Red Hat Mobile Application Platform 4.7

Server-side Developer Guide

For Red Hat Mobile Application Platform 4.7

Last Updated: 2018-10-25

Red Hat Mobile Application Platform 4.7 Server-side Developer Guide

For Red Hat Mobile Application Platform 4.7

Legal Notice

Abstract

This document provides guides for development of Cloud Apps and services in Red Hat Mobile Application Platform 4.7.

Table of Contents

CHAPTER 1. DEVELOPING CLOUD APPS

1.1. CLOUD DEVELOPMENT

1.1.1. Overview

One of the core concepts in the Red Hat Mobile Application Platform (RHMAP) are *Cloud Apps* — server-side applications which handle communication between Client Apps deployed on mobile devices, and back-end systems which contain business logic and data.

Cloud apps are developed using Node.js — *a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.*

For those not familiar with Node.js development, the Node Beginner Website is an excellent resource.

1.1.2. Cloud App Structure

When a Cloud App is created in RHMAP, this newly created Cloud App is essentially a pre-configured Node.js application. If a corresponding pre-configured Client App is also generated, the Cloud App will include all basic configuration (routes) so that the Client and Cloud App are in sync. In addition, the infrastructure for hosting the Cloud App is also in place (MBaaS) and is easily accessible.

As the configuration and infrastructure are all in place, the Cloud App can be deployed to an MBaaS and is capable of routing the Client App's requests without the need of additional code. Should an advanced Node.js developer wish to re-engineer the server or implement their own server, this is possible through the manipulation of the code within the *application.js* file.

NOTE

Ensure that the file that represents the main entry point to your App is always called *application.js*.

The following files are provided by default in a Cloud App:

1.1.2.1. application.js

This file is invoked when your application is deployed to *an MBaaS* - our cloud execution environment. Typically, you will not need to modify this file and it is by default set to handle **`fh.cloud()`** requests from the client and route them accordingly.

Should you need to modify this file, Red Hat recommends that you use *fh lib* and that you also add routes directly *after* **`app.use(mbaasExpress.fhmiddleware());`** in the application.js file otherwise there is a risk that the cloud section in the analytics panel will not display data. See the following snippet of code taken from from the application.js file for reference:

```
// Note: important that this is added just before your own Routes
app.use(mbaasExpress.fhmiddleware());

//Example of a route added in the correct position
app.use('/myroute', require('./lib/example.js')());
```

```
// Important that this is last!
app.use(mbaasExpress.errorHandler());
```

1.1.2.2. package.json

The configuration file for your Cloud App is primarily used for dependency management. If you are using third-party Node modules, they will be specified in this file.

See also:

- Using Node.js Modules in Cloud Apps
- Node.js Dependency Management Using npm
- official documentation of **`package.json`**.

1.1.3. Example

There are only a few steps to get a Cloud App running. You can explore a simple example by trying the *Hello World Project* template when creating a new project in the Studio, or by looking at the source code of the template on GitHub: feedhenry-templates/helloworld-cloud. This is a brief overview of the template.

The first step is to configure a custom route in application.js:

```
app.use('/hello', require('./lib/hello.js')());
```

In the referenced **`hello.js`** file, you would define the logic for that particular route.

```
var express = require('express');
var bodyParser = require('body-parser');
var cors = require('cors');

function helloRoute() {
  var hello = new express.Router();
  hello.use(cors()); // enables cross-origin access from all domains
  hello.use(bodyParser()); // parses POST request data into a JS object

  hello.post('/', function(req, res) {
    res.json({msg: 'Hello ' + req.body.hello});
  });
  return hello;
}
module.exports = helloRoute;
```

The client-side call to this endpoint using **`$fh.cloud`** might look like the following:

```
$fh.cloud({
    path: 'hello',
    data: {
      hello: 'World'
    }
  },
  function (res) {
```

```
    // response handler
  },
  function (code, errorprops, params) {
    // error handler
  }
);
```

1.2. ENVIRONMENTS

1.2.1. What are Environments

Environments are a way of logically separating the development cycles of Projects, Services, Forms and Apps into discrete stages. Environments are a key part of Lifecycle Management.

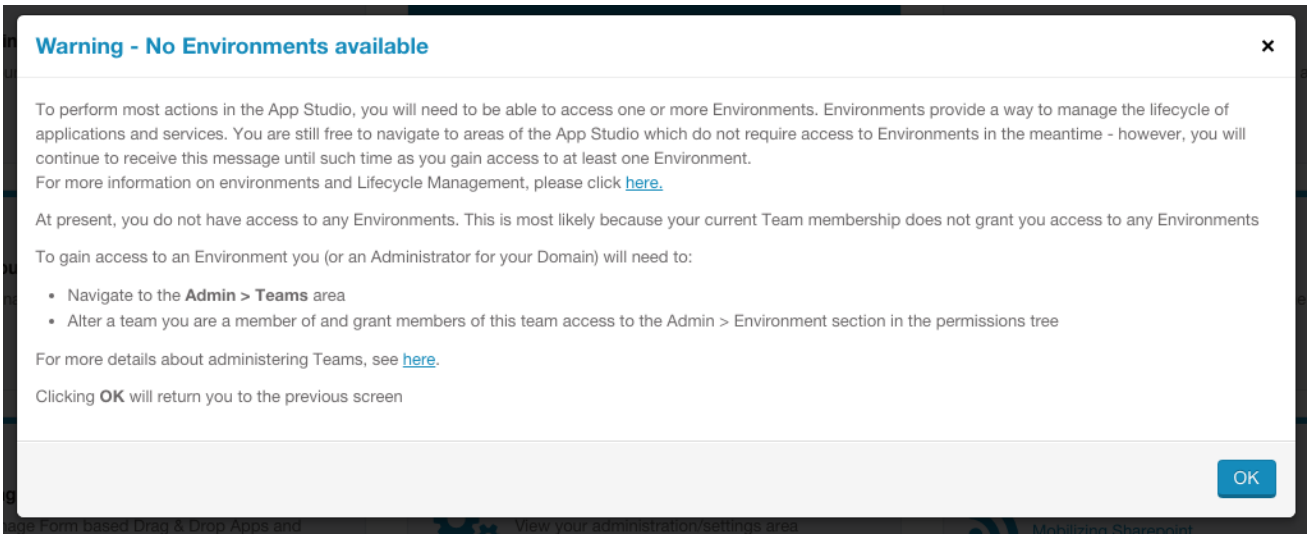
For example, a Project could have 3 stages of development: Dev, UAT & Production. Each stage is represented by an Environment. The number and make up of Environments is configurable through the Platform and can be tailored on a per-domain basis.

1.2.2. How Environments interact with Business Objects

Below is a quick reference for identifying areas in the App Studio where Environments are utilised

- Projects (including Apps)
- Services
- Forms
- Admin > Auth Policies

Should you access these areas without access to any Environments, you will see a warning similar to the one below:



1.2.3. No Environments — reduced functionality

To perform most actions relating to Projects, Services, Forms and Apps in the App Studio, you will need to be able to access one or more Environments.

If you are currently unable to access any Environments, this could be because:

- Your current Team membership excludes you from utilising any Environments
- Your Administrator has not setup any Environments yet

1.2.3.1. Update Team Membership

To gain access to an Environment you (or an Administrator for your Domain) will need to:

- Navigate to the **`Admin > Teams`** area
- Alter a team you are a member of and grant members of this team access to the **`Admin > Environment`** business object

For more details about administering Teams, see here.

1.2.3.2. Administrator Environment Setup

If you believe your Team setup and membership is correct, contact your local Administrator for further assistance. Environments need to be set up by an Administrator before access to them can be granted via Teams.

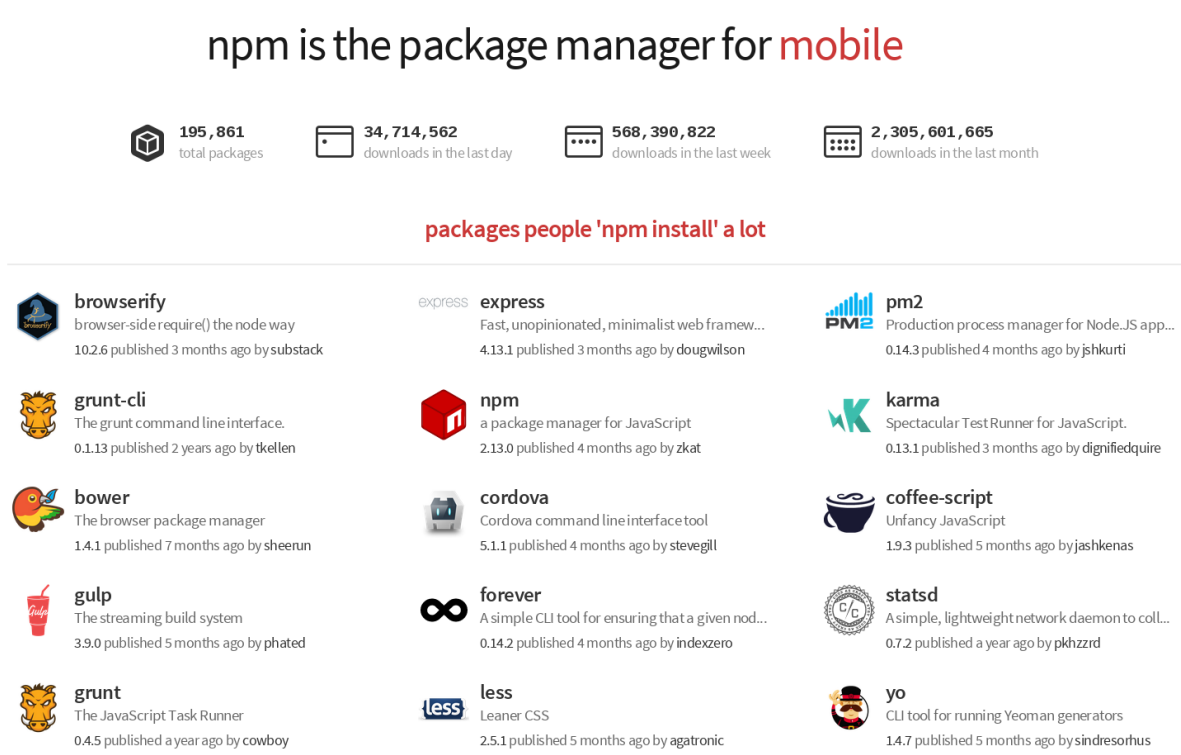
1.2.4. Other resources

- Teams and Collaboration
- Lifecycle Management

1.3. USING NODE.JS MODULES IN CLOUD APPS

1.3.1. Overview

This guide shows you how to include and use Node.js modules in your Cloud Apps. In the Red Hat Mobile Application Platform (RHMAP), you can use the modules of any Node.js package available in the registry to develop your Cloud Apps. In the public registry - npmjs.com - you can find thousands of packages - frameworks, connectors, and various tools.

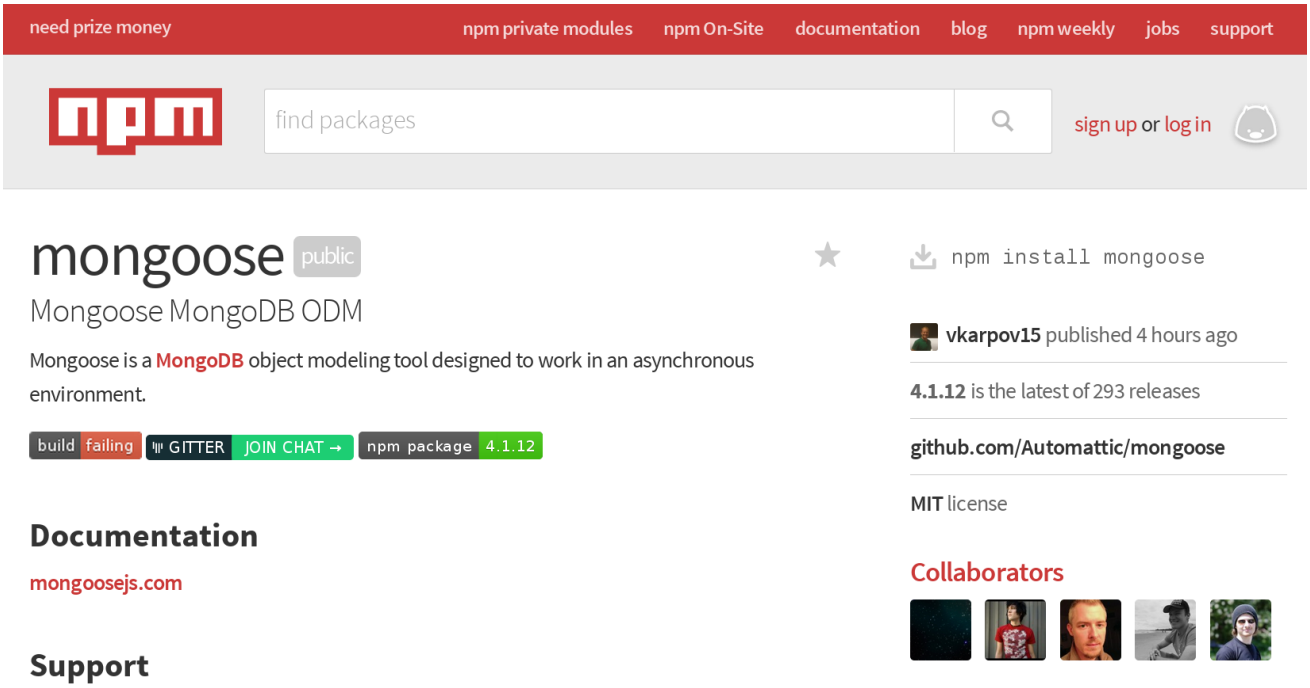


Parts of functionality of RHMAP itself are exposed as node modules. For example, when you create a new Cloud App on the Platform, by default the *package.json* will include a node module called *fh-mbaas-api*, which is the cornerstone of the Mobile Backend As A Service (MBaaS) in RHMAP.

1.3.2. Getting Started with Node.js Modules

If you need a piece of functionality in your app and you don't want to reinvent the wheel, it's worth searching for existing modules on npmjs.com.

For example, if you want to use the *Mongoose* ODM framework, you can try looking at the official page of the **mongoose** package in the registry - npmjs.com/package/mongoose.



To use the latest version of the package in your project, running the following command in your Cloud App's directory will install the package locally and add it as a dependency to the **`package.json`** file of your Cloud App:

```
npm install --save mongoose
```

Alternatively, you can add the dependency to **`package.json`** manually, by appending it at the end of the **`dependencies`** object, as in the following example:

```
{
    "name": "my-fh-app",
    "version": "0.2.0",
    "dependencies": {
        "body-parser": "~1.0.2",
        "cors": "~2.2.0",
        "express": "~4.0.0",
        "fh-mbaas-api": "~4.9.0",
        "mocha": "^2.1.0",
        "request": "~2.40.0",
        "mongoose": "~4.1.12" // added mongoose dependency
    }
}
```

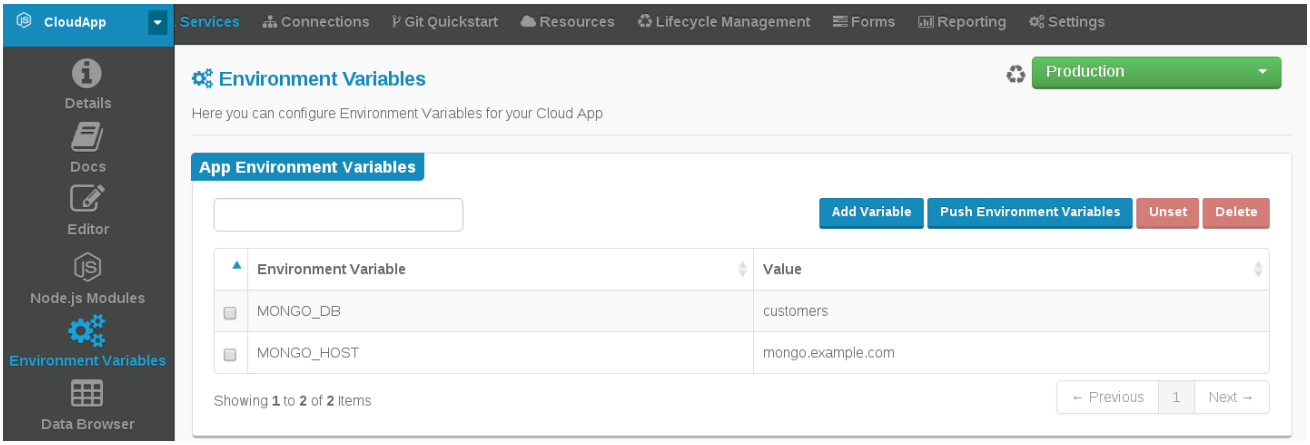
At this point, you can start using the modules provided by the package. Often, the package page on npmjs.com contains instructions for usage and links to documentation. The next step common to all modules is *requiring* the module in your code to be able to use it:

```
var mongoose = require('mongoose');
mongoose.connect('mongodb://localhost/my_database');
...
```

In this example, the database URL is hard-coded in the source. A more common scenario, however, is using different databases for different stages of development life cycle - development, testing, production. This can be resolved by defining configuration values through environment variables.

1.3.3. Environment Variables

If any part of the configuration of your Cloud App - such as hostnames, usernames, or passwords - varies between life cycle stages, you can set such configuration values using environment variables. That allows you to make configuration changes without changing the code. Every Cloud App has an *Environment Variables* section in the Studio where you can create or remove variables, set values, or push the variables to a running instance of the app.



Environment variables are especially useful in combination with the ability to *push* different values to different environments. For example, you could be using a different database host for testing and for production. Another benefit is that sensitive configuration information can be stored outside of the code base of your application.

In a Cloud App, environment variables can be accessed through the **`process.env`** object. The previous example of a Mongoose connect call changed to use environment variables would look this way:

```
mongoose.connect('mongodb://' + process.env.MONGO_HOST + '/' +
process.env.MONGO_DB);
```

1.4. NODE.JS DEPENDENCY MANAGEMENT USING NPM

1.4.1. Overview

npm, the *node package manager*, is a dependency management tool for Node.js and is an integral part of developing any Node.js application. In its principles, npm is similar to any other dependency management system, like Maven, Cocoapods, or NuGet:

- projects declare dependencies in a standardized file - **`package.json`** in case of Node.js;
- packages are hosted in a central location - the npm registry at registry.npmjs.org;
- dependency versions can be specified explicitly, or using wildcards or ranges;
- versions of transitive dependencies can be decided automatically, or fixed (see Using an npm-shrinkwrap file);
- packages are cached locally after installation (**`node_modules`** folder).

For an introduction to using Node.js modules in your Cloud Apps, see Using Node.js modules in Cloud Apps.

1.4.2. npm and App Staging

When deploying your Cloud App, the *npm* command is run in your app's environment, to download and install the dependencies of your app.

NOTE

The first time your Cloud App is deployed, the build can take some time as a full **`npm install`** is performed.

For each subsequent deployment:

- when deploying to a hosted MBaaS (RHMAP 3.x):
  - If the package.json file has changed from the previous deploy, RHMAP runs **`npm update`**.
  - If no changes have been made to package.json, no npm command is run (neither *update*, nor *install*).
  - To perform a clean stage, and run **`npm install`**:
    - In the Studio, check the *Clean Stage* check box in the *Deploy* section of the Cloud App
    - If using fhc, use the *--clean* option, that is, **`fhc app stage <app-id> <env-name> --clean`**
- when deploying to a self-managed MBaaS (RHMAP 4.x):
  - If either the gitref or the nodejs runtime are changed then a new image is built.
  - If the gitref and Node.JS Runtime are unchanged from a previous build then the existing image is used where possible.
  - To force a fresh image to be built use the *Clean Stage* check box in the *Deploy* section of the cloud App.

1.4.3. npm Best Practices

The fundamental best practice with npm is to understand how versioning works with node modules (see The semantic versioner for npm) and to avoid using **`*`** for any dependencies in package.json.

A handy tip for developing locally is to use the **`--save`** flag when installing a module using npm, for example, **`npm install request --save`**. The **`--save`** flag will append the new module to the dependencies section in your package.json with the version that has just been installed. We also recommend using a *shrinkwrap* file.

1.4.3.1. Using an npm-shrinkwrap file

An *npm-shrinkwrap.json* file locks down the versions of a package's transitive dependencies so that you can control exactly which versions of each dependency will be used when your package is installed.

To create a shrinkwrap file, run:

```
npm shrinkwrap
```

Put the resulting **`npm-shrinkwrap.json`** file to the root directory of your Cloud App. When **`npm install`** is invoked on your app in RHMAP, it will use exactly the versions defined in the shrinkwrap file.

To learn more about shrinkwrap, see the official npm documentation - Lock down dependency versions.

1.4.4. Uploading node_modules

RHMAP allows you to commit your **`node_modules`** directory and to use this in your Cloud App, even though it's **not** a recommended practice. That way, npm is not run at all when your app is staged (even if the **`clean`** option is specified), and whatever modules you've uploaded will be used to run your app. However, native node modules need to be compiled on the same architecture on which they execute, which might vary between instances of RHMAP.

1.5. SETTING THE NODE.JS VERSION

1.5.1. Using fhc

Firstly ensure you have the latest version of fhc.

```
npm install -g fh-fhc
```

You can then use the fhc runtimes command to see which runtimes are available to you in an Environment.

```
fhc runtimes --env=dev
```

This command will output the version, for example, **`4.4.7`**.

On the left is the name of the runtime and on the right is the particular version.

```
fhc app stage --app=<APP_GUID> --runtime=<node version>
```

To set the app's runtime during a stage, add the <node version>, for example, **`4.4.7`**.

This will set your apps runtime to v4.4.7 of node.

To change it to a different runtime, add the runtime argument to the stage command again

1.5.2. Using the studio

In the Studio from the deploy screen you can see the current runtime the app is set to. You can also set a new runtime here and then deploy your app.

CHAPTER 2. USING RHMAP FEATURES IN CLOUD APPS

2.1. DEVELOPING AN APPLICATION USING PUSH NOTIFICATIONS

Overview

This tutorial will guide you through the process of building a sample application which receives push notifications sent from the Platform's built-in push notification server. The application you'll create in this example is based on Cordova and targets Android and the associated Firebase Cloud Messaging (formerly Google Cloud Messaging) platform. However, the steps are analogous for all other supported platforms.

To learn more about push notification support in the Red Hat Mobile Application Platform, see Push Notifications.

2.1.1. Obtain Firebase Cloud Messaging credentials and download the `google-services.json` file

For the built-in UnifiedPush Server (UPS) to be able to access Google's push notification network, you first need to get a Server key and establish a project in the Firebase Console. See Obtaining Firebase Cloud Messaging Credentials for a detailed walk-through of getting the credentials. These instructions are for a native Android app, you do not need to complete all the process. In order to successfully complete the setup you need to obtain the following three items:

- Server key (formerly API key)
- Sender ID (formerly Project Number)
- the **`google-services.json`** file generated from the Firebase console

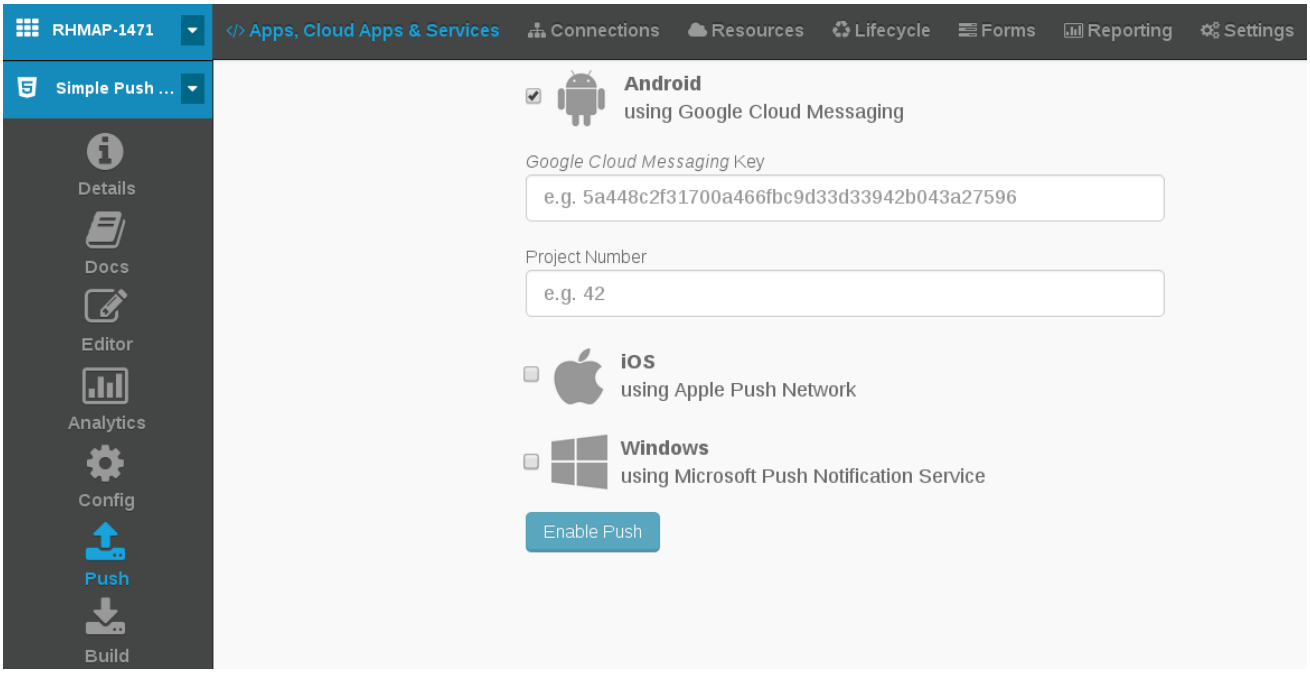
For other push networks, see Obtaining Credentials for Push Networks.

2.1.2. Create a project for your Push Notification app

Create a new project for your Push Notification app. Use the **Push Notification** templates as a reference.

2.1.3. Set up push support

Push support needs to be explicitly enabled for each Client App and you also need to provide the push network credentials obtained in the first step Obtain Firebase Cloud Messaging credentials and download the **`google-services.json`** file.



1. In the *Apps, Cloud Apps & Services* section of your project, click the *Simple Cordova Push App*.
2. Click *Push* on the left side of the screen.
3. Click *Enable Push*.
4. Select the *Android* option and enter the *Server key* and *Sender ID* obtained in the first step. Click *Enable Push*.

2.1.4. Integrate the `google-services.json` file into the Client App

1. Click *Editor* on the left-hand side of the screen.
2. Select the **`www`** folder in the Editor, click the **+** symbol in the toolbar to create a new file, and name it **`google-services.json`**.
3. Open the **`google-services.json`** file you have previously downloaded from the Firebase Console, copy the contents into the **`google-services.json`** in the Editor.
4. Click *File*, and then *Save* in the Editor toolbar to save your changes.

2.1.5. Configure config.xml file

Modify the **`widget id`** setting in the config.xml file to correspond to the package name defined in the Firebase Console.

2.1.6. Build the Client App binary

Refer to Build the Client App binary.

2.1.7. Test the app

Refer to Test the App.

2.2. DYNAMICALLY POPULATING FORM FIELDS FROM AN MBAAS SERVICE

2.2.1. Overview

You can use an endpoint of an MBaaS service as the data source for a form field in the Forms Builder. This guide shows you how to create the MBaaS service, define the data source, and populate a form field using the data source.

See Using Data Sources for more information on data sources for Forms.

2.2.2. Creating a Service for a Data Source

To define a data source, you must provide an MBaaS service endpoint which serves valid JSON responses for the data source. In this example, you'll create a new MBaaS service which serves static data from a JSON file.

1. In the Studio, navigate to *Services & APIs*. Click *Provision MBaaS Service/API*.
2. Click *Import Existing Service*, provide a name and click next
3. Select *Public Git Repository* when the *Import From* option is expanded
4. Choose the URL of one of the repositories available here and paste it into Git URL textfield
5. Click *Next*
6. From the left naavigation bar select *Deploy* and click *Deploy Cloud App* to deploy your service

2.2.3. Defining a Data Source

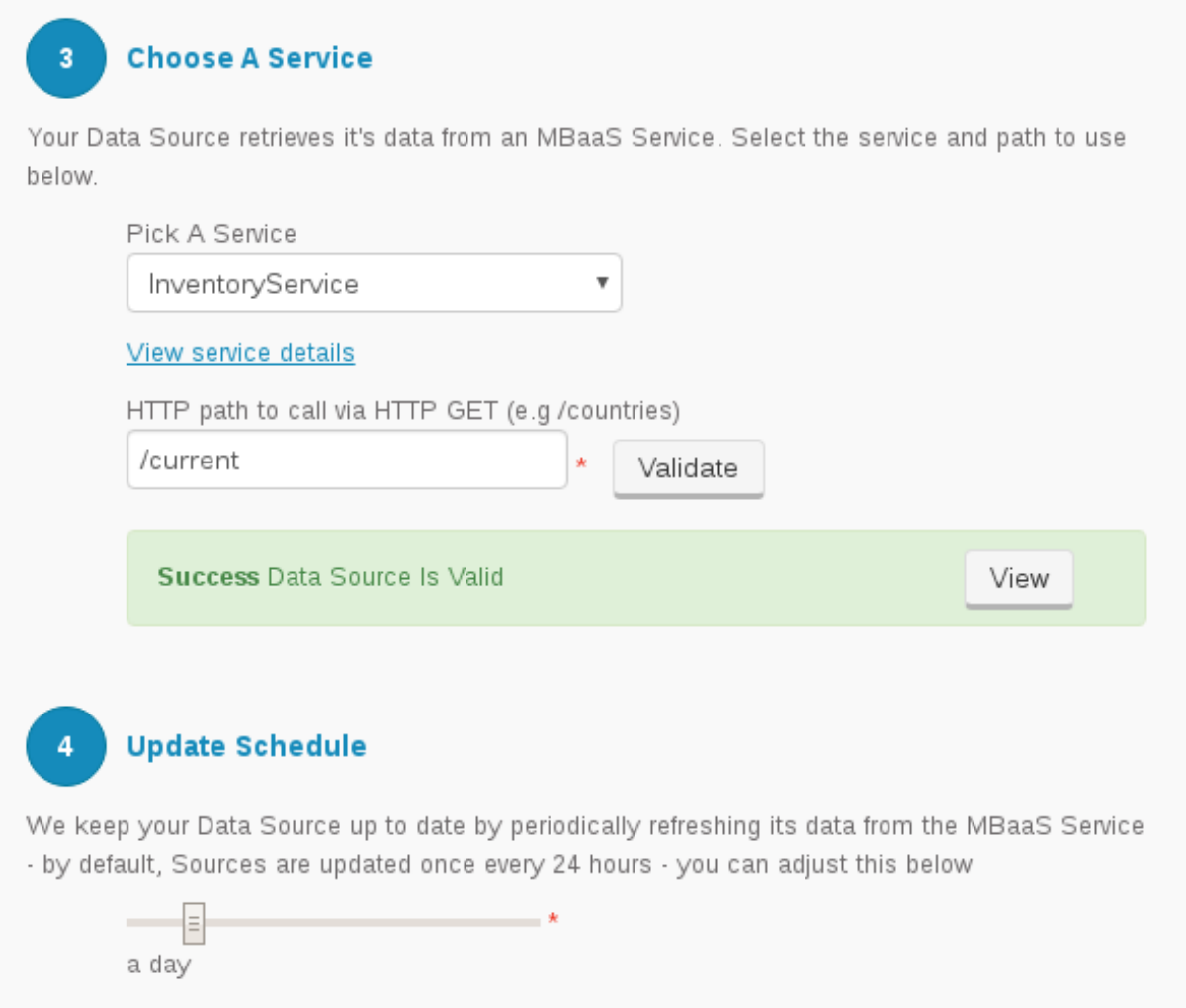
Once you've created the MBaaS service providing the data, you can now use it in a data source.

1. In the Studio, navigate to *Drag & Drop Apps > Data Sources*. Click *New Data Source*.
2. Set a name and a description. Both are required.
3. Choose the previously created MBaaS service and its endpoint to use as the data source. The *Static Data Source* template used in this example serves data at the endpoint `/static_ds/months`.

   You can check, whether the service returns data valid for use as a data source, using the *Validate* button. Before the validation, make sure the correct environment is selected using the environment selector in the top right-hand corner.

   If the data is valid, you'll see the message *"**Success** Data Source Is Valid"*. If the data is not valid, or the service can not be reached, an error message will indicate the problem.

   After a successful validation, you can view the returned data using the *View* button.



4. Choose an update frequency – how often the configured MBaaS service endpoint will be called to obtain new data. The update frequency is a value in the range of 1 minute to 7 days.

5. Choose how many responses from calls to the MBaaS service endpoint should be kept in the audit log.

6. Click *Create Data Source*. You will see the message *Data Source Created Successfully*.

Now, the data source can be used in the Forms Builder to populate a form field.

2.2.4. Using a Data Source in a Form Field

After defining the data source, you can now use it in the Forms Builder.

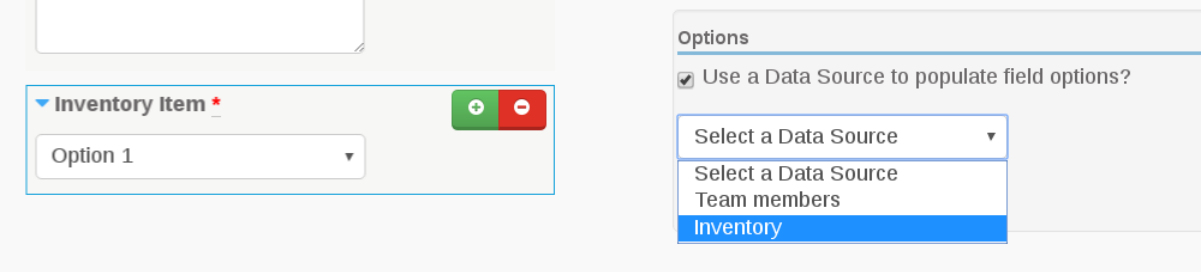
1. In the Studio, navigate to *Drag & Drop Apps > Forms Builder*.

2. Open an existing form, or create a new form. Navigate to *Edit Form*.

3. Drag one of the supported types into the form – for example, a *Dropdown* field. Select the created field.

4. In the *Options* section, check *Use a Data Source to populate field options?*.

5. Select the data source created in the previous step.



If a selected data source has loaded data at least once before – for example, while validating during creation – the last available version of the data will be displayed in the field options, as a preview.

Use the environment selector in the top right-hand corner to ensure that the same environment is selected, where the MBaaS service of the data source is deployed.

6. Save and deploy the form.

If you navigate to the *Dashboard*, you can now see the field populated by data from the data source in the preview on the right.

2.3. ADDING STATS TO YOUR APP

2.3.1. Overview

The Platform maintains counters and timers that can be accessed by the user. Some are provided by the platform, while others can be specified by the developers of an application.

Counters can be used to track the usage of particular functionality, counters can be incremented and decremented. The platform provides counters that show currently active server-side actions, that is, the number of server-side functions that have been called by a mobile application that are currently in progress.

Timers can be used to track the length of time taken by specified actions. The platform provides timers to track the execution-time of the server-side actions. Periodically, at a platform-configured interval, the current counters, and timers and pushed to a history and zeroed.

When requesting statistics from the platform, the number of recent intervals to return data for, and the length of the flush interval, in milliseconds, is returned with the data.

Timer data is returned as the number of timing events that have occurred in the interval, with upper and lower values, there are also upper and mean values for the 90th percentile.

2.3.2. Creating Counters and Timers

App developers can create their own counters and timers using:

```
$fh.stats.inc('COUNTERNAME'); // Increments a counter
$fh.stats.dec('COUNTERNAME'); // Decrements a counter
$fh.stats.timing('TIMERNAME', timeInMillis); // Store a timer value
```

For more detailed documentation about using the Stats API in your Cloud App, see:

- $fh.stats API Reference

Developers' counters and timers can be requested from the platform, by specifying the statstype of "app”.

The keynames of the counters are of the form **`DOMAIN_APPID_app_COUNTERNAME`** where **`DOMAIN_APPID`** is the name of the domain and **`APPID`** is the ID of the app, and **`COUNTERNAME`** is the developer supplied name for the counter.

The keynames of the timers are of the form **`DOMAIN_APPID_app_TIMERNAME`** where **`DOMAIN_APPID`** is the name of the domain and **`APPID`** is the ID of the app, and **`TIMERNAME`** is the developer supplied name for the timer.

2.3.3. Viewing Stats

A user can access stats data through the Studio or the raw data itself through FHC, the command line client.

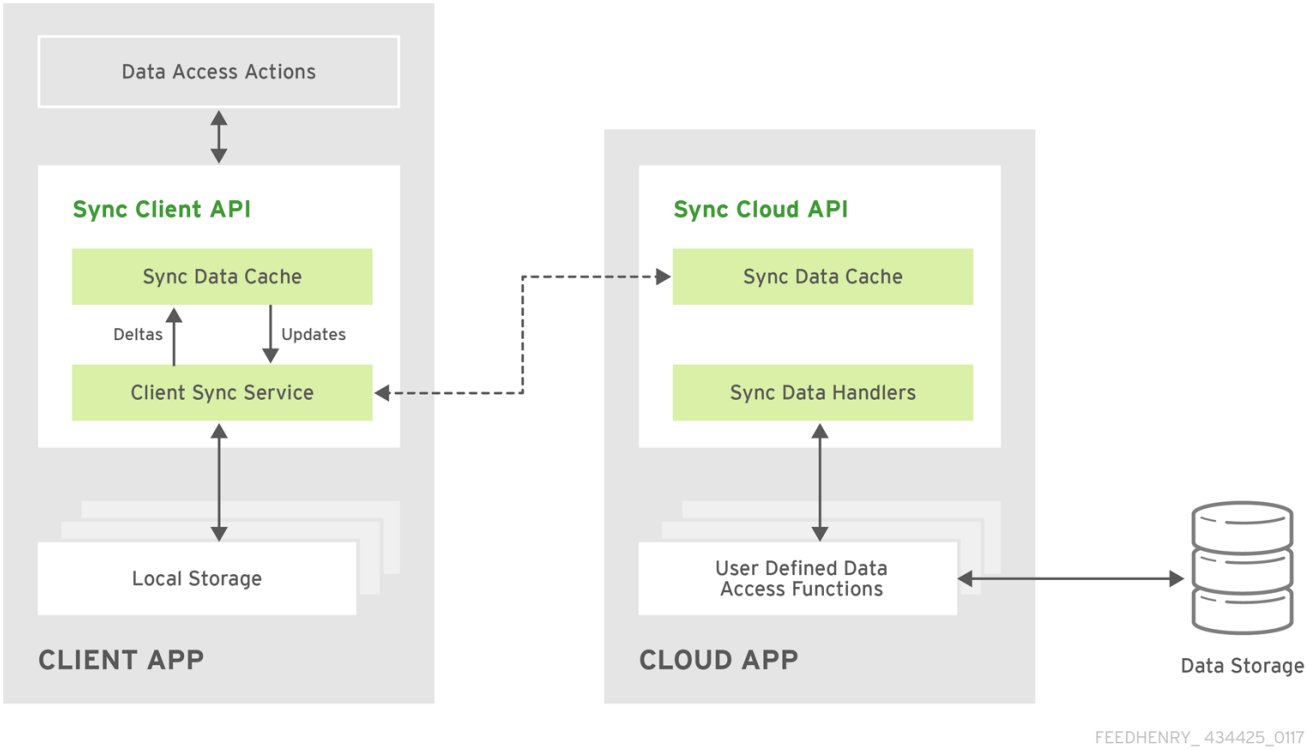
CHAPTER 3. USING RHMAP DATA SYNC FRAMEWORK

3.1. DATA SYNC FRAMEWORK

The RHMAP mobile data synchronization framework includes the following features:

- Allows mobile apps to use and update data offline (local cache)
- Provides a mechanism to manage bi-directional data synchronization from multiple Client Apps using the Cloud App and into back-end data stores
- Allows data updates (that is, deltas) to be distributed from the Cloud App to connected clients
- Enables data collision management from multiple updates in the cloud
- Allows RHMAP Apps to seamlessly continue working when the network connection is lost, and allows them to recover when the network connection is restored.

3.1.1. High Level Architecture



NOTE

Please refer to the Data Sync Framework Terminology defined in Sync terminology.

The Sync Framework comprises a set of Client App and Node.js Cloud App APIs.

The Client App does not access the back-end data directly. Instead, it uses the Sync Client API to Read and List the data stored on the device and send changes (Creates, Updates, and Deletes) to the Cloud App. Changes made to the data locally are stored in the local Sync Data Cache before being sent to the Cloud App. The Client App receives notifications of changes to the remote dataset from the Sync Client API.

The Client App then uses the Client Sync Service to receive the changes (deltas) made to the remote dataset from the Cloud App and stores them in the Sync Data Cache. The Client App also sends Updates made locally to the Cloud App using the Client Sync Service. When the Client App is off-line, cached updates are flushed to local storage on the device, allowing the changes to persist in case the Client App is closed before network connection is re-established. The changes are pushed to the Cloud App the next time the Client App goes online.

The Cloud App does not access the Back End data directly, but only through the Sync Cloud API. The Cloud App Uses the Sync Cloud API to receive updates from the Client App using the Client Sync Service. These updates are stored in the Cloud App Sync Data Cache. The Cloud App uses the Sync Cloud API to manage the Back End data in hosted storage using standard CRUDL (create, read, update, delete and list) and **`collisionHandler`** functions.

In addition to the standard data handler functions, the Cloud App can also employ user-defined data access functions.

3.1.2. API

The Client and Node.js API calls for Sync are documented in the following guides:

- JavaScript SDK API Sync section.
- Node.js API Sync section.
- iOS SDK Docs
- Android SDK Docs

3.1.3. Getting Started

To implement the Sync framework in your App:

1. Init $fh.sync on the client side

```
  //See [JavaScript SDK API](../api/app_api.html#app_api-_fh_sync)
for the details of the APIs used here

  var datasetId = "myShoppingList";

  //provide sync init options
  $fh.sync.init({
    "sync_frequency": 10,
    "do_console_log": true,
    "storage_strategy": "dom"
  });

  //provide listeners for notifications.
  $fh.sync.notify(function(notification){
    var code = notification.code
    if('sync_complete' === code){
      //a sync loop completed successfully, list the update data
      $fh.sync.doList(datasetId,
        function (res) {
          console.log('Successful result from list:',
JSON.stringify(res));
          },
```

```
        function (err) {
          console.log('Error result from list:',
JSON.stringify(err));
        });
    } else {
      //choose other notifications the app is interested in and
provide callbacks
    }
  });

  //manage the data set, repeat this if the app needs to manage
multiple datasets
  var query_params = {}; //or something like this: {"eq": {"field1":
"value"}}
  var meta_data = {};
  $fh.sync.manage(datasetId, {}, query_params, meta_data,
function(){

  });
```

About Notifications

The Sync framework emits different types of notifications during the sync life cycle. Depending on your app's requirements, you can choose which type of notifications your app listens to and add callbacks. However, it's not mandatory, the Sync framework performs synchronization without notification listeners.

Adding appropriate notification listeners helps improve the user experience of your app:

- Show critical error messages to the user in situations where Sync framework errors occur. For example, **`client_storage_failed`**.
- Log errors and failures to the console to help debugging. For example, **`remote_update_failed`**, **`sync_failed`**.
- Update the UI related to the sync data if delta is received, for example, there are changes to the data, you can use **`delta_received`**, **`record_delta_received`**.
- Monitor for collisions.
  Make sure to use $fh.sync APIs to perform CRUDL operations on the client.

2. Init $fh.sync on the cloud side
   This step is optional, and only required if you are overriding dataset options on the server, for example, modifying the sync loop frequency with the Dataset back end. See the Considerations section below if changing the default sync frequency.

```
var fhapi = require("fh-mbaas-api");
var datasetId = "myShoppingList";

var options = {
  "syncFrequency": 10
};

fhapi.sync.init(datasetId, options, function(err) {
  if (err) {
    console.error(err);
```

```
  } else {
    console.log('sync inited');
  }
});
```

You can now use the Sync framework in your app, or use the sample app to explore the basic usage: Client App and Cloud App.

If the default data access implementations do not meet your requirements, you can provide override functions.

3.1.3.1. Avoiding Unnecessary Sync Loops

Because the client and server sync frequencies are set independently, two sync loops may be invoked per sync frequency if the server-side sync frequency differs from the client-side frequency. Setting a long frequency on a client does not change the sync frequency on the server. To avoid two sync loops, set the syncFrequency value of the dataset on the server to the sync_frequency value of the corresponding dataset on the client.

For example:

- syncFrequency on the server-side dataset is set to 120 seconds.
- sync_frequency on the client-side dataset is also set to 120 seconds.

However, if you require different frequencies on the client and server, you can set different values.

3.1.4. Using Advanced Features of the Sync Framework

3.1.4.1. Define the Source Data for a Dataset

The Sync Framework provides hooks to allow the App Developer to define the source data for a dataset. Typically, the source data is an external database (MySql, Oracle, MongoDB etc), but this is not a requirement. The source data for a dataset can be anything, for example, csv files, FTP meta data, or even data pulled from multiple database tables. The only requirement that the Sync Framework imposes is that each record in the source data has a unique Id and that the data is provided to the Sync Framework as a JSON Object.

In order to synchronize with the back end data source, the App developer can implement code for synchronization.

For example, when listing data from back end, instead of loading data from database, you might want to return hard coded data:

1. Init $fh.sync on the client side
   This is the same as Step 1 in Getting Started.
2. Init $fh.sync on the cloud side and provide overrides.

```
var fhapi = require("fh-mbaas-api");
var datasetId = "myShoppingList";

var options = {
  "syncFrequency": 10
};
```

```
//provide hard coded data list
var datalistHandler = function(dataset_id, query_params, cb,
meta_data){
  var data = {
    '00001': {
      'item': 'item1'
    },
    '00002': {
      'item': 'item2'
    },
    '00003': {
      'item': 'item3'
    }
  }
  return cb(null, data);
}

fhapi.sync.init(datasetId, options, function(err) {
  if (err) {
    console.error(err);
  } else {
    $fh.sync.handleList(datasetId, datalistHandler);
  }
});
```

Check the Node.js API Sync section for information about how to create more overrides.

3.2. SYNC TERMINOLOGY

3.2.1. Sync Protocol

The protocol for communication between the Sync Client and the Sync Server.

3.2.2. Sync Server

The Sync Server is the server part of the Sync Protocol, and is included in the fh-mbaas-api module. It:

- exposes the Sync Server API for integrating with a DataSet Backend
- runs the Sync Server Loop, updating DataSets when updates from a DataSet Backend are detected.

3.2.3. Sync Client

- exposes the Sync Client API to the front end for CRUDL actions against a DataSet
- runs the Sync Client Loop, making a call to the Sync Server at the specified Sync Frequency for a particular DataSet.

The Sync Client is the client part of the Sync Protocol There are 3 Sync Client implementations available:

- Javascript, in the FeedHenry Javascript SDK
- Objective C, in the FeedHenry iOS SDK

- Java, in the FeedHenry Android SDK

3.2.4. Sync Server Loop

The Sync Server Loop is a function that runs continuously on the Sync Server with a 500ms wait between each run.
During each run, it iterates over all DataSet Clients to see if a DataSet should be synced from the DataSet Backend.

3.2.5. Sync Client Loop

The Sync Client Loop is a function that runs continuously on the Sync Client with a 500ms wait between each run.
During each run, it iterates over all DataSet Clients to see if a DataSet should be synced with the Sync Server.

3.2.6. Sync Frequency

On the Sync Client, this is the interval between checks for updates from the Sync Server for a particular DataSet.
On the Sync Server, this is the interval between checks for updates from the DataSet Backend for a particular DataSet.

For more information, see Configuring Sync Frequency.

3.2.7. DataSet

A DataSet is a collection of records synchronized between 1 or more Sync Clients, the Sync Server and the DataSet Backend.

Red Hat recommends that you use an indexing strategy when working with data sets to improve performance. For more information about indexing strategies for MongoDB, please see the MongoDB Manual.

3.2.8. DataSet Backend

The system of record for data synchronized between the Sync Client and the Sync Server.
It can be any system that provides an API and can be integrated with from the Sync Server, for example, a mysql database or a SOAP service.
The Sync Server exposes the Sync Server API for integration with a DataSet Backend using DataSet Handlers.

3.2.9. DataSet Handler

A DataSet Handler is a function for integrating the Sync Server into a DataSet Backend.
There are many handlers for doing CRUDL actions on a DataSet and managing collisions between DataSet Records.
The default implementation of these handlers uses fh.db (MongoDB backed in an RHMAP MBaaS).
You can override each of these handlers. See the Sync Server API for details.

IMPORTANT: If you are overriding handlers, Red Hat recommends overriding *all* handlers to avoid unusual behavior with some handlers using the default implementation and others using an overridden implementation.

3.2.10. DataSet Client

A DataSet Client is a configuration stored in the Sync Client & Sync Server for each DataSet that is actively syncing between the client & server.
It contains data such as:

- the Sync Frequency of the DataSet
- any query parameters to include when making calls to the DataSet Backend
- the latest Hash of the DataSet Records
- data regarding whether a sync with the DataSet Backend is currently in progress

3.2.11. DataSet Record

A DataSet Record is an individual record in a DataSet.
It contains:

- the raw data that this record represents, for example, the row values from a MySQL table
- a Hash of the raw data

3.2.12. Hash

There are 2 types of Hash used in the Sync Protocol:

- hash of an individual DataSet Record which is used to compare individual records to see if they are different.
- hash of all DataSet Records for a particular DataSet Client which is used to compare a client's set of records with the servers without iterating over all records.

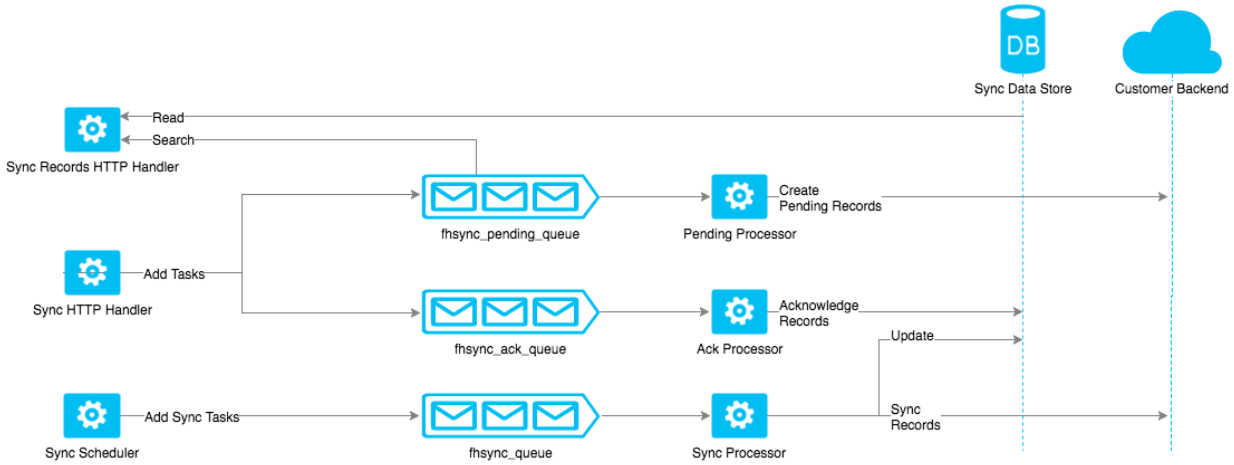
3.3. SYNC SERVER ARCHITECTURE

For a general overview of the Sync Framework, see Sync Overview and Sync Terminology.

3.3.1. Architecture

The Sync Server architecture includes: * HTTP handlers * Queues and processors * The sync scheduler

Each of these components persist data in MongoDB.



3.3.1.1. HTTP Handlers

These handlers are responsible for handling the Sync requests from Sync Clients.

3.3.1.1.1. Sync HTTP Handler

Creates or updates the Dataset Client and pushes pending records and acknowledgements on to the appropriate queues for processing.

3.3.1.1.2. Sync Records HTTP Handler

Compares up-to-date data with a client's state. After getting the delta, it checks for updates that are processed, but not yet synced. This handler iterates through all the records in the delta. If any records are in the pending queue or have been applied, this handler removes them from the delta and returns the updated delta to the client.

3.3.1.2. Queues

The following queues are used in the Sync Framework:

- **`fhsync_queue`** - jobs for datasets that require synchronization.
- **`fhsync_ack_queue`** - jobs for pending changes that require acknowledgement.
- **`fhsync_pending_queue`** - jobs for pending changes that require processing.

Messages are placed on these queues and are consumed by processors.

3.3.1.3. Processors

Each queue has a corresponding processor:

- Sync Processor - takes jobs from **`fhsync_queue`** and processes those jobs.
- Ack Processor - takes acknowledgements from **`fhsync_ack_queue`** and removes those acknowledgements from MongoDB.
- Pending Processor - takes pending items from **`fhsync_pending_queue`** and applies the changes to the Dataset Backend.

Each worker in a Sync Server has one instance of each of these processors allowing the tasks to be distributed.

3.3.1.4. Sync Scheduler

When horizontally scaled, each Sync Worker attempts to become the Sync Scheduler at fixed intervals. Each worker tries to obtain a lock which is located in MongoDB. The worker that has the Sync Scheduler lock determines which Datasets need to be synchronized by looking at the timestamp of the last synchronization and the sync frequency for the Dataset. If a Dataset needs to be synchronized, a job is added to **`fhsync_queue`**.

3.4. DATA SYNC CONFIGURATION GUIDE

Data Sync configuration can be applied to the client-side and also in the cloud (server-side).

The sync frequency on the client-side is set using the **sync_frequency** variable. To see an example of **sync_frequency** being set, see the code in this section of the documentation.

The sync frequency in the cloud is set using the **syncFrequency** variable. To see an example of **syncFrequency** being set, see the code in this section of the documentation.

3.4.1. Configuring Sync Frequency

The sync frequency is the time period the system waits between 2 sync processes.

IMPORTANT: It is possible to configure the frequency differently on the client and server. However, Red Hat recommends using the same setting to avoid the following scenarios:

- The client calls more frequently than the server checks for updates from the DataSet Backend, causing unnecessary traffic from the client.
- the client calls less frequently than the server checks for updates from the DataSet Backend, causing the server to drop its DataSet from the cache because of inactivity.

The sync frequency value of a server determines how often the sync processor runs. Every time the sync processor executes, it performs a list operation on the Dataset Backend to synchronize the data with a local copy. To determine the best value of the sync frequency for your application, review the following sections.

- How quickly do you want your clients to see changes from others?
  When a client submits changes, those changes are applied to the Dataset Backend directly. To ensure high performance, other clients get data from the local copy. This means other clients can only get the new changes after the next sync processor run. If it is required that other clients get the changes as soon as possible, then consider setting a low value for the sync frequency.
- How long it takes the sync processor to run?
  The sync frequency value determines how long the system waits between sync processor executions, that is, the sync frequency is the time from the completion of the one execution to the start time of next execution. This means there is never a situation where 2 sync processors are running at the same time. Therefore:

  ```
  actual sync period = sync processor execution time + the sync
  frequency
  ```

  This helps you calculate the number of requests the system makes to the Dataset Backend.

To determine how long each sync processor execution takes, you can query the sync stats endpoint to see the average **`Job Process Time`** it takes for the **`sync_worker`** to complete.

- How much load can the Dataset Backend service handle?
  Every time the sync processor runs, it performs a list operation on the Dataset Backend. When you configure the sync frequency, you need to estimate how many requests it generates on the backend, and make sure the backend can handle the load.

  For example, if you set the sync frequency of a dataset to 100ms, and each sync processor execution is taking 100ms to run, that means the server generates about 5 req/sec to the backend. However, if you have another dataset with a sync frequency of 100ms that uses the same backend, there will be about 10 req/sec to the backend. You can perform load tests against the backend to determine if the backend can handle that load.

  However, this value does not grow when you scale the app. For example, if you have multiple workers in your server, the sync processor executions are distributed among the workers rather than duplicated among them. This design protects the backend when the app is under heavy load.

- How much extra load does it cause to the server?
  When the data is returned from the backend, the server must save the data to the local storage (MongoDB). The system only performs updates if there are changes. But it needs to perform a read operation first to get the current data in the local storage. When there are a lot of sync processor executions, it could cause extra load on the server itself. Sometimes, you need to take this into consideration, especially if the dataset is large.

  To understand the performance of the server, you can use the sync stats endpoint to check CPU usage, and the MongoDB operation time.

You can use the sync frequency value to control the number of requests the server generates to the backend. It is acceptable to set it to 0ms, as long as the backend can handle the load, and the server itself is not over-loaded.

3.4.2. Configuring the Workers

There are different queues used to store the sync data, as described in the Sync Architecture. To process the data, a corresponding worker is created for each queue. Its sole task is to take a job off the queue, one at a time, and process it. However, there is an interval value for how long between finishing one job and getting the next available job. To maximize the worker performance, you can configure this value.

3.4.2.1. Purpose of the Intervals

The request to get a job off the queue is a non-blocking operation. When there are no jobs left on the queue, the request returns and the worker attempts to get a job again.

In this case, or if jobs are very fast to complete, a worker could overload the main event loop and slow down any other code execution. To prevent this scenario, there is an interval value configuration item for each worker:

- **`pendingWorkerInterval`**
- **`ackWorkerInterval`**
- **`syncWorkerInterval`**

The default interval value is very low (1ms), but configurable. This default value assumes the job is going to take some time to execute and have some non-blocking I/O operations (remote HTTP calls, DB calls, etc) which allows other operations to be completed on the main event loop. This low default interval allows the jobs to be processed as quickly as possible, making more efficient use of the CPU. When there are no jobs, a backoff mechanism is invoked to ensure the workers do not overload resources unnecessarily.

If the default value is causing too many requests to the Dataset Backend, or you need to change the default interval value, you can override the configuration options for one or more worker.

3.4.2.2. Worker Backoff

When there are no jobs left on a queue, each worker has a backoff strategy. This prevents workers from consuming unnecessary CPU cycles and unnecessary calls to the queue. When new jobs are put on the queue, the worker resets the interval when it next checks the queue.

You can override the behavior of each worker with the following configuration options:

- **`pendingWorkerBackoff`**
- **`ackWorkerBackoff`**
- **`syncWorkerBackoff`**

By default, all workers use an exponential strategy, with a max delay value. For example, if the min interval is set to 1ms, the worker waits 1ms after processing a job before taking another job off the queue. This pattern continues as long as there are items on the queue. If the queue empties, the interval increases exponentially (2ms, 4ms, 8ms, 16ms, … ~16s, ~32s) until it hits the max interval (for example, 60 seconds). The worker then only checks the queue every 60 seconds for a job. If it does find a job on the queue in the future, the worker returns to checking the queue every 1ms.

For more information, please refer to the Sync API Doc.

3.4.3. Managing Collisions

A collision occurs when a client attempts to send an update to a record, but the client's version of the record is out of date. Typcially, this happens when a client is off line and performs an update to a local version of a record.

Use the following handlers to deal with collisions:

- **`handleCollision()`** - Called by the Sync Framework when a collision occurs. The default implementation saves the data records to a collection named "<dataset_id>_collision".
- **`listCollision()`** - Returns a list of data collisions. The default implementation lists all the collision records from the collection name "<dataset_id>_collision".
- **`removeCollision()`** - Removes a collision record from the list of collisions. The default implementation deletes the collision records based on hash values from the collection named "<dataset_id>_collision".

You can provide the handler function overrides for dealing with data collisions. Options include:

- Store the collision record for manual resolution by a data administrator at a later date.

- Discard the update which caused the collision. To achieve this, the **`handleCollision()`** function would simply not do anything with the collision record passed to it.

WARNING

This may result in data loss as the update which caused the collision would be discarded by the Cloud App.

- Apply the update which caused the collision. To achieve this, the **`handleCollision()`** function would need to call the **`handleCreate()`** function defined for the dataset.

WARNING

This may result in data loss as the update which caused the collision would be based on a stale version of the data and so may cause some fields to revert to old values.

The native sync clients use similar interfaces. You can check the API and example codes in our iOS Github repo and Android Github repo.

3.5. SYNC SERVER UPGRADE NOTES

3.5.1. Overview

This section targets developers who:

- use Sync Server in their application
- are upgrading the version of **`fh-mbaas-api`** from **`<7.0.0`** to **`>=7.0.0`**

If you are already using **`fh-mbaas-api@>=7.0.0`**, do not follow any of the procedures in this section.

NOTE

There are no changes to the Sync Client in this upgrade.

3.5.2. Prerequisites

Prior to **`7.0.0`** the Sync Server used the **`fh.db`** API to store sync operational data in MongoDB. **`fh.db`** is a wrapper around MongoDB that may go through an intermediate http API (fh-ditch). This resulted in a resticted set of actions that could be performed on the sync operational data. It also limited the potential use of modules that connect directly to MongoDB. As of **`fh-mbaas-api@7.0.0`**, the Sync Server requires a direct connection to MongoDB.

This means:

- for a hosted MBaaS you must 'Upgrade' your App Database.
- for a self-managed MBaaS, no action is required as all Apps get their own Database in MongoDB by default

3.5.3. Data Handler Function Signature Changes

The method signature for sync data handlers are different for the new Sync Framework. If you implemented any data handler, you must change the parameter ordering. These changes are to conform to the parameter ordering convention in javascript, that is, a callback is the last parameter.

IMPORTANT Make sure that the callback function, passed to each handler as a parameter, runs for each call. This ensures that the worker can continue after the handler has completed.

The data handlers and their signature prior to and as of **`7.0.0`** are:

```
// <7.0.0
sync.handleList(dataset_id, function(dataset_id, params, callback,
meta_data) {});
sync.globalHandleList(function(dataset_id, params, callback, meta_data)
{});
// >=7.0.0
sync.handleList(dataset_id, function(dataset_id, params, meta_data,
callback) {});
sync.globalHandleList(function(dataset_id, params, meta_data, callback)
{});


// <7.0.0
sync.handleCreate(dataset_id, function(dataset_id, data, callback,
meta_data) {});
sync.globalHandleCreate(function(dataset_id, data, callback, meta_data)
{});
// >=7.0.0
sync.handleCreate(dataset_id, function(dataset_id, data, meta_data,
callback) {});
sync.globalHandleCreate(function(dataset_id, data, meta_data, callback)
{});


// <7.0.0
sync.handleRead(dataset_id, function(dataset_id, uid, callback, meta_data)
{});
sync.globalHandleRead(function(dataset_id, uid, callback, meta_data) {});
// >=7.0.0
sync.handleRead(dataset_id, function(dataset_id, uid, meta_data, callback)
{});
sync.globalHandleRead(function(dataset_id, uid, meta_data, callback) {});


// <7.0.0
sync.handleUpdate(dataset_id, function(dataset_id, uid, data, callback,
meta_data) {});
sync.globalHandleUpdate(function(dataset_id, uid, data, callback,
```

```
meta_data) {});
// >=7.0.0
sync.handleUpdate(dataset_id, function(dataset_id, uid, data, meta_data,
callback) {});
sync.globalHandleUpdate(function(dataset_id, uid, data, meta_data,
callback) {});


// <7.0.0
sync.handleDelete(dataset_id, function(dataset_id, uid, callback,
meta_data) {});
sync.globalHandleDelete(function(dataset_id, uid, callback, meta_data)
{});
// >=7.0.0
sync.handleDelete(dataset_id, function(dataset_id, uid, meta_data,
callback) {});
sync.globalHandleDelete(function(dataset_id, uid, meta_data, callback)
{});


// <7.0.0
sync.listCollisions(dataset_id, function(dataset_id, callback, meta_data)
{});
sync.globalListCollisions(function(dataset_id, callback, meta_data) {});
// >=7.0.0
sync.listCollisions(dataset_id, function(dataset_id, meta_data, callback)
{});
sync.globalListCollisions(function(dataset_id, meta_data, callback) {});


// <7.0.0
sync.removeCollision(dataset_id, function(dataset_id, collision_hash,
callback, meta_data) {});
sync.globalRemoveCollision(function(dataset_id, collision_hash, callback,
meta_data) {});
// >=7.0.0
sync.removeCollision(dataset_id, function(dataset_id, collision_hash,
meta_data, callback) {});
sync.globalRemoveCollision(function(dataset_id, collision_hash, meta_data,
callback) {});
```

3.5.4. Behavior Changes

As the sync server now connects directly to MongoDB, there is some setup time required on startup. If you currently use **`sync.init()`**, wrap these calls in a **`sync:ready`** event handler. For example, if you use the following code:

```
fh.sync.init('mydataset', options, callback);
```

Modify it to put it in an event handler.

```
fh.events.on('sync:ready', function syncReady() {
  sync.init('mydataset', options, callback);
});
```

Alternatively, you could use the event emitter from the sync API

```
fh.sync.getEventEmitter().on('sync:ready', function syncReady() {
  sync.init('mydataset', options, callback);
});
[source,json]
```

3.5.5. Logger Changes

The **`logLevel`** option passed into **`fh.sync.init()`** is no longer available. By default, the new Sync Server does not log anything. All logging uses the [debug](https://www.npmjs.com/package/debug) module. If you want log output from the Sync Server, you can set the **`DEBUG`** environment variable. For example:

```
DEBUG=fh-mbaas-api:sync
```

To see all logs from the entire SDK, you can use

```
DEBUG=fh-mbaas-api:*
```

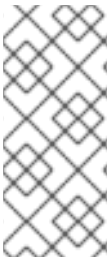
All other environment variables and behavior features of the **`debug`** module are available.

3.6. SYNC SERVER PERFORMANCE AND SCALING

3.6.1. Overview

The sync server is designed to be scalable.

This section helps you understand the performance of the sync server, and the options for scaling.



NOTE

If you are using a 1 CPU core with the sync server included with **`fh-mbaas-api`** version 7.0.0 or later, the performance could decrease compared to previous versions with default configurations. To improve performance of the sync server on a single core, consider adjusting the configuration as described in the Section 3.4, “Data Sync Configuration Guide”

3.6.2. Inspecting Performance

There are 2 options to inspect the performance of the sync server:

1. Query the **`/mbaas/sync/stats`** endpoint.
   By default, the Sync framework saves metrics data into Redis while it is running. You can then send a HTTP GET request to the **`/mbaas/sync/stats`** endpoint to view the summary of those metrics data.

   The following information is available from this endpoint:

   - CPU and Memory Usage of all the workers
   - The time taken to process various jobs

- The number of the remaining jobs in various job queues
- The time taken for various API calls
- The time taken for various MongoDB operations
  For each of those metrics, you are able to see the total number of samples, the current, maximum, minimum and average values.

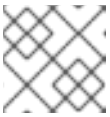
  By default, it collects the last 1000 samples for each metric, but you can control that using the **`statsRecordsToKeep`** configuration option.

  This endpoint is easy to use and provides enough information for you to understand the current performance of the sync server.

2. Visualize the metrics data with InfluxDB and Grafana
   If you want to visualize the current and historical metrics data, you can instruct the sync server to send the metrics data to InfluxDB, and view the graphs in Grafana.

   There are plenty of tutorials online to help setup InfluxDB and Grafana. For example:

   - How to setup InfluxDB and Grafana on OpenShift
     Once you have InfluxDB running, you just need to update the following configurations to instruct the sync server to send metrics data:
   - metricsInfluxdbHost
   - metricsInfluxdbPort

   NOTE

   Make sure **`metricsInfluxdbPort`** is a UDP port

   To see the metrics data graph in Grafana, you need to create a new dashboard with graphs. The quickest way is to import this Grafana databoard file. Once the app is running, you can view metrics data in the Grafana dashboard.

   For more details about how to configure the Grafana graphs, please refer to the Grafana Documentation.

3.6.3. Understanding Performance

To understand the performance of the sync server, here are some of the key metrics you need to look at:

3.6.3.1. CPU Usage

This is the most important metric. On one hand, if CPU is over loaded, then the sync server cannot respond to the client requests. On the other hand, we want to utilize the CPU usage as much as possible.

To balance that, establish a threshold to determine when to scale the sync server. The recommended value is 80%.

If CPU utilization is below that, it is not necessary to scale the sync server, and you can probably reduce a few worker interval configurations to increase the CPU usages. However, if CPU usage is above that threshold, consider the following adjustments to improve performance:

- Scaling the sync server
- Increase some of the worker intervals to reduce the load on CPU as described in Section 3.4.2, “Configuring the Workers”

3.6.3.2. Remaining Jobs in Queues

The sync server saves various jobs in queues to process them later.

If the number of jobs in queues keeps growing, and the CPU utilization is relatively low, reduce worker interval configurations to process the jobs quicker.

If the sync server is already under heavy load, consider scaling the sync server to allow new workers to be created to process the jobs.

3.6.3.3. API response time

If you observe increases in the response time for various sync APIs, and the CPU usage is going up, it means the sync server is under load, consider scaling the sync server.

However, if the CPU usage does not change much, that typically means something else is causing the slow down, and you need to investigate the problem.

3.6.3.4. MongoDB operation time

In a production environment, the time for various MongoDB operations should be relatively low and consistent. If you start observing increases of time for those operations, it could mean the sync server is generating too many operations on the MongoDB and starts to reach the limit of MongoDB.

In this case, scaling the sync server does not help, because the bottleneck is in MongoDB. There are a few options you can consider:

- Turn on caching by setting the **useCache** flag to true. This reduces the number of database requests to read dataset records.
- Increase the various worker intervals and sync frequencies.
- If possible, scale MongoDB.

3.6.4. Scaling the Sync Server

If you decide to scale the sync server, here are some of the options you can consider:

3.6.4.1. Scaling on an Hosted MBaaS

There are 2 options to scale the sync server on the RHAMP SaaS platform:

3.6.4.1.1. Use the Node.js Cluster Module

To scale inside a single app, you can use Nodejs Clustering to create more workers.

3.6.4.1.2. Deploy More Apps

Another option is to deploy more apps but point them to the same MongoDB as the existing app. This allows you to scale the sync server even further.

To deploy more apps:

- Deploy a few more apps with the same code as the existing app.
- Find out the MongoDB connection string of the existing app.
  It is listed on the **Environment Varaible** screen in the App Studio, look for a System Environment Variable called *FH_MONGODB_CONN_URL*
- Copy the value, and create a new environment variable called *SYNC_MONGODB_URL* in the newly created apps, and paste the MongoDB url as the value.
- Redeploy the apps.

With this approach, you can separate the HTTP request handling and sync data processing completely.

For example, if there are 2 apps setup like this, App 1 and App 2, and App 1 is the cloud app that accepts HTTP requests. You can then:

- Set the worker concurrencies to 0 to disable all the sync workers in App 1. It is then dedicated to handle HTTP requests.
- Increase the concurrencies of sync workers in App 2, and reduce the sync interval values.

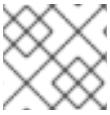
Please check $fh.sync.setConfig for more information about how to configure the worker concurrencies.

3.7. MONGODB COLLECTIONS CREATED BY THE SYNC SERVER

3.7.1. Overview

The sync server will maintain various collections in MongoDB while it's running.

This document will explain what collections the sync server will create and their purpose.



NOTE

You should not modify these collections, as it may cause data loss.

3.7.2. Sync Server Collections

All the collections created by the sync server will have the prefix **fhsync**.

3.7.2.1. fhsync_pending_queue

This collection is used to save the changes submitted from all the clients for all the Datasets.

Some of the useful fields for debugging are:

- **tries**: If the value is greater than 0, it means the change has been processed already by the sync server.
- **payload.hash**: The unique identifier of the pending change.
- **payload.cuid**: The unique id of the client.
- **payload.action**: The type of the change, like `create` or `update`.

- **payload.pre**: The data before the change was made.
- **payload.post**: The data after the change was made.
- **payload.timestamp**: When the change was made on the client.

3.7.2.2. fhsync_<datasetId>_updates

When a pending change from the **`fhsync_pending_queue`** collection is processed, the result is saved in this collection. The client will get the result when they next sync, any trigger any relevant client notifications.

Some of the useful fields for debugging are:

- **hash**: The unique identifier of the pending change from the above collection.
- **type**: If the change is applied successfully. Possible values are **`applied`**, **`failed`** or **`collision`**.

3.7.2.3. fhsync_ack_queue

After a client gets the results of its submitted changes (as saved in the **`fhsync_<datasetId>_updates`** collection), it will confirm the acknowledgements with the server so that server can remove them. This collection is used to save the acknowledgements submitted by the clients.

Some of the useful fields for debugging are:

- **payload.hash**: The unique identifier of a pending change from the **fhsync_pending_queue** collection.

3.7.2.4. fhsync_datasetClients

This collection is used to persist all the Dataset clients that are managed by the sync server.

Some of the useful fields for debugging are:

- **globalHash**: The current hash value of the Dataset Client.
- **queryParam**: The query parameters associated with the Dataset Client.
- **metaData**: The meta data associated with the Dataset Client.
- **recordUids**: The unique ids of all the records that belong to the Dataset Client.
- **syncLoopEnd**: When the last sync loop finished for the Dataset Client.

3.7.2.5. fhsync_<datasetId>_records

This data in this collection is a local copy of the data from the Dataset Backend. It will help to speed up the sync requests from the clients, and also reduce the number of requests to the Dataset Backend.

Some of the useful fields for debugging are:

- **data**: The actual data of the record returned from the Dataset Backend.

- **uid**: The unique id of the record.
- **refs**: The ids of all the Dataset Clients that contain this record.

3.7.2.6. fhsync_queue

This collection is used to save the requests to sync **fhsync_<datasetId>_records** with the Dataset Backend.

Some of the useful fields for debugging are:

- **tries**: If it's greater than 0, it means the request is already processed by the sync server.

3.7.2.7. fhsync_locks

Only 1 worker is allowed to sync with the Dataset Backend at any given time. To ensure that, a lock is used. This collection is used to persist the lock. It is unlikely you'll ever need to look at this collection, unless debugging an issue with the locking mechanism.

3.7.3. Pruning the Queue Collections

For each of the queue collections, a document is not removed immediately after being processed. Instead, it is marked as **`deleted`**. This will allow developers to use them as an audit log, and also help with debugging.

To prevent these queues from using too much space, you can set a TTL (time to live) value for those messages. Once the TTL value is reached, these messages will be deleted from the database.

For more information, see the "queueMessagesTTL" option in $fh.sync.setConfig.

3.8. SYNC SERVER DEBUGGING GUIDE

3.8.1. Client Changes Are Not Applied

To help debug this issue, answer the following questions:

Has the client sent the change to the Server?

Determine whether the change is sent by looking at the request body in sync requests after the client made the change. The change should be in the **`pending`** array. For example:

```
{
  "fn":"sync",
  "dataset_id":"myShoppingList",
  /*...*/
  "pending":[{
    "inFlight":true,
    "action":"update",
    "post":{
      "name":"Modified Name",
      "created":1495123790928
    },
    "postHash":"2e90b858164184b9ff31e0937cef8ddf4a959ac5",
    "timestamp":1495799747404,
```

```
    "uid":"591dc768a95300322eee1d1f",
    "pre":{
      "name":"Original Name",
      "created":1495123790928
    },
    "preHash":"421932b23f05f8aef528d73fff3cbf5aa00786a4",
    "hash":"f98f595974f7e7e1f07aed6220fab04446f459c9",
    "inFlightDate":1495799747850
  }]
}
```

If the change is not in the pending array, verify your device is online. Check for any errors in the Client App and verify you are calling the relevant sync action, for example, **`sync.doUpdate()`** for an update. It may help to debug or add logging in the Client App around the code where you make the change.

Is there a record for this change in the `fhsync_pending_queue` collection?

If 'No', the change was not received by the server, or there was an error receiving it.

- Verify the App successfully sent the change. If it did not, debug the App to understand the issue. There may be an error in the App, or an error in the response from the Server.
- Check the server logs for errors when receiving the change from the App. If there are no errors, see Enabling Debug Logs.

It is possible the record existed in the **`fhsync_pending_queue`** collection, but the Time To Live (TTL) period for queues has passed for the record, and it was removed. If this is the case, increasing the TTL enables debugging.

Does the record have a timestamp for the `deleted` field?

If 'No', the item has not been processed yet

- Typically, the pending worker is busy processing items ahead of it in the queue. Wait until the item gets to the top of the queue for processing.
- If an item is not processed after a significant time, or the queue is empty except for this item, check the server logs for any errors. If there are no errors, see Enabling Debug Logs.

Is there a record for the update in the `fhsync_<datasetid>_updates` collection?

If 'No', the update may have encountered an error while processing.

- Check the server logs for any errors. If there are no errors, see Enabling Debug Logs.

Is the `type` field in the record set to `failed` or `collision`?

If 'Yes', the update could not be applied to the Dataset back end.

- A 'collision' should have resulted in the collision handler being called. The collision needs resolution.
- A 'failed' update should have resulted in a notification on the client, with a reason for the failure. The reason is documented in the **`msg`** field of the record.

If 'No', and the type is **`applied`**, you need to debug the create or update handler to see why the handler assumed the change was applied to the Dataset back end.

The **type** field should never be anything other than **collision**, **failed** or **applied**.

3.8.2. Changes applied to the Dataset back end do not propagate to other Clients

Has sufficient time passed for the change to sync from the Dataset back end to clients?

After a change has been applied to the Dataset back end, there are 2 things that need to happen before other clients get that change.

- a sync loop on the server must complete to update the records cache, that is, the list handler is called for that dataset
- a sync loop on the client(s) must complete for the client(s) local record cache to be updated from the servers record cache

If sufficient time has passed, check the server logs for any errors during the sync with the Dataset back end.

Is there a recent record in the `fhsync_queue` collection for the Dataset?

If 'No', it is possible the TTL value for the record has passed and it was deleted. In this case, the TTL value can be increased to enable further debugging.

Another possibility is the sync scheduler has not scheduled a sync for that Dataset. The most likely reason is a combination of no currently active clients for that Dataset and the **`clientSyncTimeout`** has elapsed since the a client was last active for that Dataset.

Does the record have a timestamp for the `deleted` field?

If 'No', this means the sync is not processed yet.

- Typically, the sync worker is busy processing items ahead of it in the queue. Wait until the item gets to the top of the queue.
- If an item is not processed after a significant time, or the queue is empty except for this item, check the server logs for any errors. If there are no errors, see Enabling Debug Logs.

Is the record in the `fhsync_<datasetid>_records` cache up to date?

The list handler should have been called, and the result added to the records cache. To verify the records cache is updated, check the **`fhsync_<datasetid>_records`** collection for the record that was updated. The data in this record should match the data in the Dataset back end. If it does not, check the server logs for errors and the behavior of the list handler. It may help to add logging in your list handler.

Is the client sync call successful?

Check that there is a valid response from the server when the client makes its sync call. If the call is successful, verify the client is getting the updated record(s). If the updated records are not received by the client, even though the server cache has them, verify the query parameters and any meta data sent to the server are correct. Enabling the Debug logs may help determine how the incorrect data is sent back to the client.

3.8.3. Enabling Debug Logs

To enable sync logs for debugging purposes, set the following environment variable in your server.

```
DEBUG=fh-mbaas-api:sync
```

This process generates a lot of logs. Each log entry is tagged with a specific Dataset ID that is being actioned in the context of that log message, if possible. These logs can be difficult to interpret, but allow you track updates from clients and the various stages for those updates. It may help to compare logs for a successful scenario with an unsuccessful scenario, and identify which stage a failure occurs.

The most likely causes of issues are in custom handler implementations, particularly related to edge cases. It can be useful to add additional logs in your custom handlers.

Dataset back end connectivity issues, particularly intermittent issues, can be difficult to debug and identify. It can help to have external monitoring or checks on the Dataset back end.

CHAPTER 4. INTEGRATING RHMAP WITH OTHER SERVICES

4.1. INTRODUCTION TO MBAAS SERVICES

MBaaS Services are cloud/web applications that provide shared functionalities that can be used by multiple cloud applications within multiple projects. Typical use cases include:

- providing APIs to access third-party services. For example, multiple projects have a requirement to send SMS messages. You can create a service to provide the APIs to send SMS messages and that service can be accessed by multiple projects.
- providing APIs to legacy customer backend systems For example, you may have legacy systems that perform user authentication. You can create a service that provides a consistent, modern set of APIs for projects to consume, hiding all the details about how to access the legacy systems.

NOTE

MBaaS Services are not designed to be used by Client Apps directly: For example, an iOS app does not call an MBaaS service directly. A Client App calls the associated Cloud App, then the Cloud App sends requests to one or more MBaaS services.

Benefits of using MBaaS services include:

- improving code reusability. This is especially important if it is complicated to access the external services.
- protecting sensitive information. Sometimes user credentials are required to access the external services. Using services eliminates the requirement to share the credentials with multiple project developers.

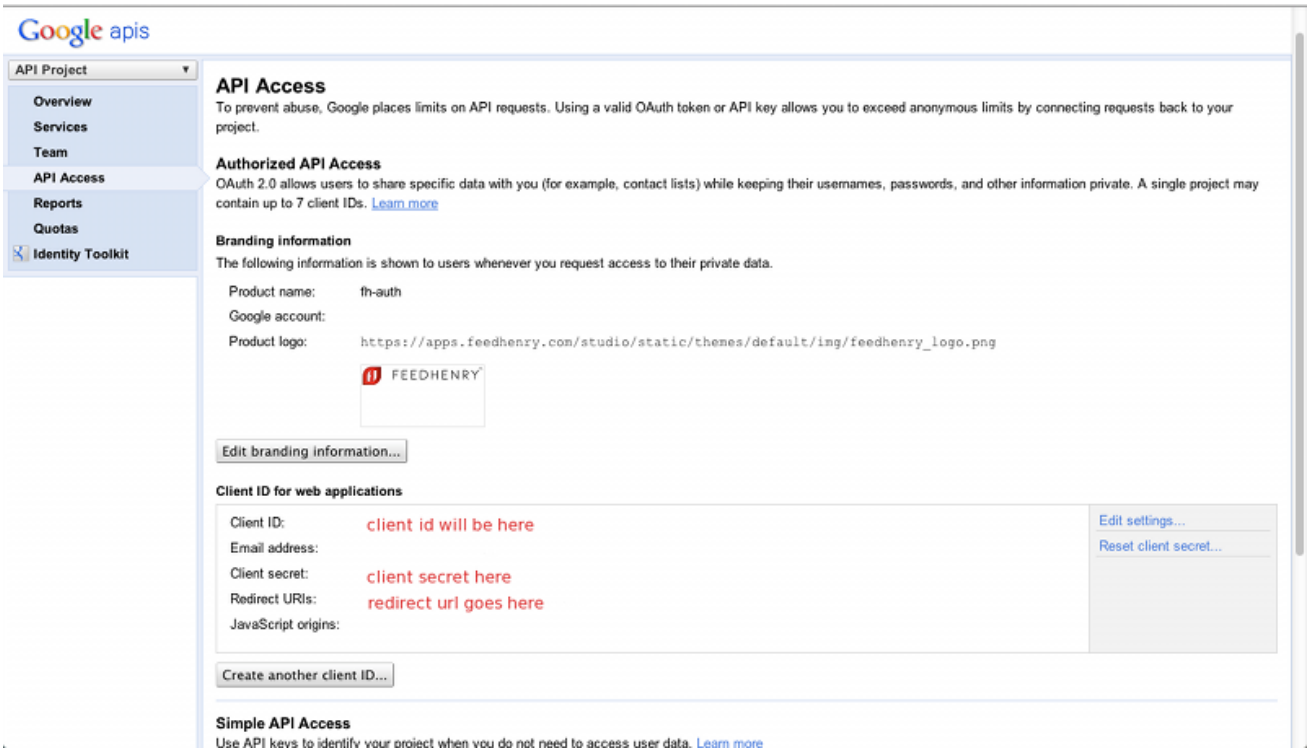
4.2. SETTING UP AN AUTH POLICY WITH GOOGLE OAUTH

4.2.1. Google OAuth Apps

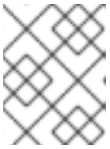
In order to perform authentication for your app's users against Google's OAuth service, there are a couple of steps you will need to go through first.

- First you will first need to set up an app with Google using the Google API console. When creating an app choose web application. Don't worry too much about the details as they can be updated afterwards. Once your app has been created take note of your client id and client secret provided by Google.
- The client id will look something like : 00000000000.apps.googleusercontent.com
- The secret will be a hash of numbers and digits.

If you look at the image below you will see in red where these items will appear in Google's console.



- Next, log in to the Studio as a user in a team with Domain Administration rights and click on the Auth Policies tab. Click the create button to make a new policy. Select the type as OAuth2 and fill in the details given to you by Google.



NOTE

In order to access the **Auth Policies** tab in the Studio, the User must be a member of a Team(s) with **View & Edit** Permissions set for **Authorization Policy**

- Add the callback url provided in the Auth Policy creation page to your Google app in the app console under Redirect URI.

4.2.2. Authorization

In the authorization panel you have two options.

- check user exists on platform.
- check if user approved for Auth.

If you do not tick any of these options, it is assumed that you wish to allow any user with an authentic Google account to have access to your app.

4.2.2.1. Check User Exists on Platform

This means that if a user has an authentic Google account and the user is registered with the Platform with the id returned by Google e.g "someuser@gmail.com" then they will be authorized to access your application.

4.2.2.2. Check if User Approved For Auth

This option means that if a user has an authentic Google account and the user id returned by Google is associated with this Auth Policy then the user will be authorized to use your applications associated with this Auth Policy.

4.2.3. Adding/Removing A User From An Auth Policy

To add an existing user to your Auth Policy click the check if user is approved for Auth option. A swap select will appear populated with the users available. To add one of these users to the policy, select the user and press the arrow pointing at the approved column. To Remove a user, select the user in the approved column and click the arrow pointing at the available column.

Once you are finished configuring your Auth Policy, click the "Update Policy" button.

CHAPTER 5. STORING DATA

5.1. ACCESSING THE DATABASE FROM CLOUD APPS

5.1.1. Overview

In an RHMAP 4.x MBaaS based on OpenShift 3, all components of the MBaaS run within a single OpenShift project, together with a shared MongoDB replica set. Depending on how the MBaaS was installed, the replica set runs either on a single node, or on multiple nodes, and may be backed by persistent storage. The recommended production-grade MongoDB setup for an MBaaS has 3 replicas, each backed by persistent storage.

Each Cloud App deployed to the MBaaS has its own OpenShift project. However, the database of a Cloud App is created in the shared MongoDB instance. Therefore, all management operations on the persistent data of Cloud Apps and the MBaaS, such as backup, or replication can be centralized. At the same time, the data of individual Cloud Apps is isolated in separate databases.

5.1.2. Accessing data in the MongoDB in the MBaaS

A simple way to store data is using the **`$fh.db`** API, which provides methods for create, read, update, delete, and list operations. See the $fh.db API documentation for more information.

If you need the full capability of a native MongoDB driver, or want to use another library to access the data, such as Mongoose, you can use the **`connectionString`** method of the **`$fh.db`** API to retrieve the connection string to the MongoDB instance:

```
$fh.db({
 "act" : "connectionString"
}, function(err, connectionString){
  console.log('connectionString=', connectionString);
});
```

NOTE

To avoid concurrency issues, we recommend using either the **`$fh.db`** API or a direct connection to the database, but not both at the same time.

CHAPTER 6. IMPORTING CLIENT APPS AND CLOUD APPS INTO PROJECTS

Overview

You can import existing Client Apps and Cloud Apps into projects. A git repository is created in RHMAP for each imported app.

The imported apps must fulfill the requirements outlined in this document:

- Client App Requirements
- Cloud App Requirements

Requirements

- Cloning, pushing, or pulling from Git repositories hosted in RHMAP requires an SSH key to be configured in the platform. If you have not already added an SSH Key, see SSH Key Setup.

6.1. CREATING AN EMPTY PROJECT

This guide assumes that you do not currently have a project in Studio. To create a bare project:

1. Log in to the Studio.
2. Navigate to **Projects**.
3. Click **+ New Project**.
4. Select the *Empty Project* template and give the new project a name.
5. Click **Create** button.
6. When the project has been created, click **Finish**.

6.2. IMPORTING A CLIENT APP

Before creating a Client App, make sure there is at least one Cloud App already created in the project. See Importing a Cloud App to import an existing Cloud App.

1. In your bare project, navigate to *Apps, Cloud Apps & Services*.
2. Click the **+** in the *Apps* box.
3. Click **Import Existing App**.
4. Select the *App Type* you wish to create. See Client App Requirements to make sure your app can be imported.
5. Click **Next** and give your imported app a name.
6. Choose how to import the app.
    - **Public Git Repository**

Enter the URL of a public Git repository containing the Client App, and a branch name. The default is `master`.

- **Zip File**
  On your computer, create a single ZIP archive of the contents of your Client App.

  ```
  zip -r app.zip ./ -x *.git*
  ```

  In Studio, choose the ZIP file to upload.

- **Bare Repo**
  This creates an empty repository where you can push your app code.

7. Click **Import & move on to Integration**. Your Client App is imported into the project.
   If you imported using the *Bare Repo* option, click *Already have a git repo* and follow the presented steps.

   If you imported using the *Zip File* or *Bare Repo* options, you will be presented with steps for integration of the RHMAP SDK.

8. Click **Finished - Take me to My App!**.

6.3. IMPORTING A CLOUD APP

1. In your bare project, navigate to *Apps, Cloud Apps & Services*.
2. Click the **+** in the *Cloud Code Apps* box.
3. Click **Import Existing App**.
4. Select the *App Type* you wish to create. See Cloud App Requirements to make sure your app can be imported.
5. Click **Next** and give your imported app a name.
6. Choose an initial deployment environment.
   Your Cloud App will be deployed to the chosen environment immediately after importing. Choose *None* to skip the initial deployment.
7. Choose how to import the app.

   - **Public Git Repository**
     Enter the URL of a public Git repository containing the Cloud App, and a branch name. The default is `master`.

   - **Zip File**
     On your computer, create a single ZIP archive of the contents of your Cloud App.

     ```
     zip -r app.zip ./ -x *.git*
     ```

     In Studio, choose the ZIP file to upload.

   - **Bare Repo**
     This creates an empty repository where you can push your app code.

8. Click **Import & move on to Integration**. Your Cloud App is imported into the project.
   If you imported using the *Bare Repo* option, click *Already have a git repo* and follow the presented steps.

9. Click **Finished - Take me to My App!**.

6.4. CLIENT APP REQUIREMENTS

Prerequisites for importing a Client App into your project depend on the type of Client App you are creating.

6.4.1. Cordova App

A standard Cordova 3.x mobile application. These apps consist of a combination of web technology and native code. The underlying native project and Cordova libraries are exposed to the developer allowing for full customization or the app, including Cordova plug-ins and third-party SDKs.

Typically, the amount of time spent writing or modifying native code for these apps is relatively small and requires only a small development team to have experience with native code. They should be able to develop, configure, and manage the native code and expose any additional plug-ins or SDKs to the web layer using the standard Cordova plug-in approach.

The majority of the development team do not need to concern themselves with the native code. They can continue to develop using pure web tech, but still be in a position to take advantage of any additional functionality that has been exposed from the native layer.

Requirements

- Minimum supported PhoneGap version: 3.0
- Must conform to PhoneGap 3.x standard structure and contain the following files/folders:
  - **`www/index.html`**
  - **`www/config.xml`**
  - **`platforms/`**
  - **`plugins/`**
  - **`merges/`**
  - **`.cordova/`**

6.4.2. Web App

A Node.js + Express web application. These apps provide advanced mobile websites and desktop browser web portals. They expose the full power of Node.js for web app development, including functionality such as server side templating (using template engines such as Jade). They also support static file serving for standard HTML5, CSS, and JavaScript.

Requirements

- Must be a Node.js app, with the following files located in `/`:

- **`package.json`**
- **`application.js`** - script that starts when the Node.js app is deployed

- Your app should listen on a port specified via the **`FH_PORT`** environment variable, for example,

```
app.listen(process.env.FH_PORT)
```

6.4.3. Native Android

A native Android app.

Requirements

- Must be a valid Android project based on Ant or Gradle (created in Android Studio based on Eclipse or IDEA). Ant-based projects must contain an **`AndroidManifest.xml`** file in `/` (project root).
- Minimum Android platform version: 2.3 (API Level 9).
- All dependencies must be available either in the Android SDK or in Maven repositories accessible from RHMAP.

6.4.4. Native iOS

A native iOS Application.

Requirements

- Must be a valid Xcode Project that builds and compiles correctly locally.
- Minimum iOS OS target version: 7.0.

6.5. CLOUD APP REQUIREMENTS

- Must be a Node.js application, with the following files are located in `/`
  - **`package.json`**
  - **`application.js`** - script that starts when the Node.js app is deployed
- Your app should listen on a port specified via the **`FH_PORT`** environment variable, for example,

```
app.listen(process.env.FH_PORT)
```