contain update functions from each component
  ...ItemUpdates,
  ...TasksUpdates
```

```
}

let config = {
  mutationCacheUpdates: updateFunctions,
}
```

getUpdateFunction を使用して、関数を自動的に生成することもできます。

```
const { createMutationOptions, CacheOperation } = require('@aerogear/voyager-client');

const updateFunctions = {
  // Can contain update functions from each component
  createTask: getUpdateFunction({
      mutationName: 'createTask',
      idField: 'id',
      updateQuery: GET_TASKS,
      operationType: CacheOperation.ADD
    }),
  deleteTask: getUpdateFunction({
      mutationName: 'deleteTask',
      idField: 'id',
      updateQuery: GET_TASKS,
      operationType: CacheOperation.DELETE
    })
}

let config = {
  ...
  mutationCacheUpdates: updateFunctions,
  ...
}
```

6.2. 指定されたミューテーションがオンラインでのみ実行されるようにする

クライアントがオンラインのときにのみ特定のミューテーションが実行されるようにする場合は、次のように GraphQL ディレクティブ **@onlineOnly** を使用します。

```
exampleMutation(...) @onlineOnly {
  ...
}
```

6.3. イベントを聞く

オフライン関連のイベントに関するすべての通知を処理するには、設定オブジェクトで **offlineQueueListener** リスナーを使用します

次のイベントが出力されます。

- **onOperationEnqueued** - 新しい操作がオフラインキューに追加されるときに呼び出されます
- **onOperationSuccess** - オンラインに戻って操作が成功したときに呼び出されます
- **onOperationFailure** - オンラインに戻り、GraphQL エラーで操作が失敗したときに呼び出されます

- **queueCleared** - オフライン操作キューがクリアされたときに呼び出されます

このリスナーを使用して、保留中の変更を表示するユーザーインターフェイスを構築できます。

6.4. キャッシュ更新ヘルパーの使用

@aerogear/voyager-client モジュールは、アプリケーションのキャッシュへの更新を管理するためのすぐに使用できるソリューションを提供します。ミューテーションとサブスクリプションの両方のキャッシュ更新メソッドをインテリジェントに生成できます。

6.4.1. ミューテーションにキャッシュ更新ヘルパーを使用する

次の例は、これらのヘルパーメソッドをミューテーションに使用する方法を示しています。これらのメソッドを使用するには、「オフラインクライアントの作成」で説明されているようにオフラインクライアントを作成し、**offlineMutate** メソッドを使用します。**offlineMutate** 関数は、**MutationHelperOptions** オブジェクトをパラメーターとして受け入れます。

```
const { createMutationOptions, CacheOperation } = require('@aerogear/voyager-client');

const mutationOptions = {
  mutation: ADD_TASK,
  variables: {
    title: 'item title'
  },
  updateQuery: {
    query: GET_TASKS,
    variables: {
      filterBy: 'some filter'
    }
  },
  typeName: 'Task',
  operationType: CacheOperation.ADD,
  idField: 'id'
};
```

updateQuery パラメーターに配列を提供することにより、キャッシュを更新するための複数のクエリーを提供することもできます。

```
const mutationOptions = {
  ...
  updateQuery: [
    { query: GET_TASKS, variables: {} }
  ]
  ,
  ...
};
```

次の例は、**mutationOptions** オブジェクトを使用してタスクを追加するためのオフラインミューテーションを準備する方法と、クライアントのキャッシュの **GET_TASK** クエリーを更新する方法を示しています。

```
const { createMutationOptions, CacheOperation } = require('@aerogear/voyager-client');

client.offlineMutate<Task>(mutationOptions);
```

オフラインクライアントを使用したくない場合は、**createMutationOptions** 関数を直接使用することもできます。この関数は、既存のクライアントに渡す Apollo 互換の **MutationOptions** オブジェクトを提供します。次の例は、**mutationOptions** が前のコード例と同じオブジェクトである場合にこの関数を使用する方法を示しています。

```
const options = createMutationOptions(mutationOptions);

client.mutate<Task>(options);
```

6.4.2. サブスクリプションにキャッシュ更新ヘルパーを使用する

@aerogear/voyager-client モジュールは、Apollo Client の **subscribeToMore** 関数で使用するために必要なオプションを生成できるサブスクリプションヘルパーを提供します。

このヘルパーを使用するには、最初にいくつかのオプションを作成する必要があります。次に例を示します。

```
const { CacheOperation } = require('@aerogear/voyager-client');

const options = {
  subscriptionQuery: TASK_ADDED_SUBSCRIPTION,
  cacheUpdateQuery: GET_TASKS,
  operationType: CacheOperation.ADD
}
```

このオプションオブジェクトは、**TASK_ADDED_SUBSCRIPTION** のために受信したすべてのデータオブジェクトについて、**GET_TASKS** クエリーもキャッシュ内で最新の状態に保つ必要があることをサブスクリプションヘルパーに通知します。

次に、必要なキャッシュ更新関数を作成できます。

```
const { createSubscriptionOptions } = require('@aerogear/voyager-client');

const subscriptionOptions = createSubscriptionOptions(options);
```

このヘルパーを使用するには、この **subscriptionOptions** 変数を **ObservableQuery** の **subscribeToMore** 関数に渡します。

```
const query = client.watchQuery<AllTasks>({
  query: GET_TASKS
});

query.subscribeToMore(subscriptionOptions);
```

データの重複排除を自動的に実行している間、キャッシュは最新の状態に保たれます。

6.4.3. 複数のサブスクリプションにキャッシュ更新ヘルパーを使用する

@aerogear/voyager-client モジュールは、**ObservableQuery** で **subscribeToMore** を自動的に呼び出す機能を提供します。これは、1 つのクエリーに影響を与える可能性のある複数のサブスクリプションがある場合に役立ちます。たとえば、**TaskAdded**、**TaskDeleted**、および **TaskUpdated** サブスクリプションがある場合は、3 つの個別の **subscribeToMore** 関数呼び出しが必要です。これを回避するには、@aerogear/voyager-client モジュールの **subscribeToMoreHelper** 関数を使用して、サブスクリプションの配列とそれに対応するクエリーを渡すことにより、これを自動的に処理します。

```
const { CacheOperation } = require('@aerogear/voyager-client');

const addOptions = {
  subscriptionQuery: TASK_ADDED_SUBSCRIPTION,
  cacheUpdateQuery: GET_TASKS,
  operationType: CacheOperation.ADD
}

const deleteOptions = {
  subscriptionQuery: TASK_DELETED_SUBSCRIPTION,
  cacheUpdateQuery: GET_TASKS,
  operationType: CacheOperation.DELETE
}

const updateOptions = {
  subscriptionQuery: TASK_UPDATED_SUBSCRIPTION,
  cacheUpdateQuery: GET_TASKS,
  operationType: CacheOperation.REFRESH
}

const query = client.watchQuery<AllTasks>({
  query: GET_TASKS
});

subscribeToMoreHelper(query, [addOptions, deleteOptions, updateOptions]);
```

第7章 ネットワークステータスの検出

NetworkStatus インターフェイスを使用して、現在のネットワークステータスを確認するか、ネットワークのステータスが変更されたときにアクションを実行するリスナーを登録します。

2 つのデフォルトの実装が提供されています。

- Web ブラウザーの **WebNetworkStatus**
- Cordova の **CordovaNetworkStatus**

次の例は、**CordovaNetworkStatus** を使用してリスナーを登録する方法を示しています。

```
import { CordovaNetworkStatus, NetworkInfo } from '@aerogear/voyager-client';
const networkStatus = new CordovaNetworkStatus();

networkStatus.onStatusChangeListener({
  onStatusChange: info => {
    const online = info.online;
    if (online) {
      //client is online, perform some actions
    } else {
      //client is offline
    }
  }
});

let config = {
  ...
  networkStatus: networkStatus,
  ...
};

//create a new client using the config
```

第8章 モバイルアプリケーションでのリアルタイム更新のサポート

8.1. リアルタイム更新の概要

いくつかのクエリーとミューテーションを開発した後、リアルタイムの更新を実装することをお勧めします。

リアルタイム更新は、**Subscription** と呼ばれる操作タイプによって GraphQL 仕様でサポートされています。実稼働環境でサブスクリプションをサポートするために、Data Sync は MQTT PubSub サブスクリプションメカニズムを使用してサブスクリプションを実装します。ただし、Apollo PubSub モジュールを使用して、概念実証アプリケーションを開発することをお勧めします。

リアルタイム更新をコーディングするときは、次のモジュールを使用します。

- @aerogear/voyager-server - voyager-client を使用して GraphQL クエリーとミューテーションを有効にするクライアントをサポートします
- @aerogear/voyager-subscriptions - voyager-client を使用して GraphQL サブスクリプションを有効にするクライアントをサポートします
- @aerogear/graphql-mqtt-subscriptions - MQTT ブローカーへの GraphQL リゾルバー接続をサポートします

GraphQL サブスクリプションを使用すると、クライアントは WebSocket 接続を介してサーバーイベントにサブスクライブできます。

フローは次のように要約できます。

- クライアントは WebSocket を使用してサーバーに接続し、特定のイベントをサブスクライブします。
- イベントが発生すると、サーバーはそれらのイベントにサブスクライブしているクライアントに通知します。
- **現在接続している**クライアントのうち、指定されたイベントを購読しているものは、更新を受け取ることができます。
- クライアントはいつでも接続を閉じることができ、更新を受信しなくなります。

更新を受信するには、クライアントが現在サーバーに接続されている必要があります。クライアントは、オフラインの間、サブスクリプションからイベントを受信しません。非アクティブなクライアントをサポートするには、プッシュ通知を使用します。

関連情報

- GraphQL サブスクリプションの詳細は、Subscriptions documentation を参照してください。

8.2. DATA SYNC サーバーでのリアルタイム更新の実装

次のコードは、サブスクリプションのないData Sync サーバーの一般的なコードを示しています。

```
const apolloServer = VoyagerServer({
  typeDefs,
  resolvers
})
```

```
const app = express()
apolloServer.applyMiddleware({ app })

app.listen({ port }, () =>
  console.log(`  Server ready at http://localhost:${port}${apolloServer.graphqlPath}`)
)
```

次のセクションでは、リアルタイム更新を有効にするために必要な手順の概要を説明します。

1. SubscriptionServer を実装する
2. パブリッシュサブスクライブメカニズムを実装する
3. スキーマでサブスクリプションを定義する
4. リゾルバーを実装する

8.2.1. voyager-subscription を使用した SubscriptionServer の実装

スキーマに GraphQL サブスクリプションタイプを作成できるようにするには:

1. **@aerogear/voyager-subscriptions** パッケージをインストールします。

```
$ npm i @aerogear/voyager-subscriptions
```

2. **@aerogear/voyager-subscriptions** を使用して SubscriptionServer を設定します

```
const { createSubscriptionServer } = require('@aerogear/voyager-subscriptions')

const apolloServer = VoyagerServer({
  typeDefs,
  resolvers
})

const app = express()
apolloServer.applyMiddleware({ app })
const port = 4000

const server = app.listen({ port }, () => {
  console.log(`  Server ready at http://localhost:${port}${apolloServer.graphqlPath}`)

  createSubscriptionServer({ schema: apolloServer.schema }, {
    server,
    path: '/graphql'
  })
})
```

createSubscriptionServer コード:

- **SubscriptionServer** インスタンスを返します
- ハンドラーをインストールします
  - WebSocket 接続を管理する

- サーバーでサブスクリプションを配信する

- **@aerogear/voyager-keycloak** などの他のモジュールとの統合を提供します。

関連情報

- 引数とオプションの詳細は、subscriptions-transport-ws モジュールを参照してください。

8.2.2. パブリッシュサブスクライブメカニズムの実装

警告

この手順では、プロトタイプ化には役立ちますが、実稼働環境には適さないメモリー内実装について説明します。Red Hat は、実稼働環境で MQTT PubSub を使用することをお勧めします。すべての実装方法の詳細は、「パブリッシュサブスクライブメカニズムの設定」を参照してください。

apollo-server によって提供されるデフォルトの **PubSub** を使用してクライアントに更新をプッシュするチャネルを提供するには、次のようにパブリッシュサブスクライブメカニズムを実装します。

```
const { PubSub } = require('apollo-server')

const pubsub = new PubSub()
```

追加情報

サブスクリプションは、サブスクリプションを通知するイベントを生成するために、publish subscribe メカニズムに依存します。**PubSubEngine** インターフェイスに基づいて利用可能な several PubSub implementations があります。

8.2.3. スキーマでのサブスクリプションの定義

サブスクリプションはルートレベルタイプです。これらは、**Query** および **Mutation** と同様のスキーマで定義されています。たとえば、次のスキーマでは、**Task** タイプが定義されており、ミューテーションとサブスクリプションも定義されています。

```
type Subscription {
  taskCreated: Task
}

type Mutation {
  createTask(title: String!, description: String!): Task
}

type Task {
  id: ID!
  title: String!
  description: String!
}
```

8.2.4. リゾルバーの実装

リゾルバーマップ内で、サブスクリプションリゾルバーはイベントをリッスンする **AsyncIterator,** を返します。イベントを生成するには、**publish** メソッドを呼び出します。**pubsub.publish** コードは通常、ミューテーションリゾルバー内にあります。

次の例では、新しいタスクが作成されると、**createTask** リゾルバーがこの変更の結果を **TaskCreated** チャネルに公開します。

```
const TASK_CREATED = 'TaskCreated'

const resolvers = {
  Subscription: {
    taskCreated: {
      subscribe: () => pubSub.asyncIterator(TASK_CREATED)
    }
  },
  Mutation: {
    createTask: async (obj, args, context, info) => {
      const task = tasks.create(args)
      pubSub.publish(TASK_CREATED, { taskCreated: task })
      return task
    }
  },
}
```

注記

このサブスクリプションサーバーは、認証または承認を実装していません。認証と承認の実装は、Supporting authentication and authorization in your mobile app を参照してください。

関連情報

- クライアントコードでサブスクリプションを使用する方法は、Realtime Updates を参照してください。

8.3. パブリッシュサブスクライブメカニズムの設定

開発には Apollo PubSub メカニズムを使用できますが、実稼働環境には MQTT PubSub メカニズムを使用する必要があります。

8.3.1. Apollo PubSub メカニズムの使用

このセクション (「Data Sync サーバーでのリアルタイム更新の実装」) では、**apollo-server** によって提供されるデフォルトの **PubSub** を設定する方法について説明します。実稼働のシステムの場合は、MQTT PubSub を使用します。

8.3.2. MQTT PubSub メカニズムの使用

@aerogear/graphql-mqtt-subscriptions モジュールは implementing subscription resolvers に使用される **AsyncIterator** インターフェイスを提供します。Data Sync サーバーを MQTT ブローカーに接続して、水平方向にスケーラブルなサブスクリプションをサポートします。

MQTT クライアントを初期化し、そのクライアントを **@aerogeaar/graphql-mqtt-subscriptions** モジュールに渡します。次に例を示します。

```
const mqtt = require('mqtt')
const { MQTTPubSub } = require('@aerogear/graphql-mqtt-subscriptions')

const client = mqtt.connect('mqtt://test.mosquitto.org', {
  reconnectPeriod: 1000,
})

const pubsub = new MQTTPubSub({
  client
})
```

この例では、**mqtt.connect** を使用して **mqtt** クライアントが作成され、このクライアントが **MQTTPubSub** インスタンスに渡されます。その後、**pubsub** インスタンスを使用して、サーバー内のイベントをパブリッシュおよびサブスクライブできます。

関連情報

- mqtt.connect documentation.
- MQTTPubSub documentation

8.4. MQTT メッセージング用の AMQ オンラインの設定

Red Hat AMQ は MQTT プロトコルをサポートしているため、大規模な GraphQL サブスクリプションを強化するための適切な PubSub テクノロジーになります。

このセクションでは、次の推奨事項を示します。

- MQTT メッセージング用の AMQ Online の設定。
- AMQ Online に接続し、サーバーアプリケーション内で pubsub として使用します。

用語

- AMQ Online は、開発者が OpenShift 内で Red Hat AMQ の機能を利用できるようにするためのメカニズムです。
- Red Hat AMQ は、インターネット向けアプリケーションに対して高速で軽量でセキュアなメッセージングを提供します。AMQ Broker は複数のプロトコルと高速メッセージの永続性をサポートします。
- MQTT は MQ Telemetry Transport の略です。パブリッシュサブスクライブ方式で、極めてシンプルかつ軽量なメッセージングプロトコルです。

AMQ Online には、アプリケーションの特定のニーズに対応する多くの設定オプションが含まれています。MQTT メッセージングに AMQ Online を使用し、GraphQL サブスクリプションを有効にするための最小設定手順は次のとおりです。

1. **AddressSpace** を作成する
2. **Address** を作成する

3. **MessagingUser** を作成する

8.4.1. アドレス空間の作成

ユーザーは、**AddressSpace** を作成することでメッセージングリソースをリクエストできます。アドレス空間には、**standard** と **brokered** の 2 種類があります。MQTT ベースのアプリケーションには、**brokered** アドレス空間を使用する必要があります。

手順

1. アドレス空間を作成します。たとえば、次のリソースは、ブローカー化された **AddressSpace** を作成します。

```
apiVersion: enmasse.io/v1beta1
kind: AddressSpace
metadata:
  name: myaddressspace
spec:
  type: brokered
  plan: brokered-single-broker
```

2. **AddressSpace** を作成します。

```
oc create -f brokered-address-space.yaml
```

3. アドレス空間のステータスを確認します。

```
oc get <`AddressSpace` name> -o yaml
```

出力には、アプリケーションの接続に必要な詳細など、アドレス空間に関する情報が表示されます。

関連情報

- 詳細は、Creating address spaces using the command line を参照してください。

8.4.2. アドレスの作成

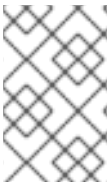
アドレスは **AddressSpace** の一部であり、メッセージを送受信するための宛先を表します。MQTT トピックを表すには、タイプ **topic** の **Address** を使用します。

1. アドレス定義を作成します。

```
apiVersion: enmasse.io/v1beta1
kind: Address
metadata:
   name: myaddressspace.myaddress # must have the format <`AddressSpace` name>.
<address name>
spec:
   address: myaddress
   type: topic
   plan: brokered-topic
```

2. アドレスを作成します。

```
oc create -f topic-address.yaml
```



注記

pubsub.asyncIterator() の使用の詳細については、Configuring your server for real-time updates ガイドを参照してください。**pubsub.asyncIterator()** に渡されるトピック名ごとにアドレスを作成します。

関連情報

- 詳細は、Creating addresses using the command line を参照してください。

8.4.3. AMQ オンラインユーザーの作成

メッセージングクライアントは、MessagingUser とも呼ばれる AMQ オンラインユーザーを使用して接続します。**MessagingUser** は、使用できるアドレスとそれらのアドレスで実行できる操作を制御する許可ポリシーを指定します。

ユーザーは **MessagingUser** リソースとして設定されます。ユーザーは、作成、削除、読み取り、更新、および一覧表示できます。

1. ユーザー定義を作成します。

```
apiVersion: user.enmasse.io/v1beta1
kind: MessagingUser
metadata:
  name: myaddressspace.mymessaginguser # must be in the format <`AddressSpace`
name>.<username>
spec:
  username: mymessaginguser
  authentication:
    type: password
    password: cGFzc3dvcmQ= # must be Base64 encoded. Password is 'password'
  authorization:
    - addresses: ["*"]
      operations: ["send", "recv"]
```

2. **MessagingUser** を作成します。

```
oc create -f my-messaging-user.yaml
```

これで、アプリケーションはこのユーザーの資格情報を使用して AMQ オンラインアドレスに接続できます。

詳細は、AMQ Online User Model を参照してください。

8.5. AMQ ONLINE での GRAPHQL MQTT PUBSUB の使用

前提条件

MQTT アプリケーションでは、次の AMQ オンラインリソースを利用できます

- AddressSpace
- アドレス
- MessagingUser

このセクションでは、**@aerogear/graphql-mqtt-subscriptions** を使用して AMQ オンライン **Address** に接続する方法について説明します。

1. 使用する **AddressSpace** の接続の詳細を取得します。

```
oc get addressspace <addressspace> -o yaml
```

2. アドレスへの接続に使用する方法を決定します。

- サービスホスト名の使用 - クライアントが OpenShift クラスター内から接続できるようにします。
  Red Hat は、OpenShift 内で実行されているアプリケーションがサービスホスト名を使用して接続することを推奨しています。サービスのホスト名には、OpenShift クラスター内でのみアクセスできます。これにより、アプリケーションと AMQ Online の間でルーティングされるメッセージは、OpenShift クラスター内にとどまり、パブリックインターネットにアクセスすることはありません。

- 外部ホスト名の使用 - クライアントが OpenShift クラスターの外部から接続できるようにします。
  外部ホスト名により、OpenShift クラスターの外部からの接続が可能になります。これは、以下の場合に役に立ちます。

  - OpenShift 接続およびメッセージの公開の外部で実行される実稼働アプリケーション。
  - クイックプロトタイピングとローカル開発。非実稼働の **AddressSpace** を作成し、開発者がローカル環境からアプリケーションに接続できるようにします。

3. サービスのホスト名を使用して AMQ オンライン**Address** に接続するには

a. サービスのホスト名を取得します。

```
oc get addressspace <addressspace name> -o jsonpath='{.status.endpointStatuses[?(@.name=="messaging")].serviceHost
```

b. 接続を作成するコードを追加します。次に例を示します。

```
const mqtt = require('mqtt')
const { MQTTPubSub } = require('@aerogear/graphql-mqtt-subscriptions')

const client = mqtt.connect({
  host: '<internal host name>',
  username: '<MessagingUser name>',
  password: '<MessagingUser password>',
  port: 5762,
})

const pubsub = new MQTTPubSub({ client })
```

c. アプリケーションと AMQ Online ブローカー間のすべてのメッセージを暗号化するには、次のように TLS を有効にします。

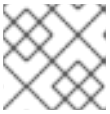
```
const mqtt = require('mqtt')
const { MQTTPubSub } = require('@aerogear/graphql-mqtt-subscriptions')

const host = '<internal host name>'

const client = mqtt.connect({
  host: host,
  servername: host,
  username: '<MessagingUser name>',
  password: '<MessagingUser password>',
  port: 5761,
  protocol: 'tls',
  rejectUnauthorized: false,
})

const pubsub = new MQTTPubSub({ client })
```

4. 外部ホスト名を使用して AMQ オンライン **Address** に接続するには:



注記

外部ホスト名は通常、TLS 接続のみを受け入れます。

a. 外部ホスト名を取得します。

```
oc get addressspace <addressspace name> -o jsonpath='{.status.endpointStatuses[?
(@.name=="messaging")].externalHost
```

b. 外部ホスト名に接続します。例を以下に示します。

```
const mqtt = require('mqtt')
const { MQTTPubSub } = require('@aerogear/graphql-mqtt-subscriptions')

const host = '<internal host name>'

const client = mqtt.connect({
  host: host,
  servername: host,
  username: '<MessagingUser name>',
  password: '<MessagingUser password>',
  port: 443,
  protocol: 'tls',
  rejectUnauthorized: false,
})

const pubsub = new MQTTPubSub({ client })
```

5. TLS を使用する場合は、次の追加の **mqtt.connect** オプションに注意してください。

- **servername** - TLS を使用して OpenShift のメッセージブローカーに接続する場合、このプロパティーを設定する必要があります。設定しないと、メッセージがプロキシー経由でルーティングされ、クライアントに複数の証明書が提示されるため、接続が失敗します。**servername** を設定することにより、クライアントは Server Name Indication (SNI) を使用して、TLS 接続設定の一部として正しい証明書をリクエストします。

- **protocol** - **'tls'** に設定する必要があります
- **rejectUnauthorizated** - false に設定する必要があります。そうしないと、接続が失敗します。これは、証明書エラーを無視するようにクライアントに指示します。繰り返しますが、これが必要なのは、クライアントに複数の証明書が提示され、証明書の 1 つがリクエストされているホスト名とは異なるホスト名のものであり、通常はエラーが発生するためです。
- **port** - サービスホスト名の場合は 5761 に、外部ホスト名の場合は 443 に設定する必要があります。

8.5.1. 設定での環境変数の使用

Red Hat では、接続に環境変数を使用することをお勧めします。次に例を示します。

```
const mqtt = require('mqtt')
const { MQTTPubSub } = require('@aerogear/graphql-mqtt-subscriptions')

const host = process.env.MQTT_HOST || 'localhost'

const client = mqtt.connect({
  host: host,
  servername: host,
  username: process.env.MQTT_USERNAME,
  password: process.env.MQTT_PASSWORD,
  port: process.env.MQTT_PORT || 1883,
  protocol: process.env.MQTT_PROTOCOL || 'mqtt',
  rejectUnauthorized: false,
})

const pubsub = new MQTTPubSub({ client })
```

この例では、接続オプションは環境変数を使用して設定できますが、**host**、**port**、および **protocol** の適切なデフォルトがローカル開発用に提供されています。

8.5.2. MQTT 接続の問題のトラブルシューティング

8.5.2.1. MQTT イベントのトラブルシューティング

mqtt モジュールは、実行時にさまざまなイベントを出力します。通常の操作とトラブルシューティングのために、これらのイベントのリスナーを追加することをお勧めします。

```
client.on('connect', () => {
  console.log('client has connected')
})

client.on('reconnect', () => {
  console.log('client has reconnected')
})

client.on('offline', () => {
  console.log('Client has gone offline')
})
```

```
client.on('error', (error) => {
  console.log(`an error has occurred ${error}`)
})
```

MQTT documentation を読んで、すべてのイベントとその原因について学習してください。

8.5.2.2. MQTT 設定の問題のトラブルシューティング

アプリケーションで接続エラーが発生している場合、確認する最も重要なことは、設定が **mqtt.connect** に渡されることです。アプリケーションはローカルまたは OpenShift で実行される可能性があるため、内部または外部のホスト名を使用して接続する場合があり、TLS を使用する場合と使用しない場合があります。誤って間違った設定を提供することは非常に簡単です。

hostname や **port** などのパラメーターが正しくない場合、Node.js **mqtt** モジュールはエラーを報告しません。代わりに、通知せずに失敗し、メッセージング機能なしでアプリケーションを起動できるようになります。

アプリケーションでこのシナリオを処理する必要がある場合があります。次の回避策を使用できます。

```
const TIMEOUT = 10 // number of seconds to wait before checking if the client is connected

setTimeout(() => {
  if (!client.connected) {
    console.log(`client not connected after ${TIMEOUT} seconds`)
 // process.exit(1) if you wish
  }
}, TIMEOUT * 1000)
```

このコードは、MQTT クライアントが接続されていないかどうかを検出するために使用できます。これは、潜在的な設定の問題を検出するのに役立ち、アプリケーションがそのシナリオに対応できるようにします。

8.6. クライアントでのリアルタイム更新の実装

GraphQL 仕様のコアコンセプトは、**Subscription** と呼ばれる操作タイプであり、リアルタイム更新のメカニズムを提供します。GraphQL サブスクリプションの詳細は、Subscriptions documentation を参照してください。

これを行うために、GraphQL サブスクリプションは WebSocket を利用して、クライアントが公開された変更をサブスクライブできるようにします。

WebSocket のアーキテクチャーは次のとおりです。

- クライアントは WebSocket サーバーに接続します。
- 特定のイベントが発生すると、サーバーはこれらのイベントの結果を WebSocket に公開できます。
- その WebSocket に **現在接続されている** クライアントは、これらの結果を受け取ります。
- クライアントはいつでも接続を閉じることができ、更新を受信しなくなります。

WebSocket は、現在アクティブなクライアントにメッセージを配信するための完璧なソリューションです。更新を受信するには、クライアントが現在 WebSocket サーバーに接続されている必要があります。クライアントがオフラインのときにこの WebSocket を介して行われた更新は、クライアントに

よって消費されません。このユースケースでは、プッシュ通知をお勧めします。

Voyager Client には、デバイスの再起動時の自動再接続やネットワークの再接続など、すぐに使用できるサブスクリプションサポートが付属しています。クライアントでサブスクリプションを有効にするには、Voyager Client 設定オブジェクトで次のパラメーターを設定します。DataSyncConfig インターフェイスを使用したい場合は、Voyager Client からも利用できます。

8.6.1. サブスクリプションを使用するようにクライアントを設定する

サブスクリプションを使用するようにクライアントを設定するには:

1. 次のように、config オブジェクトに **wsUrl** 文字列を指定します。

```
const config = {
    wsUrl: "ws://<your_websocket_url>"
}
```

ここで、**<your_websocket_url>** は、GraphQL サーバーの WebSocket エンドポイントの完全な URL です。

2. 手順 1 のオブジェクトを使用して、Voyager Client を初期化します。

```
const { createClient } = require("@aerogear/voyager-client");

const client = createClient(config)
```

8.6.2. サブスクリプションの使用

サブスクリプションを利用するための標準的なフローは次のとおりです。

1. サーバーからデータを取得するためのネットワーククエリーを作成します
2. クエリーの変更についてキャッシュを監視する
3. サーバーからプッシュされた変更をサブスクライブする
4. アクティブなサブスクリプションがあるビューを離れるときは、サブスクライブを解除してください

以下の 3 つの例では、**subscribeToMore** により、サーバーからさらに更新を受信すると、サーバーからの **subscriptionData** を使用して updateQuery 関数が強制的に呼び出されます。

subscribeToMore を使用すると、すべての GraphQL クエリーが自動的に通知されるため、キャッシュが簡単に更新されます。

詳細は、subscribeToMore documentation を参照してください。

```
getTasks() {
  const tasks = client.watchQuery({
    query: GET_TASKS
  });

  tasks.subscribeToMore({
    document: TASK_ADDED_SUBSCRIPTION,
    updateQuery: (prev, { subscriptionData }) => {
```

```
    // Update logic here.
    }
  });
  return tasks;
}
```

Voyager クライアントが **updateQuery** 関数を自動的に生成できるようにするには、Cache Update Helpers セクションを参照してください。

次に、アプリケーションでこのクエリーを使用して変更をサブスクライブできるため、サーバーから新しいデータが返されたときにフロントエンドが常に更新されます。

```
this.tasks = [];
this.getTasks().subscribe(result => {
  this.tasks = result.data && result.data.allTasks;
})
```

ページを離れるときにクエリーのサブスクライブを解除することもお勧めします。これにより、メモリーリークの可能性が防止されます。これは、次の例に示すように unsubscribe() を呼び出すことで実行できます。このコードは適切な場所に配置する必要があります。

```
this.getTasks().unsubscribe();
```

8.6.3. ネットワーク状態の変化の処理

サブスクリプションを使用してクライアントにリアルタイムの更新を提供する場合、クライアントがオフラインのときにサーバーが更新されるとクライアントが同期しなくなるため、ネットワークの状態を監視することが重要です。

これを回避するために、Voyager Client は、**NetworkInfo** インターフェイスと一緒に使用してネットワーク状態のカスタムチェックを実装できる **NetworkStatus** インターフェイスを提供します。

次の例を使用して、クライアントがオンライン状態に戻った後にクエリーを再実行します。

```
const { CordovaNetworkStatus, NetworkInfo } = require("@aerogear/voyager-client");
const networkStatus = new CordovaNetworkStatus();

networkStatus.onStatusChangeListener({
  onStatusChange(networkInfo: NetworkInfo) {
    const online = networkInfo.online;
    if (online) {
      client.watchQuery({
        query: GET_TASKS
      });
    }
  }
});
```

第9章 モバイルアプリケーションでの認証と承認のサポート

9.1. RED HAT SINGLE SIGN-ON を使用した認証と承認のためのサーバーの設定

keycloak サービスと @aerogear/voyager-keycloak モジュールを使用して、Data Sync Server アプリケーションにセキュリティーを追加することができます。

@aerogear/voyager-keycloak モジュールは、すぐに使用できる次の機能を提供します。

- 認証 - 認証されたユーザーのみが、メインの GraphQL エンドポイントを含むサーバーエンドポイントにアクセスできるようにします。
- 承認 - GraphQL スキーマ内で **@hasRole()** ディレクティブを使用して、GraphQL レベルでロールベースのアクセス制御 (RBAC) を実装します。
- GraphQL コンテキストとの統合 - GraphQL リゾルバー内の **context** オブジェクトを使用して、ユーザークレデンシャルといくつかのヘルパー関数にアクセスします。

前提条件

- Red Hat Single Sign-On サービスを利用できます。
- 有効な **keycloak.json** 設定ファイルをプロジェクトに追加する必要があります。
  - Keycloak 管理コンソールでアプリケーションのクライアントを作成します。
  - インストールタブをクリックします。
  - フォーマットオプションで **Keycloak OIDC JSON** を選択し、**Download** をクリックします。

9.1.1. Red Hat Single Sign-On を使用した Data Sync サーバーの保護

手順

1. **@aerogear/voyager-keycloak** モジュールをインポートします

```
const { KeycloakSecurityService } = require('@aerogear/voyager-keycloak')
```

2. Keycloak 設定を読み取り、それを渡して **KeycloakSecurityService** を初期化します。

```
const keycloakConfig = JSON.parse(fs.readFileSync(path.resolve(__dirname,
'./path/to/keycloak.json')))
const keycloakService = new KeycloakSecurityService(keycloakConfig)
```

3. **keycloakService** インスタンスを使用して、アプリケーションを保護します。

```
const app = express()
keycloakService.applyAuthMiddleware(app)
```

4. **keycloakService** がセキュリティーサービスとして使用されるように Voyager サーバーを設定します。

```
const voyagerConfig = {
  securityService: keycloakService
}
const server = VoyagerServer(apolloConfig, voyagerConfig)
```

Keycloak Example Server Guide には、上記の手順に基づいたサンプルサーバーがあり、サーバーを実行するために必要なすべての手順が示されています。

9.1.2. スキーマでの hasRole ディレクティブの使用

Voyager Keycloak モジュールは、スキーマでロールベースの承認を定義するための **@hasRole** ディレクティブを提供します。**@hasRole** ディレクティブは、以下に適用できる特別なアノテーションです。

- fields
- クエリー
- ミューテーション
- サブスクリプション

@hasRole の使用法は次のとおりです。

- **@hasRole(role: String)**
- 例 - **@hasRole(role: "admin"])**
- 認証されたユーザーが **admin** のロールを持っている場合、それらは承認されます。
- **@hasRole(role: [String])**
- 例 - **@hasRole(role: ["admin", "editor"])**
- 認証されたユーザーがリスト内のロールの少なくとも 1 つを持っている場合、それらは許可されます。

デフォルトの動作は、クライアントのロールをチェックすることです。たとえば、**@hasRole(role: "admin")** は、ユーザーが **admin** というクライアントロールを持っていることを確認します。**@hasRole(role: "realm:admin")** は、そのユーザーが **admin** というレルムロールを持っているかどうかを確認します

レルムのロールをチェックするための構文は、**@hasRole(role: "realm:<role>")** です。たとえば、**@hasRole(role: "realm:admin")**。ロールのリストを使用して、クライアントとレルムの両方のロールを同時にチェックすることができます。

例: @hasRole ディレクティブを使用してスキーマにロールベースの承認を適用する

次の例は、**@hasRole** ディレクティブを使用して、GraphQL スキーマのさまざまな部分でロールベースの承認を定義する方法を示しています。このサンプルスキーマは、ニュースやブログの Web サイトなどのアプリケーションの公開を表しています。

```
type Post {
  id: ID!
  title: String!
  author: Author!
  content: String!
```

```
  createdAt: Int!
}

type Author {
  id: ID!
  name: String!
  posts: [Post]!
  address: String! @hasRole(role: "admin")
  age: Int! @hasRole(role: "admin")
}

type Query {
  allPosts:[Post]!
  getAuthor(id: ID!):Author!
}

type Mutation {
  editPost:[Post]! @hasRole(role: ["editor", "admin"])
  deletePost(id: ID!):[Post] @hasRole(role: "admin")
}
```

2 つのタイプがあります:

- **Post** - 記事またはブログ投稿
- **Author** - 投稿を作成した人を表します

2 つのクエリーがあります:

- **allPosts** - 投稿のリストを返します
- **getAuthor** - 作成者に関する詳細を返します

2 つのミューテーションがあります:

- **editPost** - 既存の投稿を編集します
- **deletePost** - 投稿を削除します。

クエリーとミューテーションに関するロールベースの承認

サンプルスキーマでは、**@hasRole** ディレクティブが **editPost** および **deletePost** ミューテーションに適用されています。クエリーでも同じことができます。

- **editPost** ミューテーションを実行できるのは、ロール **editor** および/または **admin** を持つユーザーのみです。
- **admin** のロールを持つユーザーのみが **deletePost** ミューテーションを実行できます。

この例は、**@hasRole** ディレクティブをさまざまなクエリーやミューテーションで使用する方法を示しています。

フィールドでのロールベースの承認

スキーマの例では、**Author** タイプには、フィールド **address** と **age** があり、両方に **hasRole(role: "admin")** が適用されています。

これは、ロール **admin** を持たないユーザーは、クエリーまたはミューテーションでこれらのフィールドをリクエストすることが許可されていないことを意味します。

たとえば、管理者以外のユーザーは **getAuthor** クエリーを実行できますが、**address** または **age** フィールドをリクエストすることはできません。

9.2. RED HAT SINGLE SIGN-ON を使用した WEBSOCKET を介した認証

要件:

- 認証と承認のために Data Sync サーバーを設定する
- リアルタイム更新用のサーバーの設定

このセクションでは、Red Hat Single Sign-On を使用して WebSocket で認証と承認を実装する方法について説明します。WebSocket を介した認証の詳細は、Apollo の Authentication Over Websocket のドキュメントを参照してください。

Voyager クライアントは、最初の WebSocket メッセージで送信される **connectionParams** へのトークン情報の追加をサポートしています。サーバーでは、このトークンを使用して接続を認証し、サブスクリプションを続行できるようにします。Red Hat Single Sign-On Authentication in Voyager Client のセクションを読んで、Red Hat Single Sign-On トークンがサーバーに送信されていることを確認してください。

サーバーでは、**createSubscriptionServer** は、標準の **SubscriptionServer** に渡すことができる通常のオプションに加えて **SecurityService** インスタンスを受け入れます。**@aerogear/voyager-keycloak** の **KeycloakSecurityService** は、最初の WebSocket メッセージでクライアントから渡された Red Hat Single Sign-On トークンを検証するために使用されます。

```
const { createSubscriptionServer } = require('@aerogear/voyager-subscriptions')
const { KeycloakSecurityService } = require('@aerogear/voyager-keycloak')
const keycloakConfig = require('./keycloak.json') // typical Keycloak OIDC installation

const apolloServer = VoyagerServer({
  typeDefs,
  resolvers
})

securityService = new KeycloakSecurityService(keycloakConfig)

const app = express()

keycloakService.applyAuthMiddleware(app)
apolloServer.applyMiddleware({ app })

const server = app.listen({ port }, () =>
  console.log(`  Server ready at http://localhost:${port}${apolloServer.graphqlPath}`)

  createSubscriptionServer({ schema: apolloServer.schema }, {
    securityService,
    server,
    path: '/graphql'
  })
)
```

この例は、Red Hat Single Sign-On **securityService** が作成される方法と、それが **createSubscriptionServer** に渡される方法を示しています。これにより、すべてのサブスクリプションで Red Hat Single Sign-On 認証が有効になります。

9.2.1. サブスクリプションでの Red Hat Single Sign-On 認証

Red Hat Single Sign-On **securityService** は、クライアントから送信されたトークンを検証し、Token Object へ解析します。このトークンは、**context.auth** を使用してサブスクリプションリゾルバーで使用でき、よりきめ細かいロールベースのアクセス制御を実装するために使用できます。

```
const resolvers = {
  Subscription: {
    taskAdded: {
      subscribe: (obj, args, context, info) => {
        const role = 'admin'
        if (!context.auth.hasRole(role)) {
          return new Error(`Access Denied - missing role ${role}`)
        }
        return pubSub.asyncIterator(TASK_ADDED)
      }
    },
}
```

上記の例は、サブスクリプションリゾルバー内のロールベースのアクセス制御を示しています。**context.auth** は完全な Keycloak Token Object です。つまり、**hasRealmRole** や **hasApplicationRole** などのメソッドを使用できます。

ユーザーの詳細には、**context.auth.content** からアクセスできます。以下は例です。

```
{
  "jti": "dc1d6286-c572-43c1-99c7-4f36982b0e56",
  "exp": 1561495720,
  "nbf": 0,
  "iat": 1561461830,
  "iss": "http://localhost:8080/auth/realms/voyager-testing",
  "aud": "voyager-testing-public",
  "sub": "57e1dcda-990f-4cc2-8542-0d1f9aae302b",
  "typ": "Bearer",
  "azp": "voyager-testing-public",
  "nonce": "552c3cba-a6c2-490a-9914-28784ba0e4bc",
  "auth_time": 1561459720,
  "session_state": "ed35e1b4-b43c-438f-b1a3-18b1be8c6307",
  "acr": "0",
  "allowed-origins": [
    "*"
  ],
  "realm_access": {
    "roles": [
      "developer",
      "uma_authorization"
    ]
  },
  "resource_access": {
    "voyager-testing-public": {
      "roles": [
```

```
        "developer"
      ]
    },
    "account": {
      "roles": [
        "manage-account",
        "manage-account-links",
        "view-profile"
      ]
    }
  },
  "preferred_username": "developer"
}
```

ユーザーの詳細 (例えば **context.auth.content.sub** は認証されたユーザーの ID) にアクセスできることは、サブスクリプションフィルターを 実装し、ユーザーの詳細に基づいてより細かい pubsub トピックにサブスクライブすることが可能であることを意味します。

9.3. クライアントに認証と承認を実装する

Voyager Client を使用すると、ヘッダーを使用する方法とトークンを使用する方法の 2 つの方法でユーザー情報を Data Sync サーバーアプリケーションに渡すことができます。

ヘッダーは、サーバーへの HTTP リクエストを認証するために使用されます。これは、クエリーとミューテーションに使用されます。

トークンは、サブスクリプションに使用される WebSocket 接続を認証するために使用されます。

どちらの方法も、**authContextProvider** 設定オプションで設定できます。以下に例を示します。

```
//get the token value from somewhere, for example the authentication service
const token = "REPLACE_WITH_REAL_TOKEN";

const config = {
  ...
  authContextProvider: function() {
    return {
      header: {
        "Authorization": `Bearer ${token}`
      },
      token: token
    }
  },
  ...
};

//create a new client
```

サーバーで認証と承認を実行する方法は、Server Authentication and Authorization Guide を参照してください。

第10章 DATA SYNC アプリケーションの競合を解決する

10.1. はじめに

モバイルアプリケーションを使用すると、ユーザーはオフラインでデータを変更できます。これにより、競合が発生する可能性があります。

2 人以上のユーザーが同じデータを変更しようとすると、**conflict** が発生します。システムは、競合するデータを解決する必要があります。

競合の解決は、次の 2 つのフェーズで処理できます。

- **Conflict detection** は、誤ったデータが保存されている可能性を検出するアプリケーションの機能です。
- **Conflict resolution** は、正しいデータが保存されていることを確認するプロセスです。

Red Hat Data Sync の場合:

- 競合検出は、ミューテーションに関連付けられたコードでのみ実装します。
- Data Sync Server モジュールは、サーバー側で競合検出を提供します。
- Voyager Client モジュールは、クライアント側で競合解決を提供します。

10.2. サーバーでの競合の検出

競合を検出するための一般的なフローには、次の手順が含まれます。

1. **A Mutation Occurs** - クライアントが GraphQL ミューテーションを使用してサーバー上のオブジェクトを変更または削除しようとします
2. **Read the Object** - サーバーは、クライアントが変更しようとしている現在のオブジェクトをデータソースから読み取ります
3. **Conflict Detection** - サーバーは、現在のオブジェクトをクライアントから送信されたデータと比較して、競合があるかどうかを確認します。開発者は、比較の実行方法を選択します。

aerogear/voyager-conflicts モジュールは、ストレージテクノロジーに関係なく、開発者が **Conflict Detection** 手順を実行するのに役立ちますが、データのフェッチと保存は開発者の責任です。

このリリースは、次の実装をサポートします。

- **VersionedObjectState** - オブジェクトで提供されるバージョンフィールドに依存します (conflictHandler をインポートするときにデフォルトでバージョンフィールドが使用されます)。詳細は、以下を参照してください: 「バージョンベースの競合検出の実装」
- **HashObjectState** - オブジェクト全体から計算されたハッシュに依存します。詳細は、以下を参照してください: 「ハッシュベースの競合検出の実装」

これらの実装は **ObjectState** インターフェイスに基づいており、そのインターフェイスを拡張して、競合検出用のカスタム実装を提供できます。

前提条件

- リゾルバーを備えた GraphQL サーバー。
- データの競合を引き起こす可能性のあるデータベースまたはその他の形式のデータストレージ。Red Hat は、データをセキュアな場所に保存することをお勧めします。データベースを使用する場合、そのデータベースを管理、保守、およびバックアップするのはユーザーの責任です。他の形式のデータストレージを使用する場合は、データをバックアップする責任があります。

10.2.1. バージョンベースの競合検出の実装

バージョンベースの競合解決は、競合の検出と解決に推奨される最もシンプルなアプローチです。中心的な考え方は、すべてのオブジェクトに整数値の **version** プロパティーがあるということです。クライアントから送信されたバージョン番号がサーバーに保存されているバージョンと一致しない場合、**conflict** が発生します。これは、別のクライアントがすでにオブジェクトを更新していることを意味します。

手順

1. @aerogear/voyager-conflicts パッケージをインポートします。

```
const { conflictHandler } = require('@aerogear/voyager-conflicts')
```

2. 競合解決をサポートするバージョンフィールドを GraphQL タイプに追加します。バージョンもデータストレージに保存する必要があります。

```
type Task {
  title: String
  version: Int
}
```

3. ミューテーションの例を追加します。

```
type Mutation {
  updateTask(title: String!, version: Int!): Task
}
```

4. リゾルバーを実装します。すべての競合は、次のような事前定義された一連の手順を使用して処理できます。

```
// 1. Read data from data source
const serverData = db.find(clientData.id)
// 2. Check for conflicts
const conflict = conflictHandler.checkForConflicts(serverData, clientData)
// 3. If there is a conflict, return the details to the client
if(conflict) {
    throw conflict;
}
// 4. Save object to data source
db.save(clientData.id, clientData)
```

上記の例では、**throw** ステートメントにより、クライアントが競合クライアント側を解決するために必要なすべてのデータを確実に受信します。このデータの詳細は、Structure of the Conflict Error を参照してください。

競合はクライアントで解決されるため、データを保持する必要はありません。ただし、競合がない場合は、クライアントから送信されたデータを保持する必要があります。クライアント側の競合の解決の詳細は、Resolving Conflicts on the Client を参照してください。

10.2.2. ハッシュベースの競合検出の実装

ハッシュベースの競合検出は、クライアントによって更新されているオブジェクト **total** に基づいて競合を検出するメカニズムです。これは、各オブジェクトをハッシュ化し、ハッシュを比較することによって行われます。これは、オブジェクトが同等であり、競合がないと見なすことができるかどうかをサーバーに通知します。

手順

1. @aerogear/voyager-conflicts パッケージをインポートします。

```
const { HashObjectState } = require('@aerogear/voyager-conflicts')
```

2. **HashObjectState** 実装を使用する場合は、ハッシュ関数を提供する必要があります。関数のシグネチャは次のようになります。

```
const hashFunction = (object) {
  // Using the Hash library of your choice
  const hashedObject = Hash(object)
  // return the hashedObject in string form
  return hashedObject;
}
```

3. **HashObjectState** をインスタンス化するときにこの関数を提供します。

```
const conflictHandler = new HashObjectState(hashFunction)
```

4. リゾルバーを実装します。すべての競合は、次のような事前定義された一連の手順を使用して処理できます。

```
// 1. Read data from data source
const serverData = db.find(clientData.id)
// 2. Check for conflicts
const conflict = conflictHandler.checkForConflicts(serverData, clientData)
// 3. If there is a conflict, return the details to the client
if(conflict) {
    throw conflict;
}
// 4. Save object to data source
db.save(clientData.id, clientData)
```

上記の例では、**throw** ステートメントにより、クライアントが競合クライアント側を解決するために必要なすべてのデータを確実に受信します。このデータの詳細は、Structure of the Conflict Error を参照してください。

競合はクライアントで解決されるため、データを保持する必要はありません。ただし、競合がない場合は、クライアントから送信されたデータを保持する必要があります。クライアント側の競合の解決の詳細は、Resolving Conflicts on the Client を参照してください。

10.2.3. 競合エラーの構造について

サーバーとクライアントの両方の状態を含む競合が検出された場合、サーバーは特定のエラーを返す必要があります。これにより、クライアントは競合を解決できます。

```
"extensions": {
      "code": "INTERNAL_SERVER_ERROR",
      "exception": {
        "conflictInfo": {
          "serverState": {
              //..
          },
          "clientState": {
            //..
          }
        },
      }
}
```

10.3. クライアントでの競合の解決

競合を解決するための一般的なフローには、次の手順が含まれます。

1. **ミューテーションが発生する** - クライアントは、GraphQL ミューテーションを使用してサーバー上のオブジェクトを変更または削除しようとします。
2. **オブジェクトの読み取り** - サーバーは、クライアントが変更しようとしている現在のオブジェクトをデータソース (通常はデータベース) から読み取ります。
3. **競合の検出** - サーバーは、現在のオブジェクトをクライアントから送信されたデータと比較して、競合があったかどうかを確認します。競合がある場合、サーバーは Structure of the Conflict Error で概説されている情報を含む応答をクライアントに返します。
4. **競合の解決** - クライアントはこの競合を解決しようとし、このデータが競合しなくなることを期待してサーバーに新しいリクエストを行います。

競合解決の実装には、アプリケーションに次の追加が必要です。

- ミューテーションのコンテキストに追加された **returnType**。Working With Conflict Resolution on the Client を参照してください。
- 選択した競合の実装に応じて、タイプ内の追加のメタデータ (バージョンフィールドなど)。Version Based Conflict Detection を参照してください。
- 競合を最初にクライアントに返すサーバー側リゾルバー。詳細は、Server Side Conflict Detection を参照してください。

開発者は、デフォルトの競合解決の実装を使用するか、競合解決のメカニズムを使用して独自の競合解決を実装できます。

デフォルトでは、同じフィールドに変更が加えられていない場合、実装は変更されたペイロードをサーバーに再送信しようとします。サーバーとクライアントでの変更が同じフィールドに影響を与える場合、指定された競合解決のストラテジーの 1 つを使用できます。デフォルトのストラテジーでは、サーバー上のデータに加えてクライアントの変更が適用されます。開発者は、ニーズに合わせて戦略を変更できます。

10.3.1. クライアントでの競合解決の実装

競合解決を有効にするには、競合検出を実行するようにサーバー側リゾルバーを設定する必要があります。検出はさまざまな実装に依存し、競合エラーをクライアントに返すことができます。詳細は、Server Side Conflict Detection を参照してください。

手順

競合を解決するには、ミューテーションコンテキストに **returnType** パラメーターを指定します。このパラメーターは、操作対象のオブジェクトタイプを定義します。これは、次の 2 つの方法で実装できます。

- Data Sync の **offlineMutate** を使用している場合は、次のように **returnType** パラメーターを直接指定できます。

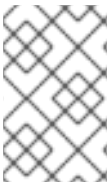
```
client.offlineMutate({
  ...
  returnType: 'Task'
  ...
})
```

- Apollo の **mutate** 関数を使用する場合は、次のように **returnType** パラメーターを指定します。

```
client.mutate({
  ...
  context: {
    returnType: 'Task'
  }
  ...
})
```

クライアントは、現在のストラテジーに基づいて競合を自動的に解決し、必要に応じてリスナーに通知します。

競合の解決は推奨されるデフォルトで機能し、クライアントで特定の処理を行う必要はありません。



注記

高度なユースケースの場合、アプリケーション設定でカスタム **conflictProvider** を指定することにより、競合の実装をカスタマイズできます。以下の Conflict Resolution Strategies を参照してください。

10.3.2. デフォルトの競合実装

デフォルトでは、競合解決は、各 GraphQL タイプの **version** フィールドに依存するように設定されています。サーバー上の変更を検出するには、バージョンフィールドをデータベースに保存する必要があります。以下に例を示します。

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

バージョンフィールドはサーバー上で制御され、サーバーから送信された最後のバージョンをマップします。バージョンフィールドに対するすべての操作は自動的に行われます。競合解決をサポートするミューテーションについては、バージョンフィールドが常にサーバーに渡されるようにしてください。

```
type Mutation {
  updateUser(id: ID!, version: String!): User
}
```

10.3.3. 競合解決戦略の実装

Data Sync を使用すると、開発者はカスタムの競合解決戦略を定義できます。提供されている **ConflictResolutionStrategies** タイプを使用して、設定内のクライアントにカスタム競合解決戦略を提供できます。デフォルトでは、**UseClient** が使用されるため、開発者は戦略を渡す必要はありません。カスタム戦略を使用して、特定の操作にさまざまな解決戦略を提供することもできます。

```
let customStrategy = {
  resolve = (base, server, client, operationName) => {
    let resolvedData;
    switch (operationName) {
      case "updateUser":
        delete client.socialKey
        resolvedData = Object.assign(base, server, client)
        break
      case "updateRole":
        client.role = "none"
        resolvedData = Object.assign(base, server, client)
        break
      default:
        resolvedData = Object.assign(base, server, client)
    }
    return resolvedData
  }
}
```

このカスタム戦略オブジェクトは、2 つの異なる戦略を提供します。戦略は、操作に一致するように名前が付けられています。オブジェクトの名前を、設定オブジェクトの conflictStrategy への引数として渡します。

```
let config = {
...
  conflictStrategy: customStrategy
...
}
```

10.3.4. 競合のリッスン

Data Sync を使用すると、開発者はデータの競合に関する情報を受け取ることができます。

競合が発生すると、Data Sync はデータのフィールドレベルの解決を実行しようとします。そのタイプのすべてのフィールドをチェックして、クライアントまたはサーバーの両方が同じフィールドを変更したかどうかを確認します。クライアントは、2 つのシナリオのいずれかで通知を受けることができます。

- クライアントとサーバーの両方が同じフィールドのいずれかを変更した場合、**ConflictListener** の **conflictOccurred** メソッドがトリガーされます。
- クライアントとサーバーが同じフィールドのいずれも変更しておらず、データを簡単にマージできる場合は、**ConflictListener** の **mergeOccurred** メソッドがトリガーされます。

開発者は、次のように独自の **conflictListener** 実装を提供できます。

```
class ConflictLogger implements ConflictListener {
  conflictOccurred(operationName, resolvedData, server, client) {
    console.log("Conflict occurred with the following:")
    console.log(`data: ${JSON.stringify(resolvedData)}, server: ${JSON.stringify(server)}, client:
${JSON.stringify(client)}, operation:  ${JSON.stringify(operationName)}`);
  }
  mergeOccurred(operationName, resolvedData, server, client) {
    console.log("Merge occurred with the following:")
    console.log(`data: ${JSON.stringify(resolvedData)}, server: ${JSON.stringify(server)}, client:
${JSON.stringify(client)}, operation:  ${JSON.stringify(operationName)}`);
  }
}

let config = {
...
  conflictListener: new ConflictLogger()
...
}
```

10.3.5. 競合前のエラーの処理

Data Sync は、開発者がミューテーションが発生する前に事前競合をチェックするためのメカニズムを提供します。送信されるデータがローカルで競合するかどうかをチェックします。これは、ミューテーション(またはミューテーションを作成する行為) が開始されたときに発生します。

たとえば、ユーザーが次のアクションを実行するとします。

1. フォームを開きます
2. このフォームに事前入力されたデータの作業を開始します
3. クライアントはサーバーからサブスクリプションの新しいデータを受信します。
4. クライアントは競合していますが、ユーザーは気づいていません
5. ユーザーが **Submit** を押すと、Data Sync はデータが競合していることに気づき、開発者にユーザーへ警告するための情報を提供します

この機能を使用して、データが確実に競合しているサーバーへの不要なラウンドトリップを省くために、開発者は Data Sync によって返されるエラーを利用できます。

開発者がこのエラーを使用する方法の例:

```
return client.offlineMutate({
  ...
}).then(result => {
  // handle the result
}).catch(error => {
```

```
  if (error.networkError && error.networkError.localConflict) {
    // handle pre-conflict here by potentially
    // providing an alert with a chance to update data before pressing send again
  }
})
```

第11章 ユーザーがモバイルアプリケーションからファイルをアップロードできるようにする

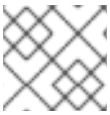
11.1. サーバーでのファイルアップロードの有効化

Data Sync Server は、GraphQL クエリーとともにバイナリーデータをアップロードするためのサポートを提供します。実装は、アップストリームの **Apollo Server** 機能に依存しています。

アップロード機能は、GraphQL マルチパートフォームリクエスト仕様を使用します。ファイルのアップロードは、サーバーとクライアントの両方に実装する必要があります。

1. クライアントでは、HTML FileList オブジェクトがミューテーションにマップされ、マルチパートリクエストでサーバーに送信されます。

2. サーバー上では、マルチパートリクエストが処理されます。サーバーはそれを処理し、リゾルバーにアップロード引数を提供します。リゾルバー機能では、アップロードプロミスがオブジェクトを解決します。



注記

ファイルのアップロードは graphql-multipart-request-spec に基づいています。

手順

ファイルのアップロードを有効にするには、スキーマを作成し、**Upload** スカラーを使用します。以下に例を示します。

```
const { ApolloServer, gql } = require('apollo-server');

const typeDefs = gql`
  type File {
    filename: String!
    mimetype: String!
    encoding: String!
  }
  type Query {
    uploads: [File]
  }
  type Mutation {
    singleUpload(file: Upload!): File!
  }
`;
```

次のスキーマはファイルのアップロードを有効にします。**Upload** スカラーは、リゾルバーの引数の 1 つとして挿入されます。**Upload** スカラーには、すべてのファイルメタデータと、ファイルを特定の場所に保存するために使用できる Readable Stream が含まれています。

```
    async singleUpload(parent, { file }) {
      const { stream, filename, mimetype, encoding } = await file;
      // Save file and return required metadata
    }
```

詳細は、Official Apollo blog post を参照してください。

11.2. クライアントでのファイルアップロードの実装

Voyager Client は、GraphQL クエリーとともにバイナリーデータをアップロードするためのサポートを提供します。バイナリーアップロードの実装では、Apollo コミュニティーによってビルドされた **apollo-upload-client** パッケージを使用します。

11.2.1. はじめに

アップロード機能は、GraphQL マルチパートフォームリクエスト仕様を使用します。ファイルのアップロードは、サーバーとクライアントの両方に実装する必要があります。

1. クライアントでは、HTML FileList オブジェクトがミューテーションにマップされ、マルチパートリクエストでサーバーに送信されます。

2. サーバー上では、マルチパートリクエストが処理されます。サーバーはそれを処理し、リゾルバーにアップロード引数を提供します。リゾルバー機能では、アップロードプロミスがオブジェクトを解決します。

注記

ファイルのアップロードは graphql-multipart-request-spec に基づいています。

11.2.2. ファイルアップロードの有効化

fileUpload フラグを config オブジェクトに渡して、ファイルアップロード機能を有効にする必要があります。

```
const config = {
  ...
  fileUpload: true
  ...
};

//create a new client
```

11.3. GRAPHQL からのファイルのアップロード

ファイルアップロード機能は、バイナリーデータを操作するミューテーションに使用できる新しい GraphQL スカラー **Upload** を追加します。**Upload** スカラーは、GraphQL スキーマの html **FileList** HTML5 オブジェクトをマッピングします。バイナリーアップロードを処理するために必要な最初のステップは、**Upload** スカラーを含むミューテーションを作成することです。次の例は、プロファイル写真をアップロードする方法を示しています。

```
import gql from 'graphql-tag'
import { Mutation } from 'react-apollo'

export const UPLOAD_PROFILE = gql`
mutation changeProfilePicture($file: Upload!) {
  changeProfilePicture(file: $file) {
    filename
    mimetype
    encoding
```

```
  }
}
`;
```

11.3.1. ミューテーションの実行

Upload スカラーは、HTML ファイル入力から返されたオブジェクトにマップされます。

次の例は、React アプリケーションでのファイルのアップロードを示しています。

```
const uploadOneFile = () => {
  return (
    <Mutation mutation={UPLOAD_PROFILE}>
      {uploadFile => (
        <input
        type="file"
        required
        onChange={({ target: { validity, files: [file] } }) =>
          validity.valid && uploadFile({ variables: { file } });
        }
       />
      )}
    </Mutation>
  );
};
```

第12章 RED HAT マネージドインテグレーションでの DATA SYNC アプリケーションの実行

12.1. DATA SYNC サーバーアプリケーションのデプロイ

前提条件

- Data Sync サーバーアプリケーションがローカルで動作している

手順

1. Solution Explorer にログインします。
2. OpenShift コンソールに移動します。
3. **Create Project** をクリックします。
4. プロンプトが表示されたら、アプリケーションの詳細を入力します。
5. **Project Overview** 画面に移動します。
6. サービスカタログで **Data Sync App** を検索します。
7. **Configuration** セクション:
   a. アプリケーションリポジトリーの Git URL を入力します。

注記

プライベートリポジトリーを使用するには、Creating New Applications を参照してください。

   b. 必須フィールドの情報を入力します(* で示されます)。
   c. 必要に応じて、オプションのフィールドに入力します。
8. ウィザードを完了して、Data Sync サーバーアプリケーションのプロビジョニングを開始します。
9. サービスが準備完了ステータスを表示するのを待ちます。
10. **Project Overview** 画面で、右上隅に表示されているアプリケーション URL を使用して、アプリケーションが使用可能であることを確認します。

12.2. DATA SYNC クライアントを DATA SYNC サーバーアプリケーションに接続する

前提条件

- Data Sync サーバーアプリケーションをデプロイしました。
- ES6 をサポートする Web プロジェクトを設定しました。以下に例を示します。

- Create React App の使用
- Ionic Getting Started の使用
- Getting Started with Angular の使用
- Webpack Getting Started Guide の使用

手順

1. Data Sync Server アプリケーションのホスト名を取得します。
   a. ターミナルで、次のコマンドを実行します。

```
$ oc get route <data-sync-application-name>
```

   b. 出力を次のように確認します。

```
NAME                          HOST/PORT              PATH     SERVICES       PORT
TERMINATION   WILDCARD
<sync-server-application-name>   <sync-server-hostname>          data-sync-app   <all>
None
```

   c. **<sync-server-hostname>** の値を記録します。

2. **@aerogear/voyager-client**、**graphql**、および **graphql-tag** ライブラリーがプロジェクトに追加されていることを確認してください。必要に応じて、次のコマンドを使用してそれらを追加します。

```
npm install @aerogear/voyager-client
npm install graphql
npm install graphql-tag
```

3. プロジェクトのソースコードで、サーバーのホスト名を使用してクライアントをインポートおよび設定します。

```
const config = {
  httpUrl: 'http://<sync-server-hostname>/graphql',
  wsUrl: 'ws://<sync-server-hostname>/graphql'
}
```

   これで、クライアントは Data Sync サーバーアプリケーションに対してクエリーとミューテーションを行う準備ができました。

改訂日時: 2022-04-26 09:32:47 +1000