Red Hat Managed Integration 2

Developing a Data Sync Application

For Red Hat Managed Integration 2

Last Updated: 2020-10-12

Red Hat Managed Integration 2 Developing a Data Sync Application

For Red Hat Managed Integration 2

Legal Notice

Abstract

This document provides a comprehensive description and usage instructions for creating a Data Sync application, Red Hat Managed Integration 2.

Table of Contents

PREFACE

Unlike other mobile services which provide a server and an API, Data Sync is a framework that allows you to develop services. Typically, you develop a Data Sync service as follows:

1. Design a GraphQL schema.
2. Develop a Data Sync server and Data Sync client, with the features you require
3. Containerize your Data Sync server and deploy it to OpenShift.
4. Configure your mobile app to point to the Data Sync server.
5. Complete your mobile app development.
6. Build and run your mobile app.

CHAPTER 1. INTRODUCTION

1.1. INTRODUCING DATA SYNC

Data Sync is a JavaScript framework that enables a developer to add real time data synchronization to both mobile and web clients. The Data Sync framework also provides offline capabilities that allow a client to continue operating offline and once connectivity is re-established, the client is automatically synchronized. An app built using the Data Sync framework typically connects to a data source for data persistence; however, an app built using the Data Sync framework works without a data source.

An app built using the Data Sync framework comprises of two components:

- The Data Sync client is a JavaScript client offering client side extensions and server side integration. The Data Sync client can be integrated into frameworks such as React and Angular.
- The Data Sync server is a framework for building Node.js based GraphQL API. The Data Sync server offers enterprise extensions for ensuring data security, integrity, and monitoring. It can be integrated into existing Node.js application.

The Data Sync framework uses the Apollo platform as the GraphQL implementation.

Additional resources

- Real-time data synchronization across mobile and web clients.
  - Websockets allow for real-time data synchronization across multiple Data Sync clients. Data Sync clients receive updates from the Data Sync server without having to explicitly query their local data as conflict detection is handled by the Data Sync server.
- A Data Sync client can perform any operation regardless of the connectivity state.
  - If network connectivity is a concern, a Data Sync client can perform any operation regardless of its connectivity state. A Data Sync client can perform the same operations when it is on-line or off-line, and this functionality ensures that you can safely use Data Sync to create business critical applications.
- Offers fully customizable conflict detection and resolution to the developer.
  - Data Sync enables users to detect and resolve conflicts on the Data Sync server resulting in the seamless transmission of data to various Data Sync clients. Data Sync also allows for conflict resolution on the Data Sync client should a developer want to adopt this strategy.
- Instant synchronous queries provide instant feedback for developers.
  - When a Data Sync client is on-line, instant queries allow a developer to quickly react to errors and display results to users when the operation is executed. Developers can retrieve an instant response or error from the Data Sync server however the Data Sync client must have a connection to the Data Sync server.
- Flexible data sources.
  - Data Sync can connect to various data sources, for example, cloud storage, databases such as MongoDB and PostgreSQL, and existing back-end data sources.

1.2. DATA SYNC TECHNICAL OVERVIEW

This section describes the technical aspects of Data Sync.

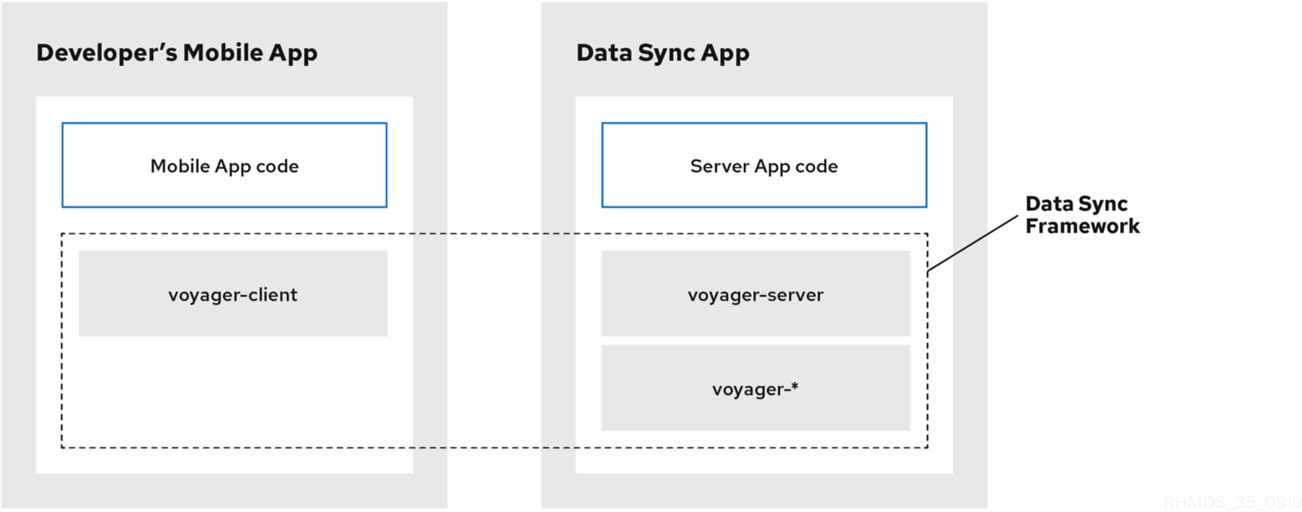


Table 1.1. Data Sync case study

Component	Technical Role
Sync Client	The Sync client is a client side JavaScript library used for building web and mobile applications. It allows for simple Sync server integration.
Sync Server	The Sync server is based on the Apollo Server framework and it performs two primary functions. It sends and retrieves data from a data source, and it syncs data across the Sync clients. The Sync server uses GraphQL to create custom connections that in turn allow various types of Sync clients to connect.
Data sources	The data source stores data. This data is typically what is synchronized across the Sync clients.

For more information about the Apollo Server framework, start here to learn about the Apollo platform .

1.3. DATA SYNC TERMINOLOGY

This section describes terminology that is associated with Data Sync.

Data Sync terms

GraphQL

A query language for your API, and a server-side runtime for executing queries that use a type system. For more information, see GraphQL.

Apollo

Apollo is an implementation of GraphQL designed for the needs of product engineering teams building modern, data-driven applications. Apollo includes two open-source libraries, Apollo Server and Apollo Client. The Data Sync Framework leverages Apollo functionality.

Sync Server

The Sync Server is a framework for building Node.js based GraphQL API.

Sync Client

The Sync Client is a JavaScript client offering client side extensions and server side integration. The Sync Client can be integrated into frameworks such as React and Angular.

Data sources

The Data Sync framework is typically used in conjunction with a data source for data persistence; however, an app built using the Data Sync framework works without a data source.

Data Sync framework

Data Sync is a JavaScript framework that enables a developer to add the capability to synchronize data in real-time for both mobile and web clients.

Additional resources

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

1.4. GETTING STARTED WITH HELLO WORLD DATA SYNC

In this example, you add the Data Sync Server library to your Express node.js project, create an **index-1.js** file, run the server, and query GraphQL.

- Data Sync Server is a set of Node.js libraries that can be used to build a Data Sync server.
- Data Sync Server is the starting point for developing a Data Sync application.

Prerequisites

- You have Node.js and npm installed.
- You have created a **node.js** project.

Procedure

1. Add libraries to your Node.js application:

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

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

(2) See https://expressjs.com/

(3) The Data Sync Server library that enables data sync

2. Create an **index-1.js** file with the following content:

```
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. Run the server:

```
$ node index-1.js

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

4. Browse **http://localhost:4000/graphql** and interact with the playground. For example:

```
{
  hello
}
```

5. Check the output. For the example above, the output should be:

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

To get started with the Data Sync framework, see the sample application. In this app, you can explore a more complex schema.

Before proceeding, make sure you have an understanding of the following GraphQL concepts:

- Schema design
- Resolvers
- Subscriptions

CHAPTER 2. QUERYING A DATA SYNC SERVER USING A DATA SYNC CLIENT

This section describes how to use the Voyager Client to create mobile and web applications that can communicate with the Voyager server application.

Data Sync provides JavaScript libraries which integrate your javascript app with a server that also uses Data Sync. The client libraries are based on the Apollo client.

You will add the libraries to your mobile project, configure the client classes, connect to the server, and confirm that it works.

Prerequisites

- You have Node.js and npm installed.
- You have created an empty web project that supports ES6, using the webpack getting started guide.
- You have completed the server getting started guide and the application is running.

Procedure

1. Create an address book server:
    a. Create an **index-2.js** file with the following content:

```
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. Run the server:

```
$ node index-2.js

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

c. Browse **http://localhost:4000/graphql** and interact with the playground. For example:

```
{
  addressBook {
    name
    address

  }
}
```

d. Check the output. For the example above, the output should be:

```
{
  "data": {
    "addressBook": [
```

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

2. Add the following libraries to your javascript client:

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

> **NOTE**
>
> A prerequisite is that you have created an empty web project that supports ES6, using the webpack getting started guide.

3. Create an **index.js** file to make the same query as step 1, but from JavaScript.
In this example, a config object is created, and the **httpUrl** field is set to the URL of the Voyager server application. If the client app uses subscriptions, then the **wsUrl** field is also required.

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. Build and run the client application.

5. Browse the client application and check the console output.
   It should include an array similar to the following:

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

CHAPTER 3. ADDING A MUTATION TO A DATA SYNC CLIENT

Prerequisites

- You have Node.js and npm installed.
- You have completed the Queries section and the server is still running.

Procedure

1. Modify the client application to perform the mutation:

   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. Build and run the client application.

3. Browse the client application and check the console output.
   It should include an array similar to the following:

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

4. Browse http://localhost:4000/graphql and enter the playground query for the addressbook. For example:

```
{
  addressBook {
    name
    address

}
```

Results should be similar to:

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

CHAPTER 4. SUPPORTING OFFLINE FUNCTIONALITY IN YOUR MOBILE APP

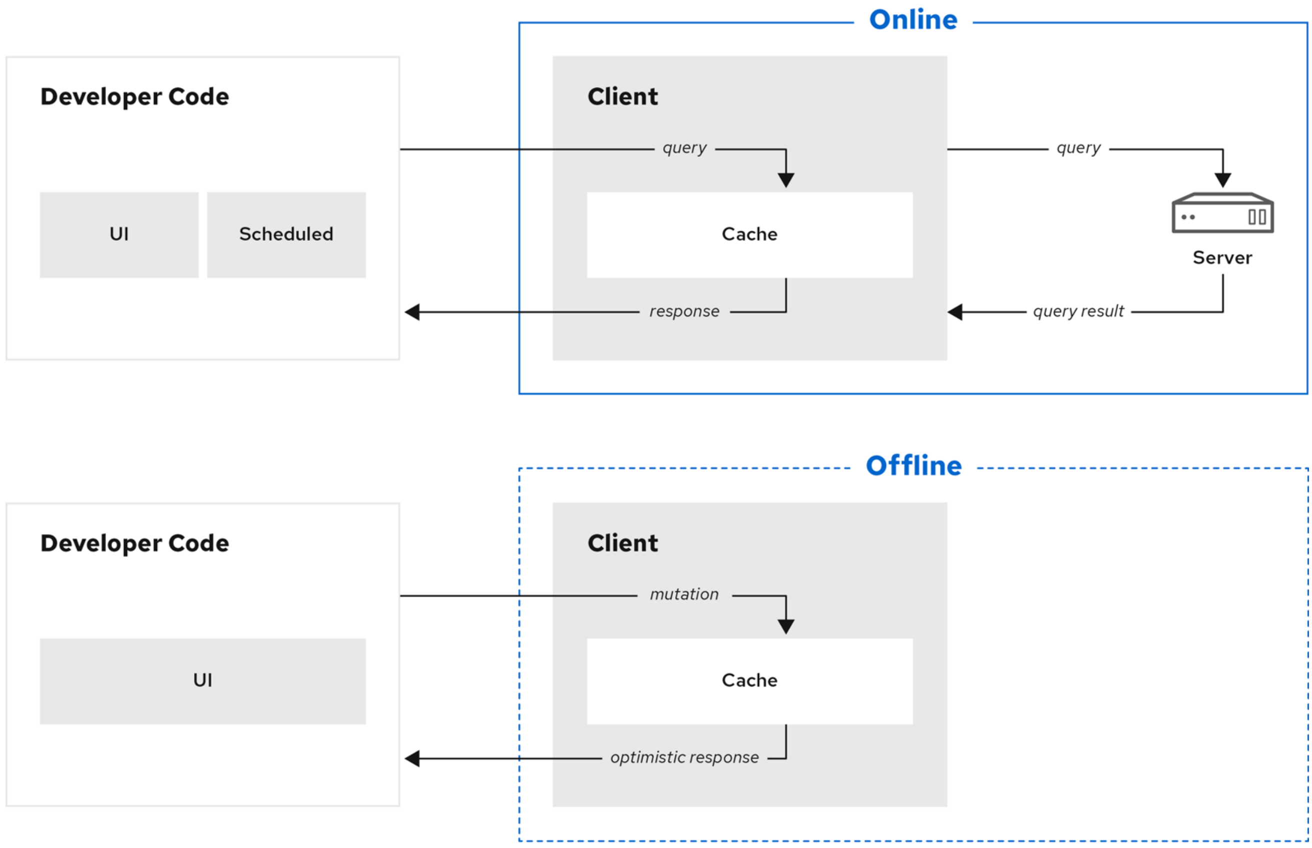
4.1. ABOUT OFFLINE FUNCTIONALITY

Your mobile app can run offline and allows users to query and create mutations using the @aerogear/voyager-client module.

All queries are performed against the cache, a mutation store (or offline store) supports offline mutations.

If a client goes offline for a long period of time, the mutation store negotiates local updates with the server using conflict resolution strategies.

When a client comes online again, the mutations are replicated back to the server.



Developers can attach listeners to get notifications about updates applied on the server or failing, and take appropriate actions.

Mutations and Local Cache

By default queries and the results of mutations are cached.

Mutations can change query results, make sure to call the **refetchQueries** or **update** options of the **mutate** method to ensure the local cache is kept up to date.

The @aerogear/voyager-client module also provides cache helper functions to reduce the amount of code required, as described in Section 6.4, “Using cache update helpers” .

For more information about **mutate** and the options available, see Apollo's document about mutations.

4.2. CREATING AN OFFLINE CLIENT

The @aerogear/voyager-client module provides an **OfflineClient** class which exposes the following functionality:

- direct access to the mutation store
- allows you to register multiple offline event listeners as described in Section 6.3, "Listening for events"
- automatically ensures the mobile app's local cache is kept up to date. This client automatically generates **update** methods as described in Section 6.4, "Using cache update helpers".

To create the client:

```
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();
```

This client can replace an Apollo client as it supports the same functionality.

CHAPTER 5. DETECTING MUTATIONS WHILE OFFLINE

If a mutation occurs while the device is offline, the **client.mutate** function:

- returns immediately
- returns a promise with an error

You can check the *error* object to isolate errors related to the offline state. Invoking the **watchOfflineChange()** method on an *error* object, watches for when an offline change is synced with the server, and sends a notification when triggered.

For example:

```
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(...)
 }
});
```

NOTE

In addition to watching individual mutations, you can add a global offline listener when creating a client as described in Section 6.3, "Listening for events".

CHAPTER 6. PERFORMING MUTATIONS WHILE OFFLINE

The @aerogear/voyager-client module provides an **offlineMutate** method which extends Apollo's mutate function with some extra functionality. This includes automatically adding some fields to each operation's context.

To set up the offline client, see Section 4.2, "Creating an offline client" .

Once set up is complete, **offlineMutate** is then available to use.

NOTE

The **offlineMutate** method accepts the same parameters as **mutate** with some additional optional parameters also available.

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

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

1. The query or queries which should be updated with the result of the mutation.
2. The type of operation being performed. Should be "add", "refresh" or "delete". Defaults to "add" if not provided.
3. The field on the object used to identify it. Defaults to "id" if not provided.
4. The type of object being operated on.

6.1. SUPPORTING APP RESTARTS WHILE OFFLINE

An Apollo client holds all mutation parameters in memory. An offline Apollo client continues to store mutation parameters and once online, it restores all mutations to memory. Any update functions that are supplied to mutations cannot be cached by an Apollo client resulting in the loss of all optimistic responses after a restart. *Update functions* supplied to mutations cannot be saved in the cache. As a result, all *optimisticResponses* disappear from the application after a restart and only reappear when the Apollo client becomes online and successfully syncs with the server.

To prevent the loss of all *optimisticResponses* after a restart, you can configure the *Update Functions* to restore all *optimisticResponses*.

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

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

You can also use **getUpdateFunction** to automatically generate functions:

```
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. ENSURING SPECIFIED MUTATIONS ARE PERFORMED ONLINE ONLY

If you wish to ensure certain mutations are only executed when the client is online, use the GraphQL directive **@onlineOnly**, for example:

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

6.3. LISTENING FOR EVENTS

To handle all notifications about offline related events, use the **offlineQueueListener** listener in the config object

The following events are emitted:

- **onOperationEnqueued** - Called when new operation is being added to offline queue
- **onOperationSuccess** - Called when back online and operation succeeds
- **onOperationFailure** - Called when back online and operation fails with GraphQL error
- **queueCleared** - Called when offline operation queue is cleared

You can use this listener to build User Interfaces that show pending changes.

6.4. USING CACHE UPDATE HELPERS

The @aerogear/voyager-client module provides an out of the box solution for managing updates to your application's cache. It can intelligently generate cache update methods for both mutations and subscriptions.

6.4.1. Using cache update helpers for mutations

The following example shows how to use these helper methods for mutations. To use these methods, create an offline client as described in Section 4.2, "Creating an offline client" and then use the **offlineMutate** method. The **offlineMutate** function accepts a **MutationHelperOptions** object as a parameter.

```
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'
};
```

You can also provide more than one query to update the cache by providing an array to the **updateQuery** parameter:

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

The following example shows how to prepare an offline mutation to add a task using the **mutationOptions** object and how to update the **GET_TASK** query for the client's cache.

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

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

If you do not want to use the offline client you can also use the **createMutationOptions** function directly. This function provides an Apollo compatible **MutationOptions** object to pass to your preexisting client. The following example shows how to use this function where **mutationOptions** is the same object as the previous code example.

```
const options = createMutationOptions(mutationOptions);

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

6.4.2. Using cache update helpers for subscriptions

The @aerogear/voyager-client module provides a subscription helper which can generate the necessary options to be used with Apollo Client's **subscribeToMore** function.

To use this helper, we first need to create some options, for example:

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

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

This options object informs the subscription helper that for every data object received because of the **TASK_ADDED_SUBSCRIPTION** the **GET_TASKS** query should also be kept up to date in the cache.

You can then create the required cache update functions:

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

const subscriptionOptions = createSubscriptionOptions(options);
```

To use this helper, pass this **subscriptionOptions** variable to the **subscribeToMore** function of our **ObservableQuery**.

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

query.subscribeToMore(subscriptionOptions);
```

The cache is kept up to date while automatically performing data deduplication.

6.4.3. Using cache update helpers for multiple subscriptions

The @aerogear/voyager-client module provides the ability to automatically call **subscribeToMore** on your **ObservableQuery**. This can be useful in a situation where you may have multiple subscriptions which can affect one single query. For example, if you have a **TaskAdded**, **TaskDeleted**, and a **TaskUpdated** subscription you require three separate **subscribeToMore** function calls. To avoid this, use the **subscribeToMoreHelper** function from the @aerogear/voyager-client module to automatically handle this by passing an array of subscriptions and their corresponding queries:

```
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]);
```

CHAPTER 7. DETECTING NETWORK STATUS

Use the NetworkStatus interface to check the current network status, or to register a listener which performs actions when the status of the network changes.

Two default implementations are provided:

- **WebNetworkStatus** for web browsers
- **CordovaNetworkStatus** for Cordova

The following example demonstrates how to register a listener using **CordovaNetworkStatus**:

```
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
```

CHAPTER 8. SUPPORTING REAL-TIME UPDATES IN YOUR MOBILE APP

8.1. INTRODUCTION TO REAL-TIME UPDATES

After developing some queries and mutations, you might want to implement real-time updates.

Real-time updates are supported in the GraphQL specification by an operation type called **Subscription**. To support subscriptions in a production environment, Data Sync implements subscriptions using an MQTT PubSub subscription mechanism; however, you might want to use the Apollo PubSub module to develop proof-of-concept applications.

When coding for real-time updates, use the following modules:

- @aerogear/voyager-server - supports clients that use voyager-client to enable GraphQL queries and mutations
- @aerogear/voyager-subscriptions - supports clients that use voyager-client to enable GraphQL subscriptions
- @aerogear/graphql-mqtt-subscriptions - supports GraphQL resolvers connections to a MQTT broker

GraphQL Subscriptions enable clients to subscribe to server events over a websocket connection.

The flow can be summarized as follows:

- Client connects to the server using websockets, and subscribes to certain events.
- As events occur, the server notifies the clients that are subscribed to those events.
- Any *currently connected* client that is subscribed to a given event receives updates.
- The client can close the connection at any time and no longer receives updates.

To receive updates, the client must be currently connected to the server. The client does not receive events from subscriptions while offline. To support inactive clients, use Push Notifications.

Additional resources

- For more information about GraphQL subscriptions, see the Subscriptions documentation.

8.2. IMPLEMENTING REAL-TIME UPDATES ON A DATA SYNC SERVER

The follow code shows typical code for a Data Sync Server without subscriptions:

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

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

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

The following sections outline the steps required to enable real-time updates:

1. Implement a SubscriptionServer
2. Implement a Publish Subscribe Mechanism
3. Define subscriptions in the schema
4. Implement resolvers

8.2.1. Implementing a SubscriptionServer using voyager-subscription

To allow you create GraphQL subscription types in your schema:

1. Install the **@aerogear/voyager-subscriptions** package:

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

2. Configure SubscriptionServer using **@aerogear/voyager-subscriptions**

   ```
   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'
     })
   })
   ```

   The **createSubscriptionServer** code:

   - returns a **SubscriptionServer** instance
   - installs handlers for
     - managing websocket connections
     - delivering subscriptions on the server
   - provides integrations with other modules such as **@aerogear/voyager-keycloak**.

Additional resources

- For more information about arguments and options, see the subscriptions-transport-ws module.

8.2.2. Implementing a Publish Subscribe Mechanism

> **WARNING**
>
> This procedure describes an in-memory implementation which is useful for prototyping but not suitable for production. Red Hat recommends using MQTT PubSub in production. See Section 8.3, "Configuring a Publish Subscribe mechanism" for more information about all the implementation methods.

To provide a channel to push updates to the client using the default **PubSub** provided by **apollo-server**, you implement a Publish Subscribe mechanism, for example:

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

const pubsub = new PubSub()
```

Addtional Information

Subscriptions depend on a publish subscribe mechanism to generate the events that notify a subscription. There are several PubSub implementations available based on the **PubSubEngine** interface.

8.2.3. Defining subscriptions in the schema

Subscriptions are a root level type. They are defined in the schema similar to **Query** and **Mutation**. For example, in the following schema, a **Task** type is defined and so are mutations and subscriptions.

```
type Subscription {
  taskCreated: Task
}

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

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

8.2.4. Implementing resolvers

Inside the resolver map, subscription resolvers return an **AsyncIterator,** which listens for events. To generate an event, call the **publish** method. The **pubsub.publish** code is typically located inside a mutation resolver.

In the following example, when a new task is created, the **createTask** resolver publishes the result of this mutation to the **TaskCreated** channel.

```
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
    }
  },
}
```

NOTE

This subscription server does not implement authentication or authorization. For information about implementing authenication and authorization, see Supporting authentication and authorization in your mobile app.

Additional resources

- For information on how to use subscriptions in your client code, see Realtime Updates.

8.3. CONFIGURING A PUBLISH SUBSCRIBE MECHANISM

You can use the Apollo PubSub mechanism for development, but you must use the MQTT PubSub mechanism for production.

8.3.1. Using the Apollo PubSub mechanism

The Section 8.2, "Implementing real-time updates on a Data Sync server" section describes how to set up the default **PubSub** provided by **apollo-server**. For a production system, use MQTT PubSub.

8.3.2. Using the MQTT PubSub mechanism

The **@aerogear/graphql-mqtt-subscriptions** module provides an **AsyncIterator** interface used for implementing subscription resolvers It connects the Data Sync server to an MQTT broker to support horizontally scalable subscriptions.

Initialize an MQTT client and pass that client to the **@aerogeaar/graphql-mqtt-subscriptions** module, for example:

```
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
})
```

In the example, an **mqtt** client is created using **mqtt.connect** and then this client is passed into an **MQTTPubSub** instance. The **pubsub** instance can then be used to publish and subscribe to events in the server.

Additional resources

- mqtt.connect documentation.
- MQTTPubSub documentation

8.4. CONFIGURING AMQ ONLINE FOR MQTT MESSAGING

Red Hat AMQ supports the MQTT protocol which makes it a suitable PubSub technology for powering GraphQL subscriptions at scale.

This section provides recommendations for:

- Configuring AMQ Online for MQTT messaging.
- Connecting to AMQ Online and using it as a pubsub within server applications.

Terminology

- AMQ Online is a mechanism that allows developers to consume the features of Red Hat AMQ within OpenShift.
- Red Hat AMQ provides fast, lightweight, and secure messaging for Internet-scale applications. AMQ Broker supports multiple protocols and fast message persistence.
- MQTT stands for MQ Telemetry Transport. It is a publish-subscribe, extremely simple and lightweight messaging protocol.

AMQ Online includes many configuration options that address the specific needs of your application. The minimum configuration steps for using AMQ Online for MQTT messaging and enabling GraphQL subscriptions are:

1. Create an **AddressSpace**
2. Create an **Address**
3. Create a **MessagingUser**

8.4.1. Creating an address space

A user can request messaging resources by creating an **AddressSpace**. There are two types of address spaces, **standard** and **brokered**. You must use the **brokered** address space for MQTT based applications.

Procedure

1. Create an address space. For example, the following resource creates a brokered **AddressSpace**:

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

2. Create the **AddressSpace**.

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

3. Check the status of the address space:

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

The output displays information about the address space, including details required for connecting applications.

Additional resources

- See Creating address spaces using the command line for more information.

8.4.2. Creating an Address

An adress is part of an **AddressSpace** and represents a destination for sending and receiving messages. Use an **Address** with type **topic** to represent an MQTT topic.

1. Create an address definition:

```
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. Create the address:

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

> **NOTE**
>
> See the Configuring your server for real-time updates guide for more information about using **pubsub.asyncIterator()**. Create an Address for each topic name passed into **pubsub.asyncIterator()**.

Additional resources

- See Creating addresses using the command line for more information.

8.4.3. Creating an AMQ Online user

A messaging client connects using an AMQ Online user, also known as a`MessagingUser`. A **MessagingUser** specifies an authorization policy that controls which addresses can be used and the operations that can be performed on those addresses.

Users are configured as **MessagingUser** resources. Users can be created, deleted, read, updated, and listed.

1. Create a user definition:

```
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. Create the **MessagingUser**.

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

An application can now connect to an AMQ Online address using this user's credentials.

For more information see the AMQ Online User Model .

8.5. USING GRAPHQL MQTT PUBSUB WITH AMQ ONLINE

Prerequisites

The following AMQ Online resources are available for MQTT Applications

- AddressSpace
- Address
- MessagingUser

This section describes how to use **@aerogear/graphql-mqtt-subscriptions** to connect to an AMQ Online **Address**.

1. Retrieve the connection details for the **AddressSpace** you want to use:

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

2. Determine which method you want to use to connect to the address:

   - Using the service hostname - Allows clients to connect from within the OpenShift cluster. Red Hat recommends that applications running inside OpenShift connect using the service hostname. The service hostname is only accessible within the OpenShift cluster. This ensures messages routed between your application and AMQ Online stay within the OpenShift cluster and never go onto the public internet.

   - Using the external hostname - Allows clients to connect from outside the OpenShift cluster. The external hostname allows connections from outside the OpenShift cluster. This is useful for the following cases:

     - Production applications running outside of OpenShift connecting and publishing messages.

     - Quick Prototyping and local development. Create a non-production **AddressSpace**, allowing developers to connect applications from their local environments.

3. To connect to an AMQ Online **Address** using the service hostname

   a. Retrieve the service hostname:

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

   b. Add code to create the connection, for example:

```
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. To encrypt all messages between your application and the AMQ Online broker, enable TLS, for example:

```
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. To connect to an AMQ Online **Address** using the external hostname:

   NOTE

   The external hostname typically accept only accept TLS connections.

   a. Retrieve the external hostname:

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

   b. Connect to the external hostname, for example:

   ```
   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. If you use TLS, note the following additional **mqtt.connect** options:

   - **servername** - when connecting to a message broker in OpenShift using TLS, this property must be set otherwise the connection will fail, because the messages are being routed through a proxy resulting in the client being presented with multiple certificates. By setting the **servername**, the client will use Server Name Indication (SNI) to request the correct certificate as part of the TLS connection setup.
   - **protocol** - must be set to **'tls'**
   - **rejectUnauthorizated** - must be set to false, otherwise the connection will fail. This tells the client to ignore certificate errors. Again, this is needed because the client is presented with multiple certificates and one of the certificates is for a different hostname than the one being requested, which normally results in an error.

- **port** - must be set to 5761 for service hostname or 443 for external hostname.

8.5.1. Using environment variables for configuration

Red Hat recommends that you use environment variables for connection, for example:

```
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 })
```

In this example, the connection options can be configured using environment variables, but sensible defaults for the **host**, **port** and **protocol** are provided for local development.

8.5.2. Troubleshooting MQTT Connection Issues

8.5.2.1. Troubleshooting MQTT Events

The **mqtt** module emits various events during runtime. It recommended to add listeners for these events for regular operation and for troubleshooting.

```
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}`)
})
```

Read the MQTT documentation to learn about all of the events and what causes them.

8.5.2.2. Troubleshooting MQTT Configuration Issues

If your application is experiencing connection errors, the most important thing to check is the configuration being passed into **mqtt.connect**. Because your application may run locally or in OpenShift, it may connect using internal or external hostnames, and it may or may not use TLS. It is very easy to accidentally provide the wrong configuration.

The Node.js **mqtt** module does not report any errors if parameters such as **hostname** or **port** are incorrect. Instead, it will silently fail and allow your application to start without messaging capabilities.

It may be necessary to handle this scenario in your application. The following workaround can be used.

```
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)
```

This code can be used to detect if the MQTT client hasn't connected. This can be helpful for detecting potential configuration issues and allows your application to respond to that scenario.

8.6. IMPLEMENTING REAL-TIME UPDATES ON ON THE CLIENT

A core concept of the GraphQL specification is an operation type called **Subscription**, they provide a mechanism for real time updates. For more information on GraphQL subscriptions see the Subscriptions documentation.

To do this GraphQL Subscriptions utilise websockets to enable clients to subscribe to published changes.

The architecture of websockets is as follows:

- Client connects to websocket server.
- Upon certain events, the server can publish the results of these events to the websocket.
- Any *currently connected* client to that websocket receives these results.
- The client can close the connection at any time and no longer receives updates.

Websockets are a perfect solution for delivering messages to currently active clients. To receive updates the client must be currently connected to the websocket server, updates made over this websocket while the client is offline are not consumed by the client. For this use case Push Notifications are recommended.

Voyager Client comes with subscription support out of the box including auto-reconnection upon device restart or network reconnect. To enable subscriptions on your client set the following paramater in the Voyager Client config object. A DataSyncConfig interface is also available from Voyager Client if you wish to use it.

8.6.1. Setting up a client to use subscriptions

To set up a client to use subscriptions:

1. Provide a **wsUrl** string in the config object as follows:
   ■

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

where **<your_websocket_url>** is the full URL of the websocket endpoint of your GraphQL server.

2. Use the object from step 1 to initialise Voyager Client:

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

const client = createClient(config)
```

8.6.2. Using Subscriptions

A standard flow to utilise subscriptions is as follows:

1. Make a network query to get data from the server
2. Watch the cache for changes to queries
3. Subscribe to changes pushed from the server
4. Unsubscibe when leaving the view where there is an active subscription

In the three examples below, **subscribeToMore** ensures that any further updates received from the server force the updateQuery function to be called with **subscriptionData** from the server.

Using **subscribeToMore** ensures the cache is easily updated as all GraphQL queries are automatically notified.

For more information, see the subscribeToMore documentation.

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

  tasks.subscribeToMore({
    document: TASK_ADDED_SUBSCRIPTION,
    updateQuery: (prev, { subscriptionData }) => {
    // Update logic here.
    }
  });
  return tasks;
}
```

To allow Voyager Client to automatically generate the **updateQuery** function for you, please see the Cache Update Helpers section.

You can then use this query in our application to subscribe to changes so that the front end is always updated when new data is returned from the server.

```
this.tasks = [];
this.getTasks().subscribe(result => {
```

```
  this.tasks = result.data && result.data.allTasks;
})
```

Note that it is also a good idea to unsubscribe from a query upon leaving a page. This prevents possible memory leaks. This can be done by calling unsubscribe() as shown in the following example. This code should be placed in the appropriate place.

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

8.6.3. Handling network state changes

When using subscriptions to provide your client with realtime updates it is important to monitor network state because the client will be out of sync if the server if updated when the the client is offline.

To avoid this, Voyager Client provides a **NetworkStatus** interface which can be used along with the **NetworkInfo** interface to implement custom checks of network status.

Use the following example to re-run a query after a client returns to an online state:

```
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
      });
    }
  }
});
```

CHAPTER 9. SUPPORTING AUTHENTICATION AND AUTHORIZATION IN YOUR MOBILE APP

9.1. CONFIGURING YOUR SERVER FOR AUTHENTICATION AND AUTHORIZATION USING RED HAT SINGLE SIGN-ON

Using the keycloak service service and the @aerogear/voyager-keycloak module, it is possible to add security to a Data Sync Server application.

The **@aerogear/voyager-keycloak** module provides the following features out of the box:

- Authentication - Ensure only authenticated users can access your server endpoints, including the main GraphQL endpoint.
- Authorization - Use the **@hasRole()** directive within the GraphQL schema to implement role based access control (RBAC) on the GraphQL level.
- Integration with GraphQL context - Use the **context** object within the GraphQL resolvers to access user credentials and several helper functions.

Prerequisites

- There is a Red Hat Single Sign-On service available.
- You must add a valid **keycloak.json** config file to your project.
  - Create a client for your application in the Keycloak administration console.
  - Click on the Installation tab.
  - Select **Keycloak OIDC JSON** for Format option, and click **Download**.

9.1.1. Protecting Data Sync Server using Red Hat Single Sign-On

Procedure

1. Import the **@aerogear/voyager-keycloak** module

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

2. Read the Keycloak config and pass it to initialise the **KeycloakSecurityService**.

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

3. Use the **keycloakService** instance to protect your app:

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

4. Configure the Voyager server so that the **keycloakService** is used as the security service:

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

The Keycloak Example Server Guide has an example server based off the instructions above and shows all of the steps needed to get it running.

9.1.2. Using the hasRole directive in a schema

The Voyager Keycloak module provides the **@hasRole** directive to define role based authorisation in your schema. The **@hasRole** directive is a special annotation that can be applied to:

- fields
- queries
- mutations
- subscriptions

The **@hasRole** usage is as follows:

- **@hasRole(role: String)**
- Example - **@hasRole(role: "admin"])**
- If the authenticated user has the role **admin** they will be authorized.
- **@hasRole(role: [String])**
- Example - **@hasRole(role: ["admin", "editor"])**
- If the authenticated user has at least one of the roles in the list, they will be authorized.

The default behaviour is to check client roles. For example, **@hasRole(role: "admin")** will check that user has a client role called **admin**. **@hasRole(role: "realm:admin")** will check if that user has a realm role called **admin**

The syntax for checking a realm role is **@hasRole(role: "realm:<role>")**. For example, **@hasRole(role: "realm:admin")**. Using a list of roles, it is possible to check for both client and realm roles at the same time.

Example: Using the @hasRole Directive to Apply Role Based Authorization in a Schema

The following example demonstrates how the **@hasRole** directive can be used to define role based authorization on various parts of a GraphQL schema. This example schema represents publishing an application like a news or blog website.

```
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")
}
```

There are two types:

- **Post** - An article or a blog post
- **Author** - Represents the person that authored a Post

There are two queries:

- **allPosts** - Returns a list of posts
- **getAuthor** - Returns details about an Author

There are two mutations:

- **editPost** - Edits an existing post
- **deletePost** - Delete a post.

Role Based Authorization on Queries and Mutations

In the example schema, the **@hasRole** directive has been applied to the **editPost** and **deletePost** mutations. The same can be done on queries.

- Only users with the roles **editor** and/or **admin** are allowed to perform the **editPost** mutation.
- Only users with the role **admin** are allowed to perform the **deletePost** mutation.

This example shows how the **@hasRole** directive can be used on various queries and mutations.

Role Based Authorization on Fields

In the example schema, the **Author** type has the fields **address** and **age** which both have **hasRole(role: "admin")** applied.

This means that users without the role **admin** are not authorized to request these fields in any query or mutation.

For example, non-admin users are allowed to run the **getAuthor** query, but cannot request the **address** or **age** fields.

9.2. AUTHENTICATION OVER WEBSOCKETS USING RED HAT SINGLE SIGN-ON

Prerequisites:

- Configure Data Sync Server for Authentication and Authorization
- Configuring Your Server for real-time updates

This section describes how to implement authentication and authorization over websockets with Red Hat Single Sign-On. For more information on authentication over websockets, read Apollo's Authentication Over Websocket documention.

The Voyager Client supports adding token information to **connectionParams** that will be sent with the first WebSocket message. In the server, this token is used to authenticate the connection and to allow the subscription to proceeed. Read the section on Red Hat Single Sign-On Authentication in Voyager Client to ensure the Red Hat Single Sign-On token is sent to the server.

In the server, **createSubscriptionServer** accepts a **SecurityService** instance in addition to the regular options that can be passed to a standard **SubscriptionServer**. The **KeycloakSecurityService** from **@aerogear/voyager-keycloak** is used to validate the Red Hat Single Sign-On token passed by the client in the initial WebSocket message.

```
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'
  })
)
```

The example shows how the Red Hat Single Sign-On **securityService** is created and how it is passed into **createSubscriptionServer**. This enables Red Hat Single Sign-On authentication on all subscriptions.

9.2.1. Red Hat Single Sign-On Authorization in Subscriptions

The Red Hat Single Sign-On **securityService** will validate and parse the token sent by the client into a Token Object. This token is available in Subscription resolvers with **context.auth** and can be used to implement finer grained role based access control.

```
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)
      }
    },
}
```

The above example shows role based access control inside a subscription resolver. **context.auth** is a full Keycloak Token Object which means methods like **hasRealmRole** and **hasApplicationRole** are available.

The user details can be accessed through **context.auth.content**. Here is an example.


```
{
  "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"
}
```

Having access to the user details (e.g. **context.auth.content.sub** is the authenticated user's ID) means it is possible to implement Subscription Filters and to subscribe to more fine grained pubsub topics based off the user details.

9.3. IMPLEMENTING AUTHENTICATION AND AUTHORIZATION ON YOUR CLIENT

With Voyager Client, user information can be passed to a Data Sync server application in two ways, by using headers or by using tokens.

Headers are used to authentication HTTP requests to the server, which are used for queries and mutations.

Tokens are used to authenticate WebSocket connections, which are used for subscriptions.

Both ways can be set by the **authContextProvider** configuration option. For example:

```
//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
```

For information about how to perform authentication and authorization on the server, see the Server Authentication and Authorization Guide.

CHAPTER 10. RESOLVING CONFLICTS IN YOUR DATA SYNC APP

10.1. INTRODUCTION

Mobile apps allow users to modify data while offline. This can result in conflicts.

A **conflict** occurs when two or more users try to modify the same data. The system needs to resolve the conflicting data.

Conflict resolution can be handled in two phases:

- **Conflict detection** is the ability of an application to detect the possibility of incorrect data being stored.
- **Conflict resolution** is the process of ensuring that the correct data is stored.

With Red Hat Data Sync:

- You implement conflict detection exclusively in the code associated with mutations.
- The Data Sync Server module provides conflict detection on the server side.
- The Voyager Client module provides conflict resolution on the client side.

10.2. DETECTING CONFLICTS ON THE SERVER

A typical flow for detecting conflicts includes the following steps:

1. **A Mutation Occurs** - A client tries to modify or delete an object on the server using a GraphQL mutation
2. **Read the Object** - The server reads the current object that the client is trying to modify from the data source
3. **Conflict Detection** - The server compares the current object with the data sent by the client to see if there is a conflict. The developer chooses how the comparison is performed.

The **aerogear/voyager-conflicts** module helps developers with the **Conflict Detection** steps regardless of the storage technology, while the fetching and storing of data is the responsibility of the developer.

This release supports the following implementations:

- **VersionedObjectState** - depends on the version field supplied in objects (the version field is used by default when importing conflictHandler). For details, please see: Section 10.2.1, “Implementing version based conflict detection”
- **HashObjectState** - depends on a hash calculated from the entire object. For details, please see: Section 10.2.2, “Implementing hash based conflict detection”

These implementations are based on the **ObjectState** interface and that interface can be extended to provide custom implementations for conflict detection.

Prerequisites

- GraphQL server with resolvers.
- Database or any other form of data storage that can cause data conflicts. Red Hat recommends that you store data in a secure location. If you use a database, it is your responsibility to administer, maintain and backup that database. If you use any other form of data storage, you are responsible for backing up the data.

10.2.1. Implementing version based conflict detection

Version based conflict resolution is the recommended and simplest approach for conflict detection and resolution. The core idea is that every object has a **version** property with an integer value. A **conflict** occurs when the version number sent by the client does not match the version stored in the server. This means a different client has already updated the object.

Procedure

1. Import the @aerogear/voyager-conflicts package.

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

2. Add a version field to the GraphQL type that should support conflict resolution. The version should also be stored in the data storage.

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

3. Add an example mutation.

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

4. Implement the resolver. Every conflict can be handled using a set of predefined steps, for example:

```
// 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)
```

In the example above, the **throw** statement ensures that the client receives all necessary data to resolve the conflict client-side. For more information about this data, please see Structure of the Conflict Error.

Since the conflict will be resolved on the client, it is not required to persist the data. However, if there is no conflict, the data sent by the client should be persisted. For more information on resolving the conflict client-side, please see: Resolving Conflicts on the Client.

10.2.2. Implementing hash based conflict detection

Hash based conflict detection is a mechanism to detect conflicts based on the *total* object being updated by the client. It does this by hashing each object and comparing the hashes. This tells the server whether or not the objects are equivalent and can be considered conflict free.

Procedure

1. Import the @aerogear/voyager-conflicts package.

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

2. When using the **HashObjectState** implementation, a hashing function must be provided. The function signature should be as follows:

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

3. Provide this function when instantiating the **HashObjectState**:

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

4. Implement the resolver. Every conflict can be handled using a set of predefined steps, for example:

```
// 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)
```

In the example above, the **throw** statement ensures the client receives all necessary data to resolve the conflict client-side. For more information about this data please see Structure of the Conflict Error.

Since the conflict will be resolved on the client, it is not required to persist the data. However, if there is no conflict, the data sent by the client should be persisted. For more information on resolving the conflict client-side, please see: Resolving Conflicts on the Client.

10.2.3. About the structure of the conflict error

The server needs to return a specific error when a conflict is detected containing both the server and client states. This allows the client to resolve the conflict.

```
"extensions": {
      "code": "INTERNAL_SERVER_ERROR",
```

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

10.3. RESOLVING CONFLICTS ON THE CLIENT

A typical flow for resolving conflicts includes the following steps:

1. **A Mutation Occurs** - A client tries to modify or delete an object on the server using a GraphQL mutation.
2. **Read the Object** - The server reads the current object the client is trying to modify from the data source (usually a database).
3. **Conflict Detection** - The server compares the current object with the data sent by the client to see if there was a conflict. If there is a conflict, the server returns a response to the client containing information outlined in Structure of the Conflict Error
4. **Conflict Resolution** - The client attempts to resolve this conflict and makes a new request to the server in the hope that this data is no longer conflicted.

The conflict resolution implementation requires the following additions to your application:

- A **returnType** added to the context of any mutation. see: Working With Conflict Resolution on the Client.
- Additional metadata inside types (for example version field) depending on the conflict implementation you chose. see: Version Based Conflict Detection.
- Server-side resolvers to return conflicts back to clients first. For more information, see: Server Side Conflict Detection.

Developers can either use the default conflict resolution implementations, or implement their own conflict resolution implementations using the conflict resolution mechanism.

By default, when no changes are made on the same fields, the implementation attempts to resend the modified payload back to the server. When changes on the server and on the client affect the same fields, one of the specified conflict resolution strategies can be used. The default strategy applies client changes on top of the server side data. Developers can modify strategies to suit their needs.

10.3.1. Implementing conflict resolution on the client

To enable conflict resolution, the server side resolvers must be configured to perform conflict detection. Detection can rely on different implementations and return the conflict error back to the client. See Server Side Conflict Detection for more information.

Procedure

Provide the mutation context with the **returnType** parameter to resolve conflicts. This parameter defines the Object type being operated on. You can implement this in two ways:

- If using Data Sync's **offlineMutate** you can provide the **returnType** parameter directly as follows:

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

- If using Apollo's **mutate** function, provide the **returnType** parameter as follows:

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

The client automatically resolves the conflicts based on the current strategy and notifies listeners as required.

Conflict resolution works with the recommended defaults and does not require any specific handling on the client.

NOTE

For advanced use cases, the conflict implementation can be customised by supplying a custom **conflictProvider** in the application config. See Conflict Resolution Strategies below.

10.3.2. About the default conflict implementation

By default, conflict resolution is configured to rely on a **version** field on each GraphQL type. You must save a version field to the database in order to detect changes on the server. For example:

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

The version field is controlled on the server and maps the last version that was sent from the server. All operations on the version field happen automatically. Make sure that the version field is always passed to the server for mutations that supports conflict resolution:

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

10.3.3. Implementing conflict resolution strategies

Data Sync allows developers to define custom conflict resolution strategies. You can provide custom conflict resolution strategies to the client in the config by using the provided **ConflictResolutionStrategies** type. By default developers do not need to pass any strategy as **UseClient** is the default. Custom strategies can also be used to provide different resolution strategies for certain operations:

```
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
  }
}
```

This custom strategy object provides two distinct strategies. The strategies are named to match the operation. You pass the name of the object as an argument to conflictStrategy in your config object:

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

10.3.4. Listening to conflicts

Data Sync allows developers to receive information about the data conflict.

When a conflict occurs, Data Sync attempts to perform a field level resolution of data - it checks all fields of its type to see if both the client or server has changed the same field. The client can be notified in one of two scenarios.

- If both client and server have changed any of the same fields, the **conflictOccurred** method of the **ConflictListener** is triggered.
- If the client and server have not changed any of the same fields, and the data can be easily merged, the **mergeOccurred** method of your **ConflictListener** is triggered.

Developers can supply their own **conflictListener** implementation, for example:

```
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. Handling pre-conflict errors

Data Sync provides a mechanism for developers to check for a 'pre-conflict' before a mutation occurs. It checks whether or not the data being sent conflicts locally. This happens when a mutation (or the act of creating a mutation) is initiated.

For example, consider a user performing the following actions:

1. opens a form
2. begins working on the pre-populated data on this form
3. the client receives new data from the server from subscriptions
4. the client is now conflicted but the user is unaware
5. when the user presses **Submit** Data Sync notices that their data is conflicted and provides the developer with the information to warn the user

To use this feature, and therefore potentially save unnecessary round-trips to the server with data which is definitely conflicted, developers can make use of the error returned by Data Sync.

An example of how developers can use this error:

```
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
  }
})
```

CHAPTER 11. ALLOWING USERS UPLOAD FILES FROM YOUR MOBILE APP

11.1. ENABLING FILE UPLOADS ON THE SERVER

Data Sync Server provides support for uploading binary data along with the GraphQL queries. The implementation relies on upstream **Apollo Server** capabilities.

The upload functionality uses the GraphQL multipart form requests specification. File upload needs to be implemented on both server and client:

1. On the client HTML FileList objects are mapped into a mutation and sent to the server in a multipart request.

2. On the server: The multipart request is handled. The server processes it and provides an upload argument to a resolver. In the resolver function, the upload promise resolves an object.

NOTE

File upload is based on graphql-multipart-request-spec.

Procedure

To enable file uploads, create a schema and use the **Upload** scalar. For example:

```
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!
  }
`;
```

The following schema enables file uploads. The **Upload** scalar will be injected as one of the arguments in the resolvers. The **Upload** scalar contains all file metadata and a Readable Stream that can be used to save the file to a specific location.

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

See Official Apollo blog post for more information.

11.2. IMPLEMENTING FILE UPLOAD ON THE CLIENT

Voyager Client provides support for uploading binary data along with the GraphQL queries. The binary upload implementation uses the **apollo-upload-client** package built by the Apollo community.

11.2.1. Introduction

The upload functionality uses the GraphQL multipart form requests specification. The File upload needs to be implemented on both server and client:

1. On the client HTML FileList objects are mapped into a mutation and sent to the server in a multipart request.

2. On the server: The multipart request is handled. The server processes it and provides an upload argument to a resolver. In the resolver function, the upload promise resolves an object.

NOTE

File upload is based on graphql-multipart-request-spec.

11.2.2. Enabling File Upload

File upload feature needs to be enabled by passing **fileUpload** flag to config object:

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

//create a new client
```

11.3. UPLOADING FILES FROM GRAPHQL

File upload capability adds a new GraphQL scalar **Upload** that can be used for mutations that operate on binary data. The **Upload** scalar maps html **FileList** HTML5 object in GraphQL schemas. The first step required to work with binary uploads is to write mutation that will contain **Upload** scalar. The following example demonstrates how to upload a profile picture:

```
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. Executing mutations

The **Upload** scalar will be mapped to object returned from HTML file input.

The following example shows file upload in a React application.

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

CHAPTER 12. RUNNING A DATA SYNC APP ON RED HAT MANAGED INTEGRATION

12.1. DEPLOYING YOUR DATA SYNC SERVER APPLICATION

Prerequisites

- You have a Data Sync server application working locally

Procedure

1. Log in to the Solution Explorer.
2. Navigate to the OpenShift console.
3. Click **Create Project**.
4. Enter the details for your application, when prompted.
5. Navigate to the **Project Overview** screen.
6. Search for the **Data Sync App** in the Service Catalog.
7. In the **Configuration** section:
    a. Enter the Git URL for the application repository.

    > **NOTE**
    >
    > To use a private repository, see Creating New Applications.

    b. Enter information for the required fields (indicated by *).

    c. Complete any optional fields, if necessary.
8. Complete the Wizard to start provisioning the Data Sync server application.
9. Wait for the service to display a ready status.
10. On the **Project Overview** screen, use the application URL displayed in the top right corner to verify your application is available.

12.2. CONNECTING THE DATA SYNC CLIENT TO YOUR DATA SYNC SERVER APPLICATION

Prerequisites

- You have deployed your Data Sync server application.
- You have set up a web project that supports ES6. For example:
    - Using Create React App

- Using Ionic Getting Started
- Using Getting Started with Angular
- Using Webpack Getting Started Guide

Procedure

1. Get the hostname of the Data Sync Server application.
   a. In your teriminal, run the command:

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

   b. Verify the output as:

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

   c. Record the value for **<sync-server-hostname>**.

2. Make sure the **@aerogear/voyager-client**, **graphql**, and **graphql-tag** libraries are added to your project. If necessary, add them by using the following commands:

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

3. In your project source code, import and configure the client using the server hostname.

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

   The client is now ready to make queries and mutations to the Data Sync server application.

Revised on 2020-10-12 16:53:10 UTC