Oracle® Cloud

Using Oracle Mobile Cloud Service

Release 18.4.3
E93987-04
September 2019

Oracle Cloud Using Oracle Mobile Cloud Service, Release 18.4.3

E93987-04

Primary Authors: Patrick Keegan, John Bassett, Chris Kutler, Jennifer Shipman, Susan Post

Contents

2 Set Up the Service

Part II Setting up Mobile Apps

3 Mobile Backends

6 Android Applications

7 iOS Applications

8 Cordova Applications

14 Storage

15 Data Offline and Sync

16 Notifications

17 Analytics

Part IV Custom APIs

20 Creating APIs Fast with the Express API Designer

21 Custom API Design

22 Implementing Custom APIs

23 Calling APIs from Custom Code

25 SOAP Connector APIs

26 ICS Connector APIs

27 Fusion Applications Connector APIs

Part VI Deployment and Lifecycle

28 MCS Environments

29 Diagnostics

32 Managing an Artifact's Lifecycle

33 Testing APIs and Mobile Backends

34 Packages

Part VII Reference

A HTTP Headers

Preface

Welcome to *Using Oracle Mobile Cloud Service*.

This guide is intended for all users of Oracle Mobile Cloud Service, whether you are a mobile app developer, service developer, enterprise architect, mobile cloud administrator, or mobile program manager.

Audience

Using Oracle Mobile Cloud Service is intended for those people who are implementing their company's mobile application strategy, including mobile application developers, API developers, system administrators, and business analysts.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Related Resources

- *What's New in Oracle Mobile Cloud Service*
- Oracle Mobile Cloud Service Getting Started Tutorials
- Oracle Mobile Cloud Service Videos
- REST API for Oracle Mobile Cloud Service
- Oracle Mobile Cloud Service Android SDK Reference
- Oracle Mobile Cloud Service iOS SDK Reference
- Oracle Mobile Cloud Service Windows SDK Reference
- Oracle Mobile Cloud Service Cordova SDK Reference
- Oracle Mobile Cloud Service JavaScript SDK Reference
- *Oracle Mobile Cloud Service Known Issues*
- *Using Mobile Application Accelerator*

Conventions

The following text conventions are used in this guide:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

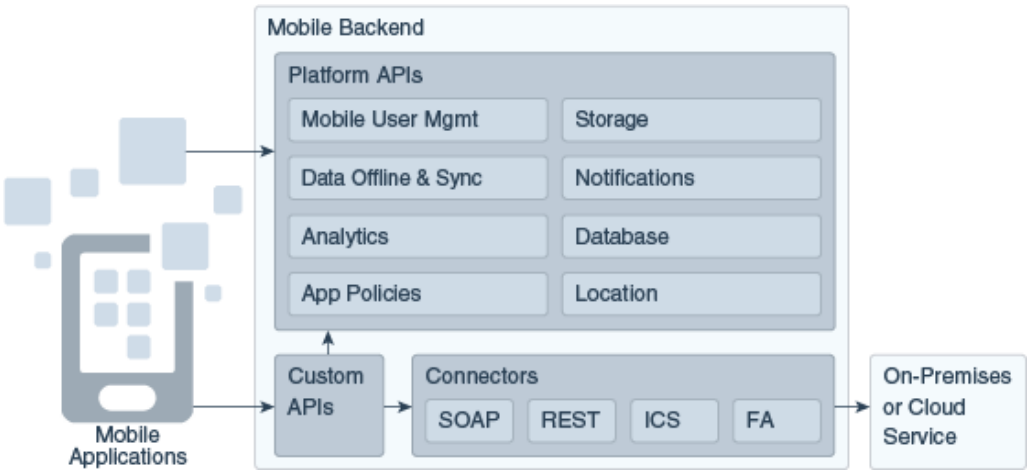
Part I
The Basics

This part contains the following chapters:

1

Get to Know Oracle Mobile Cloud Service

Welcome to Oracle Mobile Cloud Service (MCS)! MCS is a cloud-based service that provides a unified hub for developing, deploying, maintaining, monitoring, and analyzing your mobile apps and the resources that they rely on.



Your entry point into MCS depends on your role in your team's mobile project.

If you are a **mobile app developer**, you use MCS to line up and test the resources you need for your apps to work. This includes selecting from MCS platform APIs and custom APIs and collaborating with other team members to create new custom APIs.

If you are a **service developer**, you write Node.js-based JavaScript code to implement the custom APIs required by the mobile app developers on your team. You might also find yourself collaborating with mobile app developers to fine-tune API designs and creating connector APIs to connect to enterprise systems.

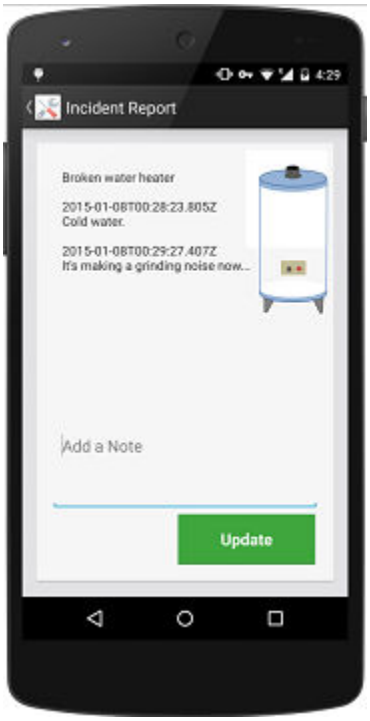
If you are the team's **enterprise architect**, you establish where desired data and functionality will come from, security and environment policies, and the roles and permissions of team members.

If you are the team's **mobile program manager**, you use the Analytics features to track usage patterns.

If you are a **mobile cloud administrator**, you work within the Administration tab to monitor the services in production, use the Diagnostics features to drill down and pinpoint problems, and handle other admin tasks such as the adding and removing of users.

To get a more concrete idea of how all of this works, let's imagine you work for a company called FixItFast (FiF) that supplies maintenance services for large in-house appliances. To help facilitate speed and quality of service, FiF management wants to roll out a mobile app which customers can use to quickly initiate a service request and

provide key information, such as by scanning the bar code of the defective appliance. They'll call this app FiF_Customer.



Here's how your team would use Oracle Mobile Cloud Service to develop that app and get the most out of it.

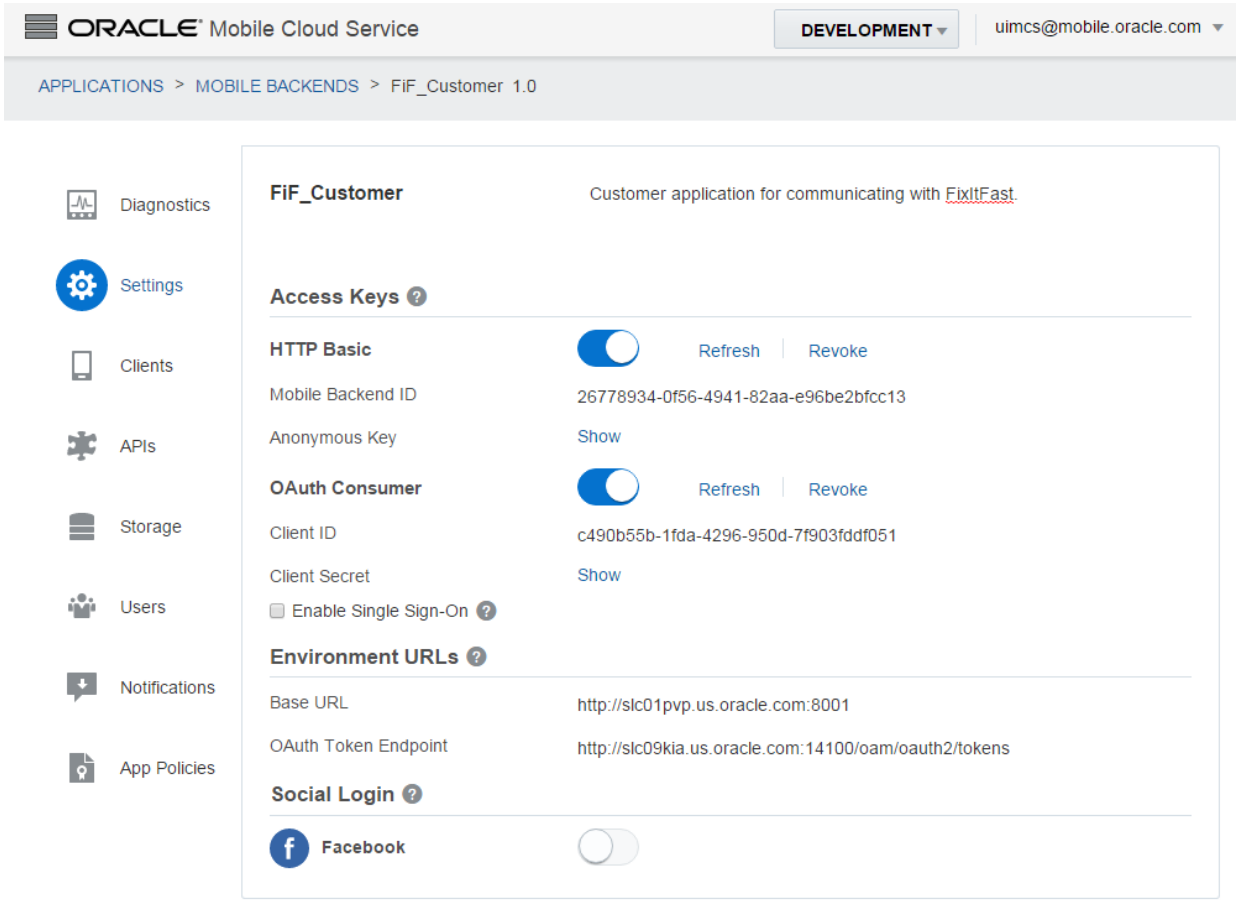
Jump in with Mobile Backends

As a mobile app developer, your first task is to set up a *mobile backend* for the app.

A mobile backend is a logical grouping of custom APIs, users, storage collections, and other resources that serves as a cloud-based companion to one or more related mobile apps. Within a mobile backend, you organize and develop resources that will be used by your apps (which they access as REST web services). The mobile backend also provides the security context (backed by an OAuth client ID/Client secret pair or by HTTP Basic authentication credentials) for accessing those services from the mobile app.

When you need apps for the same purpose on multiple platforms, all of those apps can use the same mobile backend. Likewise, completely different apps that rely on the same resources can share a mobile backend.

The screenshot below shows the Settings page of a mobile backend.



At development time, here are some of the things you do with a mobile backend:

- Browse and select the APIs to be available for the apps and test their endpoints with mock data.
- Create object storage collections and enable offline data caching.
- Specify a user realm within which you manage the mobile app users who are allowed to access the applications associated with the mobile backend.
- Set up notifications for your apps using the services provided by the platform vendors (such as Apple Push Notifications Service (APNS) for iOS, Google Cloud Messaging (GCM) for Android, and Windows Push Notification Services (WNS)). If you set up notifications for multiple platforms, you can initiate a single notification and have it delivered to apps on multiple platforms.

Later, at deployment time, the mobile backend serves as a deployment unit with dependency management for all of the artifacts you need to support the set of mobile apps.

However, before you do any of this for the FiF_Customer app that we introduced earlier, assume you're going to create a custom API to handle much of the app's heavy lifting.

Design Custom APIs

Especially if your company is new to MCS, one of the first things you'll need to do is start creating your own set of REST APIs to provide building blocks for your apps.

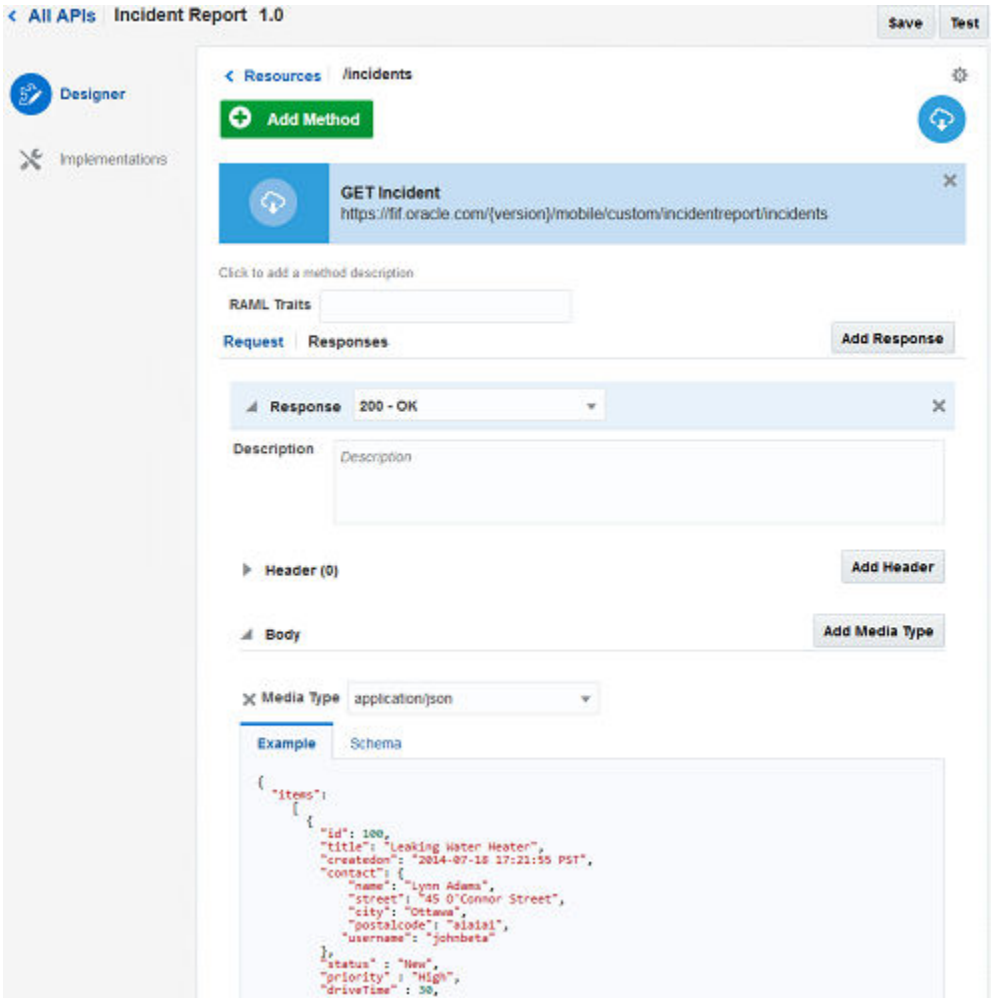
API creation is divided into two parts: designing and implementing. Let's talk about designing first.

When you design a REST API, you express the functionality that you expect in terms of resources, along with the HTTP methods they accept, and media types for the request and response bodies. In other words, you essentially define the formats for making a request on the API and for what kind of data is returned in the response. This definition is stored in a RAML (RESTful API Modeling Language) document. You *don't* actually fill in the details of how the data is produced and where it comes from right away. Those details are worked out later in the implementation.

For the previously-mentioned FiF_Customer app, you'll need an API for generating and logging incident reports. Let's call the API `incidentreports`. This report will consist of data entered by the customer, including a photo of the appliance's bar code and a description of the problem.

As the centerpiece to this API, you could define a resource called `incidents` to represent all incident reports. For that resource, you would have a GET method to retrieve all incidents and a POST to create a new incident. Further, you could define parameters for querying based on given criteria, such as the incident ID.

For the bodies of the requests sent to the `incidents` endpoint and responses returned from it, you'll define the media types (such as `application/json`) that they accept and then provide examples for those bodies. Those examples serve as mock data that is used when you (and eventually the users of your API) test the way the API works in a mobile backend.



Once you are happy with the structure of the API, a service developer can get to work on coding the implementation.

Implement APIs

As a service developer, you will work on APIs that have been sketched out for you by app developers (or perhaps on APIs that you have designed yourself). Once you have a set of endpoints to work with (like `/mobile/custom/incidentreport/incidents`, as outlined above), you can start implementing them with custom code.

This custom code takes the form of Node.js-based JavaScript. For each API implementation, you create a Node.js module. Within each module, you write a route definition for each endpoint that specifies how to respond to a client request to that endpoint. These route definitions are based on conventions promoted by the Express.js web framework for Node.js. You can also include other Node.js libraries in the module to support your custom code.

Let's go back to the `GET` method on the incidents resource that we were just talking about. Imagine that you have created a route definition for it that retrieves the incidents

via the Database Access API. The custom code implementing the endpoint might look something like this:

```
/**
 * GET ALL INCIDENTS
 */
service.get('/mobile/custom/incidentreport/incidents', function (req, res)
{
  //call to custom code SDK, which handles the interaction with the
Database Access API
  req.oracleMobile.database.getAll(
    'FIF_Incidents').then(
    function (result) {
      res.send(result.statusCode, result.result);
    },
    function (error) {
      res.send(500, error.error);
    }
  );
});
```

In the real world, your implementations will probably need additional logic and perhaps need to aggregate data with multiple API calls, but this sample should give you an idea of the basic mechanisms involved.

In much of your custom code, you'll probably also need to access various enterprise resources that reside outside of MCS, such as databases, CRM software, and other cloud services and legacy systems. Read on to learn more about accessing those resources and shaping them for use in your mobile apps.

Get the Data

Chances are that the main purpose of many of your custom APIs is to pull data into your app from various business applications and other systems maintained by your company, whether cloud or on-premises. As a service developer, your challenge is to do so in a way that's manageable, especially if you don't have detailed knowledge of the systems or the interfaces needed to access them. MCS answers this problem with connector APIs.

Connector APIs provide a bridge between your custom APIs and the enterprise services you want to process with those APIs. Using the REST, SOAP, and ICS (Integration Cloud Service) connector types, you create connector APIs for each data source that you want to access. You define a connector API by filling in info on the target resource, creating rules for the call parameters to "shape" the returned data so that it works well in a mobile context, and specifying security policies. The result is a reusable service that's exposed as a straightforward REST API that you can view in the Custom Code API Catalog. Service developers can call this connector from their custom code just like they would any other API and do not have to worry about tricky specifics like security policies and identity propagation.

For the Incident Report API, there are a number of resources that you'll want to interact with, such as an API for geolocation and customer data through your company's CRM software. In this example, the custom code calls a connector called `RightNow` to add an incident report to an Oracle Service Cloud instance that is used to manage customer service interactions.

```
/**
 *  The following example calls the 'CreateIncident' resource
 *  on a SOAP connector named '/mobile/connector/RightNow'.
 *   */
req.oracleMobile.connectors.RightNow.post('CreateIncident',
{Body: {CreateIncident: req.body}}).then(
  function(result){
    res.send(result.statusCode, result.result);
  },
```

```
  function(error){
    res.send(500, error.error);
  }
);
```

> **Note:**
>
> When you use connector APIs in your apps, you get other MCS advantages when your apps call the API, including diagnostics to measure API performance and API call analytics to evaluate how mobile apps are used.

Use Platform APIs

In addition to custom APIs, you can use MCS platform REST APIs in your apps. You can call these APIs directly from your apps and/or via the implementation code of custom APIs. You can also access many of them through MCS's **SDKs** for the iOS, Android, Windows, Cordova, and JavaScript platforms.

The available platform APIs include the following:

- **Storage** to work with collections and objects (such as images and documents) that you associate with your mobile backend.

  You set up collections in the web interface (and optionally populate them). Then you can use API calls to add, modify, and delete objects in those collections.
- **Mobile User Management** to store and retrieve data related to mobile users.
- **Location** to define location devices and places and query for them from your mobile apps.
- **Notifications** for writing code to send notifications to your mobile apps.
- **Analytics Collector** to initiate logging of specified events in the running apps. These logged events are collected and can be viewed through the prism of various reports in the Analytics tab in the MCS user interface.
- **Database Access** to access an Oracle Cloud database with REST calls. For security reasons, you can access the Database Access operations only from custom API implementations by using the custom code SDK, as described in Accessing the Database Access API from Custom Code. You can't make direct requests from client applications.
- **Database Management** to add, view, replace, and drop tables that are created (and updated) automatically when you POST or PUT a JSON object using Database Access API.
- **App Policies** to retrieve application configuration properties that you have set in the mobile backend.

Call APIs from Your App Code

Once you have selected the custom APIs to use in your mobile backend, you can call their REST endpoints from your mobile app code. Platform APIs are automatically available for all mobile backends, but calling them works the same way as calling custom APIs.

Here is a call from some Android app code to use the `incidentreport` custom API to post an incident.

```
String url = "http://<MCS_SERVER>:<PORT>/mobile/custom/incidentreport/
incidents";

HttpClient httpClient = new DefaultHttpClient();
HttpPost post = new HttpPost(url);
post.addHeader("Content-Type", "application/json");
post.addHeader("Authorization", basic bWNzOldlbGNvbWUxKg==);
try {
    JSONObject newIncidentReport = new JSONObject();
    newIncidentReport.put("EmailAddress", email);
    newIncidentReport.put("ImageLink", imageLink);
    post.setEntity(new StringEntity(newIncidentReport.toString()));
    HttpResponse response = httpClient.execute(post);
    StatusLine statusLine = response.getStatusLine();
    if (statusLine.getStatusCode() == HttpStatus.SC_OK) {
        // Success
    }
}catch (Exception e) {
    ...
}
```

And here is an example of using the Storage API in an Android app to post to a collection called `FiF_Images` that has been associated with your mobile backend:

```
String url = "http://<MCS_SERVER>:<PORT/mobile/platform/storage/
collections/FIF_Images/objects";

HttpClient httpClient = new DefaultHttpClient();
HttpPost post = new HttpPost(url);
post.setEntity(new ByteArrayEntity(imageBytes));
post.addHeader("X-Backend-Token", "FixItFast_Customer/1.0");
post.addHeader("Content-Type", "image/jpeg");
post.addHeader("Authorization", "basic bWNzOldlbGNvbWUxKg==");
HttpResponse response = httpclient.execute(post);

StatusLine statusLine = response.getStatusLine();
if (statusLine.getStatusCode() == HttpStatus.SC_CREATED) {
     // Image uploaded successfully
}
```

Call Platform APIs with Mobile SDKs

In addition to being able to call MCS APIs with straight REST calls, MCS provides SDKs to simplify use of some of the platform APIs in native code.

Here's some code for an Android app that uses the SDK classes (`StorageCollection` and `StorageObject`) for object storage.

```
Storage storage =
```

```
MobileBackendManager.getManager().getDefaultMobileBackend(context).getServi
ceProxy(Storage.class);
    try {
        StorageCollection imagesCollection =
storage.getStorageCollection("FIF_Images");
        StorageObject image = imagesCollection.get("3x4mp1e-st0r4g3-0bj3ct-
k3y");
        byte[] imageBytes = image.getPayloadBytes();
    }catch (ServiceProxyException e) {
        int errorCode = e.getErrorCode();
        ...
    }
```

A similar call with the iOS SDK might look like:

```
AppDelegate* appDelegate = [[UIApplication sharedApplication] delegate];
OMCMobileBackend* mbe = [appDelegate myMobileBackend];
OMCStorage* storage = [mbe storage];

OMCStorageCollection* aCollection = [storage getCollection:FIF_Images];
OMCStorageObject* aObject = [aCollection get:3x4mp1e-st0r4g3-0bj3ct-k3y];
NSData* data = [aObject getPayloadData];
...
```

Set Up and Manage Your Mobile App Users

With the app just about ready to go, it's now time the person on your team who has the Oracle Cloud identity domain administrator role to set up the users of the app. To manage the users of your mobile apps, you set up user *realms*. A realm is a collection of mobile app users with similar properties. Each mobile backend in an environment is associated with one realm. However, a realm can be used by multiple environments and multiple mobile backends. (Keep reading for more information on environments.)

You can also set up *roles*, which are sets of permissions that you can assign to users to control which users have permissions to what APIs and other resources. For example, you could have a role for customer service reps that provides them permissions to access the APIs that are needed for their job, such as for assigning cases. Similarly, you could have a role for technicians that allows them to access APIs relevant to their job, such as getting notifications for open cases and marking a case as resolved.

If you already have an identity provider for the future users of your apps, you can use MCS's Enterprise Single Sign-On support to enable those users to log in to apps that use MCS mobile backends. Similarly, you can use MCS's support for Facebook login for consumer apps.

Deploy Code between MCS Environments

Once your code is developed and your mobile backend is configured, it's time to proceed with deployment.

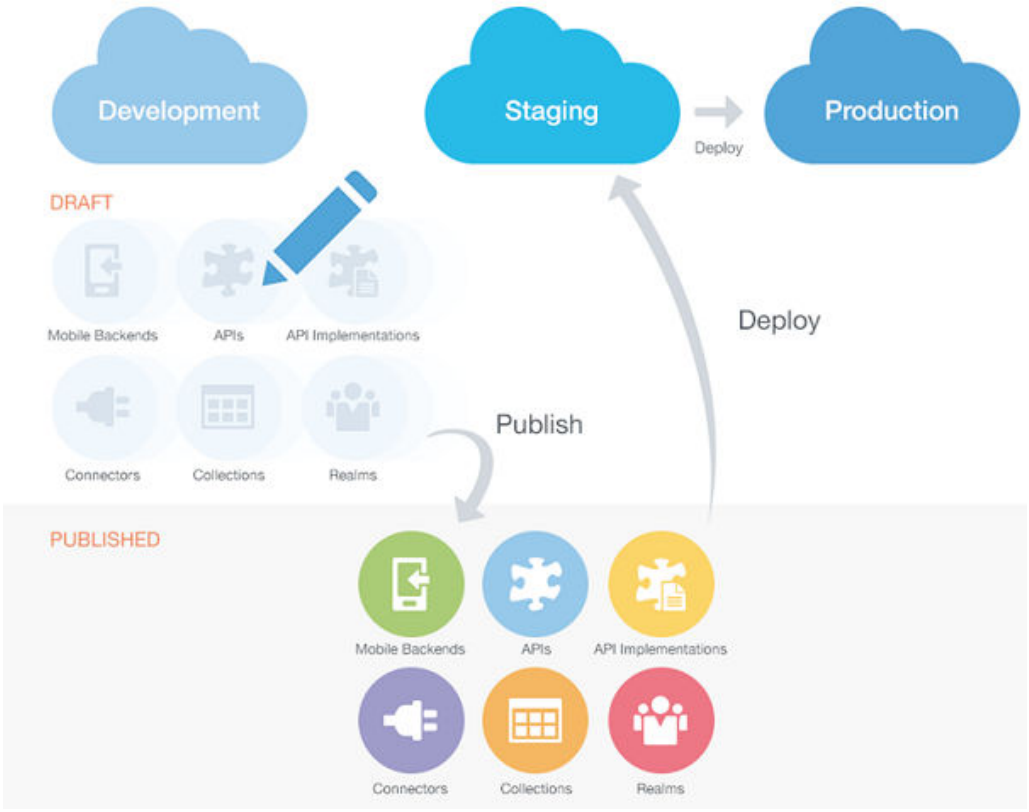
An *environment* is a predefined arena for working in MCS. You develop artifacts (mobile backends, APIs, and user realms) or custom code in a development

environment and deploy to a runtime environment for testing and distribution. You can work in one environment at a time.

For example, if you have a three-environment setup, you might use it this way:

- Designate one as a development environment to create your mobile backend, define custom APIs, create new services using custom code, set up storage for your collections, and so on. Typically, such an environment is where your team does most of its development work, and it isn't exposed to end users.
- Designate one as a staging environment where you can deploy completed project code for testing. Team members with broad permissions in the development environment might have no access to this staging environment if testing is handled by another team.
- Designate one as a production environment where you can promote fully tested code for real world distribution through an app store, such as the Apple App Store or Google Play Store. Not many team members need access to the published project code in this environment.

For more information about environments, see MCS Environments.



When you deploy, you follow this process:

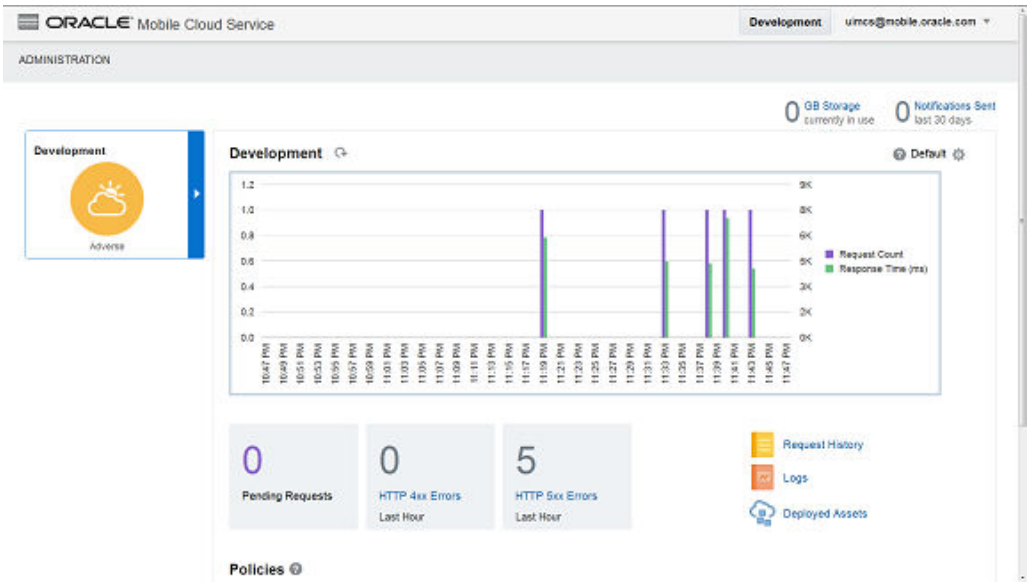
1. Publish numbered versions of your artifacts, essentially freezing them, so those versions can no longer be edited. To make a change to a published artifact, you need to create a new version, make necessary changes, and publish again.
2. Set (or verify) dependencies between relevant artifacts, such as between API versions and their implementation versions.

3. Set (or verify) environment policies to determine things such as what security credentials are associated with the environment, what versions of an app can access the mobile backend, timeouts, etc.
4. Push the artifacts to the target environment.

To release updates, you simply create a new version in your development environment and follow the deployment process again.

Monitor and Administer the Mobile Infrastructure

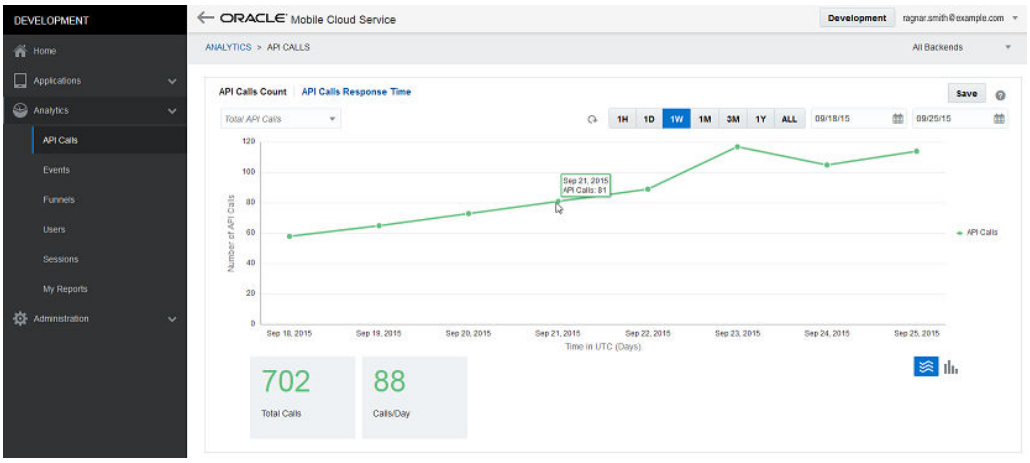
As the mobile cloud administrator, you use the Administration tab in the user interface to monitor the health and performance of your apps in all of your environments, particularly Production. The Administration tab provides graphical and tabular data on the server load and the request backlog. If any problems arise in production, you can view logs of server and app activity, filter them, and drill down to identify any trouble spots.



Analyze Your Mobile Projects

Once your apps are in production, the mobile program manager can step in to evaluate long-term usage and performance patterns in your mobile backends.

MCS comes with a host of built-in metrics such as API calls, API call response time, new users, active users, session count, and session duration.



You can also track any custom events that have been created in your apps using the Analytics API. For example, imagine your app uses the Analytics API to post an event each time a mobile app user creates a new incident report and capture properties such as appliance type, make, model, and model year. You could then generate graphs and tables based on those events and filter the data in any number of ways, such as how many incident reports were filed for water heaters every month for the last year.

What About Security?

Oracle Mobile Cloud Service is designed with enterprise-grade security baked in.

Security begins at the level of the mobile backend. For an app to access any resources through a mobile backend, its user first has to be authenticated with the mobile backend, whether it is using OAuth, enterprise single sign-on (SSO), Facebook login, or HTTP Basic authentication. See Authentication in MCS for the details.

Once a user is authenticated, access to APIs is controlled through MCS's mobile user management features. Realms allow mobile apps to use a shared set of users and data, and roles define permissions that control user access to APIs and resources from those mobile apps. For an introduction to users, roles and realms in MCS, see Set Up Mobile Users, Realms and Roles.

Security for custom APIs can be configured individually for each API. On the Security tab in the API Designer, you can decide whether or not an API can be accessed anonymously (without a user login). If you choose No, you can define the authorization policy by specifying which roles can access the API or specific endpoints. For details, see Security in Custom APIs.

MCS connector APIs also have access to security functionality, which is especially important if the connection involves transmitting proprietary or sensitive information. For details, see Security Policy Types for REST Connector APIs and Security Policy Types for SOAP Connector APIs.

Video: Security Overview

This video illustrates the key security aspects of MCS:

Video

Job Descriptions and Learning Paths

Let's take a few moments to talk about the various jobs (e.g. mobile developer, service developer, etc.) that we introduced at the beginning of the chapter. MCS is designed to meet the needs of widely disparate team roles, so the way you interact with MCS depends on your given responsibilities. To help you better understand how your responsibilities fit with MCS, here's more specific detail about what we mean by each job and links to the parts of the guide that are most relevant to those jobs.

Mobile App Developer

As a mobile app developer, it's your job to create new applications for the iOS and Android platforms. Often these apps incorporate existing enterprise functionality, which you'll need to optimize for phones and tablets.

To make things easier, you'll want to leverage existing APIs wherever possible. You can find our built-in APIs for common functions (like storage, mobile user management, notifications, and analytics) in Oracle Mobile Cloud Service's API Catalog, as well as APIs that other team members have created. When the API you need isn't available, use the API Designer to sketch out the API quickly and supply some mock data. Then you can go back to work on your app and let the service developer fill in the details (or do it yourself, if you prefer).

Here are the sections you'll be most interested in.

- For creating a mobile backend and setting up your apps to work with it:
  - Mobile Backends
  - Connecting Your App to a Mobile Backend
  - Android Applications
  - iOS Applications
  - Cordova Applications
  - JavaScript Applications
- For working with platform APIs:
  - Working with Mobile Users (for info on having test users created for you) in the Mobile User Management chapter
  - Location
  - Storage (for setting up object storage collections that your app can use)
  - Data Offline (for caching of data on your device)
  - Notifications (for setting up and sending push notifications for both iOS and Android apps)
  - Enabling Your Applications to Report Event Data in the Analytics chapter
  - App Policies (for referencing custom properties that you have defined in a mobile backend)
- For getting the ball rolling on designing APIs that you'll need in your apps:
  - Designing Custom APIs

- For info on the diagnostics features that may help you as you are testing your apps against your mobile backend:
  - Monitoring Performance and Troubleshooting
- For learning about the Oracle Mobile Application Accelerator to create mobile apps with visual tools:
  - Creating APIs Fast with the Express API Designer

If you haven't gone through them already, here are some tutorials to help you get started quickly:

- Mobile Backends (Access this tutorial by logging into MCS and clicking **Get Started** on the home page.)
- Custom APIs
- Storage
- Mobile User Management
- Notifications

Here are some other resources that you may want to look at:

- This video overview of the API designer shows you how to quickly sketch out an API design, which you can then pass to a service developer for implementation.
- The YouTube channel for Oracle Mobile Platform, which contains instructional videos covering a plethora of MCS topics, including designing and testing mobile backends, security, registering and configuring notifications, the storage API (including testing and examples), creating custom reports with the Analytics API, building connectors, and more.

Service Developer

As a service developer, your primary task in Oracle Mobile Cloud Service is to write the JavaScript code that implements the custom APIs that your organization's mobile apps rely on. These APIs might draw on existing enterprise services, platform APIs provided by Oracle Mobile Cloud Service, or other APIs your team has developed in Oracle Mobile Cloud Service.

In addition, you may be called upon to work with mobile developers to refine APIs they've already sketched out, and to create connector APIs that make it easier for your custom APIs to access enterprise resources.

Here are the chapters you'll be most interested in.

- For fine-tuning API designs and writing their implementation code:
  - Custom API Design
  - Implementing Custom APIs
  - Calling APIs from Custom Code
  - Database
- For creating connector APIs to access the enterprise system data:
  - REST Connector APIs
  - SOAP Connector APIs

  - ICS Connector APIs
- For info on testing your apps against your mobile backend:
  - Diagnostics
  - Testing APIs and Mobile Backends
- For creating packages that contain mobile backends, APIs, and other artifacts and then exporting and importing those packages:
  - Packages

If you haven't gone through them already, here are some tutorials to help you get started quickly:

- Mobile Backends (Access this tutorial by signing into MCS and clicking **Get Started** on the home page.)
- Custom APIs
- Connectors

You may also want to subscribe to The YouTube channel for Oracle Mobile Platform, which contains instructional videos covering all of the areas above.

Enterprise Architect

As the enterprise architect, you're concerned with designing a secure and scalable mobile solution for your business. It's your job to determine what can be built, where desired data and functionality will come from, and what security and environment policies need to be implemented. You're particularly attuned to establishing best practices, consistency, and reusability in your resources and repeatability in your processes.

In addition to establishing the mobile architecture, you also oversee how apps are deployed initially, updated, and patched.

You will be most interested in the following topics:

- Getting the Service Set Up
- Lifecycle
- What About Security?

You may also want to subscribe to the YouTube channel for Oracle Mobile Platform, which contains instructional videos covering all of the areas above.

Mobile Cloud Administrator

As the mobile cloud admin, you are responsible for setting up MCS for your team members and making sure that it keeps clicking both for the team members working with MCS and the end users of your apps. In your day-to-day work, you monitor the Administration tab to make sure that the service is running smoothly. When you detect issues or when problems are reported to you, the built-in diagnostics tools help you identify and fix the problems.

You will be most interested in the following chapters:

- Set Up the Service

- Lifecycle (for learning about the main principles of deployment and lifecycle of mobile backends, APIs, and other artifacts)
- Managing an Artifact's Lifecycle (for the steps on versioning, deploying, and patching)
- Diagnostics (for using diagnostics and logs)
- App Policies (for adjusting app policies for already-deployed apps)

If you haven't gone through it already, you may want to look at the Mobile User Management tutorial, which shows how to quickly set up an additional realm.

You might also want to look at this video on managing your mobile deployments as well as the YouTube channel for Oracle Mobile Platform, which contains instructional videos covering all of the areas above.

Mobile Program Manager

As the mobile program manager, you're responsible for the success of your mobile strategy. You want to know how many people are using your applications, and how they're using them. To achieve that, you will probably want to use Oracle Mobile Cloud Service's Analytics features to track standard metrics (such as registered and active users, number of transactions, etc.) and create your own events to track.

You will be most interested in the following chapter:

- Analytics

In addition, you may be interested in looking at this video on MCS's Analytics and the YouTube channel for Oracle Mobile Platform.

2

Set Up the Service

Here's what you need to know to get your team set up with Oracle Mobile Cloud Service (MCS), including activating the service, creating a service instance, and assigning team members. Be sure to go through this chapter carefully to make sure that you have fully configured the service for what your team needs.

Where Do I Sign Up?

If you haven't already purchased a subscription to Oracle Mobile Cloud Service (MCS) and would like to, you can do so in either of these ways:

- Visit https://shop.oracle.com and enter `Mobile Cloud Service` into the Search field to display the purchase options.
- Contact your sales representative. If you don't know who that is, go to the Oracle Contact List and click **Live Sales Chat**.

You can purchase a metered or non-metered subscription. For an overview, see Overview of Oracle Cloud Subscriptions in *Getting Started with Oracle Cloud*.

You can also sign up for a trial by following these steps:

1. Navigate to https://cloud.oracle.com/en_US/tryit and click **Get started for free**.
2. Click **Sign up**.
3. Fill out the online form to create an Oracle account.

   See Requesting a Free Oracle Cloud Promotion if you have any questions on how to fill out the form.

Once the request is approved, you will receive an email with details for logging in (and changing your password).

What Do I Need To Do?

MCS setup activities are divided between team members with the following administrative roles, assigned in Oracle Cloud.

Task	Who Does It?	How Do I Do It?
Activate the service and designate administrators	Your company's Oracle Cloud **account administrator**. This person is designated by your Oracle sales representative when you sign up with Oracle Cloud.	See Activate the Service

Task	Who Does It?	How Do I Do It?
Create one or more service instances (environments) and assign a service administrator	For non-metered service, it is the **account administrator** or **service administrator** designated by the account administrator. For metered service, it is the **service administrator**.	See Create Mobile Environment Service Instances
Assign MCS team member roles to define permissions	A **service administrator** for the MCS environment.	See Assign MCS Team Member Roles
Set up mobile users, realms and roles	A team member with the Oracle Cloud **identity domain administrator** role and the mobile user configuration (`MobileEnvironment_MobileUserConfig`) and mobile user management (`MobileEnvironment_MobileUserMgmt`) MCS team member roles in the MCS environment.	See Set Up Mobile Users, Realms and Roles
Set up MCS for MAX	Your company's MCS **service administrator**.	See Setting Up MAX Environments, Distinguishing Between MAX Team Member Roles for Business Users and for Mobile App Developers and Mobile Users for MAX
Log in to MCS	All MCS team members.	See Get on Board

Activate the Service

When your company submits an order for MCS, your sales representative designates an **account administrator**, who is the activator for the service. If you're that person, you'll receive an activation email to get started. If this is your first time logging in to Oracle Cloud, you'll be prompted to change your temporary password.

- Open the activation email and click **Cloud Account Services Setup**.

If you have a non-metered subscription, you've subscribed to an entitlement to create service instances of MCS (environments), so your first task is to create those environments based on your business needs, described next in Create Mobile Environment Service Instances

If you have a metered subscription, your first task is to assign MCS roles to your team, described in Assign MCS Team Member Roles.

Create Mobile Environment Service Instances

MCS uses **environments** to define the behavior of artifacts and control access to development and administrative features. As an account or service administrator, you define these environments, assign predefined MCS team member roles, and configure environment policies. For example, if you have more than one environment, you could designate one as a development environment and one as a production environment.

- **Development** could be an environment where you create your mobile backend, define your custom APIs, create new services using custom code, set up storage for your collections, and so on. It's the primary environment where you'll do most of your work.
- **Production** could be a completely separate environment, into which you can promote your completed project code for testing or public access. Developers and team members with broad permissions and easy access to features in the development environment might have little or no access to a production (or staging) environment where specific testing can be done by another team. You could also further separate the production environment to promote fully tested code for use by applications.

To create your mobile environment service instances:

1. Open the welcome email you received after being assigned as the service entitlement administrator and click **My Account**.

   You'll be prompted to change your temporary password.

2. In the Oracle Cloud Infrastructure Classic Console, click the **Create Instance** button next to Mobile Cloud Service in the list of services and complete the wizard that appears.

   Allow up to three hours for the instance to be created.

   Upon creation of your first environment, an MCS Portal instance is also created and your environment is associated with it.

3. For any additional environments you want to create, repeat step 2 of this procedure and associate them with the MCS Portal instance using the **Associations** dropdown in the wizard.

If you need more detailed information on the wizard, see Creating Service Instances in *Getting Started with Oracle Cloud*.

Setting Up MCS Environments

If you're assigned as **service administrator** for the mobile environment service instance, you're granted all MCS team member roles in the environment so you can start setting up the environment:

- To assign team member roles, open the welcome email you received when you were assigned as service administrator and follow the link to Oracle Cloud My Services. From the Oracle Cloud Infrastructure Classic Console, click the navigation menu  in the top left corner, and choose **Users**.
- From the **Users** page you can assign team member roles for the environment as described in Assign MCS Team Member Roles.
- To monitor activity, access administrative features and define environment policies, go to MCS, click  and open **Administration** from the side menu. For more information on using these features, see MCS Environments.

Setting Up MAX Environments

MAX (Mobile Application Accelerator) is a development tool that enables business users to create, test, and publish mobile apps without writing code. You can find out

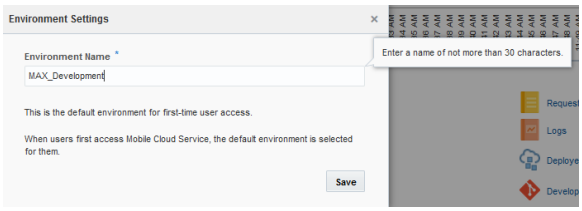
more about MAX and how it's used in Creating APIs Fast with the Express API Designer.

MCS doesn't support multiple development and production environments for MAX. You can only assign one MAX development environment and one production environment.

- A business user builds and tests apps in the MAX development environment. **The `MobileEnvironment_BusinessUser` role in the development environment limits business users to the MAX UI only.** MCS team members with the `MobileEnvironment_Develop` role in the environment also have access to MAX features.
- A business user or MCS team member can use MAX to publish apps by promoting them to the MAX production environment, making them available to other people in the organization. This requires the `MobileEnvironment_MAXApplicationDeploy` role in the environment.

For more information on MAX roles, see Distinguishing Between MAX Team Member Roles for Business Users and for Mobile App Developers.

> **Tip:**
>
> Instead of accepting the default names for the MAX development and production environments, choose names that make them easy to identify. You might consider a MAX-themed naming convention and choose simple names to help you associate MAX roles with the correct environment service instances.



Assign MCS Team Member Roles

As a **service administrator**, you use the predefined MCS team member roles to grant permissions and capabilities to your team members in each environment. Team members and their roles are managed from Oracle Cloud Infrastructure Classic Console.

> **Note:**
>
> A **service administrator** can assign MCS roles to existing team members. To create new team members, you need to be assigned the **identity domain administrator** role in Oracle Cloud by the **account administrator**.
> As **account administrator**, be judicious about granting the **identity domain administrator** role. It's required to create team members and mobile users, but it also grants broader permissions over your MCS instance in Oracle Cloud.

To add users and assign them roles:

1. Sign in to your Oracle Cloud account.
2. On the Oracle Cloud Infrastructure Classic Console, click the navigation menu in the top left corner, and choose **Users**.
3. For each team member, click **Add** and fill in the name, email, and other required information.
4. In the Simple Role Selection section, select roles for each user.

   For development environments, it's generally a good idea to assign team members all of the MCS roles described below in development environments (*except for the* `MobileEnvironment_BusinessUser` *role*) to make sure that they can complete all of the development activities. (Use the `MobileEnvironment_BusinessUser` *only* for team members you want to go straight to Mobile Application Accelerator (MAX) without seeing the rest of the MCS interface.)

   For production environments, most team members should have more limited access.

If you need more detailed instructions, see Adding Users and Assigning Roles in *Getting Started with Oracle Cloud*.

MCS Team Member Roles

MCS team member roles are predefined and can't be created or customized. Team members must be assigned at least one of the roles in the table below in each environment they should have access to.

Role Name	Privileges	Available Actions
MCS Team Member (`MobilePortal_TeamMember`)	Access to the MCS UI. All team members. *The MCS UI is represented by an environment in Oracle Cloud, called the MCS UI service. All team members must be granted this role in the MCS UI service in addition to roles granted in other MCS environments.*	• Access the MCS UI
Mobile Analytics (`MobileEnvironment_Analytics`)	Read-only access to analytics data for the environment.	• View analytics data and define custom reports

Role Name	Privileges	Available Actions
Mobile Database Management (`MobileEnvironment_DbMgmt`)	Use the Database Management API to to view, create, and drop tables.	• Access the database • Migrate data
Mobile Deploy (`MobileEnvironment_Deploy`)	Control artifact versions deployed within the environment and configure artifact policies and instance data.	• Deploy versioned artifacts • Create, modify and remove artifact policies • Modify artifact instance data
Mobile Develop (`MobileEnvironment_Develop`)	Create, configure and publish new artifacts, such as mobile APIs and custom code. Create and test mobile apps using MAX. This role is only useful in development environments.	• Create a draft of an artifact • Modify artifact metadata • Publish an artifact • Test custom code by creating mobile apps using Mobile Application Accelerator (MAX)
Mobile Location Management (`MobileEnvironment_LocationMgmt`)	Create, configure and delete location artifacts such as assets, devices and places so applications can query location data.	• View location devices, places and assets from the UI • Create location devices, places, and assets from the UI • Modify location devices, place, and assets from the UI • Delete location devices, places, and assets from the UI
Mobile System (`MobileEnvironment_System`)	Access the Location Management API from custom code.	• Create location devices, places, and assets from custom code • Modify location devices, place, and assets from custom code • Delete location devices, places, and assets from custom code
Mobile User Configuration (`MobileEnvironment_MobileUserConfig`)	Define realms and roles for mobile users so applications can use role-based access policies. *You must also be granted the role of identity domain administrator in Oracle Cloud to manage roles and realms.*	• Create a role • Delete a role • Create a realm • Modify a realm (draft/publish) – Add a user attribute – Remove a user attribute
Mobile User Management (`MobileEnvironment_MobileUserMgmt`)	Manage mobile users within a realm, including creating mobile users and assigning roles. *You must also be granted the role of identity domain administrator in Oracle Cloud to manage users.*	• Create, update, suspend, activate and remove mobile users • Assign mobile roles to mobile users • Reset a mobile user's password

Role Name	Privileges	Available Actions
Mobile Monitor (`MobileEnvironment_Monitor`)	Read-only access to diagnostics data for the environment.	• View diagnostic data and define custom reports
Mobile Notifications (`MobileEnvironment_Notifications`)	Send and receive notifications in the environment.	• Create (send) and query for notifications
Business User (`MobileEnvironment_BusinessUser`)	Access to the Mobile Application Accelerator (MAX) development UI. Blocks access to the rest of the MCS UI. **Never grant the Business User role to a MCS mobile app or service developer or assign it to a production environment.**	• Create and test mobile apps using MAX
MAX Mobile App Deployment (`MobileEnvironment_MAXApplicationDeploy`)	Access to the MAX production environment and MAX application deployment features.	• Publish mobile apps using MAX

The naming convention for Oracle Cloud roles that correspond to MCS team member roles is: `{serviceName}.{rolename}`. For example, in the environment with service name `paid1247mobsvc002dev` the name of the Oracle Cloud role for the `MobileEnvironment_Deploy` team member role would be `paid1247mobsvc002dev.MobileEnvironment_Deploy`. Service names for MCS environments are listed on the Oracle Cloud Infrastructure Classic Console.

You might see some extra roles in the list in Oracle Cloud, including a Mobile Team Management role in several environments and extra Mobile Monitor and Mobile User Management roles in the UI environment. You don't need to assign those roles to anyone, as they aren't used in this release.

Team member roles are different from the mobile user roles that you assign to end users of your apps. For details on mobile user roles, see Creating and Managing Mobile User Roles.

Distinguishing Between MAX Team Member Roles for Business Users and for Mobile App Developers

MAX (Mobile Application Accelerator) is a development tool for business users, but MCS mobile app and service developers can also use MAX to test custom code. You can find out more about MAX and how it's used in Creating APIs Fast with the Express API Designer.

To set up MCS so both business users and MCS developers can use MAX, take care in assigning roles. Both business users and MCS developers need the `MobilePortal_TeamMember` role to access the mobile portal, but these two types of users access MAX differently.

- The `MobileEnvironment_BusinessUser` role must only be assigned to a business user in the MAX development environment so they can bypass the rest of MCS. Business users with this role are MAX-only users and can't even see the MCS UI.

Never assign this role to a MCS mobile app or service developer or to a production environment.

- The `MobileEnvironment_Develop` role grants access to MAX from within the MCS UI. To make sure that MCS mobile app and service developers can open the Applications page and aren't trapped in MAX, always assign them the `MobileEnvironment_Develop` role, and not the `MobileEnvironment_BusinessUser` role.
- The `MobileEnvironment_MAXApplicationDeploy` role in the MAX production environment enables both business users and MCS developers to publish apps using MAX. When this role is assigned, MAX is included on the Applications page for the environment.

To find out more about accessing MAX, see Who Uses MAX?

Example Team Member Role Assignments

This table shows one way you could assign MCS team member roles by environment for the common jobs described in Get to Know Oracle Mobile Cloud Service. All team members also need to be assigned the `MobilePortal_TeamMember` role in the MCS UI service.

> **Caution:**
>
> When creating team member accounts for Mobile Application Accelerator (MAX), be sure to keep the roles and their associated environments straight. Do not grant the MAX `BusinessUser` role to MCS mobile app or service developers or they will be limited to the MAX UI and won't have access to MCS development features. Also, the MAX development environment is identified by the *`MobileEnvironment`*`_BusinessUser` role, so take care when choosing the service instance name in the Oracle Cloud Infrastructure Classic Console. Do not assign this role to the MAX production environment.

Job	Development Environment Roles	Staging Environment Roles	Production Environment Roles
enterprise architect	MobileEnvironment_Analytics, MobileEnvironment_DbMgmt, MobileEnvironment_Deploy, MobileEnvironment_Develop, MobileEnvironment_LocationMgmt, MobileEnvironment_System, MobileEnvironment_MobileUserConfig, MobileEnvironment_MobileUserMgmt, MobileEnvironment_Monitor, MobileEnvironment_Notifications	MobileEnvironment_Analytics, MobileEnvironment_DbMgmt, MobileEnvironment_Deploy, MobileEnvironment_LocationMgmt, MobileEnvironment_System, MobileEnvironment_MobileUserConfig, MobileEnvironment_MobileUserMgmt, MobileEnvironment_Monitor, MobileEnvironment_Notifications	MobileEnvironment_Notifications
mobile cloud administrator	MobileEnvironment_Analytics, MobileEnvironment_DbMgmt, MobileEnvironment_Deploy, MobileEnvironment_Develop, MobileEnvironment_LocationMgmt, MobileEnvironment_System, MobileEnvironment_MobileUserConfig, MobileEnvironment_MobileUserMgmt, MobileEnvironment_Monitor, MobileEnvironment_Notifications	MobileEnvironment_Analytics, MobileEnvironment_DbMgmt, MobileEnvironment_Deploy, MobileEnvironment_LocationMgmt, MobileEnvironment_System, MobileEnvironment_MobileUserConfig, MobileEnvironment_MobileUserMgmt, MobileEnvironment_Monitor, MobileEnvironment_Notifications	MobileEnvironment_Analytics, MobileEnvironment_DbMgmt, MobileEnvironment_Deploy, MobileEnvironment_LocationMgmt, MobileEnvironment_System, MobileEnvironment_MobileUserConfig, MobileEnvironment_MobileUserMgmt, MobileEnvironment_Monitor, MobileEnvironment_Notifications

Job	Development Environment Roles	Staging Environment Roles	Production Environment Roles
mobile app developer and service developer	MobileEnvironment_Analytics, MobileEnvironment_DbMgmt, MobileEnvironment_Deploy, MobileEnvironment_Develop, MobileEnvironment_LocationMgmt, MobileEnvironment_System, MobileEnvironment_MobileUserConfig, MobileEnvironment_MobileUserMgmt, MobileEnvironment_Monitor, MobileEnvironment_Notifications,	MobileEnvironment_Analytics, MobileEnvironment_MobileUserMgmt, MobileEnvironment_Monitor, MobileEnvironment_Notifications	MobileEnvironment_Notifications, MobileEnvironment_MAXApplicationDeploy
mobile program manager	MobileEnvironment_Analytics, MobileEnvironment_DbMgmt, MobileEnvironment_Deploy, MobileEnvironment_Develop, MobileEnvironment_LocationMgmt, MobileEnvironment_System, MobileEnvironment_MobileUserConfig, MobileEnvironment_MobileUserMgmt, MobileEnvironment_Monitor, MobileEnvironment_Notifications	MobileEnvironment_Analytics, MobileEnvironment_Notifications	MobileEnvironment_Analytics, MobileEnvironment_Notifications
business user	MobileEnvironment_BusinessUser	N/A	MobileEnvironment_MAXApplicationDeploy

Remember, to create new team members or mobile users, a team member also needs to be granted the identity domain administrator role in Oracle Cloud.

Set Up Mobile Users, Realms and Roles

Mobile users are your customers — the ones who use the mobile apps built with MCS. Organize your mobile users by setting up **realms** that define the user schema, and creating **roles** to grant access permissions. It's a good idea to define some realms and roles before app developers start working with MCS. You can also set up some initial mobile users for testing and maybe import larger groups of mobile users.

> **Note:**
>
> To manage mobile users, roles and realms, you need to be assigned the mobile user configuration (`MobileEnvironment_MobileUserConfig`) and mobile user management (`MobileEnvironment_MobileUserMgmt`) MCS team member roles in the environment, as well as the identity domain administrator role in Oracle Cloud.

Manage mobile users, realms and roles in MCS from **Applications > Mobile User Management**.

Creating Realms

A realm is a container for managing mobile users within an environment. Each realm includes a user schema that defines the user data that can be stored and made accessible to mobile apps. You can define custom properties for a user schema, but the following properties are required:

- user name
- password
- first name
- last name
- e-mail

To create a new realm, start in a development environment. Available realms are listed under Mobile User Management in the side menu.

1. Make sure you're in the development environment where you want to create the realm.
2. Click  to open the side menu and select **Applications > Mobile User Management**.
3. Click the **Realms** navigation link.
4. To create a realm, click **New Realm**.
5. Enter a unique name and an optional description. The realm name can't be changed after the realm is created.
6. If you want to add a custom property to the user schema, click **New Field**.
    a. Enter a unique name for the field and an optional description.

You can't use any of the following reserved field names: `firstname`, `lastname`, `email`, `username`, `password`, `createdOn`, `createdBy`, `modifiedOn`, `modifiedBy`, `id`, `roles`, and `links`.

b. Select the appropriate data **Type** for the field: string, number, date or Boolean.

c. Click **Create** to add the new field to the user schema.

7. When you've finished, click **Save** to save your changes to the realm and return to the Realms tab.

After a realm is published, the user schema can't be changed. Realms can't be deleted from MCS.
Realms are deployed automatically with the associated mobile backend. Only the user schema is deployed; no user data is migrated. For detailed information on publishing and deploying realms, see Realm Lifecycle. If you want to change the realm associated with an existing mobile backend, see Changing a Mobile Backend's Realm.

Setting the Default Realm for an Environment

When you create a new mobile backend, it's automatically associated with the default realm for the environment. You can set this default realm to any available realm in the environment.

1. Make sure you're in the environment where you want to set the default realm.
2. Click ☰ to open the side menu and select **Applications > Mobile User Management**.
3. Click the **Realms** navigation link.
4. Select the realm that you want to make the new default. Click **More** and select **Make default realm**.

Creating and Managing Mobile User Roles

Mobile user roles allow you to define permissions for your apps and assign them to mobile users. You can define as many roles as you need, and you can assign multiple roles to the same mobile user.

A mobile app can allow different access to mobile users with different roles. You could assign a Technician role to a mobile technician to grant access to specific features of the company's mobile app, and a Salesperson role to a sales rep to grant access to different features. The same mobile technician could have a Customer role in the company's supply ordering app where the sales rep has no role assigned.

To create and manage mobile user roles:

1. Make sure you're in the environment where you want to create the role(s).
2. Click ☰ to open the side menu and select **Applications > Mobile User Management**.

> **Note:**
>
> Though it's possible to create and delete mobile user roles from My Services in Oracle Cloud, you should handle all operations on mobile user roles from Mobile User Management in the MCS UI.

3. Click the **Roles** page. From here you can view and edit available mobile user roles and create new roles. As soon as you create a role, it's added to the list on the Roles page and you can define access permissions.
   - Role names are case-sensitive.
   - Roles are deployed automatically with any object that references them.

Once you've defined roles, you can use them throughout MCS:

- Assign roles to individual mobile users from the **Mobile Users** page in MCS, or use Oracle Cloud to batch assign roles to groups of mobile users, described in Importing Groups of Mobile Users Into MCS Using Oracle Cloud.
- Assign specific permissions for objects and resources to the roles you've defined, as described in Adding Access Permissions to a Collection.
- Restrict access to APIs and individual methods, as described in Setting Access to the API.

Creating Mobile Users and Assigning Roles

From the **Mobile Users** page in MCS Mobile User Management, you can create and edit users and assign roles, search for an existing user, and reset a user's password to a system-generated temporary password that is sent to the user's email address. Remember, you can only create mobile users if you have the identity domain administrator role in Oracle Cloud.

For more thorough testing or for production, you'll probably want to import a group of users. To import groups of users into MCS, use Oracle Cloud to batch assign them to a realm. You can also use Oracle Cloud to batch assign mobile user roles. For detailed instructions, see Importing Groups of Mobile Users Into MCS Using Oracle Cloud.

> **Note:**
>
> In all cases, when you a create mobile users, they are sent a temporary password. The new users need to use this temporary password to log into the Oracle Cloud Infrastructure Classic Console, change the password, and set up their challenge questions before they can be recognized as an MCS mobile user.

Creating Individual Mobile Users for Testing

You can use the MCS UI to create individual mobile users and assign roles. Here are the steps for quickly creating a test user. Some steps include suggested values that will allow app developers to seamlessly complete the *Get Started with Mobile Development* tutorial on the MCS home page.

1. Make sure you're in the environment where you want to create the mobile user(s).
2. Click  to open the side menu and select **Applications > Mobile User Management**.
3. Click **Mobile Users**.
4. Select the **Realm** where you want to create the user.
5. Click the **New User** button.
6. Enter a unique user name and fill in the remaining fields in the dialog, including an email address where you can retrieve the generated password.

   The available fields may vary depending on the realm where you're creating the user. The *Get Started with Mobile Development* tutorial uses the user name `Joe`.

   > **Note:**
   >
   > Both user name and email address must be unique across all services in Oracle Cloud.

7. If you haven't created the role you need yet, you can add a new role to the environment by clicking **Create Role** on the right side of the dialog.

   The *Get Started with Mobile Development* tutorial uses the role name `Technician` for the user Joe.

8. Click **Create** again to create the new mobile user.

   An email is sent from Oracle Cloud to the address you entered with a temporary password.

9. (Optional) Assign roles to an individual mobile user from the **Mobile Users** page in MCS.

You can only assign a mobile user to one realm via the MCS Mobile Users page, but you can associate mobile users with multiple realms using Oracle Cloud. For more thorough testing or for production, you'll also probably want to import a group of mobile users.

Importing Groups of Mobile Users Into MCS Using Oracle Cloud

You can use Oracle Cloud to import a group of users into MCS or assign MCS roles to a group of users, using the steps below. MCS mobile user realms and roles are both represented by custom roles in Oracle Cloud. As with all mobile user operations in this section, you need the identity domain manager role in Oracle Cloud to complete these steps.

1. Create the MCS realm and mobile user roles you want to assign to the group of users, if you haven't already. For detailed instructions, see Creating Realms and Creating and Managing Mobile User Roles.
2. Create a group of mobile users in Oracle Cloud using a comma-separated values (CSV) file.

   For detailed information on batch importing users, including the related CSV files, see Importing a Batch of User Accounts in *Getting Started with Oracle Cloud*.

3. Import the users into MCS by assigning the group to the Oracle Cloud custom role that represents the MCS realm you created in step 1.

   The naming convention for Oracle Cloud custom roles that represent MCS realms is: `{serviceName}_MobileEnvironment_{realmname}_{version with dots as underscores}_Realm` where `{serviceName}` is the service name of the environment in Oracle Cloud. You can find the service names for all MCS environments on the Oracle Cloud Infrastructure Classic Console. For example, for the default realm version 1.0 in the environment with service name "3240930apod" the custom role in Oracle Cloud would be `3240930apod_MobileEnvironment_Default_1_0_Realm`, or for the MyCustomers realm version 2.5 in the environment with service name "poeo342ed" it would be `poeo342ed_MobileEnvironment_MyCustomers_2_5_Realm`. For detailed instructions, see Assigning One Role to Many Users in *Getting Started with Oracle Cloud*.

4. (Optional) Assign MCS mobile user roles to the group by assigning Oracle Cloud custom roles using the same process you did for the realm in the previous step.

   The naming convention for Oracle Cloud custom roles that represent MCS mobile user roles is: `{serviceName}_MobileEnvironment_{rolename}`. For example, for a role named "APIRole" in the environment with service name "poeo342ed" the custom role in Oracle Cloud would be `poeo342ed_MobileEnvironment_APIRole`.

Mobile Users for MAX

In addition to their team member accounts, MAX (Mobile Application Accelerator) business users need mobile user accounts to test and use their mobile apps. For details on MAX team member roles, see Distinguishing Between MAX Team Member Roles for Business Users and for Mobile App Developers. For more information about MAX, see Using the Express API Designer with MAX.

Role	Definition
test user	A test user account enables MAX users to preview apps using live data. It also enables them to generate the QR code that identifies the test version of an app. For more information on creating a test user account, see Creating Individual Mobile Users for Testing.
mobile user	Mobile user accounts enable everyone (business users, MCS developers, and mobile app users) to log in to MAX and use published mobile apps. Anyone who tests or uses a mobile app built using MAX needs a mobile user account. For more information, see Importing Groups of Mobile Users Into MCS Using Oracle Cloud.

Changing a Mobile User Password

As mobile cloud administrator, you can change a mobile user's password from the Mobile Users page in MCS Mobile User Management. Mobile users can change their own passwords from Oracle Cloud Identity Self Service.

1. Click  to open the side menu and select **Applications > Mobile User Management**.
2. Click **Mobile Users**.
3. Select the mobile user on the Mobile Users page and click the **Reset password** button. MCS will send an email with a temporary password to the email address associated with the user.

Configuring Identity Management (SSO and OAuth)

MCS allows you to use single sign-on (SSO) with OAuth so your mobile apps can use your own identity provider (IdP) for authentication.

- If you want to use a third-party IdP as your identity store (without any corresponding accounts for your users in Oracle Cloud), you can use SAML and JWT tokens for authentication. See Third-Party SAML and JWT Tokens.
- If you want to use a third-party IdP in conjunction with Oracle Cloud user accounts, configure the connection between Oracle Cloud and the identity provider from the **Users** page in Oracle Cloud Infrastructure Classic Console. For detailed instructions, see Managing Single Sign On in *Administering Oracle Cloud Identity Management*.

Configuring Oracle Cloud Applications as the Identity Provider

If your team will be creating mobile apps that are designed for users of Fusion Applications-based services such as Oracle Sales Cloud, Oracle HCM Cloud, and Oracle ERP Cloud, you will probably want to enable those users to sign in to the mobile app once and not have to re-enter credentials to access the Oracle Cloud application.

For your mobile app and service developers to be able to create such apps where the user only needs to sign in once, you need to get the following things in place:

1. Have your MCS instance provisioned in the same identity domain as the Oracle Cloud application service that your apps will access.
2. Enable SSO for the identity domain and set the Oracle Cloud application service as the identity provider.
3. Enable sign—on with identity domain credentials. This enables team members to sign in with their Oracle Cloud credentials. Otherwise, they would be prompted to log in with credentials for the Oracle Cloud application service (which they might not have).

   The steps for this are:

   a. In Oracle Cloud Infrastructure Classic Console, go to the **SSO Configuration** page.
   b. Go to the **Enable Sign In to Oracle Cloud Services with Identity Domain credentials** section and click **Enable**.

> **Note:**
>
> You can only designate one identity provider to be used with SSO.

Once the services are set up in the same identity domain and SSO has been enabled, the mobile app developer can do the following to enable the app user's login credentials to propagate to the Oracle Cloud application:

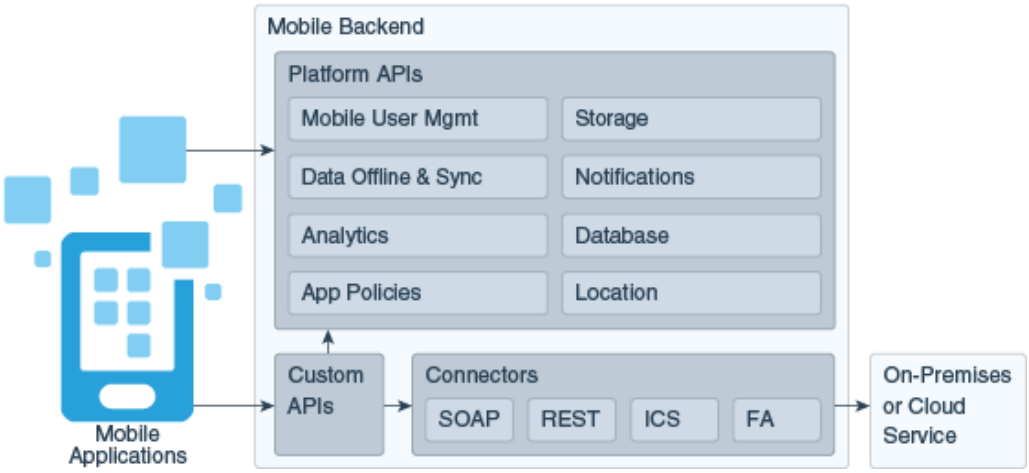
- Create a Fusion Applications connector API to connect to the Oracle Cloud application service.
- Within the connector API, designate the appropriate security policy to handle authentication and authorization with the service.
- Create a custom API that calls the connector API.
- Create a mobile backend, enable it to use SSO, and associate the custom API with it.

Get on Board

Once you're assigned a role in MCS, you can log in and get to work. To open MCS from the Oracle Cloud Infrastructure Classic Console, click the **Open Service Console** link in the MobilePortalService box. (This link is only accessible to team members with administrative roles.)

> **Note:**
>
> If you see an error when you try to access MCS, you probably don't have all the roles you need. Ask your service administrator to assign you the necessary MCS roles.

Part II
Setting up Mobile Apps

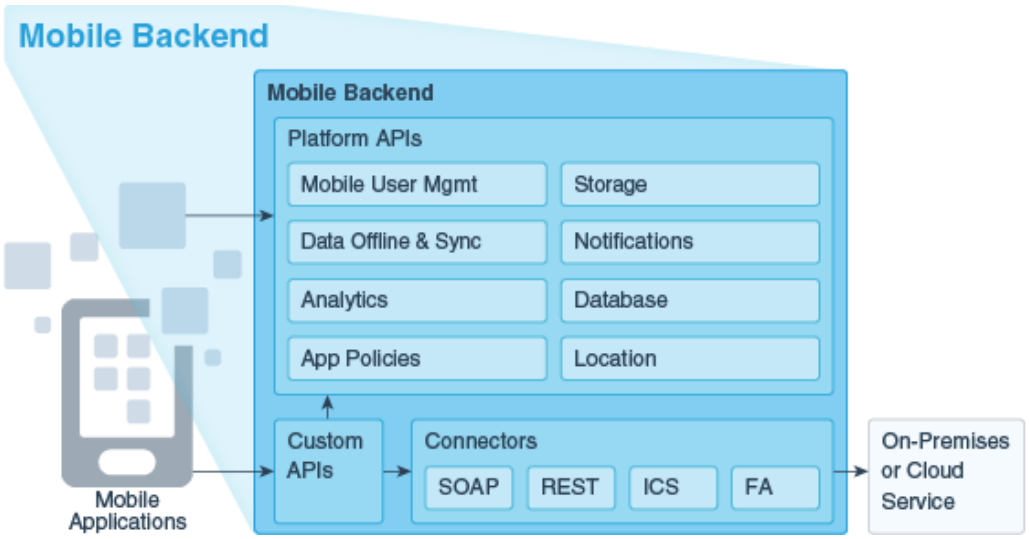


This part contains the following chapters:

3

Mobile Backends

Oracle Mobile Cloud Service (MCS) is built around the concept of mobile backends, which enables you, as a mobile app developer, to develop and deploy groupings of APIs that are designed to support a specific set of mobile apps. You can then associate one or more apps with the mobile backend to access those APIs.



What Is a Mobile Backend and How Can I Use It?

A mobile backend is a secure grouping of APIs and other resources for a set of mobile apps. Within a mobile backend, you select the APIs that you want available for those apps. For any apps that you want to receive notifications, you can also register the appropriate credentials for the given network (e.g. APNS, GCM, or WNS) in the mobile backend.

You can have multiple backends, each serving a set of applications. In addition, you can have APIs that are used by multiple backends.

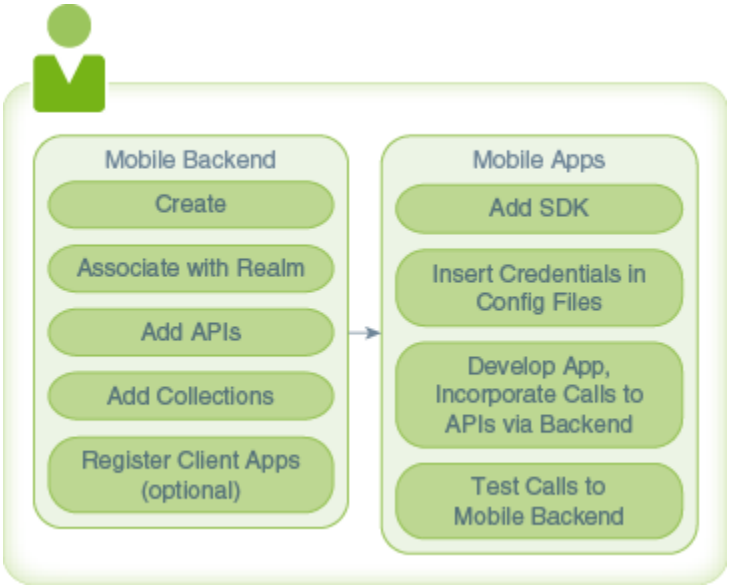
When an app accesses APIs through MCS, it is always in the context of a mobile backend. The app authenticates with credentials (OAuth Consumer or HTTP Basic Authentication) specific to the mobile backend or through an identity store (or social login provider) that is mediated by your mobile backend. If the called API includes calls to other APIs within the backend, the identity and credentials of the original caller are propagated through the chain of calls.

You don't have to start your work in MCS with a mobile backend (for example, you could start developing custom APIs or set up storage collections first without associating them with any mobile backends). But you may find it useful to do so. Working in mobile backends helps you visualize the resources available for the target apps and how they will work together. In addition, you can use the mobile backend's security context to test calls to your APIs, even in the earliest stages of development.

What's the Mobile Backend Development Process?

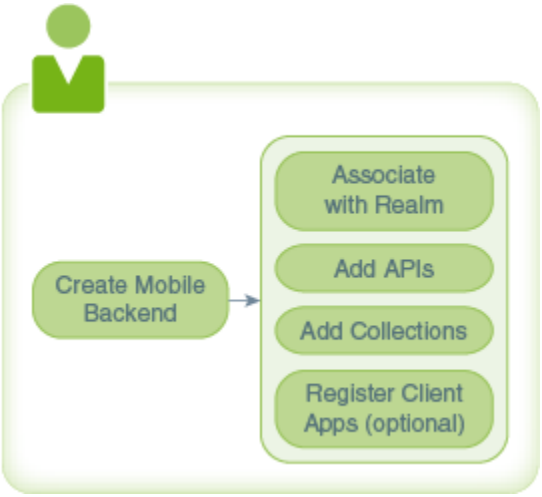
Generally speaking, using MCS entails developing APIs, grouping them in mobile backends, and developing mobile apps that use these mobile backends. The development model is flexible, allowing you to work on APIs, mobile backends, and mobile apps largely in parallel.

As shown in this figure, the general workflow includes steps both for creating and filling out the mobile backend and for setting up your app to work with the mobile backend.



Creating and Populating Mobile Backends

You create and populate mobile backends directly in Oracle Mobile Cloud Service. Once you have created a mobile backend, you can associate APIs and Storage collections with it, and register client apps that will use the mobile backend.

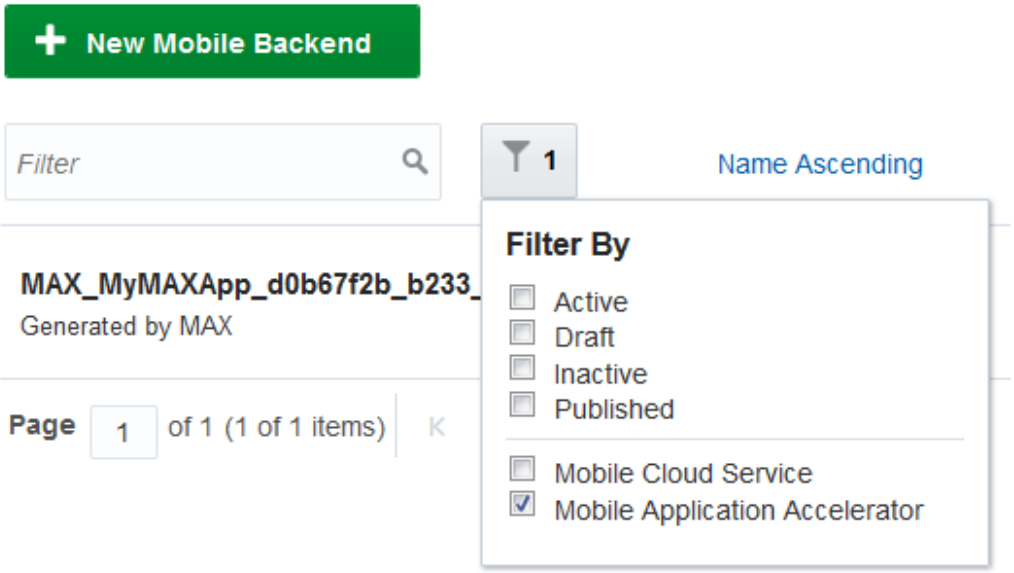


Creating a Mobile Backend

1. Make sure you're in the environment where you want to create the mobile backend.
2. Click  to open the side menu and select **Applications > Mobile Backends**.
3. Click **New Mobile Backend**.
4. Enter a name for the mobile backend and a description.

Mobile Backends for MAX Apps

A mobile backend is created on the fly whenever a Mobile Application Accelerator (MAX) user creates an app. These mobile backends are named with a *MAX_* prefix, followed by the name of the MAX app itself, another underscore (_) and an App ID. For example, *MAX_myMAXApp_0123_a4563*. (MCS inserts underscores if a MAX app's name includes spaces: *My MAX APP* becomes *My_MAX_App*, for example.) Use the **Mobile Application Accelerator** filtering option to locate these mobile backends. Although these mobile backends are created automatically and are already associated with a client app (that is, a MAX App), you can use the Settings page to update them just as you would with any other mobile backend. For example, you can add SSO support to your MAX apps.



Mobile Backend Authentication and Connection Info

The following authentication and connection details are generated when you create a mobile backend and are displayed on the mobile backend's **Settings** page. Your apps use these details to connect to and authenticate with APIs associated with that mobile backend. These credentials can be used by every application associated with that mobile backend.

- **Environment URLs**

  - The **Base URL** is needed for all API calls. This URL is distinct for each environment that you have provisioned.
  - The **OAuth Token Endpoint** is the URL that your app needs to use to make OAuth token requests.
  - A **SSO Token Endpoint** is also provided if you enable OAuth and then enable single sign-on (SSO) for your mobile backend. Your app would use this URL to obtain a single sign-on OAuth token in order to login through a remote identity provider.

- A set of **Authentication Keys**, which your app needs to access APIs through the mobile backend. Keys are generated for both OAuth Consumer and HTTP Basic authentication. Use the toggle switch next to each to enable or disable access through that protocol.

  A set of **Access Keys**, which your app needs to access APIs through the mobile backend. Keys are generated for both OAuth Consumer and HTTP Basic authentication. Use the toggle switch next to each to enable or disable access through that protocol. For OAuth, you can also enable SSO in order to allow your company's identity provider to be used authenticate users.

  OAuth Consumer keys are generated in the form of a client ID and a client secret. These two values are unique to this mobile backend.

  HTTP Basic Authentication keys are generated for you in the form of a mobile backend ID and an anonymous key.

  These keys are also unique by environment. When you deploy a mobile backend to a different environment, a new set of keys is generated for the copy of the mobile backend that is added to the target environment.

  If you suspect that these credentials have been compromised (such as by an application handling them insecurely), click **Refresh** to replace the credentials with new ones or click **Revoke** to cancel the existing credentials without generating replacements.

  > **Note:**
  >
  > Think twice before refreshing or revoking credentials, since these actions will block any calls that any existing apps make through the mobile backend. To get the apps working properly again after credentials have been revoked or refreshed, you need to rebuild the apps with the new credentials and redeploy them.

For details on using the various authentication methods, see Authentication in MCS.

To make it easier to incorporate these details in your apps, use the MCS SDKs for your app platforms. See The SDKs.

Environments and Mobile Backends

All work on mobile backends takes place in the context of an *environment*. You can use a separate environment for each phase in the mobile backend lifecycle, such as development, testing, and production.

Typically you create a mobile backend in an environment that you have designated for development, publish that mobile backend, and then deploy it to another environment

for testing. Once thoroughly tested, you would then deploy the mobile backend to your production environment.

For more on environments, see What is My Environment?.

Realms and Mobile Backends

A realm is the security context for a set of users that defines a set of properties that contain information on the user, such as user ID and user name as well as any custom information that is relevant to the purpose of the apps using that realm.

You can have different realms for different purposes. Each mobile backend in an environment can be associated with only one realm, but multiple mobile backends can be associated with the same realm, allowing them to use a shared set of users and data. When you create a mobile backend, it is assigned to the default realm for the environment.

You can change the realm associated with a mobile backend from the Users tab of the mobile backend. Realms are typically handled by users with the Oracle Cloud identity domain administrator role. If you don't have that role and you need to change the mobile backend's realm, contact someone who does have that role. For details on the default realm, see Setting the Default Realm for an Environment.

Even when a mobile backend is configured to allow login through enterprise SSO, it needs a realm that contains records for the users that log in through SSO. In this case, the realm would define only the properties needed to match the user records with those in the identity provider (such as user name or email address).

> **Note:**
>
> When you change the realm for a mobile backend, the user properties and user data also change. Make sure that the new realm includes all the properties required by any mobile apps in the mobile backend.

Changing a Mobile Backend's Realm

1. Make sure you're in the environment where you want to change the realm.
2. Click  to open the side menu and select **Applications > Mobile Backends**.
3. Open the mobile backend. (Select it and click **Open**.)
4. Click the **Users** tab. This tab lets you search for and manage users, and change the realm for the mobile backend.

Getting Test Users for a Mobile Backend

You'll probably find it useful to have one or more test users set up in the realm associated with your mobile backend. Among other things, this will make it easier to try out APIs in your mobile backend. As an app developer, you probably don't have the permissions necessary to create test users, but a person on your team with the Oracle Cloud identity domain administrator role can.

To see if you have any test users:

1. Make sure you're in the environment where you want to work with test users.
2. Click  to open the side menu and select **Applications > Mobile Backends**.
3. Select your mobile backend and click **Open**.
4. In the left navbar, click **Users**.

If you don't have any test users, see Creating Individual Mobile Users for Testing for information on creating them.

Associating APIs with a Mobile Backend

Once you have a mobile backend, you can use the API Catalog to select the custom APIs you want to access through that mobile backend. The API Catalog provides detail on each API endpoint and its documentation, as well as an opportunity to test the endpoint with mock data to see what it does.

1. Make sure you're in the environment containing the draft mobile backend.
2. Click  to open the side menu and select **Applications > Mobile Backends**.
3. Select your mobile backend and click **Open**.
4. In the left navbar, click **APIs**.
5. Click **Select APIs**.
6. Optionally, click an API's name to view its endpoints.

   At this stage, you can click **Test Endpoint** to see how the API works with mock data. To do so, you also need to provide a user name and password. If you don't yet have a test user, see Creating Test Users for info on creating one.

   For custom APIs, you can also specify that the API can be accessed without a user login. See Testing Your Custom API for more details.
7. Click the **+** (Add) icon for each API that you want to include.

> **Note:**
>
> Platform APIs (for Storage, Mobile User Management, Analytics, etc.) are automatically available in your mobile backends. If an API with the functionality that you are looking for isn't available, you can design such an API yourself. See Custom API Design.

Associating Storage Collections with a Mobile Backend

You can associate a mobile backend with collections so that your mobile apps can work with data in those collections using the MCS platform's Storage API.

To associate your mobile backend with an existing collection:

1. Make sure you're in the environment containing the draft mobile backend.
2. Click  to open the side menu and select **Applications > Mobile Backends**.
3. Select your mobile backend and click **Open**.

4. In the left navbar of the mobile backend, click **Storage**.
5. Click **Select Collections**.
6. Start typing the name of the collection that you want to add, select the collection from the drop-down list, and click **Select**.

For more on collections, including creating them, see Storage.

Clients and Mobile Backends

You can associate apps with a mobile backend by registering them as clients in MCS and then picking the mobile backend for them to use. In the process, you can also set up notifications profiles for the clients to use. See Client Management for information on registering clients.

What Can I Change in a Mobile Backend?

If you haven't yet published your mobile backend, you can change the following things that are associated with the mobile backend at any time:

- Registered clients
- Notifications credentials
- Custom APIs (and their implementations)
- Any connector APIs that are called from custom API implementations
- Storage collections
- User realm
- App policies

Once you have published a mobile backend, its content is frozen. At that point, you would need to create a new version of the mobile backend to make any changes. See Mobile Backend Lifecycle if you are interested in a rundown of publishing, deploying, and versioning mobile backends.

> **Note:**
>
> Though you can't change the list of app policies in a published mobile backend, you can change their *values*.

Video: Mobile Backend Design Considerations

Before you start creating mobile backends, you should spend some time analyzing what your apps need from the mobile backends, what different apps will have in common, and what kind of approach will be easiest to maintain. To help you think about these questions, watch the following video on the Oracle Mobile Platform channel on YouTube:

Video

The SDKs

MCS provides client SDKs for multiple platforms to help you use MCS APIs in your apps. The SDKs simplify app development in the following ways:

- Simplify the passing of access keys and environment details in all of your API calls, including for custom APIs. All APIs in MCS are REST APIs that are called with an HTTPS request, including headers containing security credentials and mobile backend environment details. With each SDK, you use a configuration file to hold these values in one place so that they do not have to be hard-coded into each API call.
- Provide wrapper classes for key endpoints in the platform APIs.
- Set up the network connection between your mobile app and its mobile backend.

You can get the SDKs from the Oracle Technology Network's MCS download page.

For specific info on each SDK, see Android Applications, iOS Applications, Cordova Applications, JavaScript Applications, Xamarin Android Applications, and Xamarin iOS Applications.

There is also a utility for accessing MCS from Oracle Mobile Application Framework (MAF) apps that is available in a MAF sample app. Go to the Oracle Mobile Application Framework Samples page to get the sample and download the MAF MCS Utility Developer Guide to learn more about using it.

> **Note:**
>
> For information on using the REST APIs directly, see the platform's REST API reference docs.

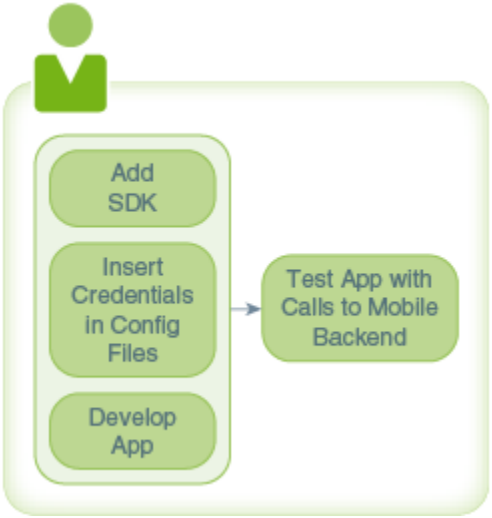
Connecting Your App to a Mobile Backend

Once you have a mobile backend set up and a client application registered with that mobile backend, you need to configure your app code to access the mobile backend.

Connecting your app to a mobile backend involves these basic steps:

- Adding the SDK libraries to your app. (This step is optional, but highly recommended.)
- Adding a configuration file to your app to hold environment information that your app needs to access the mobile backend. The SDK classes that you use to make calls to the mobile backend use the values in this file so that you don't have to manually include them in each of your calls.
- Adding calls to MCS APIs in your app.

  The APIs available include MCS platform APIs and any custom APIs that you or other members of your team have developed in MCS.
- Testing your app.



For platform-specific details on setting up your apps, see:

- iOS Applications
- Android Applications
- Cordova Applications
- JavaScript Applications

4

Client Management

To simplify handling of notifications and management of the application lifecycle in Oracle Mobile Cloud Service (MCS), you can register your mobile apps in MCS as clients and associate them with a mobile backend and a Notifications profile. When it comes time to deploy an app, you can deploy the client you have registered and have its associated mobile backend and its dependencies deployed as well.

Registering a client accomplishes the following things:

- Enables you to store the ID that is needed for the app store.
- Enables the app to receive notifications via MCS.
- Simplifies lifecycle management of the app and its associated mobile backend and related artifacts.
- Enables collection of data specific to that app through the Analytics API.

How Clients Work in MCS

Here are the principles behind client registration in MCS:

- A client in MCS represents a single version of a single app binary.

  For example, if you have both iOS and Android versions of an app, you would register a client for each. Similarly, if you provide an upgraded version of the app, you would create a new client to hold its metadata.
- When you register a client, you specify metadata such as the application ID that is required by the platform vendor's app store, the app version number, and a profile that contains notifications credentials.
- Once the client is registered an application key is generated. In turn, you can use this key in your apps to access the client metadata. Each of the SDKs has a configuration file where you can insert this application key.
- A client can only be associated with one version of a mobile backend.

  This means that when you create a new version of a mobile backend, that mobile backend doesn't inherit any clients that you associated with the previous version of the mobile backend. So, as you create new versions of your mobile apps that use a new version of a mobile backend, you should create corresponding clients in MCS.
- A client can be published and deployed in a way similar to other artifacts. When a client is deployed, its mobile backend and other dependencies are deployed with it.

  For a rundown on publishing, deploying, and versioning clients, see Client Lifecycle.

Profiles

Profiles serve as a place to store credentials for notification services. After you create a profile, you can associate it with multiple clients.

Creating a Profile

You create profiles to hold notification credentials that your clients need.

To create a profile:

1. Make sure you're in the environment where you want to create the profile.
2. Click  to open the side menu and select **Applications > Client Management**.
3. Click **Profiles**.
4. In the **New Profile** dialog:
   - Fill in the **Name**. This can be whatever name that will help you identify the profile most easily.
   - Select the **Notification Service**.
   - Fill in the rest of the dialog with the information required by the notification service. For details on getting credentials from your notification provider, including any additional setup steps, see Setting Up a Mobile App for Notifications.

     For Apple Push Notification Services (APNS), you need to register a certificate obtained from the Apple Developer portal.

     For Firebase Cloud Messaging (FCM) and Google Cloud Messaging (GCM), you must register server credentials obtained from the Developers Console for an Android application. (However, providing the package name is optional, because credentials may or may not be scoped to a specific app.)

     For Windows Notification Service (WNS), you register your app in the Windows Store Dashboard to get the credentials required to authenticate with the Windows Notification Service.

     For Syniverse (SMS), fill in the required fields:

     – **Channel ID** or sender address. A Channel represents a collection of sender addresses, for example, a set of SMS short codes that can be used to send text-based messages. A sender address can be any long code, short code or alphanumeric ID that applications can send SMS messages from. You can use your own sender address or purchase a sender address owned by Syniverse. When sending messages via a Channel, the Syniverse Messaging API service chooses the most appropriate sender address for each message and recipient. To get a Syniverse-provisioned test channel ID for testing SMS in the U.S. or Canada, go to your Syniverse Dashboard > Service Offerings > Messaging Accounts > Public Channels (U.S. apps must use the “US MT Test Channel”). To test in the U.S. or Canada, you also need to whitelist test phone numbers as described in Setting Up a Mobile App for Notifications.

- The authentication keys you got from Syniverse: **Consumer Key**, **Consumer Secret** and **Access Token**.
- By default, consent management is handled by Syniverse, but if you want your app to handle consent management or you want to register devices through the MCS UI, deselect **Consent Management Enabled**.

5. Click **Create**.

Once a profile is created, you can add it to a client by opening the client, selecting its **Profiles** tab, and clicking **Select Profile**.

You can add a profile to any client whose platform is valid for the profile's notification service and whose application ID matches that of the profile. If an FCM or GCM profile does not specify a package name, the profile may be used with any Android client.

Registering an App as a Client in MCS

1. Copy the bundle ID (for iOS), package name (for Android), or application ID (for Windows) so that you have it ready when creating the client.

   Once you create a client, you can't change this value, and the value needs to match that of the profile that you associate with the client.

   > **Note:**
   >
   > You might find it more convenient to create your profiles before registering the clients so that you have these credentials in hand when creating the client. Also, you might have multiple clients that use the same profile.

2. Make sure you're in the environment containing the version of the client you want to register.
3. Click  to open the side menu and select **Applications > Client Management**.
4. Click **New Client**.
5. In the **New Client** dialog:
   - Fill in the **Client Display Name** and **Client Name**.

     These can be whatever names that will help you identify the client most easily. The former can have spaces and the latter can't.

     In most places in the user interface, the client display name is used. The client name is used for clients in packages and the trash.
   - Select the **Platform** (iOS, Android, Windows, or Web).
   - Fill in the **Version Number** field. *This version must match the version number of the app as registered with your platform vendor*.
   - Fill in the fully-qualified app ID. You obtain this from the platform vendor.

     For Apple, it is the **Bundle ID** assigned to the application in the Xcode project.

     For Google, it is the **Package Name** for the application as declared in its manifest file.

For Microsoft, it is the **Application ID** you gave your app when you registered it in the Windows Dashboard.

For Web, it can be any unique identifier that distinguishes it from other web applications that you register.

6. Click **Create**.
7. On the **Settings** page, select a mobile backend to associate with the client from the **Mobile Backend** dropdown.
8. Click the **Profiles** tab and select one or more notifications profiles that you want to associate with the client.

> **Note:**
>
> If the notifications profile is for the notifications service of the app's vendor (e.g. APNS for an iOS app or FCM for an Android app), the app ID (bundle ID for iOS, package name for Android, or package SID for Microsoft) for the profile must match the app ID specified for the client. A client can only be associated with a single SMS profile.

Legacy Client Behavior

In versions of MCS previous to 16.4.1, there were some differences in how clients were handled:

- Client registrations and notifications profiles were not divided. Instead of referring to notifications profiles, client registrations held notifications credentials directly.
- Client registrations could apply to multiple versions of a mobile backend.

When your environment was upgraded to 16.4.1, these differences were reconciled in the following way:

- Any existing clients were split into clients and profiles.
- For any client that was associated with multiple versions of a mobile backend, the client only remained associated with the version of the mobile backend in which it was created.

5

Authentication in MCS

In Oracle Mobile Cloud Service (MCS), all resources are secured and can only be accessed by authenticated users that are authorized to access those resources. As a mobile app developer, you enable one or more authentication methods in the mobile backend and then write app code to use one of these methods.

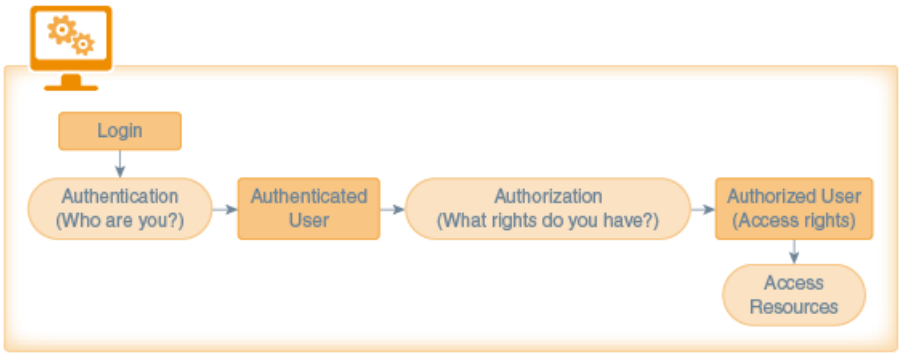
The authentication methods available are:

- OAuth Consumer
- HTTP Basic
- Enterprise Single Sign-On (SSO)

  This method includes variants for browser-based SSO and use of third-party tokens.
- Facebook Login

Before getting into the specifics of each authentication method, let's go over how authentication relates to authorization:

- **Authentication** is the process of identifying an individual, usually based on a user name and password, often in combination with other credentials such as an application key. Authentication ensures that the user is who he or she claims to be. This chapter explains how to use these features in your mobile apps.
- **Authorization** is the process of determining what an individual has permission to do. After the user gains access through authentication, the system grants access according to the settings configured for the user. The MCS Mobile User Management features let you configure an intelligent authorization policy based on user roles. For an introduction to MCS Mobile User Management, see Set Up Mobile Users, Realms and Roles.



OAuth Consumer Authentication in MCS

The ability to use OAuth as your authentication mechanism is built in to all mobile backends and enabled by default. Whenever you create a mobile backend, the OAuth Consumer keys are generated for you.

To enable or disable OAuth Consumer as an authentication method:

1. Open the mobile backend and select the **Settings** page.
2. Under **Access Keys**, set the **OAuth Consumer** switch to **ON** or **OFF**.

For details on the access keys and environment details provided, see Mobile Backend Authentication and Connection Info.

Once you have these keys, you can use them in your apps. When using the MCS SDK for a given mobile platform, you insert these access keys in the configuration file provided by the SDK and then the SDK uses them when constructing calls to REST APIs associated with the mobile backend. If you are coding the REST calls manually, see Authenticating with OAuth in Direct REST Calls.

HTTP Basic Authentication in MCS

The ability to use HTTP Basic as your authentication mechanism is built in to all mobile backends and enabled by default.

To enable or disable HTTP Basic as an authentication method:

1. Open the mobile backend and select the **Settings** page.
2. Under **Access Keys**, set the **HTTP Basic** switch to **ON** or **OFF**.

   When switched to **ON**, the access keys that you need are displayed.

For details on the access keys and environment details provided, see Mobile Backend Authentication and Connection Info.

Once you have these keys, you can use them in your apps. When using the MCS SDK for a given mobile platform, you insert these access keys in the configuration file provided by the SDK and then the SDK uses them when constructing calls to REST APIs associated with the mobile backend. If you are coding the REST calls manually, see Authenticating with HTTP Basic in Direct REST Calls.

Enterprise Single Sign-On in MCS

If you want to use your own identity provider to authenticate users of your apps, you can enable Oracle Cloud's single sign-on (SSO) capability to connect with that identity provider and then configure your mobile backends to use it. This is particularly useful if you are rolling out apps for your company's employees and you want them to be able to sign into the apps using their existing employee login credentials. Similarly, this could work for consumer applications where the customers already have user accounts for corresponding web applications.

You can set up SSO to work in either of the following ways:

- **Third-Party SAML and JWT tokens**. The app obtains a token from a trusted 3rd-party issuer, makes an API call to the MCS token exchange endpoint, and

receives back an MCS-issued token, which you include as a bearer token on each subsequent MCS API call.

- **Browser-Based SSO through MCS**. The app opens the MCS SSO URL in a browser and, after a series of redirects, displays the login screen of the remote identity provider. Once the user successfully enters their credentials, they receive an OAuth token, which you include as a bearer token on each subsequent MCS API call.

  In the case of JWT tokens, MCS uses the OpenID Connect discovery protocol.

Third-Party SAML and JWT Tokens

MCS supports the use of tokens from third-party providers in two cases:

- With zero footprint SSO, where no user accounts are stored in Oracle Cloud. Instead, all of the information for the user, including user roles, is derived from the third-party token. Such users are referred to as *virtual users*.
- With a token that identifies a user that has been provisioned in both Oracle Cloud and the third-party IdP. Roles are assigned to the users in MCS.

SAML Tokens and Virtual Users

If you have users set up in a third-party IdP that supports the SAML 2.0 spec, you can authenticate those users in MCS via SAML tokens.

Here are the general steps to get this to work with virtual users (in other words, without having to also provision the users in Oracle Cloud):

1. You configure your mobile backend to use HTTP Basic authentication. (This is required for you to be able to get the token.)

   You do this by selecting the backend in MCS, selecting the backend's **Settings** page, and setting the switch for HTTP Basic Authentication to ON.

   > **Note:**
   >
   > To test authentication through the API Test page, you'll need to enable SSO for your mobile backend. You can check if your instance of MCS is configured for SSO from the Settings page of the mobile backend. Select the **Enable Single Sign-On** option if it's not selected. If you don't see the **Enable Single Sign-On** checkbox, you need to enable SSO for your Oracle Cloud account. See Configuring Identity Management (SSO and OAuth). After SSO is set up, you may need to log out and back into MCS for it to take effect.

2. Your administrator configures the IdP to generate a SAML token when the user logs in.
3. Your administrator registers the third-party token issuer and one or more token certificates in MCS.

   As part of this process, she can also associate MCS roles with tokens in one of the following ways.

   - By designating MCS roles to be associated with all tokens based on a given certificate.

- By deriving role names (that match existing MCS roles) from given token attributes.
- By mapping given token attribute values to existing MCS roles (where the attribute values don't already match the MCS names).

4. You code your app to do the following:
   a. Obtain a token from the third-party IdP upon user login.
   b. Send that token to an MCS token exchange endpoint to get an MCS-issued token in return.
   c. Use the MCS token for all subsequent API calls to MCS.

Configuring SAML Tokens for Virtual Users

To enable the authentication of virtual users via SAML tokens, you need to create a SAML app in your IdP. This is a special app that mediates the creating and passing of the SAML tokens.

Though the workflow varies by IdP, you generally need to do the following key tasks:

1. Create a SAML 2.0 app.
2. Configure the SAML 2.0 app by specifying the following:
   a. Redirect URL.

      You'll configure your app to use the redirect URL to obtain the token. How the token is obtained depends on the operating system you use (iOS or Android) . Avoid entering an address to an actual live site. Use a fictitious address URL request, for example,
      `http://hostname/mobile/platform/sso/redirect`

      Be sure the redirect URL you provide is formed correctly, that is it should match the expected redirect URL value.

   b. Audience.

      SAML tokens have the concept of an *audience*. An audience is the intended recipient of the SAML response (the token). It restricts the set of URLs against which the token can be used. You configure the audience to the URL for the MCS SSO token endpoint.

      You construct this endpoint by appending `/mobile/platform/sso/exchange-token` to your instance's base URL. You can determine the base URL by opening any mobile backend in MCS, clicking its **Settings** tab, and looking in the Environment URLs section.

   c. An assertion that lists the applicable roles for the user.

For concrete examples, see Use Case: Configuring OKTA to Obtain a SAML Token and Use Case: Configuring AD FS to Obtain a SAML Token.

Registering the Token Issuer in MCS

Before your apps can use tokens issued by a third-party IdP to authenticate with a backend, an administrator needs to register the IdP as a token issuer in MCS. Here are the steps:

1. In MCS, click ☰ and select **Administration** from the side menu.

2. Select an environment and click **Keys & Certificates**.
3. Click the **Web Service and Token Certificates** tab.
4. Click **Add** and provide the following information:
   - In the **Alias** field, enter a unique identifiable name for the certificate.
   - In the text field, paste the definition of the token certificate that was provided by the identity provider.
5. Click **Save**.
6. Wait for the token certificate to be propagated in the system. This should take no longer than 10 minutes.
7. Click the **Token Issuers** tab.
8. Click **New Issuer**.
9. Enter the name of the token issuer in the **Name** field under **Issuer Details**.
10. Next to the Certificate Subject Names panel, click **Add** (+) .
11. From the Select Certificate Subject Names dialog, select at least one name and click **Save**.

    Typically the name is the subject name of the token certificate you added previously.
12. Back on the Token Issuers tab, click **Rules**.
13. Select **Enable Virtual User**.
14. Optionally, create a **User Mapping** rule to designate the name of the token's attribute that identifies the user.

    See Configuring Rules for information on creating rules.
15. Optionally, designate user roles and mappings. The next topic has more information on how this works.
16. Click **Save and Close**.

Associating Roles with a SAML Token

If you want to set up role-based access for users that authenticate with SAML tokens, you do so when registering the token issuer in MCS. You have the following possibilities:

- Use roles already defined in the token that match the names of MCS roles.

  You do this by creating a **Role Attribute** rule and providing a comma-separated list of token attribute names. The roles are then derived from the values of these attributes.
- If the role names defined in the token don't match role names defined in MCS, provide a mapping between the two.

  You do this by:

  1. Creating a **Role Attribute** rule and providing a comma-separated list of token attributes that contain the role names.
  2. Creating a **Role Mapping** rule to create a mapping between a role derived from the token (via the role attribute rule) with one or more MCS user roles.

You can create multiple mappings.

- Apply one or more MCS roles to all tokens issued with a given certificate (unless roles were already applied via the role attribute or role mapping rules).

  You do this by creating a **Default Role** rule.

See Configuring Rules for the steps to create rules.

Extracting the SAML Assertion

After you've obtained a SAML token from an IdP, you need to decode it to extract the SAML assertion from its response. You then GZIP compress that assertion and base64 encode it again before submitting it to the MCS token exchange to receive an MCS token.

One way to extract the assertion is to follow these steps:

1. Open a browser and enter the address for the identity provider:

   For example, if you configured a SAML token with AD FS: `https://` *domain_name*`/adfs/ls/idpinitiatedsignon`

   You're taken to the Test Local Federation page.
2. Enter the user name and password credentials for the user you created and click **Sign In**.
3. After the page refreshes, select the SAML app you created and click **Sign in** again.

   You are redirected to the endpoint URL and the SAML token is displayed in the browser URL field.
4. Copy the response beginning with `SAML Response=`.
5. Since you'll need to base64 decode and inflate the SAML response, go to a SAML decoder tool such as SAML Decoder at https://www.samltool.com/decode.php.
6. Go to the base64 Decode and Inflate page and paste the response into the **Decode and Inflate XML** field.
7. Click **DECODE AND INFLATE XML**.
8. Extract the SAML assertion from the XML field.
9. Gzip compress the extracted assertion.
10. Base64 encode the assertion.

Now you can call the token exchange, pass the assertion, and receive the MCS token.

Using a SAML Token to Authenticate with MCS

Once you have obtained a valid SAML token, you can use it to authenticate with MCS. You do so by passing the token to MCS's token exchange endpoint. In exchange, you get an OAuth token issued by MCS that can be used for subsequent API calls during the session.

MCS's client SDKs support authentication via the token exchange. Here is some sample code you can use with those SDKs.

Android

```
private AuthorizationAgent mAuthorization;
private MobileBackend mobileBackend;

try {
  mobileBackend = MobileBackendManager.getManager().getMobileBackend(this);
} catch (ServiceProxyException e) {
      e.printStackTrace();
}

mAuthorization = mobileBackend.getAuthorization(AuthType.TOKENAUTH);
```

iOS

```
-(void) authenticateSSOTokenExchange: (NSString*) token
                    storeAccessToken:(BOOL) storeToken
                     completionBlock: (OMCErrorCompletionBlock)
completionBlock;
```

Cordova and JavaScript

```
mcs.mobileBackend.setAuthenticationType(mcs.AUTHENTICATION_TYPES.token);
mcs.mobileBackend.authorization.authenticate(token).then(callback).catch(er
rorCallback);
```

Coding the SAML Token Exchange Manually

If you are not using a client SDK, you need to manually code your app to exchange that token for an MCS token, with which you then authenticate.

1. In the app's login sequence, call the MCS token exchange endpoint to exchange the third-party token for an MCS-issued OAuth token:
   - The token exchange request is a simple GET request with no parameters.
   - It must include an Authorization header of the form:

     `Authorization: Bearer` *`external-token`*
   - It must also include the `oracle-mobile-backend-id` header with the value of the Basic Auth mobile backend ID for the mobile backend that you're using.

   The token exchange endpoint is formed by starting with the base URL for your environment (which you can get from the Settings page of a mobile backend) and appending `/mobile/platform/sso/exchange-token`.
2. In all REST calls to MCS APIs, include the given token in the `Authorization` header.

   The header takes the form `Bearer` *`access-token`*.

   The *access-token* value includes the mobile backend ID from the original request so you don't have to include the ID in a separate header.

JWT Tokens and Virtual Users

If you have users set up in a third-party IdP that supports JWT, you can authenticate those users in MCS via JWT tokens.

Here are the general steps to get this to work with virtual users (in other words, without having to also provision the users in Oracle Cloud):

1. You configure your backend to use both HTTP Basic and OAuth Consumer authentication.

   You can do this by selecting the backend in MCS, selecting the backend's **Settings** page, and setting the switches for HTTP Basic and OAuth Consumer authentication to ON.
2. Your administrator configures the IdP to generate a JWT token when the user logs in.
3. Your administrator registers the third-party token issuer via a policy in MCS.

   As part of this process, she can also associate MCS roles with tokens in one of the following ways.

   - By designating MCS roles to be associated with all tokens based on a given certificate.
   - By deriving role names (that match existing MCS roles) from given token attributes.
   - By mapping given token attribute values to existing MCS roles (where the attribute values don't already match the MCS names).
4. You code your app to do the following:

   a. Obtain a token from the third-party IdP upon user login.

   b. Send that token to an MCS token exchange endpoint to get an MCS-issued token in return.

   c. Use the MCS token for all subsequent API calls to MCS.

> **Note:**
>
> This mode of integrating with an IdP is based on enhanced features that are specific to working with JWT tokens (such as JWKS support) and includes other features, such as the ability to configure allowed audience values and username attribute. You can also use the process that is used for integrating with SAML-based IdPs, though this provides you with less flexibility. See SAML Tokens and Virtual Users.

Registering a JWT Token Issuer in MCS

Before your apps can use JWT tokens issued by a third-party IdP to authenticate with a backend, an administrator needs to register the IdP as a token issuer in MCS. Here's how it works:

1. You create a configuration that holds information that is needed to integrate with the token issuer. This integration takes the form of a JSON object.

2. You flatten the configuration into a single line.
3. You insert the configuration as the value of the `Security_AuthTokenConfiguration` policy.

   See Modifying an Environment Policy.

The following several topics provide some examples of creating the configuration file for a token issuer.

Minimal IdP Configuration

Here is an example of a configuration file that covers a basic use case, where:

- The user name can be derived from the token's `sub` claim.
- The token issuer is configured so that you can use discovery to obtain the issuer's current keys and/or certificates.
- You are using MCS's virtual user (zero footprint) capability so that you don't need to have corresponding records for the user in Oracle Cloud.
- User roles are specified in a token attribute named `roles`.
- The token's audience (`aud`) claim is set to the JWT auth token endpoint for your MCS instance (*`MCS-BASE-URL`*`/mobile/platform/auth/token`) so there is no need to override the default audience validation behavior.

```
{
  "issuers": [
    {
      "issuerName": "TOKEN-ISSUER-URL",
      "jwks": {
        "discoveryUri": "TOKEN-ISSUER-URL/.well-known/openid-configuration"
      },
      "virtualUserEnabled": true,
      "roleAttributes": [
        "roles"
      ]
    }
  ]
}
```

IdP Configuration with Audience

Here is an example of a configuration file that covers a basic use case, where:

- The user name can be derived from the token's `sub` claim.
- The token issuer is configured so that you can use discovery to obtain the issuer's current keys and/or certificates.
- You are using MCS's virtual user (zero footprint) capability so that you don't need to have corresponding records for the user in Oracle Cloud.
- User roles are specified in a token attribute named `roles`.

- The token's audience (aud) claim is set to GUID-12345678-ABCD-EFAB-CDEF-123456789ABC (which is a value that does not match MCS's auth token endpoint).

```
{

  "issuers": [
    {
      "issuerName": "TOKEN-ISSUER-URL",
      "audience": [
        "GUID-12345678-ABCD-EFAB-CDEF-123456789ABC"
      ],
      "jwks": {
        "discoveryUri": "TOKEN-ISSUER-URL/.well-known/openid-configuration"
      },
      "virtualUserEnabled": true,
      "roleAttributes": [
        "roles"
      ]
    }
  ]
}
```

IdP Configuration with Audience and Username Attribute

Here is an example of a configuration file that covers a basic use case, where:

- The username is specified in the unique_name claim (rather than the sub claim).
- The token issuer is configured so that you can use discovery to obtain the issuer's current keys and/or certificates.
- You are using MCS's virtual user (zero footprint) capability so that you don't need to have corresponding records for the user in Oracle Cloud.
- User roles are specified in a token attribute named roles.
- The token's audience (aud) claim is set to GUID-12345678-ABCD-EFAB-CDEF-123456789ABC (which is a value that does not match MCS's auth token endpoint).

```
{

  "issuers": [
    {
      "issuerName": "BASE-TOKEN-ISSUER-URL",
      "usernameAttribute": "unique_name",
      "audience": [
        "GUID-12345678-ABCD-EFAB-CDEF-123456789ABC"
      ],
      "jwks": {
        "discoveryUri": "BASE-TOKEN-ISSUER-URL/.well-known/openid-
configuration"
      },
      "virtualUserEnabled": true,
      "roleAttributes": [
```

```
        "roles"
      ]
    }
  ]
}
```

Associating Roles with a JWT Token

If you want to set up role-based access for users that authenticate with JWT tokens, you do so when registering the token issuer in MCS via the `Security_AuthTokenConfiguration` policy. You have the following possibilities:

- Use roles already defined in the token that match the names of MCS roles.

  You do this by creating a `roleAttributes` array for the issuer and populate it with claims in the token that you want to derive roles from.

- If the role names defined in the token don't match role names defined in MCS, provide a mapping between the two.

  You do this by:

  1. Creating a `roleAttributes` array for the issuer and populate it with claims in the token that you want to derive roles from.
  2. Creating a `roleMappings` array rule to create a mapping between a role derived from the token (via the `roleAttributes` array) with one or more MCS user roles.

     You can create multiple mappings.

- Apply one or more MCS roles to all tokens issued with a given certificate (unless roles were already applied via `roleAttributes` or `roleMappings`).

  You do this by creating a `defaultRoles` array.

- Apply one or more MCS roles to all tokens issued with a given certificate (whether or not roles were already applied via `roleAttributes` or `roleMappings`).

  You do this by creating an `issuerRoles` array.

See JWT Configuration Reference for details on the syntax of the configuration file.

Converting a JSON Object to One Line

You might find it useful to have some tools to convert JSON objects from multi-line objects to single-line objects and vice versa. Here are some examples of Python commands that you can use for that purpose,

To output the JSON content in file `/scratch/jsmith/authTokenConfig.json` as a single line:

```
cat /scratch/jsmith/authTokenConfig.json | python -c 'import
json,sys;obj=json.load(sys.stdin);print json.dumps(obj);'
```

To output the JSON content in file `/scratch/jsmith/authTokenConfig.json` in “pretty print" form:

```
cat /scratch/jsmith/authTokenConfig.json | python -c 'import
json,sys;obj=json.load(sys.stdin);print json.dumps(obj, indent=4,
sort_keys=False);'
```

JWT Configuration Reference

Here are the fields that can be used in the JSON object that serves as the configuration for a JWT identity provider.

Root Fields

- `issuers` — Required. A JSON array of trusted issuers objects. Each trusted issuer is defined as a JSON object, with a combination of the following fields.
- `policyMinReloadInterval` — Optional. If a token exchange request is received, and the specified issuer is not found in the configuration cache, the configuration cache will automatically be reloaded from the stored policy in order to check for changes, unless the amount of time since the last configuration cache reload is less than the `policyMinReloadInterval`. The default value for this interval is 10 seconds. The `policyMinReloadInterval` configuration field can be used to override the default value with a specified integer value in seconds.
- `policyMaxReloadInterval` — Optional. If a token exchange request is received, if the elapsed time since the last time the configuration cache was reloaded is in excess of `policyMaxReloadInterval`, the configuration cache will automatically be reloaded from the stored policy in order to check for changes. The default value for this interval is 120 seconds. The `policyMaxReloadInterval` configuration field can be used to override the default value with a specified integer value in seconds.
- `certificatesMinReloadInterval` — Optional. If a token exchange request is received, and a required certificate is not found in the certificates cache, the certificates cache will automatically be reloaded from Oracle Keystore Service (KSS) in order to check for changes, unless the amount of time since the last certificates cache reload is less than the `certificatesMinReloadInterval`. The default value for this interval is 10 seconds. The `certificatesMinReloadInterval` configuration field can be used to override the default value with a specified integer value in seconds.
- `certificatesMaxReloadInterval` — Optional. If a token exchange request is received, if the elapsed time since the last time the certificates cache was reloaded is in excess of `certificatesMaxReloadInterval`, the certificates cache will automatically be reloaded from KSS in order to check for changes. The default value for this interval is 300 seconds. The `certificatesMaxReloadInterval` configuration field can be used to override the default value with a specified integer value in seconds.

Issuer Fields

- `issuerName` — Required. A JSON string which specifies the issuer name. This value must match the value of the `iss` claim in tokens from the associated token issuer.

- `enabled` — Optional. A JSON boolean which can be used to enable or disable the token issuer. If the token issuer is disabled, any attempt to exchange a token from that issuer will fail. The default value is `true`.
- `audience` — Optional. A JSON array of string values, specifying valid audience values for the external token. If the external token contains an `aud` claim and none of the associated values exactly matches one of the values in the specified list, then the external token will be treated as invalid.

  The default behavior if this field is not specified (or contains an empty list) is to compare the `aud` values in the external token to the following values:

  - *`base-URL`*
  - *`base-URL`*`/`
  - *`base-URL`*`/mobile`
  - *`base-URL`*`/mobile/`
  - *`base-URL`*`/mobile/platform`
  - *`base-URL`*`/mobile/platform/`
  - *`base-URL`*`/mobile/platform/auth`
  - *`base-URL`*`/mobile/platform/auth/`
  - *`base-URL`*`/mobile/platform/auth/token`
  - *`base-URL`*`/mobile/platform/auth/token/`

  If none of the `aud` values in the external token match any of the above values, the external token will be treated as invalid.
- `virtualUserEnabled` — Optional. If `true` the virtual user (zero footprint) feature is enabled for this issuer, meaning your users can authenticate with third-party tokens without having corresponding user accounts in Oracle Cloud. The default value is `false`.
- `usernameAttribute` — Optional. A JSON string specifying the name of a JWT token claim from which a username is extracted. If no value is provided, the value of the `sub` claim will be used as the username.
- `requireClientAuth` — Optional. A JSON boolean which can be used to configure whether client authentication is required for this token issuer.
  - If the value is `true`, full client authentication is required.
  - If the value is `false`, a token exchange request can contain a `client-id` value in the POST body, with no `client_secret` value provided. This is intended only for cases where devices are not able to protect the `client_secret`.

  The default value is `true`.
- `clientIdAttribute` — Optional. A JSON string specifying the name of a JWT token claim which contains the client ID of the OAuth client on the external token issuer which was used to obtain the external token. If a `clientIdAttribute` value is specified, the specified attribute is present in a token, and its value matches the username associated with the token, then the token exchange request will be rejected, because client tokens shouldn't be exchanged for MCS user tokens.

  If no `clientIdAttribute` value is provided, this check will not be performed.

- `tokenTimeoutSeconds` — Optional. A JSON integer specifying the token lifetime (i.e. from `iat` to `exp`) in seconds for MCS tokens issued in exchange for tokens from this issuer. If this field is not specified, the token lifetime will be governed by the `Security_TokenExchangeTimeoutSecs` policy. If the `Security_TokenExchangeTimeoutSecs` policy has not been defined, the default token lifetime is 28800 seconds (i.e. 8 hours).

  The token lifetime is also governed by the `tokenTimeoutPolicy`.

- `tokenTimeoutPolicy` — Optional. A JSON string specifying the policy used to control the token lifetime (i.e. from `iat` to `exp`) for MCS tokens issued in exchange for tokens from this issuer. Three policy values are supported:

  – `FromTimeoutSecs` — The token lifetime is governed by the `tokenTimeoutSeconds` value.

  – `FromExternalToken` — The MCS-issued token will expire at the same time the external token being exchanged will expire (i.e. `tokenTimeoutSeconds` is ignored).

  – `FromExternalTokenLimitedByTimeoutSecs` — The MCS-issued token will expire at the same time the external token being exchanged or after the token timeout value, whichever comes first.

  If this field is not specified, the token timeout policy lifetime will be governed by the `Security_TokenExchangeTimeoutPolicy` policy. If the `Security_TokenExchangeTimeoutPolicy` policy has not been defined, the default token timeout policy is `FromTimeoutSecs`.

- `jwks`— Optional. A JSON object which specifies the URI(s) and other configuration options associated with loading keys and/or certificates from the external token issuer on the fly.

  Use this object if you are using a discovery URI to load keys and/or certificates (and you are not using a `certificateSubjectNames` object).

  See jwks Fields for the options.

- `certificateSubjectNames` — Optional. A JSON array of strings containing a list of the certificate subject names of certificates that have been uploaded into MCS through the Administration tab's Keys and Certificates page. (See Configuring a Web Service or Token Certificate.)

  Use this object if you are *not* using a discovery URI to load keys and/or certificates (and therefore are not using a `jwks` object).

- `filters` — Optional. A JSON array of filter objects. Each filter is defined as a JSON object, with a combination of these fields:

  – `name` — Required. A JSON string specifying the name of an attribute or claim to which the filter will be applied.

  – `type` — Optional. A JSON string specifying whether the filter is an `include` filter or an `exclude` filter.

    An include filter is satisfied if the token contains a value which matches one or more of the specified filter values (i.e. presence of a "match" causes the filter to be satisfied). An exclude filter is satisfied if the token does not contains a value which matches any of the specified filter values (i.e. absence of a "match" causes the filter to be satisfied).

    The default value is `include`.

  - `values` — Required. A JSON array of string values which will be compared to the value of the attribute or claim in the external token as identified by the `name` field.

    Filter values may contain the * character as a wildcard for matching purposes.

  Each filter in the array must be satisfied in order for the external token to be considered valid.

  > **Note:**
  >
  > If a filter is specified incorrectly or incompletely (e.g. missing name, invalid type, missing or empty values array) the filter will always be considered to be not satisfied. The rationale is that the admin who configured the filter was trying to filter out something, and if we cannot figure out what that something is, it is better to err on the side of caution, and reject the external token.

- `allowedMbes` — Optional. A JSON array of JSON objects which identify mobile backends can be used with this token issuer.

  You can specify a mobile backend including the `name` and `version`, or by including just `clientId`.

  If this field isn't specified, the issuer can be used with any mobile backend.

  Here are the possible entries:

  - `name` — Optional. A JSON string specifying the name of a mobile backend. If you include this field, you must also include `version`.
  - `version` — Optional. A JSON string specifying the mobile backend version. If you include this field, you must also include `name`.
  - `clientId` — Optional. A JSON string specifying the OAuth client ID of a mobile backend.

- `userMappingAttribute` — Optional. A JSON string identifying the user attribute used to search for an Oracle Cloud user to be associated with the token exchange.

  This attribute is ignored if `virtualUserEnabled` is set to `true`.

  The string can have one of the following values:

  - `uid` — Search for an Oracle Cloud user whose username matches the username extracted from the external token.
  - `mail` — Search for an Oracle Cloud user whose email address matches the username extracted from the external token.

  The default value is `uid`.

> **Note:**
>
> If a `usernameAttribute` hasn't been configured, the username extracted from the external token will be the value of the `sub` claim. If a `usernameAttribute` has been configured, the username extracted from the external token will be the value of the whatever claim is identified by the `usernameAttribute` value.

- `defaultRoles` — Optional. A JSON array of strings, where each string is the name of an MCS role which should be granted to a virtual user in the case where no `roleAttributes` value has been configured or where a `roleAttributes` value is configured but the specified attributes are either absent from the external token or are empty.
- `issuerRoles` — Optional. A JSON array of strings, where each string is the name of an MCS role which should be always granted to a virtual user when a token from this external issuer is exchanged. The difference between default roles and issuer roles is that default roles are granted only when no roles have been found during processing of role attributes, while issuer roles are always granted.
- `roleAttributes` — Optional. A JSON array of strings where each string is the name of a token attribute (i.e. claim) which should be searched for role values. If a specified token attribute is not present in the external token, no roles will be added for that attribute. Otherwise, the token attribute value will be processed as follows:
  - If the token attribute value contains a JSON string, the string value will be granted as a role, subject to role mapping (see the`roleMappings` field).
  - If the token attribute value contains a JSON array of JSON string values, each of the string values will be granted as a role, subject to role mapping.

  If no `roleAttributes` array is provided, the external token will not be searched for roles, and the roles to be granted to the user will be based on `defaultRoles` and/or `issuerRoles` configuration, where provided.
- `roleMappings` — Optional. A JSON array of role mapping objects, each of which specifies a mapping from a token role value (i.e. a value obtained from `roleAttributes`) and one or more MCS roles. Use this field when the values derived from role attributes do not match MCS role names.

  Here are the fields for a role mapping object:
  - `tokenRole` — Required. A JSON string specifying a token role name.
  - `mappedRoles` — Required. A JSON array of string values. Each string value should match an MCS role name.

  .

jwks Fields

- `discoveryUri` — Optional. A JSON string specifying the URI from which the token issuer's discovery information can be loaded. The discovery information provided by the external token issuer must be in accordance with the following specification:

  http://openid.net/specs/openid-connect-discovery-1_0.html

  The discovery URI for a token issuer will typically be of the form *`base-url`*`/.well-known/openid-configuration`, but MCS does not require this to be the case.

If a `discoveryUri` is configured for a token issuer, the MCS token exchange service will make a GET request to that URL to obtain the discovery information as needed. Once the discovery information has been obtained, MCS will typically use the `jwks_uri` value specified in the discovery information to obtain the issuer's current keys and/or certificates.

If no `discoveryUri` is configured, then a `jwksUri` value must be configured.

- `jwksUri` — Optional. A JSON string specifying the URI from which the token issuer's JWKS information can be loaded. The information provided by the external token issuer must be in accordance with the following specification:

  https://tools.ietf.org/html/rfc7517

  If a `jwksUri` is configured for a token issuer, the MCS token exchange service will make a GET request to that URL to obtain the current keys and/or certificates for that issuer as needed.

  If both a `discoveryUri` and a `jwksUri` are specified in the configuration, the configured `jwksUri` value will be used, overriding the value in the issuer's discovery information.

- `allowHttp` — Optional. A JSON boolean indicating that HTTP `discoveryUri` and `jwksUri` values should be allowed.

  For security reasons, `discoveryUri` and `jwksUri` values for external token issuers in production should always use HTTPS URLs, so that the server providing the information can be verified using its SSL certificate. However, in certain non-production test scenarios, it may be helpful to allow HTTP URIs to be used.

  The default value is `false`.

- `minReloadInterval` — Optional. If a token exchange request is received, and the key and/or certificate needed to validate the external token cannot be found, MCS will automatically reload the discovery and JWKS information in order to check for changes (e.g. key rotation), unless the amount of time since the discovery/JWKS reload is less than this value (in seconds, expressed as an integer).

  The default value is 60.

- `maxReloadInterval` — Optional. If a token exchange request is received and if the elapsed time since the last time the discovery and JWKS information was reloaded is in excess of this value (in seconds, expressed as an integer), the discovery and JWKS information will automatically be reloaded from the external token issuer in order to check for changes.

  The default value is 28800 (i.e. 8 hours).

- `connectTimeout` — Optional. A JSON integer specifying the default connect timeout for discovery and/or JWKS requests. The default is 30 seconds.

- `readTimeout` — Optional. A JSON integer specifying the default read timeout for discovery and/or JWKS requests. The default is 60 seconds

- `tlsVersions` — Optional. A JSON array of string values, listing the SSL/TLS which will be allowed when connecting to the external token issuer for Discovery and/or JWKS requests. Valid version names are:

  - SSL
  - SSLv2
  - SSLv3

  - TLS
  - TLSv1
  - TLSv1.1
  - TLSv1.2

  The default value is `["TLSv1.1", "TLSv1.2"]`.

  > **Note:**
  >
  > Older SSL/TLS versions are considered insecure, and should be avoided.

- `authorizationHeader` — Optional. A JSON string specifying an Authorization header value which should be included in discovery and/or JWKS requests. In most cases, discovery and JWKS web pages are public and no authorization is required. This property is intended primarily for test purposes (e.g. when setting up a custom service to act as a discovery and/or JWKS endpoint).

Obtaining a JWT Token Using an Embedded Browser

If you use an embedded browser to obtain JWT tokens, you'll need to perform the following actions:

1. Create a delegate object (for iOS) or client (for Android) to intercept the web request that contains the token. The delegate (or client) implements a method that allows your app to preview any web requests. For iOS, create a `UIWebViewDelegate` object. For Android, create a `WebViewClient` object.
2. Register the delegate or client object with the embedded browser.
3. Modify the method to look for a redirect URL or a form post URL, depending on how the IdP is configured to deliver it.

   When the specified request is located, the method should extract the token from the query string (or post body) and indicate to the browser to stop the request and close or hide the browser.

For either iOS or Android, you'll need a web view class, a delegate (or client) class, and the delegate (or client) implementation method name.

For iOS, use the `UIWebView` object and the `UIWebViewDelegate` method:

```
#pragma mark - UIWebViewDelegate

- (BOOL)webView:(UIWebView *)webView shouldStartLoadWithRequest:
(NSURLRequest *)
request navigationType:(UIWebVeiwNavigationType)navigationType
```

For Android, use the `WebView` client and the `WebVewClient` method:

```
public class MainActivity extends Activity {
    private  Activity mCtx;
    private static  final  String TAG = "TokenExchange";
    private String remoteIDPURL = "https://hostname/mobile/platform/sso/
```

```
redirect/saml";
    private WebView myWebView = null;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.content_main);
        mCtx = MainActivity.this;
        myWebView = (WebView) findViewById(R.id.webview);
        initWebView();
    }
private class MyBrowser extends WebViewClient {
        @Override
        public void onReceivedSslError(WebView view, SslErrorHandler
handler,
SslError error){
            handler.proceed();
        }
@Override
        public void onPageStarted(WebView view, String url, Bitmap
favicon) {
            super.onPageStarted(view, url, favicon);
           if(url.contains("http://localhost:port")) {
               // get value of SAMLResponse form field
                myWebView.loadUrl("javascript:window.HtmlViewer.showHTML" +
"('<html>'+document.getElementsByName('SAMLResponse')[0].value+'</
html>');");
            }
        }
    }
class MyJavaScriptInterface
    {
        @JavascriptInterface
        @SuppressWarnings("unused")
        public void showHTML(String html){
            Log.i(TAG, "===== html is "+html);
            String samlToken = html.substring(html.indexOf("<html>") + 6,
html.indexOf("</html>"));
            Log.i(TAG, "SAML Token = " + samlToken);
            runOnUiThread(new Runnable() {
                @Override
                public void run() {
                    myWebView.stopLoading();
                    myWebView.setVisibility(View.INVISIBLE);
                    myWebView.destroy();
                    finish();
                }
            });
        }
    }
private void initWebView(){
        myWebView.setWebViewClient(new MyBrowser());
        myWebView.getSettings().setJavaScriptEnabled(true);
        myWebView.addJavascriptInterface(new MyJavaScriptInterface(),
"HtmlViewer");
        myWebView.getSettings().setLoadWithOverviewMode(true);
```

```
        myWebView.getSettings().setUseWideViewPort(false);
        myWebView.loadUrl(remoteIDPURL);
    }
private void showMessage(final String message){
        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                Toast.makeText(mCtx, message, Toast.LENGTH_LONG).show();
            }
        });
    }
}
```

When the app is launched, it's directed to the `remoteIDPURL` (the redirect URL). When you enter your login credentials, the page is redirected. The `onPageStarted` method intercepts the response and the `showHTML` method retrieves the token

Obtaining a JWT Token Using a System Browser

If you use a system browser to obtain the token, your app must relinquish control to the system browser app. When the login process is complete, you'll need to return control to your app. You can return control via a redirect to a custom app scheme for which your app has registered.

For either iOS or Android, you'll need to perform the following actions:

1. Register the custom scheme for your app as dictated by the operating system. The custom scheme URL tells the mobile OS that requests to the given scheme should be sent to your app.
2. Edit your app to handle the redirection. You'll need to implement a method to handle the incoming redirect, which contains the token.

Coding Your Android App to Obtain a JWT Token

For Android apps, you need to register a custom URL scheme and then code the app to handle requests associated with that scheme. You do this by editing the `AndroidManifest.xml` file:

```
<activity android:name=".MainActivity">
    <intent-filter>
       <action android:name="android.intent.action.VIEW"/>
       <category android:name="android.intent.category.DEFAULT"/>
       <category android:name="android.intent.category.BROWSABLE"/>
       <data android:scheme="http"
           android:host="mytest.com"
           android:pathPrefix="/"/>
    </intent-filter>
</activity>
```

The following example shows how to extract the token from the custom URL scheme in the Android activity class:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
```

```
    super.onCreate(savedInstanceState);
    setContentView(R.layout.content_main);
    Uri uri = getIntent().getData();
    if(uri != null) {
        String token = uri.getQueryParameter("token");
        Logger.debug(TAG, "token is : " + token);
    }
}
```

When you open the link to `mytest.com`, you'll have the option to open the link with the app. This will launch the Android activity from where the JWT token is retrieved.

Coding Your iOS App to Obtain a JWT Token

To obtain a third-party token via a system browser for an iOS app, you need to perform the following actions:

1. Declare a custom URL scheme by editing the app's `Info.plist` configuration file.

   The scheme tells the mobile operating system to route to your app the request that contains the token.

2. Edit your app to implement the method to handle requests associated with that scheme.

To register a custom URL scheme with your iOS app, you must include the `CFBundleURLTypes` in your app's `Info.plist` file. `CFBundleURLTypes` is an array of dictionaries. Each dictionary defines a URL scheme that the app supports. `CFBundleURLTypes` contains the following keys:

- `CFBundleURLName` - a string that contains the abstract name of the URL scheme. This name should be unique. To ensure the name is unique, specify it as a reverse DNS style of identifier, such as `com.company.myscheme`.

  This string is also used as a key in your app's `InfoPlist.strings` file. The value of the key is the human-readable scheme name.

- `CFBundleURLSchemes` - An array of string s that contain the URL scheme names. For example: `http`, `mailto`, `tel`, and `sms`.

> **Note:**
>
> If multiple third-party apps register to handle the same URL scheme, there's no way to determine which app is given the scheme.

Here's an example of how to implement support for the custom URL scheme:

```
<key>CFBundleURLTypes</key>
<array>
   <dict>
       <key>CFBundleURLName</key>
       <string>oracle.cloud.mobile.URLDemo</string>
       <key>CFBundleURLSchemes</key>
       <array>
           <string>urldemo</string>
       </array>
```

```
      <key>CFBundleTypeRole</key>
     <string>Viewer</string>
   </dict>
</array>
```

This stipulates that any URL specifying the scheme, `urlScheme`, is redirected to your app.

When the iOS system browser encounters a URL with this custom scheme, it launches your app, if necessary, and passes the URL to your app delegate. To handle incoming URLs, your app delegate must implement the `application:openURL:options:` method. For example:

```
- (BOOL)application:(UIApplication*)application
           openURL:(NSURL*)url
           options:
(NSDictionary<UIApplicationOpenURLOptionsKey,id>*)options
{
   NSLog(@"Open URL: %@", url.absoluteString);
   NSLog(@"Open URL options: %@", options);
   if ([url.scheme isEqualToString:@"urldemo"]) {
      [self viewController].incomingURL = url;
      return YES;
   }
   return NO;
}
```

This implementation parses the incoming URL and extracts a ‘token’ query argument and stores it in an instance variable for later use. The implementation assumes the token is passed via the URL’s query string. Your implementation might differ and the token could be stored somewhere else in the URL. After your app extracts the token from the URL, the token can be exchanged for an MCS-issued token.

If you’re not familiar with creating URL schemes or implementing them in your app, see Apple’s documentation, specifically Using URL Schemes to Communicate with Apps.

Using a JWT Token to Authenticate with MCS

Once you have obtained a valid JWT token, you can use it to authenticate with MCS. You do so by passing the token to MCS’s token exchange endpoint. In exchange, you get a token issued by MCS that can be used for subsequent API calls during the session.

MCS’s client SDKs support authentication via the token exchange. Here is some sample code you can use with those SDKs.

Android

```
private AuthorizationAgent mAuthorization;
private MobileBackend mobileBackend;

try {
  mobileBackend = MobileBackendManager.getManager().getMobileBackend(this);
```

```
} catch (ServiceProxyException e) {
      e.printStackTrace();
}

mAuthorization = mobileBackend.getAuthorization(AuthType.TOKENAUTH);
```

iOS

```
-(void) authenticateSSOTokenExchange: (NSString*) token
                    storeAccessToken:(BOOL) storeToken
                     completionBlock: (OMCErrorCompletionBlock)
completionBlock;
```

Cordova and JavaScript

```
mcs.mobileBackend.setAuthenticationType(mcs.AUTHENTICATION_TYPES.token);
mcs.mobileBackend.authorization.authenticate(token).then(callback).catch(er
rorCallback);
```

Coding the JWT Token Exchange Manually

Once your mobile administrator has registered an IdP as a token issuer in your environment and you have code in your app to acquire a 3rd-party token, you can use the MCS client SDK for your platform to handle the complete login sequence.

If you are not using a client SDK, you need to code your app to exchange that token for an MCS token, with which you then authenticate.

In the app's login sequence, you call the MCS token exchange endpoint to exchange the third-party token for an MCS-issued OAuth token.

The token exchange request is an HTTP POST request, with an `application/x/www-form-urlencoded` request body, to the token exchange URL: *`base-URL`*`/mobile/platform/auth/token`.

The token exchange request must provide:

- The external token (a.k.a. "user assertion") being exchanged in the form `assertion=`*`external-token`*.
- Client authentication for the MCS mobile backend for which a new token is being requested, to prove that it is a valid user of that mobile backend.

Client authentication can be provided in any of the following ways:

- Encode the `client_id` and `client_secret` in basic auth form in the Authorization header.

  In this case, the following headers are required:

  ```
  Content-Type: application/x/www-form-urlencoded
  Authorization: Bearer Base64(client_id:client_secret)
  ```

And the body of the POST must contain these values:

```
grant_type=urn:ietf:params:oauth:grant-type:jwt-bearer
assertion=external-token
```

- Encode the `client_id` and `client_secret` as `application/x/www-form-urlencoded` form values in the POST body.

  In this case, the following header is required:

  ```
  Content-Type: application/x/www-form-urlencoded
  ```

  And the body of the POST must contain these values:

  ```
  grant_type=urn:ietf:params:oauth:grant-type:jwt-bearer
  assertion=external-token
  client_id=client-id
  client_secret=client-secret
  ```

  If this option is used, the `client_secret` can be omitted if the `requireClientAuth` value in the configuration is set to `false` for the given issuer. This option is provided for clients that are unable to securely protect a client secret value. Even if the `client_secret` is omitted, the `client_id` value must still be provided, in order to identify the MCS mobile backend for which a token is being requested.

- Provide a valid client assertion as an `application/x/www-form-urlencoded` form value in the POST body.

  In this case, the following header is required:

  ```
  Content-Type: application/x/www-form-urlencoded
  ```

  And the body of the POST must contain these values, where *`client-token`* is client token obtained from Oracle Cloud for the OAuth client associated with the MCS mobile backend for which a user token is being requested.

  ```
  grant_type=urn:ietf:params:oauth:grant-type:jwt-bearer
  assertion=external-token
  client_assertion_type=urn:ietf:params:oauth:client-assertion-type:jwt-
  bearer
  client_assertion=client-token
  ```

If the token exchange is successful, the response will have a 200 status, and will include an `application/json` body similar to this:

```
{

"access_token":"123456789iJKV1QiLA0KICJhbGciOiJIUzI1NiJ9.abcdefiOiJqb2UiLA0
KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290Ijp0cnVlfQ
.dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWFOEjXk",
    "token_type":"Bearer",
    "id_token":null,
    "expires_in":28800 }
```

Mapping Users from a Third-Party IdP to Oracle Cloud Users

It is also possible to have enable authentication with 3rd-party tokens where there are matching records for the users in Oracle Cloud. This enables you to apply roles to users directly in MCS.

For this matching to work, the following conditions apply:

- The Oracle Cloud users have been assigned to the realm that your mobile backend uses.
- When registering the token issuer in MCS, your mobile administrator *didn't* select the **Enable Virtual User** option.
- In SAML tokens, the subject must identify the user's username as defined in Oracle Cloud.
- In JWT tokens, the `sub` or `prn` attributes must identify either the user's username or email address as defined in Oracle Cloud.

User roles can be applied in any of these ways:

- By assigning roles to individual users on the Applications > Mobile User Management page of MCS.
- By doing batch assignments of roles in the Oracle Cloud Infrastructure Classic Console. To do this, you need to have the identity domain administrator role for your account in Oracle Cloud.
- By having your administrator, in the process of registering the IdP as a token issuer in MCS, specify one or more mobile roles to give to users authenticated with this IdP (via the default role rule).
- By having your administrator, in the process of registering the IdP as a token issuer in MCS, create rules to map information extracted from the token (such as role names) to MCS mobile roles (via role attribute rules).

  If the role names defined in the IdP don't match the role names defined in MCS, your administrator can configure role apping rules to map the token role names to the MCS role names.

If you want to use this approach but don't yet have user accounts set up in Oracle Cloud, follow the instructions at Importing Groups of Mobile Users Into MCS Using Oracle Cloud.

Getting a Single Sign-On OAuth Token through a Browser

For an app to authenticate through a single sign-on identity provider, it first needs to get an SSO OAuth token. Using the MCS SDK for your platform simplifies this process. However, if you are making the REST calls directly from your app (or you are testing API calls using another tool, such as cURL or Postman), you need to get the token manually.

1. On the mobile backend's **Settings** page, gather the following information:
   - (OAuth Consumer) Client ID
   - Base URL
2. Form the SSO token endpoint by appending `/mobile/platform/sso/exchange-token` to the base URL.

3. Form a URL that combines the SSO token endpoint and a query parameter for the client ID. For example:

   ```
   <SSO_Token_Endpoint>?clientID=<client_ID>
   ```

4. Open a private or incognito browser window, paste the URL into the address bar, and press Enter.

   (You need to use an incognito or private window because cookies stored in your browser for whatever reason, such as from having logged in to MCS, will interfere with your SSO token request.)

5. In the page that appears, enter the SSO user name and password and press Enter.

6. Open a private or incognito browser window, paste the URL into the address bar, and press Enter.

   The browser window will then display your token.

   You can use this in any REST calls you make to APIs through that mobile backend.

> **Note:**
>
> If you want to obtain a new token, do it from a fresh incognito or private window. If you use the same window from which you previously obtained a token, the correct token might not be returned.

Enabling Browser-Based SSO through MCS

Setting up browser-based single sign-on (SSO) in MCS consists of steps both in MCS and in the Oracle Cloud Infrastructure Classic Console. To follow these steps, you need to have the identity domain administrator role for your Oracle Cloud account.

To set up SSO for a group of users, you need to:

1. Create a realm in MCS for those users by following the steps at Creating Realms.
2. Configure your Oracle Cloud identity domain to allow SSO.

   To do so, go to the **Users** section of the Oracle Cloud Infrastructure Classic Console. See Configuring Identity Management (SSO and OAuth). After SSO is set up, you may need to log out and back into MCS for it to take effect.

3. Create user accounts in Oracle Cloud for the app users and have them assigned to the realm that you have just set up.

   These accounts correspond with the user accounts in your identity provider but only contain limited information, such as user name and email address. The password is not stored in the Oracle Cloud user account.

   To get user accounts set up, follow the instructions at Importing Groups of Mobile Users Into MCS Using Oracle Cloud.

4. (Optional) Assign the roles to the users that they need to access the APIs. (This step assumes that the given APIs are role-based.)

This step is not a prerequisite for developing APIs and mobile backends that use SSO as the authentication method. However, you might find it convenient to assign roles to the mobile users as they are created, especially if your team has decided on the mobile user roles to create and what users and APIs to associate them with.

In addition, as an identity domain administrator, you can do batch assignments of roles in the Oracle Cloud Infrastructure Classic Console. Mobile app developers can use the Mobile User Management interface in MCS to assign roles, but only one at a time. This is useful for testing purposes, but might be cumbersome when setting up more than a handful of users.

> **Note:**
>
> If you're not sure whether your instance of MCS is already configured to allow SSO, you can quickly check by opening the **Settings** page of any mobile backend, enabling **OAuth Consumer** authentication, and looking for the **Enable SSO** checkbox under the OAuth settings. If SSO isn't configured for your identity domain, you'll see the message **SSO is not set up for your Oracle Cloud account**.

Enabling Single Sign-On for a Mobile Backend

1. Open the mobile backend and select the **Settings** page.
2. Under **Access Keys**, make sure **OAuth Consumer** is enabled.

   The **Enable Single Sign-On** checkbox appears.
3. Select **Enable Single Sign-On**.

> **Note:**
>
> If the **Enable Single Sign-On** checkbox does not appear, you need to enable SSO for your Oracle Cloud account. See Configuring Identity Management (SSO and OAuth).

After you enable single sign-on, an SSO token endpoint is displayed under **Environment URLs**. You use this token endpoint to obtain the SSO authentication token. When using the MCS SDK for a given mobile platform, you insert this token endpoint into the configuration file provided by the SDK and SDK code handles the obtaining of the token.

Getting an SSO Token Using Form Post Response Mode

If you want to use MCS's SSO login feature with browser-based apps, you use the form post response type to get the OAuth token from the SSO token relay and have it posted back to the app through a redirect URI.

So that you don't make yourself vulnerable to having OAuth tokens generated on your behalf and then sent to a URI out of your control, you also have to specify acceptable values for the redirect URI in the `Security_SsoRedirectWhitelist` environment policy.

To code the call to the SSO token relay:

1. On the mobile backend's **Settings** page, gather the following information:
   - (OAuth Consumer) Client ID
   - SSO token endpoint
2. In your code, form a URL that combines the SSO token endpoint, a query parameter for the client ID, and a parameter for the redirect URI. For example:

   ```
   <SSO_Token_Endpoint>?clientID=<client_ID>&redirect_uri=<Redirect_URI>
   ```

3. From your code, call that URL.

   When that URL is called, the app user is redirected to a login page where they can sign in.

To set the `Security_SsoRedirectWhitelist` environment policy, see Modifying an Environment Policy.

The value for the `Security_SsoRedirectWhitelist` environment policy is a comma-separated list of simple URL patterns. For example:

```
https://www.example.com, https://*.example2.com
```

The pattern `https://www.example.com` will match the URLs `https://www.example.com/path1`, `https://www.example.com/path1/path2`, and so on.

Similarly, the pattern `http://www.example.com/path1` will match URLs `http://www.example.com/path1`, `http://www.example.com/path1/path2`, `http://www.example.com/path1/path2/path3` and so on, but will not match URL `http://www.example.com/other-path`.

Here are some other rules for the environment policy value:

- You must include the port, unless you are using the default port for the URL scheme. For example, the pattern `http://www.example.com` matches the URL `http://www.example.com` or the URL `http://www.example.com:80`, but not `http://www.example.com:8080`.
- You can use an asterisk (*) as a wildcard character within a URL segment but it doesn't apply across dot (`.`), forward slash (`/`), or colon (`:`) characters.

  For example, `https://example*:8080` would match `https://example-source:8080`, but it wouldn't match `https://example.com:8080`. This restriction is designed to prevent matching unintended sites. (Imagine something like `http://example.imposter.com:8080` which you would not want your wildcard to match.)
- Simple path values don't require a wildcard. For example, if a redirect URI of `https://example.com/apps/customer` is passed to the mobile backend and compared to the white list value in the above policy example, it will be accepted.
- The protocol (`https://` in the above example) must be included.

Testing APIs in a Mobile Backend with SSO Login

Once you add an API to a mobile backend with SSO login enabled, you can use the API tester with SSO as the authentication method. This helps you ensure that the API

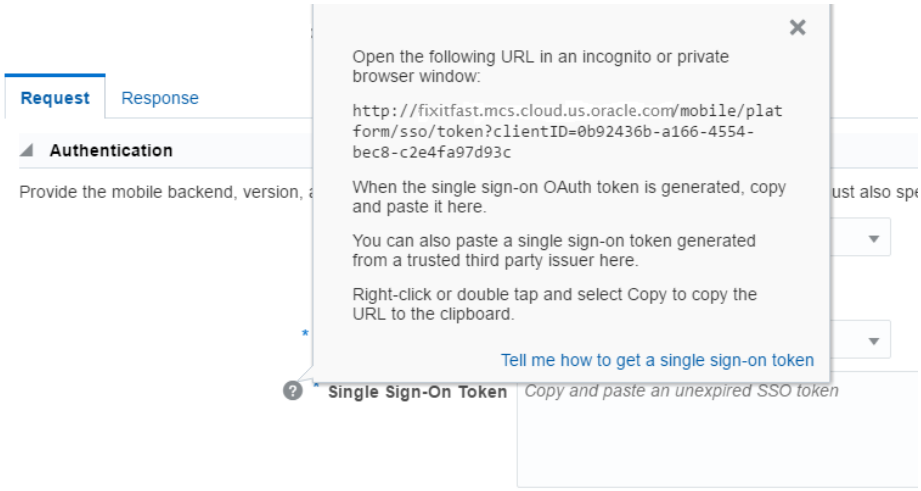
call works end to end. You can test with the MCS-issued SSO token or a token from a third-party provider.

To test a custom API with SSO login:

1. Click and select **Applications > Mobile Backends** from the side menu.
2. Select your mobile backend and click **Open**.
3. In the left navbar of the mobile backend, select **APIs**.
4. Click the API that you want to test.
5. If the user that you plan to authenticate in the test has not yet been assigned the role that is needed to access the API, click the **Security** navigation link and switch **Login Required** to OFF.
6. Click the **Endpoints** navigation link and scroll to the endpoint that you want to test.
7. From the **Authentication Method** dropdown, select **Single Sign-On Token** .
8. Obtain a valid SSO token for the mobile backend.

   If you are using web SSO, the fastest way to do this is to:

   a. Mouse over the info tip next to the **Single Sign-On Token** field, select the token endpoint URL that is in the info tip, and select **Copy** from your browser's menu (pressing Ctrl-C might not work).



   b. Open a private or incognito browser window, paste the URL into the address bar, and press Enter.

   c. In the page that appears, enter the SSO user name and password and press Enter.

      The token should appear in the page that is returned.

9. In the **Single Sign-On Token**, text field, paste the SSO token.

   If you have a token from your third-party provider, you can paste it in this field to authentication.

10. Click **Test Endpoint**.

    If successful, a test response will appear with an appropriate HTTP code, such as 200.

Token Expiration for SSO Login

When you use SSO as your login mode, the token expires after six hours by default, meaning that the app user will need to log in again after that time. The length of the timeout is governed by the `Security_TokenExchangeTimeoutSecs` policy, which is given in seconds. See Environment Policies for information on changing the policy.

Facebook Login in MCS

You can configure mobile backends to enable users to log in through Facebook. This mode of authentication is particularly useful for apps targeting consumers (as opposed to employees of your business).

When you enable users to log in to an app through Facebook, you can do the following things in the app:

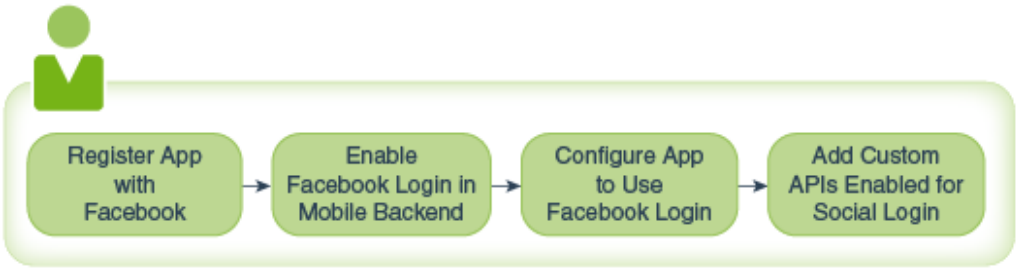
- Call any custom APIs that allow access with a social identity login.
- In the implementation code of such custom APIs, use the custom code SDK to call MCS platform APIs (with the exception of any APIs that are role-based).
- Register for notifications.

The main steps for setting up an app to use Facebook for login are:

1. Registering the app itself with Facebook.
2. Configuring Facebook login in the mobile backend that the app will be using.

> **Note:**
>
> This mobile backend can *only* be used for Facebook login. If you wish to have apps access the mobile backend using different authentication methods, you must create a separate mobile backend for that purpose.

3. Configuring the app itself to use Facebook for logging in.
4. In the mobile backend, adding custom APIs that allow access through Facebook login.



Registering an App for Login Through Facebook

Before you can enable login through Facebook, you need to register your app with Facebook using the Facebook SDK for your platform. From the registration process

Facebook will give you a Facebook app ID and secret which you will next configure in MCS.

For details, see Facebook's documentation at https://developers.facebook.com/docs/apps/register.

Enabling Facebook Login in a Mobile Backend

Once you have registered your app with Facebook, you can enable Facebook login in a mobile backend.

1. In MCS, open the mobile backend and select the **Settings** page.
2. Under **Social Login**, switch on **Facebook**.
3. In the **Facebook Settings** dialog, enter the app ID and app secret that you obtained when registering the app with Facebook.
4. On the same page, make sure that **HTTP Basic** authentication is enabled.

   (HTTP Basic authentication is needed for the first part of the authentication process when the app requests the Facebook access token.)

> **Note:**
>
> If you also want to make an app accessible through any other authentication method, create a separate mobile backend for which Facebook Login is not enabled. Then, in the configuration file provided by the MCS client SDK for the given platform (e.g. `OMC.plist` for iOS and `oracle_mobile_cloud_config.xml` for Android), add the details for that mobile backend. The app can then use both mobile backends, depending on how the user authenticates.

Configuring an App to Use Facebook Login

Once you have registered your app with Facebook and have configured a mobile backend to work with Facebook login, you can configure your app to log users in with their Facebook identities. You need to:

- Specify that Facebook is the identity provider.
- Provide the Facebook App ID.
- Provide the mobile backend ID and HTTP Basic anonymous key.

The easiest way to get this working is by using the MCS client SDK for the app's platform, which enables you to specify all of the credentials in a single configuration file. See The SDKs.

Adding APIs to a Mobile Backend with Facebook Login

You can add the following types of APIs to a mobile backend configured for Facebook login.

- Custom APIs that have the **Login Required** switch set to OFF.

- Custom APIs that have the **Login Required** switch set to ON *and* the **Social Login** switch set to ON.
- Any MCS platform APIs endpoints that allow anonymous access. The Analytics Collector, App Policies, Devices, MCS, and Location APIs all have endpoints that can be accessed anonymously. The Database Access API and Notifications API can be accessed from any custom API, including custom APIs that allow anonymous access.

To add an API to a mobile backend with Facebook login:

1. Make sure that the API allows social login. For custom APIs, you can check by following these steps:
   a. Click ☰ and select **Applications > APIs** from the side menu.
   b. Select the API that you want to add and click **Open**.
   c. In the API Designer, select the **Security** tab and check the settings.

> **Note:**
>
> APIs that you design for use with Facebook login can not be used with other authentication types. If you want an API's functionality to be available for apps with Facebook login and apps that are based on other types of authentication (such as OAuth, enterprise SSO, or HTTP Basic anonymous access), you need separate variants of the API, each with the appropriate security settings. For more information on API security, see Security in Custom APIs.

2. Add the API to the mobile backend:
   a. Click ☰ and select **Applications > Mobile Backends** from the side menu.
   b. Select your mobile backend and click **Open**.
   c. In the left navbar of the mobile backend, select **APIs**.
   d. Click **Select APIs**.
   e. Click the **+** (Add) icon for the API.

Getting a Facebook User Access Token Manually

For an app to authenticate through Facebook, it needs to get a user access token from Facebook. Using the MCS client SDK for your platform simplifies this process.

However, if you are testing an API with the API tester or another tool (such as cURL or Postman) or making the REST calls directly from your app, you need to get the user access token yourself. If you are the person who registered the app with Facebook, you can do this by following these steps:

1. Log into your Facebook account (the one with which you registered the mobile app).
2. Navigate to https://developers.facebook.com/tools/accesstoken/ and find your app.
3. Click the **You need to grant permissions to your app to get an access token** link to generate the token. A token is generated for you on the next page.

> **Note:**
>
> If you anticipate testing the app over a period of several weeks, you might find it convenient to extend the validity of your access token. You can do so by clicking **Extend Access Token**.

For more information, see Facebook's documentation on user access tokens at https://developers.facebook.com/docs/facebook-login/access-tokens#usertokens.

Headers Needed for API Calls with Facebook Authentication

When you call custom APIs from apps that use Facebook login, headers need to be passed to handle authentication. If you are using the SDKs for your platform, these headers are constructed for you based on values that you have entered into the SDK's configuration file.

If you are making REST calls to the APIs directly from your app (or from a separate tool, such as cURL), you need to add the following headers in your calls manually:

- `Authorization: Basic {anonymousKey}`
- `Oracle-Mobile-Backend-ID: {mobileBackendID}`
- `Oracle-Mobile-Social-Identity-Provider : facebook`
- `Oracle-Mobile-Social-Access-Token : {YOUR_FACEBOOK_USER_ACCESS_TOKEN}`

Authenticating in Direct REST Calls

When your app uses the MCS client SDK, you store the authentication credentials in one place so that you don't need to manually insert them into each call. In addition, the SDK handles the encoding of the username and password. However, if you are making the REST calls directly from your app (or you are testing API calls using another tool, such as cURL or Postman), you need to handle the authentication in each call. The value you send in the `Authorization` header depends on the type of authentication.

Authenticating with OAuth in Direct REST Calls

When you have OAuth enabled as an authentication mechanism for a mobile backend, an app can authenticate itself by sending the mobile backend's OAuth credentials (client ID and client secret) plus a user name and password to get an OAuth access token. If the API that is being called does not require a logged-in user, then the user name and password are not needed. The app then uses the OAuth token to make REST calls to APIs in the mobile backend.

You need the following information from the Settings page for the mobile backend:

- OAuth token endpoint
- Client ID
- Client secret

If the API is configured to require login, you also need the user name and password for a mobile user.

To construct a REST call to authenticate via OAuth:

1. Send the request to retrieve an access token:
   a. Base64 encode the `clientID:clientSecret` string.
   b. Set the `Authorization` header to `Basic` *client id:client secret-Base64-encoded-string*.
   c. Set the `Content-Type` to `application/x-www-form-urlencoded; charset=utf-8`.
   d. Set the request body to the appropriate grant type:
      - For access without a logged-in user, use: `grant_type=client_credentials`
      - For access with a logged-in user, use: `grant_type=password&username=username&password=password`. The user name and password must be URL encoded.
   e. `POST` the request to the OAuth token endpoint. For example, in cURL:

```
curl -i
-H "Authorization: Basic clientId:clientSecret-encoded-string"
-H "Content-Type: application/x-www-form-urlencoded; charset=utf-8"
-d "grant_type=client_credentials"
--request POST oauthTokenEndpoint
```

2. In the response, find the `access_token` property, as shown below (the value is truncated in this example).

```
{"oracle_client_assertion_type":"urn:ietf:params:oauth:client-assertion-
type:jwt-bearer",
"expires_in":604800,
"token_type":"Bearer",
"oracle_tk_context":"client_assertion",
"access_token":"eyJhbGciOiJ...FIqFiA"}
```

3. Copy the `access_token` property's value into the value of the `Authorization` header.

   The header takes the form `Bearer access_token`.

Authenticating with HTTP Basic in Direct REST Calls

When you have HTTP Basic enabled as an authentication mechanism for a mobile backend, an app can authenticate itself by sending the mobile backend ID, a user name, and a password. You pass the username and password as a Base64–encoded string. If the API that is being called is set to allow anonymous access, then you pass an anonymous access key instead of a user name and password.

Remember, if your app uses the MCS client SDK, the authentication credentials are stored in one place so you don't need to manually insert them.

To authenticate with MCS using HTTP Basic, you send a method to any platform endpoint with these headers:

- `Oracle-Mobile-Backend-ID`: The mobile backend ID is listed on the Settings tab for the mobile backend.
- `Authorization: Basic`: For basic authentication this header should include the mobile user's name and password encoded in Base64 or the anonymous key. If the anonymous key is available, it will also be displayed on the Settings tab for the mobile backend.

For example:

```
curl -X GET
     -H "Authorization: Basic {Base64 of
mobileUsername:mobileUserPassword} or {anonymousKey}"
     -H "Oracle-Mobile-Backend-ID: {mobileBackendID}"
                 {baseUri}/mobile/platform/users/~
```

For this call, the response would be one of the following:

- In the case of `200: Success`, the payload returned from MCS contains a JSON object with the user information.
- In case of an error, a JSON error message is returned.

For more information about Base64 encoding, see Base64 Decode and Encode.

How OAuth Works in MCS

This section provides some background on how MCS takes advantage of OAuth. You don't necessarily need to read this section to do your work, but you might find the conceptual background useful.

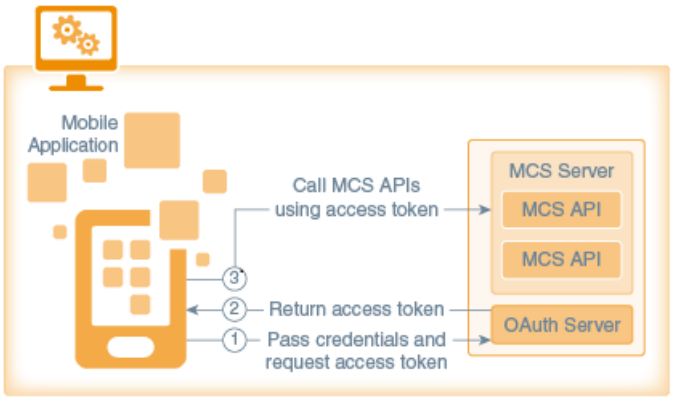
OAuth 2.0 is explicitly designed with REST in mind. It supports a variety of different client types that access REST APIs, including mobile apps. Secured APIs are made available only after a mobile app presents a valid OAuth access token.

Oracle Mobile Cloud Service's implementation of OAuth uses a model with the following roles:

- Resource Owner: The resource owner is responsible for entering credentials to grant authorizations to protected resources. The resource owner is often the app user.
- Mobile Application: The mobile app is the client that accesses protected resources and makes calls to secure APIs.
- MCS Server: The MCS server provides the interface for accessing the protected resources.
- OAuth Server: The OAuth server manages authorizations by the resource owner and issues access tokens. Typically, this role is also handled by the MCS server.

In OAuth 2.0, the client uses an access token issued by the OAuth server to access protected resources hosted by the MCS server.

1. The mobile app sends credentials to the OAuth server in an HTTP header.
2. The OAuth server returns an access token.
3. The mobile app uses the access token to access secure MCS APIs.



This enables MCS to manage permissions and grant applications access to services without requiring a separate login for each individual service. Credentials are issued for each mobile backend. Each mobile app registered with the mobile backend uses those credentials to authenticate with any API associated with that mobile backend.

Before a mobile application can access MCS APIs, it must first register with the MCS OAuth server. The registration is typically a one-time task and is done when the mobile backend is created. Once registered, the registration remains valid unless revoked. For details on registering a mobile app with a mobile backend, see Registering Applications.

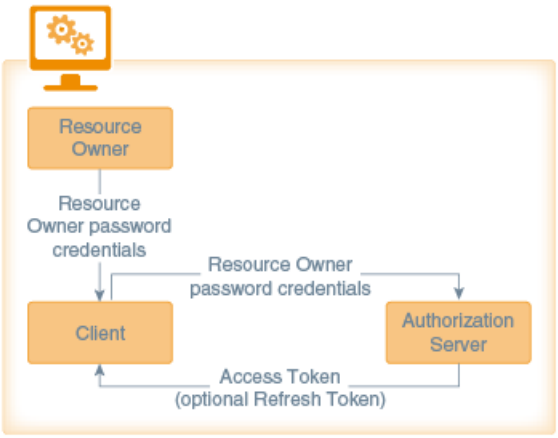
For every custom API in Mobile Cloud Service, the mobile developer decides whether or not authentication is required. This determines which OAuth flow is used.

- Resource Owner Password Credentials Grant - Authenticated Access
- Client Credentials Grant - Unauthenticated Access

Resource Owner Password Credentials Grant - Authenticated Access

The resource owner password credentials grant flow is suitable for highly trusted mobile applications because the client could abuse the credentials, or they could unintentionally be disclosed to an attacker. This grant type requires direct access to user credentials, but credentials are used for a single request and are exchanged for an access token. This grant type can eliminate the need for the mobile app to store user credentials for future use, by exchanging them for a long-lived access token or refresh token.

> **Note:**
>
> If you want to use a refresh token, you need to use a 3rd-party identity provider. MCS's OAuth server does not support refresh tokens.



The resource owner password credentials grant flow involves the following steps:

1. The mobile app prompts the user (resource owner) to enter a username and password.
2. The mobile app authenticates with the OAuth server through the token endpoint and requests an access token using the credentials entered by the user. The request contains the following parameters:
   - `grant_type` — Required. Must be set to `password`.
   - `username` — Required. The username of the resource owner (user).
   - `password` — Required. The password of the resource owner (user).
   - `scope` — Optional. The scope of the authorization.
3. The OAuth server validates the credentials and issues an access token.
   - `access_token`
   - `token_type`
   - `expires_in` (the number of seconds before the access token is no longer valid; expiration is optional)
4. The mobile app passes the access token to the MCS service, which validates the token and grants access.

For example, the mobile app makes the following HTTP request using transport-layer security:

```
POST /token HTTP/1.1
Host: server.example.com
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW
Content-Type: application/x-www-form-urlencoded
grant_type=password&username=johndoe&password=A3ddj3w
```

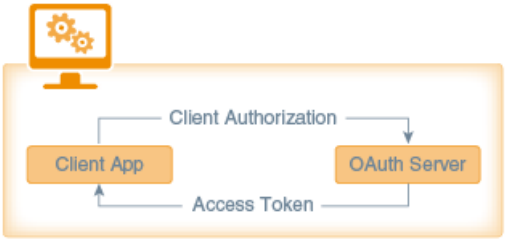
After the OAuth server accepts these values, it returns the following response with an access token:

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Cache-Control: no-store
Pragma: no-cache
{
```

```
"access_token":"2YotnFZFEjr1zCsicMWpAA",
"token_type":"example",
"expires_in":3600,

"example_parameter":"example_value"
}
```

Client Credentials Grant - Unauthenticated Access

The client credentials grant flow can be used when the authorization scope is limited to protected resources under the control of the mobile app. A registered trusted app is allowed to obtain an access token by providing only the client credentials to the OAuth server.



This flow is applicable in the following situations:

- The mobile app is requesting access to protected resources under its control. For example, unauthenticated access to APIs in the Mobile Backend, such as when a mobile banking app retrieves a list of ATMs based on location.
- The mobile app is requesting access to a protected resource where authorization has been previously arranged with the OAuth server.

The client credentials grant flow involves the following steps:

1. The mobile app authenticates with the OAuth server through the token endpoint and requests an access token. The request contains the following parameters:
   - `grant_type` — Required. Must be set to `client_credentials`.
   - `scope` — Optional. The scope of the authorization.
2. The OAuth server validates the credentials and issues an access token.

   The access token has the following parameters:
   - `access_token`
   - `token_type`
   - `expires_in` (the number of seconds before the access token is no longer valid; expiration is optional)
3. The mobile app passes the access token to the service. The service accepts the token and allows access.

For example, the mobile app makes the following HTTP request using transport-layer security

```
POST /token HTTP/1.1
Host: server.example.com
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW
```

```
Content-Type: application/x-www-form-urlencoded
grant_type=client_credentials
```

The OAuth server MUST authenticate the client. It returns the following response with an access token:

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Cache-Control: no-store
Pragma: no-cache
{
"access_token":"2YotnFZFEjr1zCsicMWpAA",
"token_type":"example",
"expires_in":3600,
"example_parameter":"example_value"
}
```

Securing Cross-Site Requests to MCS APIs

In addition to setting authentication methods, it's very important that you manage cross-origin resource sharing (CORS) for access to MCS APIs. You do so through the `Security_AllowOrigin` environment policy.

For browser-based applications, particularly those that use Single-Sign On (SSO) authentication, you should either not allow cross-site access at all or restrict access only to trusted origins where authorized applications are known to be hosted to mitigate vulnerability to Cross-Site Request Forgery (CSRF) attacks. If you're not using browser-based applications, it's best to use the default value, `disallow`, for `Security_AllowOrigin`.

Control cross-site access by setting the `Security_AllowOrigin` environment policy value to either `disallow` (the default value) or to a comma separated list of URL patterns, which specifies a whitelist of trusted URLs from which cross-site requests can be made. If the origin of a cross-site request matches at least one of the patterns in the whitelist, the request is allowed.

For example, the URL value for `Security_AllowOrigin` might look like this:

`https://myexample.com`, `https://*.example.com`, `https://*.example2.com`

When specifying a URL, note the following:

- You must include the port, unless you are using the default port for the URL scheme. For example, the pattern `http://www.example.com` matches the URL `http://www.example.com` or the URL `http://www.example.com:80`, but not `http://www.example.com:8080`.
- When specifying values for `Security_AllowOrigin`, don't include path parts and don't include a trailing forward slash, '/', character. For example, the pattern `http://www.example.com/` won't match `http://www.example.com`.
- You can use an asterisk (*) as a wildcard character within a URL segment but it doesn't apply across dot (.), forward slash (/), or colon (:) characters.

  For example, if the URL is `https://example.example.com:8080`, the following patterns match:

  - `https://*.example.com:8080`
  - `https://*.example.com:*`

- `https://ex*.example.com:*`

These patterns, however, won't match:

- `https://*.example.com*`
- `https://example*.oracle.com:*`

These restrictions are designed to prevent matching unintended sites.

> **Note:**
>
> For convenience, during the development of a browser-based application or during testing of a hybrid application running in the browser, you can set `Security_AllowOrigin` to `http://localhost:[port]`, but be sure to update the value in production.

6
Android Applications

If you are an Android app developer, you can use the SDK that Oracle Mobile Cloud Service (MCS) provides for Android. This SDK simplifies authentication with MCS and provides native wrapper classes for MCS platform APIs.

Getting the SDK for Android

To get the MCS client SDK for Android, go to the Oracle Technology Network's MCS download page.

To use the MCS SDK for Android, you should have the following software on your system:

- Android Studio, or the standalone Android SDK Tools from Google.

  See https://developer.android.com/studio/index.html for info on getting and using Android Studio.

- Java Development Kit (JDK) 1.7.0_67 or compatible.

  See http://www.oracle.com/technetwork/java/javase/downloads/index.html for JDK downloads.

Contents of the Android SDK

The following SDK libraries (JAR files) are included in the Android SDK:

- `mcs-android-sdk-shared-<version-number>.jar` - The base library for the SDK, including functionality required by the other libraries as well as utility classes for accessing and authenticating with mobile backends.
- `mcs-android-sdk-analytics-<version-number>.jar` - The Analytics library, which lets you insert custom events into your code that can then be collected and analyzed from the Analytics console.
- `mcs-android-sdk-location-<version-number>.jar` - The Location library, which lets you access details about location devices that have been registered in MCS and the places and assets they are associated with.
- `mcs-android-sdk-fcm-notifications-<version-number>.jar` - The Notifications library for FCM, which lets you set up your application to receive notifications sent from your mobile backend. If your app still uses GCM, the SDK also includes `/gcm/mcs-android-sdk-notifications-<version-number>.jar`. (The two notifications modules can't be used at the same time.)
- `mcs-android-sdk-social-<version-number>.jar` - The Social Login library, which allows you to set up your app to use Facebook login.
- `mcs-android-sdk-storage-<version-number>.jar` - The Storage library, which lets you write code to access storage collections that are set up with your mobile backend.

- `mcs-android-sdk-sync-<version-number>.jar` - The Sync Client library, which allows you to cache application data when the device running your app is disconnected from the network, then sync up the data when the network connection is reestablished.
- `IDMMobileSDK.jar` - The identity management library used by all applications.

The SDK also includes these tools and examples:

- `mcs-tools.zip` - The MCS Custom Code Test Tools, a set of command line tools for debugging custom APIs that you have associated with your app's mobile backend. Detailed instructions are located in the `README` file included in the zip.
- `mobile-log-download.zip` - A command-line tool that allows you to download logs from MCS for viewing or archiving.
- `oracle_mobile_cloud_config.xml` - A sample configuration file. You can adjust its properties based on the environment details of the mobile backend that your app will use and then copy the file to the `assets` folder you created when adding the SDK to your app.
- `examples.zip` - Sample mobile apps that demonstrate how to use the SDK.
- `Javadoc.zip` - Complete SDK API documentation. You can also reference the API documentation online: https://docs.oracle.com/en/cloud/paas/mobile-cloud/mcssa/index.html

Android SDK Dependencies

The SDK is modular, so you can package just the libraries that your app needs. Just be aware of the following dependencies:

- Every Android application developed for MCS must have the shared (`oracle-mobile_android_shared-<version-number>.jar`) and `IDMMobileSDK.jar` libraries.
- If the Storage library is installed, the Sync Client library *must* also be installed.

Adding the SDK to an Android App

1. If you haven't already done so, unzip the Android SDK zip.
2. Copy the SDK jars into the `/libs` folder in your app's project. If this folder doesn't exist, create it at the same level in your hierarchy as your `/src` and `/build` folders.
3. Decide which notifications library you need (FCM or GCM) and delete the .jar you are not using: `mcs-android-sdk-fcm-notifications-<version-number>.jar` or `/gcm/mcs-android-sdk-notifications-<version-number>.jar`. These modules can't be used at the same time.
4. In the source tree for the application, create a folder called `/assets` (at the same level as the `/java` and `/res` folders).
5. In the SDK bundle, locate the `oracle_mobile_cloud_config.xml` file and copy it to the `/assets` folder.
6. In your app's `build.gradle` file, make sure the following are among the dependencies registered so that the SDK libraries are available to the app.

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
```

```
    compile 'org.slf4j:slf4j-jdk14:1.7.13'
}
```

7. Open `/assets/oracle_mobile_cloud_config.xml` and fill in the environment details for the mobile backend that the app will be using. See Configuring SDK Properties for Android.

Upgrading an Android App from SDK 17.x and Before

1. Remove the following SDK jar files from the `libs` folder in your app's project (if they exist):
   - `IDMMobileSDK.jar`
   - `IDMMobileSDK.zip`
   - `mcs-android-sdk-vanalytics-<version>.jar`
   - `mcs-android-sdk-vIDMSDK-<version>.jar`
   - `mcs-android-sdk-vlocation-<version>.jar`
   - `mcs-android-sdk-vnotifications-<version>.jar`
   - `mcs-android-sdk-vshared-<version>.jar`
   - `mcs-android-sdk-vsocial-<version>.jar`
   - `mcs-android-sdk-vstorage-<version>.jar`
   - `mcs-android-sdk-vsync-<version>.jar`
2. Unzip the new MCS Android SDK zip if you haven't already.
3. Copy the new SDK jar files into the `libs` folder in your app's project.
4. Decide which notifications library you need (FCM or GCM) and delete the .jar you are not using: `mcs-android-sdk-fcm-notifications-<version-number>.jar` or `/gcm/mcs-android-sdk-notifications-<version-number>.jar`. These modules can't be used at the same time.
5. In your app's `settings.gradle` file, make sure that `IDMMobileSDK` is NOT an include. (Remove it if it is.)
6. In your app's `build.gradle` file, make sure the following is removed from the dependencies registered:

   ```
   compile project(':IDMMobileSDK')
   ```

7. In your app's `build.gradle` file, add the following to the dependencies registered:

   ```
   compile 'org.slf4j:slf4j-jdk14:1.7.13'
   ```

   So, the final dependencies should include:

   ```
   dependencies {
       compile fileTree(dir: 'libs', include: ['*.jar'])
       compile 'org.slf4j:slf4j-jdk14:1.7.13'
   }
   ```

Follow the rest of the instructions in this chapter to configure SDK properties and your Android manifest file.

Configuring SDK Properties for Android

To use the SDK in an Android app, you need to add the `oracle_mobile_cloud_config.xml` configuration file to the app and fill it in with environment details for your mobile backend. In turn, the SDK classes use the information provided in this file to access the mobile backend and construct HTTP headers for REST calls made to APIs.

You package the configuration file in your app's main bundle in the `assets` folder at the same level as the `java` and `res` folders. For example, in the demo application `FixItFast`, it's in `/app/src/main/assets`.

The following code sample shows the structure of a `oracle_mobile_cloud_config.xml` file.

```
<mobileBackends>
  <mobileBackend>
    <mbeName>MBE_NAME</mbeName>
    <mbeVersion>MBE_VERSION</mbeVersion>
    <default>true</default>
    <appKey>APPLICATION_KEY</appKey>
    <baseUrl>BASE_URL</baseUrl>
    <networkConnectionTimeOut>CONNECTION_TIMEOUT</networkConnectionTimeOut>
    <enableAnalytics>true</enableAnalytics>
    <enableLogger>true</enableLogger>
    <authorization>
      <offlineAuthenticationEnabled>true</offlineAuthenticationEnabled>
      <authenticationType>AUTH_TYPE</authenticationType>
      <oauth>
        <oAuthTokenEndPoint>OAUTH_URL</oAuthTokenEndPoint>
        <oAuthClientId>CLIENT_ID</oAuthClientId>
        <oAuthClientSecret>CLIENT_SECRET</oAuthClientSecret>
      </oauth>
      <basic>
        <mobileBackendID>MOBILE_BACKEND_ID</mobileBackendID>
        <anonymousKey>ANONYMOUS_KEY</anonymousKey>
      </basic>
    </authorization>
    <!-- additional properties go here -->
  </mobileBackend>
</mobileBackends>
```

Here's a list of the file's elements. The values that you need to fill in for a given mobile backend can be found on the Settings and Clients pages for that mobile backend.

- `mobileBackends` — The config file's root element, containing one or more `mobileBackend` elements.
- `mobileBackend` — The element for a mobile backend.
- `mbeName` — The name of the mobile backend associated with your app.
- `mbeVersion` — The version number of your app (for example, `1.0`).

- `default` — If `true`, that mobile backend is treated as the default and thus can be easily referenced using the `getDefaultMobileBackend(Context context)` method in the SDK's `MobileBackendManager` class.
- `appKey` — The application key, which is a unique string assigned to your app when you register it as a client in MCS. This key is only required if you are using notifications. See Registering an App as a Client in MCS.
- `baseUrl` — The URL your app uses to connect to its mobile backend.
- `networkConnectionTimeOut` — (Optional) The connection timeout value in seconds. The default is 60 seconds. This element was added in 17.4.5.
- `enableLogger` — When set to `true`, logging is included in your app.
- `enableAnalytics` — When set to `true`, analytics on the app's use can be collected.
- `authorization` — Use the sub-elements of this element to define the authentication the app will be using and specify the required credentials.
  - `offlineAuthenticationEnabled` — If set to `true`, offline login will be allowed. For this to work, you also need to add the following to the app's `AndroidManifest.xml` file:

    ```
    <receiver android:name="oracle.cloud.mobile.network.NetworkHelper"
      <intent-filter>
        <action android:name="android.net.conn.CONNECTIVITY_CHANGE" />
      </intent-filter>
    </receiver>
    ```

  - `authenticationType` — Define the kind of authentication mechanism being used to connect your app to MCS. Possible values are `oauth` (for OAuth Consumer) , `basic` (for HTTP Basic), `sso`, `tokenAuth` (for SSO token exchange), and `facebook` (for logging in with Facebook credentials). If this element isn't specified, OAuth Consumer is used. The other contents and sub-elements of the `authorization` element depend on the type of authentication.

OAuth Consumer

For OAuth, set the value of the `<authenticationType>` element to `oauth` and fill in the OAuth credentials provided by the mobile backend.

- `oAuthTokenEndPoint` — The URL of the OAuth server your app goes to, to get its authentication token.
- `oAuthClientId` — The unique client identifier assigned to all apps when they're first created in your mobile backend.
- `oAuthClientSecret` — The unique secret string assigned to all apps they're first created in your mobile backend.

The resulting `authorization` element might look something like this:

```
<authorization>
  <offlineAuthenticationEnabled>true</offlineAuthenticationEnabled>
  <authenticationType>oauth</authenticationType>
  <oauth>
    <oAuthTokenEndPoint>http://oam-server.oracle.com/oam/oauth2/tokens</
oAuthTokenEndPoint>
```

```
    <oAuthClientId>f2d3ca5c-7e6f-4d1c-aabc-a2f3caf7ec4e</oAuthClientId>
    <oAuthClientSecret>vZMRkgniIbhNUiPnSRT2</oAuthClientSecret>
  </oauth>
</authorization>
```

Enterprise SSO

For SSO, set the value of the `<authenticationType>` element to `sso`, fill in the OAuth credentials provided by the mobile backend, and add the `ssoTokenEndpoint`.

The resulting `authorization` element might look something like this:

```
<authorization>
  <offlineAuthenticationEnabled>true</offlineAuthenticationEnabled>
  <authenticationType>sso</authenticationType>
  <oauth>
    <oAuthTokenEndPoint>host/mobile/platform/sso/token</oAuthTokenEndPoint>
    <oAuthClient>f2d3ca5c-7e6f-4d1c-aabc-a2f3caf7ec4e</oAuthClient>
    <oAuthClientSecret>vZMRkgniIbhNUiPnSRT2</oAuthClientSecret>
    <ssoTokenEndpoint>https://development-
mcspmtrial90.mobileenv.oracle.com:443/mobile/platform/sso/token</
ssoTokenEndpoint>
</oauth>
</authorization>
```

SSO with a Third Party Token

For SSO with a third-party token, set the value of the `<authenticationType>` element to `tokenAuth`. You also need to fill in authentication credentials provided by the mobile backend, depending on how you have integrated the token issuer.

If you are using JWT tokens and have integrated the token issuer by registering a configuration via a policy in MCS, you need to nest the mobile backend's OAuth credentials. The resulting `authorization` element might look something like this:

```
<authorization>
  <offlineAuthenticationEnabled>true</offlineAuthenticationEnabled>
  <authenticationType>tokenAuth</authenticationType>
  <oauth>
    <oAuthTokenEndPoint>http://oam-server.oracle.com/oam/oauth2/tokens</
oAuthTokenEndPoint>
    <oAuthClientId>f2d3ca5c-7e6f-4d1c-aabc-a2f3caf7ec4e</oAuthClientId>
    <oAuthClientSecret>vZMRkgniIbhNUiPnSRT2</oAuthClientSecret>
  </oauth>
</authorization>
```

If you have integrated the IdP token issuer by uploading certificates into MCS, you need to nest the mobile backend's HTTP Basic credentials. The resulting `authorization` element might look something like this:

```
<authorization>
  <offlineAuthenticationEnabled>true</offlineAuthenticationEnabled>
  <authenticationType>tokenAuth</authenticationType>
  <basic>
```

```
    <mobileBackendID>6d3744b8-cab2-479c-998b-ebba2c31560f</mobileBackendID>
    <anonymousKey>UFJJTUVfREVDRVBUSUNPTl9NT0JJTEVfQU5PTll</anonymousKey>
  </basic>
</authorization>
```

HTTP Basic

For HTTP Basic authentication, you need to set the value of the `<authenticationType>` element to `basic` and fill in the HTTP Basic auth credentials provided by the mobile backend.

- `mobileBackendID` — The unique identifier assigned to a specific mobile backend. It gets passed in an HTTP header of every REST call made from your app to MCS, to connect it to the correct mobile backend. When calling platform APIs, the SDK handles the construction of the authentication headers for you.
- `anonymousKey` — A unique string that allows your app to access APIs that don't require login. In this scenario, the anonymous key is passed to MCS instead of an encoded user name and password combination.

The resulting `authorization` element might look something like this:

```
<authorization>
  <offlineAuthenticationEnabled>true</offlineAuthenticationEnabled>
  <authenticationType>basic</authenticationType>
  <basic>
    <mobileBackendID>6d3744b8-cab2-479c-998b-ebba2c31560f</mobileBackendID>
    <anonymousKey>UFJJTUVfREVDRVBUSUNPTl9NT0JJTEVfQU5PTll</anonymousKey>
  </basic>
</authorization>
```

Facebook

For Facebook login, you need to set the value of the `<authenticationType>` element to `facebook`, fill in the HTTP Basic auth credentials provided by the mobile backend, and add the `facebook` element, where you specify the Facebook credentials.

- `facebookAppId` — The Facebook application ID.
- `scopes` — You can use this element to specify Facebook permissions (optional).

The resulting `authorization` element might look something like this:

```
<authorization>
  <offlineAuthenticationEnabled>true</offlineAuthenticationEnabled>
  <authenticationType>facebook</authenticationType>
  <basic>
    <mobileBackendID>6d3744b8-cab2-479c-998b-ebba2c31560f</mobileBackendID>
    <anonymousKey>UFJJTUVfREVDRVBUSUNPTl9NT0JJTEVfQU5PTll</anonymousKey>
  </basic>
  <facebook>
    <facebookAppId>123456789012345</facebookAppId>
    <scopes>public_profile,user_friends,email,user_location,user_birthday</
scopes>
  </facebook>
</authorization>
```

Configuring Your Android Manifest File

Permissions for operations such as accessing the network and finding the network state are controlled through permission settings in your application's manifest file, `AndroidManifest.xml`. These permissions are required:

- `permission.INTERNET` — Allows your app to access open network sockets.
- `permission.ACCESS_NETWORK_STATE` — Allows your app to access information about networks.

Other permissions are optional. For example, the Analytics platform API uses location to provide detailed information about the usage and performance of your app. If you're including the Analytics library from the SDK, you'll want to add these permissions as well.

- `permission.ACCESS_COARSE_LOCATION`— Allows your app to access approximate location information, derived from sources such as wi-fi and cell tower positions.
- `permission.ACCESS_FINE_LOCATION` — Allows your app to access precise location information, derived from sources such as GPS.

For more information about permissions in your Android application, see Android Manifest Permissions in the Google documentation.

Add the permissions at the top of your `AndroidManifest.xml` file, as shown in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
package="oracle.cloud.mobile.photobox" >
  <uses-permission android:name="android.permission.INTERNET" />
  <uses-permission
android:name="android.permission.ACCESS_NETWORK_STATE" />
  <uses-permission
android:name="android.permission.ACCESS_FINE_LOCATION" />
  <uses-permission
android:name="android.permission.ACCESS_COARSE_LOCATION" />
</manifest>
```

> **Note:**
>
> Versions of the SDK before 17.4.5 used a NetworkHelper class that is no longer required. If your manifest file includes the following section, it can be deleted:
>
> ```
> <application>
>     <receiver
> android:name="oracle.cloud.mobile.network.NetworkHelper"
>       <intent-filter>
>         <action
> android:name="android.net.conn.CONNECTIVITY_CHANGE" />
>       </intent-filter>
>     </receiver>
>     (.....)
> </application>
> ```

Adding the SDK to your application may require you to configure your `AndroidManifest.xml` file to add new permissions or activities. For example, if you add the Notifications individual SDK library, you may also need to add a new broadcast receiver. For more information, see Setting Up a Mobile App for Notifications.

Loading a Mobile Backend's Configuration into an Android App

For any calls to MCS APIs using the Android SDK to successfully complete, you need to have the mobile backend's configuration loaded from the app's `oracle_mobile_cloud_config.xml` file. You do this using the `MobileBackendManager` and `MobileBackend` classes:

```
MobileBackendManager.getManager().getMobileBackend("My_Backend_Name")
```

Authenticating and Logging In Using the SDK for Android

Here is some sample code that you can use for authentication through MCS in your Android apps.

OAuth Consumer

First you initialize the authorization agent and set the authentication type to `OAUTH`.

```
private AuthorizationAgent mAuthorization;
private MobileBackend mobileBackend;
Context mCtx = getApplicationContext();
mobileBackend =
MobileBackendManager.getManager().getDefaultMobileBackend(mCtx);
mAuthorization = mobileBackend.getAuthorization(AuthType.OAUTH);
```

Then you use the `authenticate` method to attempt authentication. The call includes parameters for Android context, user name, password, and a callback that completes the authorization process.

```
TextView username, password;
username = (TextView) findViewById(R.id.username);
password = (TextView) findViewById(R.id.password);
String userName = username.getText().toString();
String passWord = password.getText().toString();
mAuthorization.authenticate(mCtx, userName, passWord, mLoginCallback);
```

Here's the definition for the callback.

```
AuthorizationCallback mLoginCallback = new AuthorizationCallback() {
   @Override
   public void onCompletion(ServiceProxyException exception) {
     Log.d(TAG, "OnCompletion Auth Callback");
      if (exception != null) {
        Log.e(TAG, "Exception while receiving the Access Token",
exception);
     } else {
        Log.e(TAG, "Authorization successful");
     }
   }
 }
```

Enterprise SSO

First you initialize the authorization agent and set the authentication type to `SSO`. (For SSO third-party token exchange, see the next example.)

```
private AuthorizationAgent mAuthorization;
private MobileBackend mobileBackend;
Context mCtx = getApplicationContext();
mobileBackend =
MobileBackendManager.getManager().getDefaultMobileBackend(mCtx);
mAuthorization = mobileBackend.getAuthorization(AuthType.SSO);
```

Then you create a thread to handle the authentication call and its callback.

```
private final Object lock = new Object();
new Thread(new Runnable() {
    @Override
    public void run() {
        mAuthorization.authenticateSSO(mCtx, cookies.isChecked(), new
AuthorizationCallback() {
            @Override
            public void onCompletion(ServiceProxyException exception) {
                if (exception != null)
                    Logger.debug(TAG, "Exception " +
exception.getMessage());
                else {
                    Logger.debug(TAG, "SSO Auth Succeeded");
```

```
                }
                synchronized (lock) {
                    lock.notifyAll();
                }
            }
        });

        synchronized (lock) {
            try {
                lock.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}).start();
```

SSO with a Third-Party Token

First, your app needs to get a token from the third-party token issuer. The way you can obtain the token varies by issuer. For detailed information on obtaining third-party tokens and configuring identity providers in MCS, see Third-Party SAML and JWT Tokens.

Once you have the token, initialize the authorization agent and use the token in your authorization call.

```
private AuthorizationAgent mAuthorization;
private MobileBackend mobileBackend;

Context mCtx = getApplicationContext();
mobileBackend =
MobileBackendManager.getManager().getDefaultMobileBackend(mCtx);
mAuthorization = mobileBackend.getAuthorization(AuthType.TOKENAUTH);
mAuthorization.authenticateUsingTokenExchange(mCtx, token, false,
mLoginCallback);
```

Here's the callback:

```
AuthorizationCallback mLoginCallback = new AuthorizationCallback() {
    @Override
    public void onCompletion(ServiceProxyException exception) {
        if (exception == null) {
            //log event with Analytics
            mAnalyticsAgent.logEvent("Login with 3rd party token
successfully");
            mAnalyticsAgent.flush();

            //redirect to another Activity after login
            Intent intent = new Intent(mCtx, ContentActivity.class);
            startActivity(intent);

        } else {
            Log.e(TAG, "Exception during token exchange:", exception);
```

```
            finish();
        }
    }
};
```

SSO with a Third-Party Token — Staying Logged In

You can also code the app to keep the user logged in, even when closing and restarting the app.

In the above example, the `authenticateUsingTokenExchange()` method is called with the third parameter (`storeToken`) set to `false`. If you set this parameter to `true` and the token exchange is successful, the MCS token is stored in a secure store and the user remains logged in until the token expires.

You can then use the `loadSSOTokenExchange` method on the `Authorization` object to load the stored token. If a token can't be retrieved from the secure store, the method returns `false`.

Here's some code that tries to load a saved token and, if it fails, restarts the authentication process:

```
try {
    mAuthorization =
MobileBackendManager.getManager().getDefaultMobileBackend(mCtx).getAuthoriz
ation();
    if (!mAuthorization.loadSSOTokenExchange(mCtx)) {
        //user not logged in, so need to initiate login
        mAuthorization.authenticateUsingTokenExchange(mCtx, token, true,
mLoginCallback);
    }
```

When you have the token stored in the secure store, it remains associated with the mobile backend that the app originally used. Therefore, if the app is updated to use a different mobile backend (or mobile backend version), you need to clear the saved token and re-authenticate.

```
mAuthorization.clearSSOTokenExchange(mCtx);
mAuthorization.authenticateUsingTokenExchange(mCtx, token, true,
mLoginCallback);
```

> **Note:**
>
> The default expiration time for a stored token that was obtained through token exchange is 6 hours. You can adjust this time by changing the `Security_TokenExchangeTimeoutSecs` policy.

HTTP Basic Authentication

The code for handling login with HTTP Basic is nearly the same as the code for OAuth.

First you initialize the authorization agent and set the authentication type to `BASIC_AUTH`.

```
private AuthorizationAgent mAuthorization;
private MobileBackend mobileBackend;
Context mCtx = getApplicationContext();
mobileBackend =
MobileBackendManager.getManager().getDefaultMobileBackend(mCtx);
mAuthorization = mobileBackend.getAuthorization(AuthType.BASIC_AUTH)
```

Then you use the `authenticate` method to attempt authentication. The call includes parameters for Android context, user name, password, and a callback that completes the authorization process.

```
TextView username, password;
username = (TextView) findViewById(R.id.username);
password = (TextView) findViewById(R.id.password);
String userName = username.getText().toString();
String passWord = password.getText().toString();
mAuthorization.authenticate(mCtx, userName, passWord, mLoginCallback);
```

Here's the definition for the callback.

```
AuthorizationCallback mLoginCallback = new AuthorizationCallback() {
   @Override
   public void onCompletion(ServiceProxyException exception) {
     Log.d(TAG, "OnCompletion Auth Callback");
     if (exception != null) {
       Log.e(TAG, "Exception while receiving the Access Token", exception);
     } else {
   Log.e(TAG, "Authorization successful");
   }
  }
 }
```

Facebook

For Facebook login, you use classes in the `oracle_mobile_android_social` library.

First you initialize the authorization agent and set the authentication type to `Facebook`.

```
private AuthorizationAgent mAuthorization;
private SocialMobileBackend socialMobileBackend;
Context mCtx = getApplicationContext();
socialMobileBackend =
SocialMobileBackendManager.getManager().getDefaultMobileBackend(mCtx);
mAuthorization = socialMobileBackend.getAuthorization(AuthType.Facebook);
mAuthorization.setAuthType(AuthType.Facebook);
```

Using a `CallbackManager` object from Facebook's SDK, initiate authentication.

```
private CallbackManager callbackManager;
mAuthorization.setup(getApplicationContext(), callback);
```

```
callbackManager = mAuthorization.getCallBackManager();
mAuthorization.authenticateSocial(mCtx);
```

Here's code you can use for the `callback` that is passed above.

```
private FacebookCallback<LoginResult> callback = new
FacebookCallback<LoginResult>() {
    @Override
    public void onSuccess(LoginResult loginResult) {
        Log.e(TAG, "facebook login successful.");
    }
    @Override
    public void onCancel() {
    }
    @Override
    public void onError(FacebookException e) {
    }
};
```

Override the `onActivityResult()` method to use the callback.

```
@Override
public void onActivityResult(int requestCode, int resultCode, Intent data)
{
    super.onActivityResult(requestCode, resultCode, data);
    callbackManager.onActivityResult(requestCode, resultCode, data);
```

Calling Platform APIs Using the SDK for Android

Once the mobile backend's configuration info is loaded into the app, you can make calls to SDK classes.

The root class in the Android SDK is the `MobileBackendManager`. An instance of `MobileBackendManager` manages one or more`MobileBackend` objects. A `MobileBackend` object is used to manage connectivity, authentication, and other transactions between your application and its associated mobile backend, including calls to platform APIs and any custom APIs you have defined. In turn, a `MobileBackend` instance manages instances of `ServiceProxy`. These instances correspond to platform services in MCS (for example, Analytics, Notifications, Sync, and so on).

The `MobileBackend` instance retrieves the information it needs about each mobile backend (the mobile backend name, version, and ID, as well as authentication information) from the app's `oracle_mobile_cloud_config.xml` file.

Here's an example of how you would use these classes to make calls into the Analytics API to create a new analytics event. The `ServiceProxy` instance created here manages calls to the Analytics platform API, including the constructing of the HTTP headers with the mobile backend credentials necessary to access the API:

```
...
byte[] imageBytes = new byte[0];
try {
```

```
    Analytics analytics =
MobileBackendManager.getManager().getDefaultMobileBackend(this).getServiceP
roxy(Analytics.class);
    Event customEvent = new Event("App Submission", new Date(), null);
    customEvent.addProperty("Image Attached", new
Boolean(imageBytes.length > 0).toString());
    analytics.logEvent(customEvent);
    analytics.flush();
} catch(ServiceProxyException e) {int errorCode = e.getErrorCode();
...
}
```

Here's how you could upload an image using the Storage API:

```
try {
    Storage storage =
MobileBackendManager.getManager().getDefaultMobileBackend(this).getServiceP
roxy(Storage.class);
    StorageCollection imagesCollection =
storage.getStorageCollection("FIF_Images");
    StorageObject imageToUpload = new StorageObject(null, imageBytes,
"image/jpeg");
    StorageObject uploadedImage = imagesCollection.post(imageToUpload);
} catch(ServiceProxyException e) {int errorCode = e.getErrorCode();
            ...
}
```

And here's how you could retrieve an image using the Storage API:

```
try {
    Storage storage =
MobileBackendManager.getManager().getDefaultMobileBackend(this).getServiceP
roxy(Storage.class);
    StorageCollection imagesCollection =
storage.getStorageCollection("FIF_Images");
    StorageObject image = imagesCollection.get("3x4mp1e-st0r4g3-0bj3ct-
k3y");byte[] imageBytes = image.getPayloadBytes();
} catch(ServiceProxyException e) {int errorCode = e.getErrorCode();
 ...
}
```

For more information on the individual platform APIs, see Platform APIs.

Calling Custom APIs Using the SDK for Android

The SDK provides the `CustomHttpResponse` class, the `GenericCustomCodeClientCallBack` interface, and the `invokeCustomCodeJSONRequest` method in the authorization classes to simplify the calling of custom APIs in MCS. You can call a REST method (GET, PUT, POST, or DELETE) on an endpoint where the request payload is JSON or empty and the response payload is JSON or empty.

You use `GenericCustomCodeClientCallBack` to create a handler for the response (which is returned in the form of a `CustomHttpResponse` object.)

Then, to call the custom API, you call `invokeCustomCodeJSONRequest(GenericCustomCodeClientCallBack restClientCallback, JSONObject data, String functionName, RestClient.HttpMethod httpMethod)` on your `Authorization` object.

To make a call to a custom API endpoint, you could use something like this:

```
import org.json.JSONObject;
import oracle.cloud.mobile.customcode.CustomHttpResponse;
import oracle.cloud.mobile.customcode.GenericCustomCodeClientCallBack;
import oracle.cloud.mobile.mobilebackend.MobileBackendManager;
.......

final GenericCustomCodeClientCallBack genericCustomCodeClientCallBack =
new GenericCustomCodeClientCallBack() {
    @Override
    public void requestCompleted(CustomHttpResponse response, JSONObject
data, Exception e) {
        boolean getResponse = (response.getHttpStatus() >=200 &&
response.getHttpStatus() <300);

        // write any logic based on above response
    }
};
AuthorizationAgent authorization =
MobileBackendManager.getManager().getDefaultMobileBackend(mContext).getAuth
orization();

authorization.authenticate(mContext, "user1", "pass1", successCallback);

........
// after the user successfully authenticates, make a call to the custom
API endpoint
authorization.invokeCustomCodeJSONRequest(genericCustomCodeClientCallBack,
null, "TaskApi/tasks", RestClient.HttpMethod.GET);
```

Video: Configuring an Existing Android App to Work with Mobile Cloud

For a demonstration on how to configure an Android app to use mobile backends and call MCS platform APIs, see this video on YouTube channel for the Oracle Mobile Platform:


Video

7

iOS Applications

If you are an iOS app developer, you can use the client SDK that Oracle Mobile Cloud Service (MCS) provides for iOS. This SDK simplifies authentication with MCS and provides Objective-C wrapper classes for MCS platform APIs.

Getting the SDK for iOS

To get the MCS client SDK for iOS, go to the Oracle Technology Network's MCS download page.

Contents of the iOS SDK

The iOS SDK contains the following items:

- `Documentation` - Contains web-browser based documentation (`html.zip`) and a docset for browsing and accessing context-sensitive help from Xcode (`oracle.mobile.cloud.Mobile_Client_SDK.docset.zip`). To use `html.zip`, unzip the file and browse the main page from `index.html`. To use the docset, unzip the file into the usual location for Xcode docsets, typically something like `~/Library/Developer/Shared/Documentation/DocSets`, where `~` is your home directory.

  This folder also contains a sample copy of the `OMC.plist` file that you will need to add to your app and populate with the configuration details for your mobile backend.
- `release-iphoneos` - Release versions of the static libraries and header files. Also contains SyncStore initialization data. The static libraries are Universal (fat) binaries that contain `armv7*` code and support both the iPhone Simulator and real devices. The following static libraries are included:
  - `libOMCCore.a` - The Core static library file shared by all iOS applications. Contains the common libraries required by all other libraries.
  - `libOMCAnalytics.a` - The Analytics static library file, which allows you to insert events in your code that can then be collected and analyzed from the Analytics console.
  - `libOMCLocation.a` - The Location library, which lets you access details about location devices that have been registered in MCS and the places and assets they are associated with.
  - `libOMCNotifications.a` - The Notifications static library file, which allows you to set up your application to receive notifications sent from your mobile backend.
  - `libOMCStorage.a` - The Storage static library file, which allows you to write code to access storage collections that are set up with your mobile backend.
  - `libOMCSynchronization.a` - The Data Offline static library file, which allows you to cache application data when the device running your app is

disconnected from the network, then synchronize the data when the network connection is reestablished.

- `thirdParty` - The static library (`libIDMMobileSDK.a`), headers, and resource strings for the identity management (IDM) library.
- `mcs-tools.zip` - The MCS Custom Code Test Tools, a set of command line tools for debugging custom APIs that you have associated with your app's mobile backend. Detailed instructions are located in the `README` file included in the zip.

Prerequisites for Developing iOS Apps

Before you start developing your app, you need to do some basic setup, such as adding iOS SDK frameworks, modifying configuration settings, and other steps.

Here's what we assume:

- You're familiar with Xcode as your development environment. If you're just starting, see https://developer.apple.com/xcode/.
- You've already obtained the following things from Apple:
  - An Apple Developer account.
  - A unique secure certificate installed on your Mac or iPad (that is, on the machine where you'll be developing your app).
  - An Application ID, which is used as the bundle identifier for your application in Xcode.
  - A Provisioning Profile. If you intend to install the Notifications static library from the client SDK and receive notifications in your iOS app, your Provisioning Profile must be enabled for notifications.

    If you haven't done these things yet, see the iOS developer documentation at http://developer.apple.com.

> **Note:**
>
> You can also use the client SDK with Swift apps. See Writing Swift Applications Using the iOS SDK.

Adding the SDK to an iOS App

1. Unzip the download file, `oracle_mobile_ios_sdk-{n}.zip` (where {n} is the version number of the SDK) into some directory on your machine.
2. Drag and drop the contents of the zip to the Xcode project navigator.
   - Select **Copy items if needed**.
   - Select **Create Groups**.
   - Click **Finish**.

   Once the `.a` file for a specific library has been copied into your application's development tree in Xcode, the corresponding platform API is available to your app through SDK calls. At this point, all of the SDK's static libraries are available to

your app. However, you need to complete the next steps so that the Identity Management library works properly.

3. Select the target for your project, select the **Build Phases** tab, expand **Link Binary with Libraries**, click the **+** button, and add the following frameworks:
   - `SystemConfiguration.framework`
   - `Security.framework`
   - `CoreLocation.framework`
4. Add the `-ObjC` flag to the Other Linker Flags settings.
5. Expand the `Documentation` folder of the unpacked zip, copy the `OMC.plist` file, and place it in the root of your app's main application bundle.
6. Fill in your mobile backend environment details. See Configuring SDK Properties for iOS.
7. If you are using Xcode 7 or higher, you need to account for the Application Transport Security (ATS) policy, which enforces remote communications to be over HTTPS.

   For development purposes only, add the following key in app's `Info.plist` file to turn off the ATS policy for the app.

```
<key>NSAppTransportSecurity</key>
<dict>
 <key>NSAllowsArbitraryLoads</key>
 <true/>
</dict>
```

> **Note:**
>
> You shouldn't use this setting in production. To make sure you provide optimal security for your app, study Apple's documentation for NSAppTransportSecurity and follow Apple's recommendations for disabling ATS for specific domains and applying proper security reductions for those domains.

iOS SDK Interdependencies

The client SDK is modular, so you can package just the libraries that your app needs. Just be aware of the following dependencies:

- Every app must have the `libOMCCore.a` static library file.
- If your app uses `libOMCStorage.a`, you must also include `lilbOMCSynchronization.a`.
- If your app uses `lilbOMCSynchronization.a`, you must also include the `SyncStore.momd` folder, which contains initialization data.
- If your app uses `libOMCCxAEngagement.a`, you must also include `libOMCCxAAnalytics.a`.

Configuring SDK Properties for iOS

To use the SDK in an iOS app, you need to add the `OMC.plist` configuration file to the app and fill it in with environment details for your mobile backend. In turn, the SDK classes use this information to access the mobile backend and construct HTTP headers for REST calls made to APIs.

You package the configuration file in the root of your app's main bundle.

Here's an example of the contents of the `OMC.plist` file. Pay careful attention to the hierarchy of elements.

Key	Type	Value
▼ Root	Dictionary	(2 items)
▼ mobileBackends	Dictionary	(1 item)
▼ FixItFast_Customer	Dictionary	(4 items)
appKey	String	ebfbc8ea-9173-442b-8a5e-2fae63c64422
▼ authorization	Dictionary	(2 items)
authenticationType	String	oauth
▼ OAuth	Dictionary	(3 items)
tokenEndpoint	String	https://oam.oracle.com/oam/oauth2/tokens
clientID	String	ddb7ff5a-0d86-4b4a-8164-ddad03734249
clientSecret	String	pFmzazXzNTBNVDyraQs7
baseURL	String	https://fif.cloud.oracle.com
default	Boolean	YES
logLevel	String	debug

Here's the source code for the same example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/
DTDs/PropertyList-1.0.dtd"
<plist version="1.0">
<dict>
  <key>mobileBackends</key>
  <dict>
    <key>FixItFast_Customer</key>
    <dict>
      <key>default</key>
      <true/>
      <key>baseURL</key>
      <string>https://fif.cloud.oracle.com</string>
      <key>appKey</key>
      <string>ebfbc8ea-9173-442b-8a5e-2fae63c64422</string>
      <key>authorization</key>
      <dict>
        <key>authenticationType</key>
        <string>OAuth</string>
        <key>OAuth</key>
        <dict>
          <key>tokenEndpoint</key>
          <string>https://oam.oracle.com/oam/oauth2/tokens</string>
```

```
            <key>clientID</key>
            <string>ddb7ff5a-0d86-4b4a-8164-ddad03734249</string>
            <key>clientSecret</key>
            <string>pFmzazXzNTBNVDyraQs7</string>
          </dict>
        </dict>
      </dict>
    </dict>
    <key>logLevel</key>
    <string>debug</string>
</dict>
</plist>
```

Here are the key entries in the `OMC.plist` file. You can obtain the necessary environment details from the Settings and Clients pages of the mobile backend.

- `mobileBackends` — a dictionary entry containing a nested dictionary for your mobile backend such as `FixItFast_Customer`. (When you call `OMCMobileBackend` in an app, you need to supply the value of that entry as a parameter to `OMCMobileBackendManager`.) That entry, in turn, contains entries for `appKey`, `baseURL`, `authenticationType`, `mobileBackendID`, `anonymousKey`, and, optionally, `networkConnectionTimeout`. See the example below.
- `baseURL` — The URL your application uses to connect to its mobile backend.
- `appKey` — The application key, which is a unique string assigned to your app when you register it as a client in MCS. See Registering an App as a Client in MCS. If you have not registered the app as a client in MCS, assign a placeholder value for this entry.
- `networkConnectionTimeout` — (Optional) The network timeout for API calls, in seconds. Should you need to do any network performance tuning, you can add this property, though you should use it with care. Keep in mind that app responsiveness issues might be better addressed in the app design itself. The default timeout is 60 seconds.
- `logLevel` — Determines how much SDK logging is displayed in the app's console. The default value is `error`. Other possible values (in increasing level of detail) are `warning`, `info`, and `debug`. It is also possible to set the value to `none`.
- `logHTTPRequestBody` — When set to `true`, the SDK will also log the HTTP and HTTPS headers and body in the requests to MCS.
- `logHTTPResponseBody` — When set to `true`, the SDK will also log the HTTP and HTTPS headers and body in responses from MCS.
- `authorization` — Use this key to define the type of authentication the app will be using and specify the required credentials. The contents and sub-elements of the `authorization` key depend on the type of authentication.
  - `authenticationType` — Defines the type of authentication mechanism being used in your mobile application. Possible values are `OAuth` (for OAuth Consumer), `basic` (for HTTP Basic), `SSO`, `SSOTokenExchange` and `Facebook`. Include a dictionary for each supported authentication type with the required credentials as explained in the sections that follow.
  - `offlineAuthenticationEnabled` — If set to `true`, offline login will be allowed. Offline login is not supported for OAuth so this key will be ignored.

OAuth Consumer

For OAuth, set the value of the `authenticationType` property to `OAuth` and fill in the OAuth credentials provided by the mobile backend.

- `tokenEndpoint` — The URL of the OAuth server your application goes to, to get its authentication token.
- `clientID` — The unique client identifier assigned to all applications when they're first created in your mobile backend.
- `clientSecret` — The unique secret string assigned to all applications when they're first created in your mobile backend.

The resulting `authorization` property might look something like this:

```
<key>authorization</key>
<dict>
    <key>authenticationType</key>
    <string>oauth</string>
      <key>OAuth</key>
      <dict>
        <key>tokenEndpoint</key>
        <string>https://oam.oracle.com/oam/oauth2/tokens</string>
        <key>clientID</key>
        <string>ddb7ff5a-0d86-4b4a-8164-ddad03734249</string>
        <key>clientSecret</key>
        <string>pFmzazXzNTBNVDyraQs7</string>
      </dict>
 </dict>
```

SSO

For SSO, set the value of the `authenticationType` property to `SSO` and fill in the OAuth credentials provided by the mobile backend. (For `tokenEndpoint`, you use the mobile backend's OAuth token endpoint.)

The resulting `authorization` property might look something like this:

```
<key>authorization</key>
<dict>
    <key>authenticationType</key>
    <string>SSO</string>
        <key>SSO</key>
        <dict>
      <key>tokenEndpoint</key>
      <string>https://oam-server.oracle.com/oam/oauth2/tokens</string>
      <key>clientID</key>
      <string>ddb7ff5a-0d86-4b4a-8164-ddad03734249</string>
      <key>clientSecret</key>
      <string>pFmzazXzNTBNVDyraQs7</string>
    </dict>
</dict>
```

SSO with a Third-Party Token

For SSO with a third-party token, set `authenticationType` to `SSOTokenExchange` and fill in the appropriate credentials. You also need to fill in auth credentials provided by the mobile backend, depending on how you have integrated the token issuer.

If you are using JWT tokens and have integrated the token issuer by registering a configuration via a policy in MCS, you need to nest the mobile backend's HTTP Basic credentials and then include the mobile backend's OAuth credentials as a separate key. The resulting `authorization` property might look something like this:

```
<key>authorization</key>
<dict>
  <key>authenticationType</key>
  <string>SSOTokenExchange</string>
  <key>SSOTokenExchange</key>
  <dict>
    <key>mobileBackendID</key>
    <string>ddb7ff5a-0d86-4b4a-8164-ddad03734249</string>
    <key>anonymousKey</key>

<string>UFJJTUVfREVDRVBUSUNPTl9NT0JJTEVfQU5PTllNT1VTX0FQUElEOnZrZWJxUmwuamE
wbTdu</string>
  </dict>
  <key>OAuth</key>
  <dict>
    <key>tokenEndpoint</key>
    <string>https://p2mob1813rc1f.identity.dc1.c9dev2.oraclecorp.com/oam/
oauth2/tokens</string>
    <key>clientID</key>
    <string>c437c1ed-fef0-4e88-802c-b85525fa0d6d</string>
    <key>clientSecret</key>
    <string>MtHoeHcRrWlDLiKcHJC8</string>
    </dict>
</dict>
```

If you have integrated the IdP token issuer by uploading certificates into MCS, you need to nest the mobile backend's HTTP Basic credentials. The resulting `authorization` property might look something like this:

```
<key>authorization</key>
<dict>
    <key>authenticationType</key>
    <string>SSOTokenExchange</string>
    <key>SSOTokenExchange</key>
    <dict>
      <key>mobileBackendID</key>
      <string>ddb7ff5a-0d86-4b4a-8164-ddad03734249</string>
      <key>anonymousKey</key>

<string>UFJJTUVfREVDRVBUSUNPTl9NT0JJTEVfQU5PTllNT1VTX0FQUElEOnZrZWJxUmwuamE
wbTdu</string>
```

```
    </dict>
</dict>
```

HTTP Basic

For HTTP Basic authentication, set the value of the `authenticationType` property to `basic` and fill in the HTTP Basic credentials provided by the mobile backend.

- `mobileBackendID` — The unique identifier assigned to a specific mobile backend. It gets passed in an HTTP header in every REST call made from your application to MCS, to connect it to the correct mobile backend. When calling platform APIs, the SDK handles the construction of the `mobileBackendID` header for you.
- `anonymousKey` — When using HTTP Basic authentication, a unique string that allows your app to access APIs that don't require login. In this scenario, the anonymous key is passed to MCS instead of an encoded user name and password combination.

You can also enable offline login for Basic authentication by setting the `offlineAuthenticationEnabled` property to true.

The resulting `authorization` property might look something like this:

```
<key>authorization</key>
<dict>
    <key>authenticationType</key>
    <string>Basic</string>
    <key>offlineAuthenticationEnabled</key>
    <true/>
    <key>Basic</key>
    <dict>
      <key>anonymousKey</key>

<string>UFJJTUVfREVDRVBUSUNPTl9NT0JJTEVfQU5PTllNT1VTX0FQUElEOml6LmQxdTlCaWFrd2Nz</string>
      <key>mobileBackendID</key>
      <string>4fb9cabd-d0e2-40f8-87b5-d2d44cdd7c68</string>
    </dict>
</dict>
```

Loading a Mobile Backend's Configuration into an iOS App

For any calls to MCS APIs using the iOS SDK to successfully complete, you need to have the mobile backend's configuration loaded from the app's `OMC.plist` file. You do this using the `OMCMobileBackend` class:

```
/**
 * Returns the mobile backend named "FixItFast_Customer" that is
configured in the OMC.plist file
 */
- (OMCMobileBackend *) myMobileBackend{

    return [[OMCMobileBackendManager sharedManager]
mobileBackendForName:@"FixItFast_Customer"];
```

```
}
```

Authenticating and Logging In Using the SDK for iOS

Here is are some methods you can use for authentication through MCS in your iOS apps. All of code given uses the `OMCAuthorization.h` class and relies on the following imports:

```
#import "OMCCore/OMCAuthorization.h"
#import "OMCCore/OMCMobileBackend.h"
#import "OMCCore/OMCMobileBackendManager.h"
```

OAuth Consumer and HTTP Basic

You can use the following method to handle a user logging in with a user name and password.

```
- (void) authenticate:(NSString *)userName
             password:(NSString *)password
      completionBlock: (OMCAuthorizationAuthCompletionBlock)
completionBlock;
```

This method terminates the connection to MCS and clears the user name and password from the iOS keychain:

```
-(void) logout: (OMCAuthorizationLogoutCompletionBlock) completionBlock;
```

SSO

For apps that allow login through enterprise SSO, use:

```
-(void) authenticateSSO: (UIViewController*) presentingViewController
           clearCookies: (BOOL) clearCookies
        completionBlock:(OMCAuthorizationAuthCompletionBlock)
completionBlock;
```

SSO with a Third-Party Token

First, your app needs to get a token from the third-party token issuer. The way you can obtain the token varies by issuer. For detailed information on obtaining third-party tokens and configuring identity providers in MCS, see Third-Party SAML and JWT Tokens.

Once you have the token, initialize the authorization agent and use the token in your authorization call.

```
-(void) authenticateSSOTokenExchange:(NSString*) token
                    storeAccessToken:(BOOL) storeToken
                     completionBlock:(OMCAuthorizationAuthCompletionBlock)
completionBlock;
```

```
-(NSError*) authenticateSSOTokenExchange:(NSString*) token
                        storeAccessToken:(BOOL) storeToken;
```

SSO with a Third-Party Token — Staying Logged In

You can also code the app to keep the user logged in, even when closing and restarting the app.

In the `authenticateSSOTokenExchange` method, if `storeAccessToken` is set to `YES`, the token is stored in secure store and the user remains logged in until the token expires.

You can use the `loadSSOTokenExchange()` method in the app launch sequence to load the token from the keychain. (If a token can't be retrieved, the method returns `NO`).

Here's some code that tries to load a saved token and, if it fails, restarts the authentication process:

```
OMCAuthorization* auth;
if ( [auth loadSSOTokenExchange] ){
    NSLog(@"## Token already found, login skipped.");
    ...
}
else{
    [auth authenticateSSOTokenExchange:thirdPartyToken
                  storeAccessToken:YES
                   completionBlock:^(NSError * _Nullable error) {

                       if( error ){
                           //Show error popup
                       }
                       else{
                            // Login success.
                            ...
                       }
                   }];
}
```

When you have the token stored in the secure store, it remains associated with the mobile backend that the app originally used. Therefore, if the app is updated to use a different mobile backend (or mobile backend version), you need to clear the saved token (using `clearSSOTokenExchange`) and re-authenticate.

> **Note:**
>
> The default expiration time for a stored token that was obtained through token exchange is 6 hours. You can adjust this time by changing the `Security_TokenExchangeTimeoutSecs` policy.

Calling Platform APIs Using the SDK for iOS

Once the mobile backend's configuration info is loaded into the app, you can make calls to SDK classes based on the iOS Core library classes.

The iOS Core library (`libOMCCore.a`) provides three public interfaces that are common across all other iOS libraries:

- `OMCMobileBackendManager`
- `OMCMobileBackend`
- `OMCServiceProxy`

The root class in the SDK is the `OMCMobileBackendManager`. An instance of `OMCMobileBackendManager` manages one or more `OMCMobileBackend` objects. An `OMCMobileBackend` object is used to manage connectivity, authentication, and other transactions between your application and its associated mobile backend, including calls to platform APIs and any custom APIs you have defined. In turn, an `OMCMobileBackend` instance manages instances of `OMCServiceProxy`. These instances correspond to platform services in MCS (for example, Analytics, Notifications, and so on).

It retrieves the information it needs about each mobile backend (the mobile backend name and ID, as well as authentication information) from the app's `OMC.plist` file.

Here's an example of using these classes to call APIs.

```
//Get mobile backend, here "FixItFast_Customer" is your backend name
from the OMC.plist configuration.
OMCMobileBackend* mbe = [[OMCMobileBackendManager sharedManager]
mobileBackendForName:@"FixItFast_Customer"];

//Authenticate with your credentials; if returns nil, then authenticated
successfully.
NSError* error = [mbe.authorization authenticate:@"username"
                                        password:@"password"];
//Get analytics client
OMCAnalytics* analytics = [mbe analytics];

//Get storage client
OMCStorage* storage = [mbe storage];

//Get notifications client
OMCNotifications* notifications = [mbe notifications];
```

To access the required headers to compile the preceding code, you need to import the following headers into your code:

```
#import "OMCMobileBackend.h"
#import "OMCMobileBackendManager.h"
#import "OMCAuthorization.h"
#import "OMCAnalytics.h"
#import "OMCMobileBackend+OMC_Analytics.h"
#import "OMCStorage.h"
```

```
#import "OMCMobileBackend+OMC_Storage.h"
#import "OMCNotifications.h"
#import "OMCMobileBackend+OMC_Notifications"
```

> **Note:**
>
> Methods written in Objective-C that are used in the MCS SDK for iOS can also be mapped to Swift. For more information, see Writing Swift Applications Using Mobile Client SDK.

Calling Custom APIs Using the SDK for iOS

The SDK provides the `OMCCustomCodeClient` class to simplify the calling of custom APIs in MCS. You can call a REST method (GET, PUT, POST, or DELETE) on an endpoint where the request payload is JSON or empty and the response payload is JSON or empty.

Using this class, you invoke a REST method (GET, PUT, POST, or DELETE) on an endpoint where the request payload is JSON or empty and the response payload is JSON or empty.

In addition you can provide a completion handler to be called when the method invocation is complete (meaning that the handler runs asynchronously).

If the completion handler is set, it will be invoked in the UI (main) thread upon completion of the method invocation, allowing update of UI items. The completion block will contain the format-specific data for a JSON object, namely an `NSDictionary` or `NSArray`. Use the completion block for any returned data or errors, HTTP or system.

All of the required MCS headers, such as `Authorization` (assuming the user has authenticated), will automatically be inserted into the request.

Use of `OMCCustomCodeClient` might look something like this:

```
#import "OMCCore/OMCMobileBackend.h"
#import "OMCCore/OMCCustomCodeClient.h"
...

// A GET, PUT, POST, or DELETE method may be specified here - sent or
returned JSON data object may be nil as appropriate.
OMCMobileBackend *backend = ...
OMCCustomCodeClient *ccClient = backend.customCodeClient;
NSDictionary *jsonPayload = @{@"myKey", @"myValue"};
[ccClient invokeCustomRequest: @"API2/endpoint2"
                       method: "@PUT"
                         data: jsonPayload,
                   completion: ^(NSError* error,
                                 NSHTTPURLResponse *response,
                                 id responseData) {
        // error will be nil if no problems occurred, otherwise it will
contain the error object
        // response will be complete HTTP response
        // response data will be Map or Array for JSON object if success
```

```
or nil if error
    }];
```

Video: Configuring an Existing iOS App to Work with Mobile Cloud

For a demonstration on how to configure an iOS app to use mobile backends and call MCS platform APIs, see this video on YouTube channel for the Oracle Mobile Platform:

Video

8
Cordova Applications

If you develop hybrid apps based on the Apache Cordova framework, you can use the client SDK that Oracle Mobile Cloud Service (MCS) provides for Cordova. This SDK simplifies authentication with MCS and provides Cordova wrapper classes for MCS platform APIs as well as libraries for Data Offline and Sync and Sync Express.

If you are new to Cordova itself and still need to set it up on your system, you can follow the Getting Started with JET Hybrid Apps tutorial for an end-to-end look at creating a Cordova app and connecting it with a mobile backend.

> **Note:**
>
> This SDK supports Cordova apps for the iOS and Android platforms. Apps for Microsoft Windows are not supported.

Getting the SDK for Cordova

To get the MCS client SDK for Cordova, go to the Oracle Technology Network's MCS download page.

Contents of the Cordova SDK Bundle

The Cordova SDK contains the following items:

- `jsdocs.zip` — The compiled documentation for the library.
- `mcs.js` — The uncompressed version of the SDK. This version contains code comments and is best used as you are developing and debugging your app.
- `mcs.sync.js` — The uncompressed version of the SDK Data Offline and Sync and Sync Express libraries.
- `mcs.min.js` — The compressed version of the SDK. Use this version when you deploy the completed app.
- `mcs.sync.min.js` — The compressed version of the SDK Data Offline and Sync and Sync Express libraries.
- `oracle_mobile_cloud_config.js` — An MCS configuration file, in which you can insert environment and authentication details for the mobile backends that your app will access.
- `oracle_mobile_js_sdk_cookies_cordova_plugin[VERSION].zip` — A Cordova plugin that's necessary if you are developing Cordova apps that authenticate with MCS via SSO.
- `\pako` — This folder includes the `pako` JavaScript library, which is required to use SSO with a third-party token.

- `mcs-tools.zip` - The MCS Custom Code Test Tools, a set of command line tools for debugging custom APIs that you have associated with your app's mobile backend. Detailed instructions are located in the `README` file included in the zip.

Adding the SDK to a Cordova App

1. If you haven't already done so, unzip the Cordova SDK zip.
2. Copy `mcs.min.js` and `oracle_mobile_cloud_config.js` into the directory where you keep your JavaScript libraries.
3. Fill in your mobile backend details in `oracle_mobile_cloud_config.js`. See Configuring SDK Properties for Cordova.
4. If you will be using SSO or Facebook authentication in your apps, add the Cordova `inappbrowser` to your project:

   ```
   cordova plugin add cordova-plugin-inappbrowser -save
   ```

5. If you will be using SSO in your apps, install the `oracle_mobile_js_sdk_cookies_cordova_plugin` plugin:
   a. Unzip `oracle_mobile_js_sdk_cookies_cordova_plugin[VERSION].zip`.
   b. At the command line, type:

      ```
      cordova plugin add <PLUGIN_FOLDER>
      ```

      where *<PLUGIN_FOLDER>* is the path to the unpacked plugin.
6. Load `mcs.min.js` in your app using RequireJS or a HTML script tag.

> **Note:**
>
> In addition to `mcs.min.js`, if your app uses Sync Express, `mcs.sync.min.js` must be fetched and executed as the first script in the main page of your app, before any other script, including RequireJS. For detailed instructions on adding Sync Express to your app, see Building Apps that Work Offline Using Sync Express.

Configuring SDK Properties for Cordova

To use the SDK in a Cordova app, add the `oracle_mobile_cloud_config.js` configuration file to the app and fill it in with environment details for your mobile backend. The SDK classes draw on this file for the details needed to access the mobile backend and use them to construct HTTP headers for REST calls made to APIs.

> **Note:**
>
> If any of your apps will be browser-based, you need to manage cross-origin resource sharing (CORS) for access to MCS APIs. See Securing Browser-Based Apps Against Cross-Site Request Forgery Attacks.

Package the configuration file in the same folder as the `mcs.min.js` file.

The following example shows the structure of a generic `oracle_mobile_cloud_config.js` file:

```
var mcs_config = {
  "logLevel": mcs.LOG_LEVEL.INFO,
  "logHTTP": true,
  "mobileBackends": {
    "YOUR_BACKEND_NAME": {
      "default": true,
      "baseUrl": "YOUR_BACKEND_BASE_URL",
      "applicationKey": "YOUR_BACKEND_APPLICATION_KEY",
      "authorization": {
        "basicAuth": {
          "backendId": "YOUR_BACKEND_ID",
          "anonymousToken": "YOUR_BACKEND_ANONYMOUS_TOKEN"
        },
        "oAuth": {
          "clientId": "YOUR_CLIENT_ID",
          "clientSecret": "YOUR_ClIENT_SECRET",
          "tokenEndpoint": "YOUR_TOKEN_ENDPOINT"
        },
        "facebookAuth":{
          "facebookAppId": "YOUR_FACEBOOK_APP_ID",
          "backendId": "YOUR_BACKEND_ID",
          "anonymousToken": "YOUR_BACKEND_ANONYMOUS_TOKEN"
        },
        "ssoAuth":{
          "clientId": "YOUR_CLIENT_ID",
          "clientSecret": "YOUR_CLIENT_SECRET",
          "tokenEndpoint": "YOUR_TOKEN_ENDPOINT"
        },
        "tokenAuth":{
          "backendId": "YOUR_BACKEND_ID"
        }
      }
    }
  },
  "syncExpress": {
    "handler": "OracleRestHandler",
    "policies": [
      {
        "path": '/mobile/custom/firstApi/tasks/:id(\\d+)?',
      },
      {
        "path": '/mobile/custom/secondApi/tasks/:id(\\d+)?',
```

```
      }
    ]
  }
};
```

Here's a list of the file's elements. The values that you need to fill in for a given mobile backend can be found on the Settings and Clients pages for that mobile backend. For details on sync elements, see Building Apps that Work Offline Using Sync Express.

- `logLevel` — Determines how much SDK logging is displayed in the app's console. The default value is `mcs.LOG_LEVEL.INFO` (where only important events are logged). Other possible values are `mcs.LOG_LEVEL.ERROR` (only errors are logged) and `mcs.LOG_LEVEL.VERBOSE`.
- `logHTTP` — When set to true, enables additional logging capability that includes the complete HTTP headers and body in requests and responses to MCS.
- `mobileBackends` — The config file's root element, containing a JSON object for each mobile backend.
- `baseUrl` — The URL your app uses to connect to its mobile backend.
- `applicationKey` — The application key, which is a unique string assigned to your app when you register it as a client in MCS. See Registering an App as a Client in MCS.
- `authorization` — JSON object containing the authentication details for connecting your app to MCS. In turn, it must contain one or more objects of type `basicAuth`, `oAuth`, `ssoAuth`, `tokenAuth` or `facebookAuth`. The contents of the object depend on the type of authentication.

OAuth Consumer

For OAuth, nest an `oAuth` object within the `authorization` object and fill in the OAuth credentials provided by the mobile backend.

- `clientID` — The unique client identifier assigned to all apps when they're first created in your mobile backend.
- `clientSecret` — The unique secret string assigned to all apps they're first created in your mobile backend.
- `tokenEndpoint` — The URL of the OAuth server your app goes to, to get its authentication token.

The resulting `authorization` property might look something like this:

```
"authorization": {
  "oAuth": {
    "clientID": "b20a34b4-e646-44dc-a787-3a8715f4bb46",
    "clientSecret": "chIkehuDPYsaosPEMyE2",
    "tokenEndpoint": "http://abc09xyz.oracle.com:14100/oam/oauth2/tokens",
  }
}
```

HTTP Basic

For HTTP Basic, nest a `basicAuth` object within the `authorization` object and fill in the HTTP Basic credentials provided by the mobile backend.

- `backendId` — The unique identifier assigned to a specific mobile backend.
- `anonymousToken` — A unique string that allows your app to access APIs that don't require login. In this scenario, the anonymous key is passed to MCS instead of an encoded user name and password combination.

The resulting `authorization` property might look something like this:

```
"authorization": {
  "basicAuth": {
    "backendId": "3b113ad5-07dc-4143-8b6a-a2ef62a175c1",
    "anonymousToken":
"UFJJTUVfREVDRVBUSUNPTl9NT0JJTEVfQU5PTllNT1VTX0FQUElEOnZrZWJxUmwuamEwbTdu"
  }
}
```

SSO

For SSO, nest an `ssoAuth` object within the `authorization` object and fill in the OAuth credentials provided by the mobile backend. The resulting `authorization` property might look something like this:

```
"authorization": {
  "ssoAuth": {
    "clientID": "b20a34b4-e646-44dc-a787-3a8715f4bb46",
    "clientSecret": "chIkehuDPYsaosPEMyE2",
    "tokenEndpoint": "http://abc09xyz.oracle.com:14100/oam/oauth2/tokens",
  }
}
```

SSO with a Third-Party Token

For SSO with a third-party token, nest a `tokenAuth` object within the `authorization` object and fill in credentials, depending on how you have the token issuer integrated with MCS.

If you are using JWT tokens and have integrated the token issuer by registering a configuration via a policy in MCS, you need to include the mobile backend ID and the OAuth credentials for the backend. The resulting `authorization` property might look something like this:

```
"authorization": {
  "tokenAuth": {
    "backendId": "3b113ad5-07dc-4143-8b6a-a2ef62a175c1",
    "clientId": "b20a34b4-e646-44dc-a787-3a8715f4bb46",
    "clientSecret": "chIkehuDPYsaosPEMyE2"  }
}
```

If you have integrated the IdP token issuer by uploading certificates into MCS you just nest the mobile backend ID. The resulting `authorization` property might look something like this:

```
"authorization": {
  "tokenAuth": {
```

```
    "backendId": "3b113ad5-07dc-4143-8b6a-a2ef62a175c1",
  }
}
```

Facebook

For Facebook login, nest a `facebookAuth` object within the `authorization` object, fill in the HTTP Basic credentials provided by the mobile backend, and add the `facebookAppId`. The resulting `authorization` property might look something like this:

```
"authorization": {
  "basicAuth": {
    "backendId": "3b113ad5-07dc-4143-8b6a-a2ef62a175c1",
    "anonymousToken":
"UFJJTUVfREVDRVBUSUNPTl9NT0JJTEVfQU5PTllNT1VTX0FQUElEOnZrZWJxUmwuamEwbTdu",
    "facebookAppId": "123456789012"
  }
}
```

Loading a Mobile Backend's Configuration in a Cordova App

For any calls to MCS APIs using the Cordova SDK to successfully complete, you need to have the mobile backend's configuration loaded. You do this using the `mobileBackendManager` and `mobileBackend` objects.

The root object in the SDK is the `mcs.mobileBackendManager`. The `mcs.mobileBackendManager` object manages one or more `mobileBackend` objects. A `mobileBackend` object is used to manage connectivity, authentication, and other interactions between your application and its associated mobile backend, including calls to platform APIs and any custom APIs you have defined.

Use `mobileBackendManager.setConfig` to specify a configuration, defined in a local JavaScript object or in the app's `oracle_mobile_cloud_config.js` file. This configuration includes info such as the mobile backend name and version, base URL, and authentication details.

Here's some code you can insert into the app class to retrieve data from the `oracle_mobile_cloud_config.js` file:

```
function initializeMCS(){
  mcs.mobileBackendManager.platform = new mcs.CordovaPlatform();
  mcs.mobileBackendManager.setConfig(mcs_config);
  backend = mcs.mobileBackendManager.getMobileBackend("YOUR_BACKEND_NAME");
  if(backend != null){
    backend.setAuthenticationType("oAuth");
  }
}
```

Authenticating and Logging In Using the SDK for Cordova

Here are some examples of using the Cordova SDK's `Authorization` class.

OAuth and HTTP Basic

Get the mobile backend and set the authentication type to `oAuth` (or `basicAuth`).

```
function initializeMCS(){
  mcs.mobileBackendManager.platform = new mcs.CordovaPlatform();
  mcs.mobileBackendManager.setConfig(mcs_config);
  mcsBackend =
mcs.mobileBackendManager.getMobileBackend("YOUR_BACKEND_NAME");
  if(mcsBackend != null){
    mcsBackend.setAuthenticationType("oAuth");
  }
}
```

Then add a function that calls `Authorization.authenticate` and pass it the MCS mobile backend and a user name and password.

```
function login(username, password){
  return mcsBackend
    .authorization
    .authenticate(username, password)
    .then(succeed)
    .catch(failed);

  function succeed(response){
    logAnalyticsEvent();
    console.log(response.statusCode + " with message: " + response.data);
    return response;
  }

  function failed(response){
    console.log(response.statusCode + " with message: " + response.data);
    return response;
  }
}
```

SSO

Get the mobile backend and set the authentication type to `ssoAuth`.

```
function initializeMCS(){
  mcs.mobileBackendManager.platform = new mcs.CordovaPlatform();
  mcs.mobileBackendManager.setConfig(mcs_config);
  mcsBackend =
mcs.mobileBackendManager.getMobileBackend("YOUR_BACKEND_NAME");
  if (mcsBackend != null) {
    mcsBackend.setAuthenticationType("ssoAuth");
  }
},
```

Then add a function that calls `Authorization.authenticate`.

```
function ssoLogin() {
  mcsBackend.authorization.authenticate().then(
    function (response) {
      console.log(response.statusCode + " with message: " + response.data);
    }).catch(
    function (response) {
      console.log(response.statusCode + " with message: " + response.data);
    });
}
```

SSO with a Third-Party Token

To use SSO with a third-party token, first your app needs to get a token from the third-party token issuer. The way you can obtain the token varies by issuer. For detailed information on obtaining third-party tokens and configuring identity providers in MCS, seeThird-Party SAML and JWT Tokens.

Get the mobile backend and set the authentication type to `tokenAuth`.

```
function initializeMCS(){
  mcs.mobileBackendManager.platform = new mcs.CordovaPlatform();
  mcs.mobileBackendManager.setConfig(mcs_config);
  mcsBackend =
mcs.mobileBackendManager.getMobileBackend("YOUR_BACKEND_NAME");
  if (mcsBackend != null) {
    mcsBackend.setAuthenticationType("tokenAuth");
  }
},
```

Then pass the token you got from the third-party token issuer to a function that calls `Authorization.authenticate`.

```
function ssoLoginToken() {
  mcsBackend.Authorization.authenticate(thirdPartyToken).then(
    function() {
      console.log("MCS authenticate() worked");
     }
   ).catch(
     function() {
       console.log("MCS authenticate() FAILED");
     });
}
```

Facebook

Get the mobile backend and set the authentication type to `facebookAuth`.

```
function initializeMCS(){
  mcs.mobileBackendManager.platform = new mcs.CordovaPlatform();
  mcs.mobileBackendManager.setConfig(mcs_config);
  mcsBackend =
mcs.mobileBackendManager.getMobileBackend("YOUR_BACKEND_NAME");
```

```
  if (mcsBackend != null) {
    mcsBackend.setAuthenticationType("facebookAuth");
},
```

Then add a function that calls `Authorization.authenticate`.

```
function facebookLogin() {
  mcsBackend.authorization.authenticate().then(
    function (response) {
      console.log(response.statusCode + " with message: " + response.data);
    }).catch(
    function (response) {
      console.log(response.statusCode + " with message: " + response.data);
    });
}
```

Setting Up a Cordova App for FCM or GCM Notifications

If you want to use Firebase Cloud Messaging (FCM) or Google Cloud Messaging (GCM) in a Cordova app, follow the instructions below.

For more information on using notifications in MCS, see Notifications.

FCM

These steps configure a Cordova app to use Firebase Cloud Messaging (FCM).

1. Create a project in Firebase. Record the **Server Key** and **Sender ID** (Project Number), and download the `google-service.json` file. For details on setting up a Firebase project, see Set Up a Firebase Cloud Messaging Client App on Android on Google's developer site.
2. Create a client for your mobile app and configure notifications profile(s) by entering the credentials you got in step 1. See Client Management.
3. Copy the `google-service.json` file you downloaded in step 1 to the root of your project, typically `app/`.
4. Add following lines to the application config.xml in the `platform` tag for Android:
   - cordova-android 7.0 or above:

     ```
     <platform name="android">
         <resource-file src="google-services.json" target="app/google-
     services.json" />
     </platform>
     ```

   - cordova-android 6.x or earlier:

     ```
     <platform name="android">
         <resource-file src="google-services.json" target="google-
     services.json" />
     </platform>
     ```

5. Add the `phonegap-plugin-push` Cordova plugin to your application.

```
cordova plugin add phonegap-plugin-push
```

6. From the application code, after the device ready event, register the device.

```
const push = PushNotification.init({
        android: { }
      });

      push.on('registration', (data) => {

backend.notifications.registerForNotifications(data.registrationId,
appId, appVersion, 'FCM');
      });

      push.on('notification', (data) => {
        // data.message,
        // data.title,
        // data.count,
        // data.sound,
        // data.image,
        // data.additionalData
       console.log(data);
      });

      push.on('error', (e) => {
        console.error(e.message);
      });
      function success(data) {
        console.log('Registered successfully');
      }
```

7. For next steps and more information, see Setting Up Android Notifications and Sending Notifications to and from Your App.

GCM

These steps configure a Cordova app to use Google Cloud Messaging (GCM).

> **Note:**
>
> Google Cloud Messaging (GCM) is being phased out, so new apps should be configured with FCM.

1. Open your project in Google console and record the **API Key** and **Sender ID** (Project Number).
2. Create a client for your mobile app and configure notifications profile(s) by entering the credentials you got in step 1. See Client Management.

3. Add the `phonegap-plugin-push` Cordova plugin to your application.

```
cordova plugin add phonegap-plugin-push@v1.9.0 --variable
SENDER_ID="SENDER_ID_FROM_FIRST_STEP"
```

4. From the application code, after the device ready event, register the device.

```
const push = PushNotification.init({
        android: {
            senderID: "SENDER_ID_FROM_FIRST_STEP"
        }
      });

      push.on('registration', (data) => {

backend.notifications.registerForNotifications(data.registrationId,
appId, appVersion, 'GCM');
      });

      push.on('notification', (data) => {
        // data.message,
        // data.title,
        // data.count,
        // data.sound,
        // data.image,
        // data.additionalData
       console.log(data);
      });

      push.on('error', (e) => {
        console.error(e.message);
      });
      function success(data) {
        console.log('Registered successfully');
      }
```

5. For next steps and more information, see Setting Up Android Notifications and Sending Notifications to and from Your App.

Securing Browser-Based Apps Against Cross-Site Request Forgery Attacks

If any of your apps will be browser-based, you need to manage cross-origin resource sharing (CORS) for access to MCS APIs to protect against Cross-Site Request Forgery (CSRF) attacks. Do this by setting the `Security_AllowOrigin` environment to either `disallow` (the default value) or to a comma-separated whitelist of trusted URLs from which cross-site requests can be made. For more information and details on how to use the wildcard character (*), see Securing Cross-Site Requests to MCS APIs.

> **Note:**
>
> For convenience, during the development of a browser-based application or during testing of a hybrid application running in the browser, you can set `Security_AllowOrigin` to `http://localhost:[port]`, but be sure to update the value in production.

Calling Platform APIs Using the SDK for Cordova

Once you include the SDK libraries in your application, and adjust configuration settings, you're ready to use the SDK classes in your apps.

The root object in the Cordova SDK is the `mcs.mobileBackendManager`. An instance of `mcs.mobileBackendManager` manages one or more `mobileBackend` objects. A `mobileBackend` object is used to manage connectivity, authentication, and other transactions between your application and its associated mobile backend, including calls to platform APIs and any custom APIs you have defined. In turn, a `mobileBackend` instance manages instances of `ServiceProxy`. These instances correspond to platform services in MCS (for example, Analytics, Notifications, Offline Data, and so on).

Here's an example of how you could use these classes to get a Storage collection in the mobile backend, create a storage object (in this case, a text file), and then upload that object to the collection. The code here manages calls to the Storage API, including the constructing of the HTTP headers with the mobile backend credentials necessary to access the API:

```
var backend;
var collection_id = 'YOUR_STORAGE_COLLECTION_NAME';

function uploadTextFile() {

  return getCollection()
    .then(success);

  function success(collection){
    var obj = new mcs.StorageObject(collection);
    obj.setDisplayName("JSFile.txt");
    obj.loadPayload("Hello World from Oracle Mobile Cloud Service Cordova
SDK", "text/plain");

    return postObject(collection, obj).then(function(object){
      return readObject(collection, object.id);
    });
  }
}

function getCollection(){
  //return a storage collection with the name assigned to the
collection_id variable.
  return backend
    .storage
    .getCollection(collection_id, null)
```

```
    .then(onGetCollectionSuccess)
    .catch(onGetCollectionFailed);

  function onGetCollectionSuccess(collection){
    console.log('onGetCollectionSuccess:', collection);
    return collection;
  }

  function onGetCollectionFailed(response){
    console.log('onGetCollectionFailed:', response);
    return response.statusCode;
  }
}

function postObject(collection, obj){
  return collection
    .postObject(obj)
    .then(onPostObjectSuccess)
    .catch(onPostObjectFailed);

  function onPostObjectSuccess(object){
    console.log('onPostObjectSuccess:', object);
    return object;
  }

  function onPostObjectFailed(response){
    console.log('onPostObjectFailed:', response);
    return response.statusCode;
  }
}
```

For more information on the individual platform APIs, see Platform APIs.

Calling Custom APIs Using the SDK for Cordova

The SDK provides the `CustomCode` class to simplify the calling of custom APIs in MCS. You can call a REST method (GET, PUT, POST, or DELETE) on an endpoint where the request payload is JSON or empty and the response payload is JSON or empty.

To call a custom API endpoint, you could use something like this:

```
mcs.mobileBackendManager.platform = new mcs.CordovaPlatform();
mcs.mobileBackendManager.setConfig(mcs_config);
backend = mcs.mobileBackendManager.getMobileBackend("CordovaJSBackend");
.....

backend.CustomCode.invokeCustomCodeJSONRequest("TaskApi1/tasks/100" ,
"GET" , null).then(function(response){
    //The response parameter returns the status code and HTTP payload from
the HTTP REST Call.
    console.log(response);
    // Example: { statusCode: 200, data: {} }
    //Depends on the response format defined in the API.
  }).catch(function(response){
```

```
  //The response parameter returns the status code and HTTP payload, if
available, or an error message, from the HTTP REST Call.
  console.log(response);
  /*
    Example:
      { statusCode: 404,
        data: {
      "type":"http://www.w3.org/Protocols/rfc2616/rfc2616-
sec10.html#sec10.4.1",
        "status":404,"title":"API not found",
        "detail":"We cannot find the API cordovaJSApi2 in Mobile Backend
CordovaJSBackend(1.0). Check that this Mobile Backend is associated with
the API.",
        "o:ecid":"005Bojjhp2j2FSHLIug8yf00052t000Jao, 0:2",
"o:errorCode":"MOBILE-57926", "o:errorPath":"/mobile/custom/cordovaJSApi2/
tasks" } }
   */
  //Depends on the response format defined in the API.
  });
```

9
JavaScript Applications

If you develop JavaScript-based mobile apps, you can use the client SDK that Oracle Mobile Cloud Service (MCS) provides for JavaScript. This SDK simplifies authentication with MCS and provides JavaScript wrapper classes for MCS platform APIs.

This SDK is primarily geared toward browser-based apps but can also be used for hybrid frameworks. If you develop Cordova-based apps, use the Cordova SDK. See Cordova Applications.

Getting the SDK for JavaScript

To get the MCS client SDK for JavaScript, go to the Oracle Technology Network's MCS download page.

Contents of the JavaScript SDK Bundle

The JavaScript SDK contains the following items:

- `jsdocs.zip` - The compiled documentation for the library.
- `mcs.js` - The uncompressed version of the SDK. This version contains code comments and is best used as you are developing and debugging your app.
- `mcs.sync.js` - The uncompressed version of the Sync Express library.
- `mcs.min.js` - The compressed version of the SDK. Use this version when you deploy the completed app.
- `mcs.sync.min.js` - The compressed version of the Sync Express library.
- `oracle_mobile_cloud_config.js` - The MCS configuration file. In this file, you insert environment and authentication details for the mobile backends that your app will access.
- `\pako` - This folder includes the `pako` JavaScript library, which is required to use SSO with a third-party token.
- `mcs-tools.zip` - The MCS Custom Code Test Tools, a set of command line tools for debugging custom APIs that you have associated with your app's mobile backend. Detailed instructions are located in the `README` file included in the zip.

Adding the SDK to a JavaScript App

1. If you haven't already done so, unzip the JavaScript SDK zip.
2. Copy `mcs.min.js` and `oracle_mobile_cloud_config.js` into the directory where you keep your JavaScript libraries.
3. Fill in your backend details in `oracle_mobile_cloud_config.js`.

Configuring SDK Properties for JavaScript

To use the SDK in a JavaScript app, you need to add the `oracle_mobile_cloud_config.js` configuration file to the app and fill it in with environment details for your mobile backend. In turn, the SDK classes draw on this file for the details needed to access the mobile backend and use them to construct HTTP headers for REST calls made to APIs.

You package the configuration file in the same folder as the `mcs.min.js` file.

The following example shows the structure of a generic `oracle_mobile_cloud_config.js` file:

```
var mcs_config = {
  "logLevel": mcs.LOG_LEVEL.INFO,
  "logHTTP": true,
  "mobileBackends": {
    "YOUR_BACKEND_NAME": {
      "default": true,
      "baseUrl": "YOUR_BACKEND_BASE_URL",
      "applicationKey": "YOUR_BACKEND_APPLICATION_KEY",
      "authorization": {
        "basicAuth": {
          "backendId": "YOUR_BACKEND_ID",
          "anonymousToken": "YOUR_BACKEND_ANONYMOUS_TOKEN"
        },
      }
    }
  }
};
```

Here's a list of the file's elements. The values that you need to fill in for a given mobile backend can be found on the Settings and Clients pages for that mobile backend.

- `logLevel` — Determines how much SDK logging is displayed in the app's console. The default value is `mcs.LOG_LEVEL.INFO` (where only important events are logged). Other possible values are `mcs.LOG_LEVEL.ERROR` (only errors are logged) and `mcs.LOG_LEVEL.VERBOSE`.
- `logHTTP` — When set to true, enables additional logging capability that includes the complete HTTP headers and body in requests and responses to MCS.
- `mobileBackends` — The config file's root element, containing a JSON object for each mobile backend.
- `baseUrl` — The URL your app uses to connect to its mobile backend.
- `applicationKey` — The application key, which is a unique string assigned to your app when you register it in MCS.
- `backendId` — The unique identifier assigned to a specific mobile backend.
- `anonymousToken` — A unique string that allows your app to access APIs that don't require login. In this scenario, the anonymous key is passed to MCS instead of an encoded user name and password combination.

- `authorization` — JSON object containing the authentication details for connecting your app to MCS. In turn, it must contain one or more objects of type `basicAuth`, `oAuth`, or `tokenAuth`. The contents of the object depend on the type of authentication.

HTTP Basic

For HTTP Basic, you need to nest an `basicAuth` object within the `authorization` object and fill in the HTTP Basic credentials provided by the mobile backend. The resulting authorization property might look something like this:

```
"authorization": {
  "basicAuth": {
    "backendId": "3b113ad5-07dc-4143-8b6a-a2ef62a175c1",
    "anonymousToken":
"UFJJTUVfREVDRVBUSUNPTl9NT0JJTEVfQU5PTllNT1VTX0FQUElEOnZrZWJxUmwuamEwbTdu"
  }
}
```

OAuth Consumer

For OAuth, you need to nest an `oAuth` object within the `authorization` object and fill in the OAuth credentials provided by the mobile backend. The resulting authorization property might look something like this:

```
"authorization": {
  "oAuth": {
    "clientID": "b20a34b4-e646-44dc-a787-3a8715f4bb46",
    "clientSecret": "chIkehuDPYsaosPEMyE2",
    "tokenEndpoint": "http://abc09xyz.oracle.com:14100/oam/oauth2/tokens",
  }
}
```

SSO with a Third-Party Token

For SSO with a third-party token, nest a `tokenAuth` object within the `authorization` object and fill in credentials, depending on how you have the token issuer integrated with MCS.

If you are using JWT tokens and have integrated the token issuer by registering a configuration via a policy in MCS, you need to include the mobile backend ID and the OAuth credentials for the backend. The resulting `authorization` property might look something like this:

```
"authorization": {
  "tokenAuth": {
    "backendId": "3b113ad5-07dc-4143-8b6a-a2ef62a175c1",
    "clientId": "b20a34b4-e646-44dc-a787-3a8715f4bb46",
    "clientSecret": "chIkehuDPYsaosPEMyE2"  }
}
```

If you have integrated the IdP token issuer by uploading certificates into MCS you just nest the mobile backend ID. The resulting `authorization` property might look something like this:

```
"authorization": {
  "tokenAuth": {
    "backendId": "3b113ad5-07dc-4143-8b6a-a2ef62a175c1",
  }
}
```

Loading a Mobile Backend's Configuration into a JavaScript App

For any calls to MCS APIs using the JavaScript SDK to successfully complete, you need to have the mobile backend's configuration loaded. You do this using the `mobileBackendManager` and `mobileBackend` objects.

The root object in the SDK is the `mcs.mobileBackendManager`. The `mcs.mobileBackendManager` object manages one or more `mobileBackend` objects. A `mobileBackend` object is used to manage connectivity, authentication, and other transactions between your application and its associated mobile backend, including calls to platform APIs and any custom APIs you have defined.

Using `mobileBackendManager.setConfig`, you specify a configuration that is defined in the app's `oracle_mobile_cloud_config.js` file. This configuration includes info such as the mobile backend name and version, base URL, and authentication details.

Here's some code you can insert into the app class establish the mobile backend and retrieve data from the `oracle_mobile_cloud_config.js` file.

```
mcs.mobileBackendManager.platform = new mcs.BrowserPlatform();
mcs.mobileBackendManager.setConfig(mcs_config);

this.backend =
mcs.mobileBackendManager.getMobileBackend("YOUR_BACKEND_NAME");
```

Authenticating and Logging In Using the SDK for JavaScript

Here are some examples of how to use the `Authorization` class of the JavaScript SDK in your code.

OAuth and HTTP Basic

Get the mobile backend and set the authentication type to `oAuth` or `basicAuth`.

```
function initializeMCS(){
  mcs.mobileBackendManager.setConfig(mcs_config);
  mcsBackend =
mcs.mobileBackendManager.getMobileBackend("YOUR_BACKEND_NAME");
  if(mcsBackend != null){
    mcsBackend.setAuthenticationType("oAuth");
```

```
  }
}
```

Then add a function that calls `Authorization.authenticate` and pass it a user name and password.

```
function login(username, password){
  var deferred = $q.defer();
  mcsBackend.Authorization.authenticate(username, password, success,
failed);

  return deferred.promise;

  function success(response,data){
    deferred.resolve();
    logAnalyticsEvent();
  }

  function failed(statusCode,data){
    deferred.reject();
  }
}
```

SSO with a Third-Party Token

To use SSO with a third-party token, first your app needs to get a token from the third-party token issuer. The way you can obtain the token varies by issuer. For detailed information on obtaining third-party tokens and configuring identity providers in MCS, see Third-Party SAML and JWT Tokens.

> **Note:**
>
> Third-party token exchange requires the pako JavaScript library, so make sure to add it to your app. Pako is distributed with the SDK in the \pako subdirectory.

Get the mobile backend and set the authentication type to `tokenAuth`.

```
function initializeMCS(){
  mcs.mobileBackendManager.platform = new mcs.JSPlatform();
  mcs.mobileBackendManager.setConfig(mcs_config);
  mcsBackend =
mcs.mobileBackendManager.getMobileBackend("YOUR_BACKEND_NAME");
  if (mcsBackend != null) {
    mcsBackend.setAuthenticationType("tokenAuth");
  }
},
```

Then pass the token you got from the third-party token issuer to a function that calls `Authorization.authenticate`.

```
mcsBackend.Authorization.authenticate(thirdPartyToken).then(
    function() {
      console.log("MCS authenticate() worked");
     }
   ).catch(
     function() {
       console.log("MCS authenticate() FAILED");
     }
  );
```

Securing Browser-Based Apps Against Cross-Site Request Forgery Attacks

If any of your apps will be browser-based, you need to manage cross-origin resource sharing (CORS) for access to MCS APIs to protect against Cross-Site Request Forgery (CSRF) attacks. Do this by setting the `Security_AllowOrigin` environment to either `disallow` (the default value) or to a comma-separated whitelist of trusted URLs from which cross-site requests can be made. For more information and details on how to use the wildcard character (*), see Securing Cross-Site Requests to MCS APIs.

> **Note:**
>
> For convenience, during the development of a browser-based application or during testing of a hybrid application running in the browser, you can set `Security_AllowOrigin` to `http://localhost:`*`[port]`*, but be sure to update the value in production.

Calling Platform APIs Using the SDK for JavaScript

Once you include the SDK libraries in your application, and adjust configuration settings, you're ready to use the SDK classes in your apps.

The root class in the JavaScript SDK is the `mcs.mobileBackendManager`. An instance of `mcs.mobileBackendManager` manages one or more`mobileBackend` objects. A `mobileBackend` object is used to manage connectivity, authentication, and other transactions between your application and its associated mobile backend, including calls to platform APIs and any custom APIs you have defined. In turn, a `mobileBackend` instance manages instances of `ServiceProxy`. These instances correspond to platform services in MCS (for example, Analytics, Notifications, Offline Data, and so on).

It retrieves the information it needs about each mobile backend (such as the mobile backend name and authentication information) from the app's `oracle_mobile_cloud_config.js` file.

Here's an example of how you could use these classes to get a Storage collection in the mobile backend, create a storage object (in this case, a text file), and then upload

that object to the collection. The code here manages calls to the Storage API, including the constructing of the HTTP headers with the mobile backend credentials necessary to access the API:

```
var backend;
var collection_id = 'YOUR_STORAGE_COLLECTION_NAME';

function uploadTextFile() {

  return getCollection()
    .then(success);

  function success(collection){
    //create new Storage object and set its name and payload
    var obj = new mcs.StorageObject(collection);
    obj.setDisplayName("JSFile.txt");
    obj.loadPayload("Hello World from Oracle Mobile Cloud Service
Javascript SDK", "text/plain");

    return postObject(collection, obj).then(function(object){
      return readObject(collection, object.id);
    });
  }
}

function getCollection(){
  var deferred = $q.defer();

  //return a storage collection with the name assigned to the
collection_id variable.
  backend.Storage.getCollection(collection_id, null,
onGetCollectionSuccess, onGetCollectionFailed);

  return deferred.promise;

  function onGetCollectionSuccess(collection){
    deferred.resolve(collection);
  }

  function onGetCollectionFailed(statusCode, headers, data){
    deferred.reject(statusCode);
  }
}

function postObject(collection, obj){
  var deferred = $q.defer();

  //post an object to the collection
  collection.postObject(obj, onPostObjectSuccess, onPostObjectFailed);

  return deferred.promise;

  function onPostObjectSuccess(object){
    deferred.resolve(object);
  }
```

```
  function onPostObjectFailed(statusCode, headers, data){
    deferred.reject(statusCode);
  }
}
```

For more information on the individual platform APIs, see Platform APIs.

Avoiding Unsafe Header Errors

When you have JavaScript web apps that call the Storage APIs, you need to set the `Security_ExposeHeaders` policy to allow headers returned by these APIs to be accessed by the browser. For example, setting the value of that policy to the following would allow you to use all Storage API endpoints:

```
*.*.Security_ExposeHeaders=Oracle-Mobile-Created-By,Oracle-Mobile-Created-
On,Oracle-Mobile-Modified-By,Oracle-Mobile-Modified-On,Accept-
Encoding,Oracle-Mobile-Name,ETag
```

For instructions on setting policies, see Environment Policies.

Calling Custom APIs Using the SDK for JavaScript

The SDK provides the `CustomCode` class to simplify the calling of custom APIs in MCS. You can use this class to call a REST method (GET, PUT, POST, or DELETE) on an endpoint where the request payload is JSON or empty and the response payload is JSON or empty.

To make a call to a custom API endpoint, you could use something like this:

```
mcs.mobileBackendManager.setConfig(mcs_config);
backend = mcs.mobileBackendManager.getMobileBackend("JSBackend");
.....

backend.CustomCode.invokeCustomCodeJSONRequest("TaskApi1/tasks/100" ,
"GET" , null, function(statusCode, data){
  mcs._Logger.log(mcs.LOG_LEVEL.INFO, statusCode);
  //The statusCode parameter returns the status code from the HTTP REST
Call.
  mcs._Logger.log(mcs.LOG_LEVEL.INFO, data);
  //The data parameter is the HTTP payload from the server, if available,
or an error message.
    Example:
      statusCode: 200,
      data: {}
      //Depends on the response format defined in the API.
},
function(statusCode, data){
  mcs._Logger.log(mcs.LOG_LEVEL.INFO, statusCode);
  //The statusCode parameter returns the status code from the HTTP REST
Call.
  mcs._Logger.log(mcs.LOG_LEVEL.INFO, data);
  //The data parameter is the HTTP payload from the server, if available,
```

```
or an error message.
    Example:
      statusCode: 404,
      data: {
        "type":"http://www.w3.org/Protocols/rfc2616/rfc2616-
sec10.html#sec10.4.1",
        "status":404,"title":"API not found",
        "detail":"We cannot find the API JSApi2 in Mobile Backend
JSBackend(1.0). Check that this Mobile Backend is associated with the
API.",
        "o:ecid":"005Bojjhp2j2FSHLIug8yf00052t000Jao, 0:2",
"o:errorCode":"MOBILE-57926", "o:errorPath":"/mobile/custom/JSApi2/tasks" }
        //Depends on the response format defined in the API.
  });
```

10

Xamarin Android Applications

If you use the Xamarin platform to develop Android apps, you can use the SDK that Oracle Mobile Cloud Service (MCS) provides for Xamarin Android apps. This SDK simplifies authentication with MCS and provides native wrapper classes for MCS platform APIs.

Getting the SDK for Xamarin Android

To get the MCS client SDK for Xamarin Android, go to the Oracle Technology Network's MCS download page.

To use this SDK, you should have the following software on your system:

- Microsoft Visual Studio, with support for Xamarin development.
- Java Development Kit (JDK) 1.7.0_67 or compatible.

  See http://www.oracle.com/technetwork/java/javase/downloads/index.html for JDK downloads.

Adding the SDK to a Xamarin Android Project

1. If you haven't already done so, extract the contents from the SDK zip.
2. In Visual Studio, create a Visual C# Android app.
3. Make sure you can connect to the internet from Visual Studio connection so that NuGet packages are reachable.
4. Add GCM and Facebook dependencies to your project:
   - If a Packages node appears in the Solution Explorer for your project, do the following:
     a. Right-click the **Packages** node.
     b. Type `GCM` in the search field, select `Xamarin.GooglePlayServices.Gcm` (*not* `Crosslight.Xamarin.GooglePlayServices.GCM`), and click **Add Package**. The remaining GCM dependencies will be added automatically.
     c. Accept the terms to add the packages successfully.
     d. Add `Xamarin.Facebook.Android` by searching for it in the NuGet packages and adding it in the same way you added the GCM packages.
   - If a Packages node `doesn't` appear in the Solution Explorer for your project, do the following:
     a. Select **Tools > NuGet Package Manager > Manage NuGet Packages for Solution**.
     b. Select the **Browse** tab.

c. Type `GCM` in the search field, select `Xamarin.GooglePlayServices.Gcm` (*not* `Crosslight.Xamarin.GooglePlayServices.GCM`), select the checkbox for your app, and click **Install**. The remaining GCM dependencies will be added automatically.

d. After previewing the changes, click **OK**.

e. Add `Xamarin.Facebook.Android` by searching for it in the NuGet packages and adding it in the same way you added the GCM packages.

5. At the end make sure you have all the below dependencies. If any of them are missing, search for them in the NuGet package manager.

```
<packages>
  <package id="Bolts" version="1.4.0.1"
targetFramework="monoandroid71" />
  <package id="Xamarin.Android.Support.Animated.Vector.Drawable"
version="25.4.0.2" targetFramework="monoandroid71" />
  <package id="Xamarin.Android.Support.Annotations" version="25.4.0.2"
targetFramework="monoandroid71" />
  <package id="Xamarin.Android.Support.Compat" version="25.4.0.2"
targetFramework="monoandroid71" />
  <package id="Xamarin.Android.Support.Core.UI" version="25.4.0.2"
targetFramework="monoandroid71" />
  <package id="Xamarin.Android.Support.Core.Utils" version="25.4.0.2"
targetFramework="monoandroid71" />
  <package id="Xamarin.Android.Support.CustomTabs" version="25.4.0.2"
targetFramework="monoandroid71" />
  <package id="Xamarin.Android.Support.Design" version="25.4.0.2"
targetFramework="monoandroid71" />
  <package id="Xamarin.Android.Support.Fragment" version="25.4.0.2"
targetFramework="monoandroid71" />
  <package id="Xamarin.Android.Support.Media.Compat" version="25.4.0.2"
targetFramework="monoandroid71" />
  <package id="Xamarin.Android.Support.Transition" version="25.4.0.2"
targetFramework="monoandroid71" />
  <package id="Xamarin.Android.Support.v4" version="25.4.0.2"
targetFramework="monoandroid71" />
  <package id="Xamarin.Android.Support.v7.AppCompat" version="25.4.0.2"
targetFramework="monoandroid71" />
  <package id="Xamarin.Android.Support.v7.CardView" version="25.4.0.2"
targetFramework="monoandroid71" />
  <package id="Xamarin.Android.Support.v7.RecyclerView"
version="25.4.0.2" targetFramework="monoandroid71" />
  <package id="Xamarin.Android.Support.Vector.Drawable"
version="25.4.0.2" targetFramework="monoandroid71" />
  <package id="Xamarin.Build.Download" version="0.4.7"
targetFramework="monoandroid80" />
  <package id="Xamarin.Facebook.Android" version="4.26.0"
targetFramework="monoandroid80" />
  <package id="Xamarin.Google.ZXing.Core" version="3.3.0"
targetFramework="monoandroid80" />
  <package id="Xamarin.GooglePlayServices.Base" version="42.1021.1"
targetFramework="monoandroid71" />
  <package id="Xamarin.GooglePlayServices.Basement" version="42.1021.1"
targetFramework="monoandroid71" />
```

```
  <package id="Xamarin.GooglePlayServices.Gcm" version="42.1021.1"
targetFramework="monoandroid71" />
  <package id="Xamarin.GooglePlayServices.Iid" version="42.1021.1"
targetFramework="monoandroid71" />
  <package id="Xamarin.GooglePlayServices.Tasks" version="42.1021.1"
targetFramework="monoandroid71" />
</packages>
```

6. Add the SDK's DLL file to your app by right-clicking the project's **References** node and selecting **Edit References** or **Add Reference** (depending on which menu item is available).
   - If you select **Edit References**, click the **.NET Assembly** tab, and then browse to the `Android.dll` file in the extracted SDK zip.
   - If you select **Add Reference**, click the **Browse** tab, click the **Browse** button, and then navigate to the `Android.dll` file in the extracted SDK zip.
7. Add the configuration file to the app by right-clicking the project's **Assets** node and selecting either **Add > Add Files** or **Add > Existing File** (depending which is available) and then navigating to the SDK's `oracle_mobile_cloud_config.xml` file.
8. Select the node for `oracle_mobile_cloud_config.xml` so that it's properties are displayed in the Properties pane. Then make sure that the **Build Action** property is set to `AndroidAsset`.
9. Open `oracle_mobile_cloud_config.xml` and fill in the environment details for the mobile backend that the app will be using. See Configuring SDK Properties for Xamarin Android.
10. Update the `AndroidManifest.xml` file with the necessary properties as detailed in Configuring Your AndroidManifest.xml File.

Configuring SDK Properties for Xamarin Android

To use the SDK in an Android app, you need to add the `oracle_mobile_cloud_config.xml` configuration file to the app and fill it in with environment details for your mobile backend. In turn, the SDK classes use the information provided in this file to access the mobile backend and construct HTTP headers for REST calls made to APIs.

The following code sample shows the structure of a `oracle_mobile_cloud_config.xml` file:

```
<mobileBackends>
  <mobileBackend>
    <mbeName>MBE_NAME</mbeName>
    <mbeVersion>MBE_VERSION</mbeVersion>
    <default>true</default>
    <appKey>APPLICATION_KEY</appKey>
    <baseUrl>BASE_URL</baseUrl>
    <enableAnalytics>true</enableAnalytics>
    <enableLogger>true</enableLogger>
    <authorization>
      <offlineAuthenticationEnabled>true</offlineAuthenticationEnabled>
      <authenticationType>AUTH_TYPE</authenticationType>
```

```
      <oauth>
        <oAuthTokenEndPoint>OAUTH_URL</oAuthTokenEndPoint>
        <oAuthClientId>CLIENT_ID</oAuthClientId>
        <oAuthClientSecret>CLIENT_SECRET</oAuthClientSecret>
      </oauth>
      <basic>
        <mobileBackendID>MOBILE_BACKEND_ID</mobileBackendID>
        <anonymousKey>ANONYMOUS_KEY</anonymousKey>
      </basic>
    </authorization>
    <!-- additional properties go here -->
  </mobileBackend>
</mobileBackends>
```

Here's a list of the file's elements. The values that you need to fill in for a given mobile backend can be found on the Settings and Clients pages for that mobile backend.

- `mobileBackends` — The config file's root element, containing one or more `mobileBackend` elements.
- `mobileBackend` — The element for a mobile backend.
- `mbeName` — The name of the mobile backend associated with your app.
- `mbeVersion` — The version number of your app (for example, `1.0`).
- `default` — If `true`, that mobile backend is treated as the default and thus can be easily referenced using the `getDefaultMobileBackend(Context context)` method in the SDK's `MobileBackendManager` class.
- `appKey` — The application key, which is a unique string assigned to your app when you register it as a client in MCS. See Registering an App as a Client in MCS.
- `baseUrl` — The URL your app uses to connect to its mobile backend.
- `enableLogger` — When set to `true`, logging is included in your app.
- `enableAnalytics` — When set to `true`, analytics on the app's use can be collected.
- `authorization` — Use the sub-elements of this element to define the authentication the app will be using and specify the required credentials.
  - `offlineAuthenticationEnabled` — If set to `true`, offline login will be allowed. For this to work, you also need to add the following to the app's `AndroidManifest.xml` file:

    ```
    <receiver android:name="oracle.cloud.mobile.network.NetworkHelper"
      <intent-filter>
        <action android:name="android.net.conn.CONNECTIVITY_CHANGE" />
      </intent-filter>
    </receiver>
    ```

  - `authenticationType` — Define the kind of authentication mechanism being used to connect your app to MCS. Possible values are `oauth` (for OAuth Consumer) , `basic` (for HTTP Basic), `sso`, `tokenAuth` (for SSO token exchange), and `facebook` (for logging in with Facebook credentials). If this element isn't specified, OAuth Consumer is used. The other contents and sub-elements of the `authorization` element depend on the type of authentication.

OAuth Consumer

For OAuth, set the value of the `<authenticationType>` element to `oauth` and fill in the OAuth credentials provided by the mobile backend.

- `oAuthTokenEndPoint` — The URL of the OAuth server your app goes to, to get its authentication token.
- `oAuthClient` — The unique client identifier assigned to all apps when they're first created in your mobile backend.
- `oAuthClientSecret` — The unique secret string assigned to all apps they're first created in your mobile backend.

The resulting `authorization` element might look something like this:

```
<authorization>
  <offlineAuthenticationEnabled>true</offlineAuthenticationEnabled>
  <authenticationType>oauth</authenticationType>
  <oauth>
    <oAuthTokenEndPoint>http://oam-server.oracle.com/oam/oauth2/tokens</
oAuthTokenEndPoint>
    <oAuthClient>f2d3ca5c-7e6f-4d1c-aabc-a2f3caf7ec4e</oAuthClient>
    <oAuthClientSecret>vZMRkgniIbhNUiPnSRT2</oAuthClientSecret>
  </oauth>
</authorization>
```

Enterprise SSO

For SSO, set the value of the `<authenticationType>` element to `sso`, fill in the OAuth credentials provided by the mobile backend, and add the `ssoTokenEndpoint`.

The resulting `authorization` element might look something like this:

```
<authorization>
  <offlineAuthenticationEnabled>true</offlineAuthenticationEnabled>
  <authenticationType>sso</authenticationType>
  <oauth>
    <oAuthTokenEndPoint>host/mobile/platform/sso/token</oAuthTokenEndPoint>
    <oAuthClient>f2d3ca5c-7e6f-4d1c-aabc-a2f3caf7ec4e</oAuthClient>
    <oAuthClientSecret>vZMRkgniIbhNUiPnSRT2</oAuthClientSecret>
    <ssoTokenEndpoint>https://development-
mcspmtrial90.mobileenv.oracle.com:443/mobile/platform/sso/token</
ssoTokenEndpoint>
</oauth>
</authorization>
```

SSO with a Third Party Token

For SSO with a third-party token, set the value of the `<authenticationType>` element to `tokenAuth` and fill in the HTTP Basic auth credentials provided by the mobile backend (described next).

The resulting `authorization` element might look something like this:

```
<authorization>
  <offlineAuthenticationEnabled>true</offlineAuthenticationEnabled>
  <authenticationType>tokenAuth</authenticationType>
  <basic>
    <mobileBackendID>6d3744b8-cab2-479c-998b-ebba2c31560f</mobileBackendID>
    <anonymousKey>UFJJTUVfREVDRVBUSUNPTl9NT0JJTEVfQU5PTll</anonymousKey>
  </basic>
</authorization>
```

HTTP Basic

For HTTP Basic authentication, you need to set the value of the `<authenticationType>` element to `basic` and fill in the HTTP Basic auth credentials provided by the mobile backend.

- `mobileBackendID` — The unique identifier assigned to a specific mobile backend. It gets passed in an HTTP header of every REST call made from your app to MCS, to connect it to the correct mobile backend. When calling platform APIs, the SDK handles the construction of the authentication headers for you.
- `anonymousKey` — A unique string that allows your app to access APIs that don't require login. In this scenario, the anonymous key is passed to MCS instead of an encoded user name and password combination.

The resulting `authorization` element might look something like this:

```
<authorization>
  <offlineAuthenticationEnabled>true</offlineAuthenticationEnabled>
  <authenticationType>basic</authenticationType>
  <basic>
    <mobileBackendID>6d3744b8-cab2-479c-998b-ebba2c31560f</mobileBackendID>
    <anonymousKey>UFJJTUVfREVDRVBUSUNPTl9NT0JJTEVfQU5PTll</anonymousKey>
  </basic>
</authorization>
```

Facebook

For Facebook login, you need to set the value of the `<authenticationType>` element to `facebook`, fill in the HTTP Basic auth credentials provided by the mobile backend, and add the `facebook` element, where you specify the Facebook credentials.

- `facebookAppId` — The Facebook application ID.
- `scopes` — You can use this element to specify Facebook permissions (optional).

The resulting `authorization` element might look something like this:

```
<authorization>
  <offlineAuthenticationEnabled>true</offlineAuthenticationEnabled>
  <authenticationType>facebook</authenticationType>
  <basic>
    <mobileBackendID>6d3744b8-cab2-479c-998b-ebba2c31560f</mobileBackendID>
    <anonymousKey>UFJJTUVfREVDRVBUSUNPTl9NT0JJTEVfQU5PTll</anonymousKey>
  </basic>
```

```
  <facebook>
    <facebookAppId>123456789012345</facebookAppId>
    <scopes>public_profile,user_friends,email,user_location,user_birthday</
scopes>
  </facebook>
</authorization>
```

Configuring Your AndroidManifest.xml File

Permissions for operations such as accessing the network and finding the network state are controlled through permission settings in `AndroidManifest.xml`. These permissions are required:

- `permission.INTERNET` — Allows your app to access open network sockets.
- `permission.ACCESS_NETWORK_STATE` — Allows your app to access information about networks.

Other permissions are optional. For example, the Analytics platform API uses location to provide detailed information about the usage and performance of your app. If you're using the Analytics library from the SDK, you'll want to add these permissions as well.

- `permission.ACCESS_COARSE_LOCATION`— Allows your app to access approximate location information, derived from sources such as wi-fi and cell tower positions.
- `permission.ACCESS_FINE_LOCATION` — Allows your app to access precise location information, derived from sources such as GPS.

For more information about permissions in your Android application, see Android Manifest Permissions in the Google documentation.

Add the permissions at the top of your `AndroidManifest.xml` file, as shown in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
package="oracle.cloud.mobile.photobox" >
  <uses-permission android:name="android.permission.INTERNET" />
  <uses-permission
android:name="android.permission.ACCESS_NETWORK_STATE" />
  <uses-permission
android:name="android.permission.ACCESS_FINE_LOCATION" />
  <uses-permission
android:name="android.permission.ACCESS_COARSE_LOCATION" />
  <application>
    <receiver android:name="oracle.cloud.mobile.network.NetworkHelper"
      <intent-filter>
        <action android:name="android.net.conn.CONNECTIVITY_CHANGE" />
      </intent-filter>
    </receiver>
    (.....)
  </application>
</manifest>
```

If you are using the Notifications API in your app, you may also need to add a broadcast receiver element. See Setting Up a Mobile App for Notifications.

Loading a Mobile Backend's Configuration into a Xamarin Android App

For any calls to MCS APIs using the Xamarin Android SDK to successfully complete, you need to have the mobile backend's configuration loaded from the app's `oracle_mobile_cloud_config.xml` file. You do this using the `MobileBackendManager` class:

```
MobileBackendManager.Manager.GetMobileBackend(context,"GCMBackend");
```

Authenticating and Logging In Using the SDK for Xamarin Android

Here is some sample code that you can use for authentication through MCS in your Xamarin Android apps.

OAuth Consumer

First you initialize the authorization agent and set the authentication type to `OAUTH`.

```
MobileBackend mobileBackend;
IAuthorizationAgent mAuthorization;
mobileBackend = MobileBackendManager.Manager.GetDefaultMobileBackend(mCtx);
mAuthorization = mobileBackend.GetAuthorization(AuthType.Oauth);
```

Then you use the `authenticate` method to attempt authentication. The call includes parameters for Android context, user name, password, and a callback that completes the authorization process.

```
TextView username, password;
username = (AutoCompleteTextView)FindViewById(Resource.Id.username);
password = (EditText)FindViewById(Resource.Id.password);
String userName = username.Text;
String passWord = password.Text;
mAuthorization.Authenticate(mCtx, userName, passWord, new
AuthorizationCallback());
```

Here's the definition for the callback.

```
Authorization CallBack
private class AuthorizationCallback : Java.Lang.Object,
IAuthorizationCallback
{
    public void OnCompletion(ServiceProxyException exception)
    {
        if (exception != null)
```

```
        {
            Logger.Error(TAG, "Exception while receiving the Access
Token", exception);
        }
        else
        {
            Logger.Error(TAG, "Authorization successful");
        }
    }
}
```

Enterprise SSO

```
mAuthorization.authenticateSSO(mCtx, false, new AuthorizationCallback());


private class AuthorizationCallback : Java.Lang.Object,
IAuthorizationCallback
{
    public void OnCompletion(ServiceProxyException exception)
    {
        if (exception != null)
            Logger.Debug(TAG, "Exception " + exception.Message;
        else
        {
            Logger.Debug(TAG, "SSO Auth Succeeded");
        }
    }
}
```

SSO with a Third-Party Token

First, your app needs to get a token from the third-party token issuer. The way you can obtain the token varies by issuer. For detailed information on obtaining third-party tokens and configuring identity providers in MCS, see Third-Party SAML and JWT Tokens.

Once you have the token, initialize the authorization agent and use the token in your authorization call. The example below checks to see if the token is already stored in MCS before logging in again.

> **Note:**
>
> The default expiration time for storing a third-party token in MCS is 6 hours. You can adjust this time by changing the `Security_TokenExchangeTimeoutSecs` policy.

```
IAuthorizationAgent mAuthorization;
MobileBackend mobileBackend;
mobileBackend = MobileBackendManager.Manager.GetDefaultMobileBackend(mCtx);
mAuthorization = mobileBackend.GetAuthorization(AuthType.Tokenauth);
```

```
// Check whether credential exists in secure store
Boolean isCredentialLoaded = mAuthorization.LoadSSOTokenExchange(mCtx);

if(isCredentialLoaded){
    // Credentials found in secure store - redirect to main activity
    Logger.Info(TAG, "Credentials got loaded successfully from secure
store.");
    Intent intent = new Intent(mCtx, typeof(ContentActivity));
    StartActivity(intent);

} else {
    // Credentials not found - authenticate using token exchange
    Logger.Info(TAG, "Credentials could not be found in secure store.");
    mAuthorization.AuthenticateUsingTokenExchange(mCtx, token, true,
mLoginCallback);
}
```

Here's the callback:

```
private class AuthorizationCallback : Java.Lang.Object,
IAuthorizationCallback
{
    public void OnCompletion(ServiceProxyException exception)
    {
        if (exception == null)
        {
            //log event with Analytics
            mAnalyticsAgent.LogEvent("Login with 3rd party token
successfully");
            mAnalyticsAgent.Flush();

            //redirect to another Activity after login
            Intent intent = new Intent(mCtx, typeof(ContentActivity));
            LoginActivity.activity.StartActivity(intent);

        } else {
            Logger.Error(TAG, "Exception during token exchange:",
exception);
            LoginActivity.activity.Finish();
        }
    }
}
```

HTTP Basic Authentication

The code for handling login with HTTP Basic is nearly the same as the code for OAuth.

First you initialize the authorization agent and set the authentication type to `BASIC_AUTH`.

```
MobileBackend mobileBackend;
IAuthorizationAgent mAuthorization;
```

```
mobileBackend = MobileBackendManager.Manager.GetDefaultMobileBackend(mCtx);
mAuthorization = mobileBackend.GetAuthorization(AuthType.BasicAuth);
```

Then you use the `Authenticate` method to attempt authentication. The call includes parameters for Android context, user name, password, and a callback that completes the authorization process.

```
TextView username, password;
username = (AutoCompleteTextView)FindViewById(Resource.Id.username);
password = (EditText)FindViewById(Resource.Id.password);
String userName = username.Text;
String passWord = password.Text;
mAuthorization.Authenticate(mCtx, userName, passWord, new
AuthorizationCallback());
```

Here's the definition for the callback.

```
private class AuthorizationCallback : Java.Lang.Object,
IAuthorizationCallback
{
    public void OnCompletion(ServiceProxyException exception)
    {
        Logger.Debug(TAG, "OnCompletion Auth Callback");
        if (exception != null)
        {
            Logger.Error(TAG, "Exception while receiving the Access
Token", exception);
        }
        else
        {
            Logger.Error(TAG, "Authorization successful");
        }
    }
}
```

Facebook

First you initialize the authorization agent and set the authentication type to `Facebook`.

```
ISocialAuthorizationAgent mAuthorization;
SocialMobileBackend socialMobileBackend;
socialMobileBackend =
SocialMobileBackendManager.Manager.GetDefaultMobileBackend(context);
mAuthorization =
socialMobileBackend.GetSocialAuthorization(SocialAuthType.Facebook);
mAuthorization.SetAuthType(AuthType.Facebook);
```

Using a `CallbackManager` object from Facebook's SDK, initiate authentication.

```
ICallbackManager callbackManager;
mAuthorization.Setup(context, new FacebookCallback());
```

```
callbackManager = mAuthorization.CallBackManager;
mAuthorization.AuthenticateSocial(activity);
```

Here's code you can use for the `callback` that is passed above.

```
private class FacebookCallback : Java.Lang.Object, IFacebookCallback
{
    public void OnSuccess(Java.Lang.Object loginResult)
    {
        Logger.Error(TAG, "facebook login successful.");
    }

    public void OnCancel()
    {
    }

    public void OnError(FacebookException error)
    {
    }
}
```

Override the `OnActivityResult()` method to use the callback.

```
protected override void OnActivityResult(int requestCode, Result
resultCode, Intent data)
{
    Logger.Debug(TAG, "In OnActivity Result    onActivityResult");

    base.OnActivityResult(requestCode, resultCode, data);
    callBackManger.OnActivityResult(requestCode, (int)resultCode, data);
}
```

Calling Platform APIs Using the SDK for Xamarin Android

Once the mobile backend's configuration info is loaded into the app and you have made a call to get the mobile backend, you can use SDK classes for various platform APIs.

Here are some code snippets that illustrate how to access these APIs with the SDK.

User Management

Getting a User

```
IAuthorizationAgent authorizationAgent = authentication.Authorization;
authorizationAgent.FetchCurrentUser(new UserRegistrationCallback());
private class UserRegistrationCallback : Java.Lang.Object,
IUserRegistrationCallback
{
    public void OnComplete(ServiceProxyException exception, User user)
    {
        if (exception == null)
```

```
        {
            mUser = user;
            setText("User " + mUser.Username + " details have been fetched
successfully.");
        }
        else
        {
            //Handle Error
        }
    }
}
```

Updating a User

```
//creating map with properties
IDictionary<string, Object> map = new Dictionary<string,Object>();
map.Add("age", 26);
map.Add("address", "india");
authorizationAgent.UpdateUser(new UserRegistrationCallback(), map);

private class UserRegistrationCallback : Java.Lang.Object,
IUserRegistrationCallback
{
    public void OnComplete(ServiceProxyException exception, User user)
    {

        if (exception == null)
        {
            setText("User " + user.Username + " details have been updated
successfully.");
        }
        else
        {
            /Handle Error
        }
    }
}
```

Location

Initialization

```
Location location =
(Location)mobileBackend.GetServiceProxy(Class.FromType(typeof(Location));
```

Places, Devices, and Assets

```
static Location location;
static LocationPlace place;
static LocationDevice device;
static LocationAsset asset;
```

```
location =
(Location)mobileBackend.GetServiceProxy(Class.FromType(typeof(Location)));

LocationPlaceQuery locationPlaceQuery = location.BuildPlaceQuery();

locationPlaceQuery.Name = "West";

locationPlaceQuery.OrderByAttributeType =
LocationDeviceContainerQuery.LocationDeviceContainerQueryOrderByAttributeTy
pe.LocationDeviceContainerQueryOrderByAttributeTypeName;
locationPlaceQuery.Format =
LocationObjectQuery.LocationObjectQueryFormatType.LocationObjectQueryFormat
TypeShort;

locationPlaceQuery.Execute(new LocationObjectQueryCallback());

LocationDeviceQuery locationDeviceQuery = location.BuildDeviceQuery();

locationDeviceQuery.Name = "Beacon";

locationDeviceQuery.OrderByAttributeType =
LocationDeviceQuery.LocationDeviceQueryOrderByAttributeType.LocationDeviceQ
ueryOrderByAttributeTypeName;
locationDeviceQuery.Format =
LocationObjectQuery.LocationObjectQueryFormatType.LocationObjectQueryFormat
TypeShort;

locationDeviceQuery.Execute(new LocationObjectQueryCallback());



LocationAssetQuery locationAssetQuery = location.BuildAssetQuery();

locationAssetQuery.Name = "Joe";

locationAssetQuery.OrderByAttributeType =
LocationDeviceContainerQuery.LocationDeviceContainerQueryOrderByAttributeTy
pe.LocationDeviceContainerQueryOrderByAttributeTypeName;
locationAssetQuery.Format =
LocationObjectQuery.LocationObjectQueryFormatType.LocationObjectQueryFormat
TypeShort;

locationAssetQuery.Execute(new LocationObjectQueryCallback());
```

Fetching a Place

```
private class LocationObjectQueryCallback : Java.Lang.Object,
ILocationObjectsQueryCallback
{

    public void OnComplete(LocationObjectQueryResult queryResult,
ServiceProxyException exception)
    {
```

```
        if (mProgressDialog != null && mProgressDialog.IsShowing)
        {
            mProgressDialog.Dismiss();
        }

        if (exception != null)
        {
            Logger.Debug(TAG, exception.Message);

            setText(exception.Message);
        }
        else
        {

            foreach (LocationObject locationobject in queryResult.Items)
            {

                if
(locationobject.GetType().Equals(typeof(LocationPlace))) {

                    place = (LocationPlace)locationobject;

                    location.FetchPlace(place.Id, new
LocationObjectFetchCallback());

                    lock (obj)
                    {
                        Monitor.Wait(obj);
                    }

                    Logger.Debug(TAG, place.Name +"  " +
place.HasChildren);
                }
                else
if(locationobject.GetType().Equals(typeof(LocationDevice)))
                {

                    device = (LocationDevice)locationobject;

                    location.FetchDevice(device.Id, new
LocationObjectFetchCallback());

                    lock (obj)
                    {
                        Monitor.Wait(obj);
                    }

                    Logger.Debug(TAG, device.Name + "  ");
                }
                else if
(locationobject.GetType().Equals(typeof(LocationAsset)))
                {

                    asset = (LocationAsset)locationobject;
```

```
                    location.FetchAsset(asset.Id, new
LocationObjectFetchCallback());

                    lock (obj)
                    {
                        Monitor.Wait(obj);
                    }

                    Logger.Debug(TAG, asset.Name + "  ");
                }

            }
        }
    }
}


private class LocationObjectFetchCallback : Java.Lang.Object,
ILocationObjectFetchCallback
{

    public void OnComplete(LocationObject locationObject,
ServiceProxyException exception)
    {
        if (mProgressDialog != null && mProgressDialog.IsShowing)
        {
            mProgressDialog.Dismiss();
        }

        if (exception != null)
        {
            Logger.Debug(TAG, exception.Message);

            setText(exception.Message);
        }
        else
        {
            Logger.Debug(TAG, locationObject.Name );

        }

        lock (obj)
        {
            Monitor.PulseAll(obj);
        }

    }
}
```

Refreshing

```
private class LocationObjectQueryCallback : Java.Lang.Object,
ILocationObjectsQueryCallback
{
```

```
    public void OnComplete(LocationObjectQueryResult queryResult,
ServiceProxyException exception)
    {
        if (mProgressDialog != null && mProgressDialog.IsShowing)
        {
            mProgressDialog.Dismiss();
        }

        if (exception != null)
        {
            Logger.Debug(TAG, exception.Message);

            setText(exception.Message);
        }
        else
        {

            foreach (LocationObject locationobject in queryResult.Items)
            {

                if
(locationobject.GetType().Equals(typeof(LocationPlace))) {

                    place = (LocationPlace)locationobject;

                    place.Refresh(new LocationObjectFetchCallback());

                    lock (obj)
                    {
                        Monitor.Wait(obj);
                    }

                    Logger.Debug(TAG, place.Name +"  " +
place.HasChildren);
                }
                else
if(locationobject.GetType().Equals(typeof(LocationDevice)))
                {

                    device = (LocationDevice)locationobject;

                    device.Refresh(new LocationObjectFetchCallback());

                    lock (obj)
                    {
                        Monitor.Wait(obj);
                    }

                    Logger.Debug(TAG, device.Name + "  ");
                }
                else if
(locationobject.GetType().Equals(typeof(LocationAsset)))
                {
```

```
                    asset = (LocationAsset)locationobject;

                    asset.Refresh(new LocationObjectFetchCallback());

                    lock (obj)
                    {
                        Monitor.Wait(obj);
                    }

                    Logger.Debug(TAG, asset.Name + "  ");
                }

            }
        }
    }
}

private class LocationObjectFetchCallback : Java.Lang.Object,
ILocationObjectFetchCallback
{

    public void OnComplete(LocationObject locationObject,
ServiceProxyException exception)
    {
        if (mProgressDialog != null && mProgressDialog.IsShowing)
        {
            mProgressDialog.Dismiss();
        }

        if (exception != null)
        {
            Logger.Debug(TAG, exception.Message);

            setText(exception.Message);
        }
        else if(locationObject != null)
        {
            Logger.Debug(TAG, locationObject.Name );

        }

        lock (obj)
        {
            Monitor.PulseAll(obj);
        }

    }
}
```

Storage

Initialization

```
Storage storage =
(Storage)mobileBackend.GetServiceProxy(Class.FromType(typeof(Storage)));
```

Getting a Collection

```
StorageCollection storageCollection =
storage.GetStorageCollection("FullCoverage_Private");
StorageObject storageObject = storageCollection.Get("ab911696-7e61-4fcd-
a244-b26adb6183ba");
string str =
Encoding.UTF8.GetString(Decompress(storageObject.GetPayloadBytes()));
```

Getting an Object

```
storageObject = storageCollection.Get("d4400472-b912-4f7a-b4f5-
e32523e5c1f3");
Logger.Debug(TAG, "Storage Object:  " + storageObject.DisplayName);
```

Getting All Objects

```
IList<StorageObject> list =  storageCollection.Get(0, 100, true);

IEnumerator<StorageObject> iEnumerator = list.GetEnumerator();
while(iEnumerator.MoveNext()){
    storageObject = iEnumerator.Current;
    Logger.Debug(TAG, "Storage Object:  " + storageObject.DisplayName);

}
```

Uploading a Text File

```
Java.Lang.String str = new Java.Lang.String("This is sample txt file");

storageObject = new StorageObject("textfile.txt");
storageObject.SetPayload(str.GetBytes(), "text/plain");
storageCollection.Put(storageObject);
```

Uploading an Image

```
System.IO.Stream imageBytes = getFileFromAssets("mcs_oracle.png");

storageObject = new StorageObject("mcs_oracle.png", imageBytes, "image/
jpeg");
var imagePosted = storageCollection.Post(storageObject);
```

Decompressing

```
static byte[] Decompress(byte[] data)
{
    using (var compressedStream = new MemoryStream(data))
    using (var zipStream = new GZipStream(compressedStream,
CompressionMode.Decompress))
    using (var resultStream = new MemoryStream())
    {
        zipStream.CopyTo(resultStream);
        return resultStream.ToArray();
    }
}
```

Notifications

Initialization

```
LocalBroadcastManager.GetInstance(context)
        .RegisterReceiver(new MBroadcastReceiver(),
        new IntentFilter(NotificationsConfig.RegistrationComplete));

Notifications notifications =
(Notifications)mobileBackend.GetServiceProxy(Java.Lang.Class.FromType(typeo
f(Notifications)));
```

Registering for Notifications

```
bool result = notifications.Initialize(context, "Sender ID");
```

Broadcast Receiver

```
private class MBroadcastReceiver : BroadcastReceiver
{

    public override void OnReceive(Context context, Intent intent)
    {

        if (mProgressDialog != null && mProgressDialog.IsShowing)
        {
            mProgressDialog.Dismiss();
        }

        ISharedPreferences prefs =
PreferenceManager.GetDefaultSharedPreferences(context);
        bool sentToken =
prefs.GetBoolean(NotificationsConfig.SentTokenToServer, false);
        if (sentToken)
        {
            Logger.Debug(TAG, "Token retrieved and sent to server! App can
use GCM");
```

```
        }
        else
        {
            Logger.Debug(TAG, "An error occurred while either fetching the
InstanceID");
        }

    }
}
```

Analytics

Initialization

```
static Analytics analyticsAgent =
        (Analytics)mobileBackend
        .GetServiceProxy(Class.FromType(typeof(Analytics)));
analyticsAgent.SetContext(activity);
```

Logging an Event

```
if (analyticsAgent != null)
    analyticsAgent.LogEvent("This is Event No. : " + i);
```

Setting Context Location

```
analyticsAgent.SetContextLocation("India", "Telangana", "Hyderabad",
"500081");
```

Flushing an Event

```
analyticsAgent.Flush();
```

App Policies

Loading the App Config and Getting Policies

```
if (mobileBackend != null)
{
     mobileBackend.LoadAppConfig(new AAppConfigCallBack());
     mProgressDialog = ProgressDialog.Show(activity, "Please Wait", "App
Config is being loaded.");

     lock(obj){
         Monitor.Wait(obj);
     }

     AppConfig oMCAppConfig = mobileBackend.AppConfig;
```

```
//Getting String:

     string str = oMCAppConfig.GetString("Test_String", "No value
configured");

     setText("AppConfig: String:  " + str);


//Getting Number
     Number number = oMCAppConfig.GetNumber ("Test_number", new
Java.Lang.Double(1.0));

     setText("AppConfig: Number:  " + number);


//Getting boolean
     bool boolean = oMCAppConfig.GetBoolean("Test_Boolean", false);

     setText("AppConfig: Boolean:  " + boolean);

}

private class AAppConfigCallBack : AppConfigCallback
{
     public override void OnResult(Oracle.Cloud.Mobile.Utils.McsError
error, AppConfig config)
  {
      if (mProgressDialog != null && mProgressDialog.IsShowing)
      {
          mProgressDialog.Dismiss();
      }

      lock(obj){
          Monitor.PulseAll(obj);
      }
  }
}
```

Calling Custom APIs Using the SDK for Xamarin Android

The SDK provides the `CustomHttpResponse` class, the `GenericCustomCodeClientCallBack` interface, and the `InvokeCustomCodeJSONRequest` method in the authorization classes to simplify the calling of custom APIs in MCS. You can call a REST method (GET, PUT, POST, or DELETE) on an endpoint where the request payload is JSON or empty and the response payload is JSON or empty.

You use `GenericCustomCodeClientCallBack` to create a handler for the response (which is returned in the form of a `CustomHttpResponse` object.)

Then, to call the custom API, you call `InvokeCustomCodeJSONRequest(GenericCustomCodeClientCallBack restClientCallback, JSONObject data, String functionName, RestClient.HttpMethod httpMethod)` on your `Authorization` object.

To make a call to a custom API endpoint, you could use something like this:

```
IAuthorizationAgent mAuthorization =
MobileBackendManager.Manager.GetDefaultMobileBackend(context).Authorization
;

mAuthorization.Authenticate(mActivity, "user1", "pass1", new
AuthorizationCallback());

........
// after the user successfully authenticates, make a call to the custom
API endpoint
mAuthorization.InvokeCustomCodeJSONRequest(new
GenericCustomCodeClientCallBack(), null, "TaskApi/tasks",
RestClient.HttpMethod.Get);


private class GenericCustomCodeClientCallBack : Java.Lang.Object,
IGenericCustomCodeClientCallBack
{
    public void RequestCompleted(CustomHttpResponse response, JSONObject
data, Java.Lang.Exception exception)
    {
        Logger.Debug(TAG, response.HttpStatus + "");
    }
}
```

11

Xamarin iOS Applications

If you use the Xamarin platform to develop iOS apps, you can use the SDK that Oracle Mobile Cloud Service (MCS) provides for Xamarin iOS apps. This SDK simplifies authentication with MCS and provides native wrapper classes for MCS platform APIs.

Getting the SDK for Xamarin iOS

To get the MCS client SDK for Xamarin iOS, go to the Oracle Technology Network's MCS download page.

To use this SDK, you should have the following software on your system:

- Microsoft Visual Studio, with support for Xamarin development.
- Xcode 9.1 or later and iphoneos 11.0.

  See http://www.oracle.com/technetwork/java/javase/downloads/index.html for JDK downloads.

Adding the SDK to a Xamarin iOS Project

1. If you haven't already done so, extract the contents from the SDK zip.
2. In Visual Studio, create a Visual C# iOS app.
3. Add the SDK's DLL file to your app by right-clicking the project's **References** node and selecting **Edit References**, clicking the **.NET Assembly** tab, and then browsing to the `IOS.dll` file in the extracted SDK zip.
4. Add the configuration file to the app by right-clicking the project's root node and selecting **Add > Add Files** and then navigating to the SDK's `OMC.plist` file.
5. Select the node for `OMC.plist` so that it's properties are displayed in the Properties pane. Then make sure that the **Build Action** property is set to `BundleResource`.
6. Add the `SynchStore.momd` folder to the app by right-clicking the project's root node and selecting **Add > Add Existing Folder** and then navigating to the SDK's `SynchStore` folder.
7. For all of the files in the `SynchStore.momd` folder, make sure that the **Build Action** property is set to `BundleResource`.
8. Open `OMC.plist` and fill in the environment details for the mobile backend that the app will be using. See Configuring SDK Properties for Xamarin iOS.

Configuring SDK Properties for Xamarin iOS

To use the SDK in a Xamarin iOS project, you need to add the `OMC.plist` configuration file to the app and fill it in with environment details for your mobile backend. In turn, the SDK classes use this information to access the mobile backend and construct HTTP headers for REST calls made to APIs.

You package the configuration file in the root of your app's main bundle.

Here's an example of the contents of the `OMC.plist` file. Pay careful attention to the hierarchy of elements.

Key	Type	Value
▼ Root	Dictionary	(2 items)
▼ mobileBackends	Dictionary	(1 item)
▼ FixItFast_Customer	Dictionary	(4 items)
appKey	String	ebfbc8ea-9173-442b-8a5e-2fae63c64422
▼ authorization	Dictionary	(2 items)
authenticationType	String	oauth
▼ OAuth	Dictionary	(3 items)
tokenEndpoint	String	https://oam.oracle.com/oam/oauth2/tokens
clientID	String	ddb7ff5a-0d86-4b4a-8164-ddad03734249
clientSecret	String	pFmzazXzNTBNVDyraQs7
baseURL	String	https://fif.cloud.oracle.com
default	Boolean	YES
logLevel	String	debug

Here's the source code for the same example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/
DTDs/PropertyList-1.0.dtd"
<plist version="1.0">
<dict>
  <key>mobileBackends</key>
  <dict>
    <key>FixItFast_Customer</key>
    <dict>
      <key>default</key>
      <true/>
      <key>baseURL</key>
      <string>https://fif.cloud.oracle.com</string>
      <key>appKey</key>
      <string>ebfbc8ea-9173-442b-8a5e-2fae63c64422</string>
      <key>authorization</key>
      <dict>
        <key>authenticationType</key>
        <string>OAuth</string>
        <key>OAuth</key>
        <dict>
          <key>tokenEndpoint</key>
          <string>https://oam.oracle.com/oam/oauth2/tokens</string>
          <key>clientID</key>
          <string>ddb7ff5a-0d86-4b4a-8164-ddad03734249</string>
          <key>clientSecret</key>
          <string>pFmzazXzNTBNVDyraQs7</string>
        </dict>
      </dict>
    </dict>
  </dict>
  <key>logLevel</key>
```

```
  <string>debug</string>
</dict>
</plist>
```

Here are the key entries in the `OMC.plist` file. You can obtain the necessary environment details from the Settings and Clients pages of the mobile backend.

- `mobileBackends` — a dictionary entry containing a nested dictionary for your mobile backend such as `FixItFast_Customer`. (When you call `OMCMobileBackend` in an app, you need to supply the value of that entry as a parameter to `OMCMobileBackendManager`.) That entry, in turn, contains entries for `appKey`, `baseURL`, `authenticationType`, `mobileBackendID`, `anonymousKey`, and, optionally, `networkConnectionTimeout`. See the example below.
- `baseURL` — The URL your application uses to connect to its mobile backend.
- `appKey` — The application key, which is a unique string assigned to your app when you register it as a client in MCS. See Registering an App as a Client in MCS. If you have not registered the app as a client in MCS, assign a placeholder value for this entry.
- `authorization` — Use this key to define the type of authentication the app will be using and specify the required credentials. The contents of the `authorization` key depend on the type of authentication.
  - `authenticationType` — Defines the type of authentication mechanism being used in your mobile application. Possible values are `OAuth` (for OAuth Consumer), `basic` (for HTTP Basic), `SSO`, `SSOTokenExchange` and `Facebook`. Include a dictionary for each supported authentication type with the required credentials as explained in the sections that follow.
- `networkConnectionTimeout` — (Optional) The network timeout for API calls, in seconds. Should you need to do any network performance tuning, you can add this property, though you should use it with care. Keep in mind that app responsiveness issues might be better addressed in the app design itself. The default timeout is 60 seconds.
- `logLevel` — Determines how much SDK logging is displayed in the app's console. The default value is `error`. Other possible values (in increasing level of detail) are `warning`, `info`, and `debug`. It is also possible to set the value to `none`.
- `logHTTPRequestBody` — When set to `true`, the SDK will also log the HTTP and HTTPS headers and body in the requests to MCS.
- `logHTTPResponseBody` — When set to `true`, the SDK will also log the HTTP and HTTPS headers and body in responses from MCS.
- `offlineAuthenticationEnabled` — If set to `true`, offline login will be allowed.

The contents and sub-elements of the `authorization` dictionary depend on what kind of authentication the app will be using.

OAuth Consumer

For OAuth, set the value of the `authenticationType` property to `OAuth` and fill in the OAuth credentials provided by the mobile backend.

- `tokenEndpoint` — The URL of the OAuth server your application goes to, to get its authentication token.

- `clientID` — The unique client identifier assigned to all applications when they're first created in your mobile backend.
- `clientSecret` — The unique secret string assigned to all applications when they're first created in your mobile backend.

The resulting `authorization` property might look something like this:

```
<key>authorization</key>
<dict>
    <key>authenticationType</key>
    <string>oauth</string>
      <key>OAuth</key>
      <dict>
        <key>tokenEndpoint</key>
        <string>https://oam.oracle.com/oam/oauth2/tokens</string>
        <key>clientID</key>
        <string>ddb7ff5a-0d86-4b4a-8164-ddad03734249</string>
        <key>clientSecret</key>
        <string>pFmzazXzNTBNVDyraQs7</string>
      </dict>
 </dict>
```

SSO

For SSO, set the value of the `authenticationType` property to `SSO` and fill in the OAuth credentials provided by the mobile backend. (For `tokenEndpoint`, you use the mobile backend's OAuth token endpoint.)

The resulting `authorization` property might look something like this:

```
<key>authorization</key>
<dict>
    <key>authenticationType</key>
    <string>SSO</string>
        <key>SSO</key>
        <dict>
      <key>tokenEndpoint</key>
      <string>https://oam-server.oracle.com/oam/oauth2/tokens</string>
      <key>clientID</key>
      <string>ddb7ff5a-0d86-4b4a-8164-ddad03734249</string>
      <key>clientSecret</key>
      <string>pFmzazXzNTBNVDyraQs7</string>
    </dict>
</dict>
```

SSO with a Third-Party Token

For SSO with a third-party token, set `authenticationType` to `SSOTokenExchange` and fill in the appropriate credentials.

The resulting `authorization` property might look something like this:

```
<key>authorization</key>
<dict>
```

```
    <key>authenticationType</key>
    <string>SSOTokenExchange</string>
    <key>SSOTokenExchange</key>
    <dict>
      <key>mobileBackendID</key>
      <string>ddb7ff5a-0d86-4b4a-8164-ddad03734249</string>
      <key>anonymousKey</key>

<string>UFJJTUVfREVDRVBUSUNPTl9NT0JJTEVfQU5PTllNT1VTX0FQUElEOnZrZWJxUmwuamE
wbTdu</string>
    </dict>
</dict>
```

HTTP Basic

For HTTP Basic authentication, set the value of the `authenticationType` property to `basic` and fill in the HTTP Basic credentials provided by the mobile backend.

- `mobileBackendID` — The unique identifier assigned to a specific mobile backend. It gets passed in an HTTP header in every REST call made from your application to MCS, to connect it to the correct mobile backend. When calling platform APIs, the SDK handles the construction of the `mobileBackendID` header for you.
- `anonymousKey` — When using HTTP Basic authentication, a unique string that allows your app to access APIs that don't require login. In this scenario, the anonymous key is passed to MCS instead of an encoded user name and password combination.

The resulting `authorization` property might look something like this:

```
<key>authorization</key>
<dict>
    <key>authenticationType</key>
    <string>Basic</string>
    <key>Basic</key>
    <dict>
      <key>anonymousKey</key>

<string>UFJJTUVfREVDRVBUSUNPTl9NT0JJTEVfQU5PTllNT1VTX0FQUElEOml6LmQxdTlCaWF
rd2Nz</string>
      <key>mobileBackendID</key>
      <string>4fb9cabd-d0e2-40f8-87b5-d2d44cdd7c68</string>
    </dict>
</dict>
```

Facebook

For Facebook, set the value of the `authenticationType` property to `Facebook` and fill in the HTTP Basic auth credentials provided by the mobile backend plus the `facebookAppID`.

The resulting `authorization` property might look something like this:

```
<key>authorization</key>
<dict>
    <key>authenticationType</key>
```

```
    <string>Facebook</string>
    <key>Facebook</key>
    <dict>
      <key>mobileBackendID</key>
      <string>11d1fc49-7574-4b24-82f3-74a3720ce154</string>
      <key>anonymousKey</key>

<string>UFJJTUVfREVDRVBUSUNPTl9NT0JJTEVfQU5PTllNT1VTX0FQUElEOml6LmQxdTlCaWFrd2Nz</string>
      <key>facebookAppID</key>
      <string>154198719279</string>
    </dict>
</dict>
```

Loading a Mobile Backend's Configuration into a Xamarin iOS App

For any calls to MCS APIs using the iOS SDK to successfully complete, you need to have the mobile backend's configuration loaded from the app's `OMC.plist` file. You do this using the `OMCMobileBackend` class:

```
OMCMobileBackend oMCMobileBackend =
OMCMobileBackendManager.SharedManager.MobileBackendForName("MBE_FullCoverage");
```

Authenticating and Logging In Using the SDK for Xamarin iOS

Here is some sample code that you can use for authentication through MCS in your iOS apps.

Oauth

You can use the following method to handle a user logging in with a user name and password.

```
OMCAuthorization authorization = oMCMobileBackend.Authorization;
authorization.AuthenticationType = OMCAuthenticationType.OAuth;
authorization.Authenticate(username.Text, password.Text);
```

This method terminates the connection to MCS and clears the user name and password from the iOS keychain:

```
authorization.Logout(HandleOMCAuthorizationLogoutCompletionBlock);

void HandleOMCAuthorizationLogoutCompletionBlock(NSError nsError)
{
    if(nsError == null){
        Console.WriteLine("Logout success!");
```

```
    }
}
```

HTTP Basic

You can use the following method to handle a user logging in with a user name and password.

```
OMCAuthorization authorization = oMCMobileBackend.Authorization;
authorization.AuthenticationType = OMCAuthenticationType.HTTPBasic;
authorization.Authenticate(username.Text, password.Text);
```

This method terminates the connection to MCS and clears the user name and password from the iOS keychain:

```
authorization.Logout(HandleOMCAuthorizationLogoutCompletionBlock);

void HandleOMCAuthorizationLogoutCompletionBlock(NSError nsError)
{
    if(nsError == null){
        Console.WriteLine("Logout success!");
    }
}
```

SSO

For apps that allow login through enterprise SSO, use:

```
OMCAuthorization oMCAuthorization = oMCMobileBackend.Authorization;
oMCAuthorization.AuthenticationType = OMCAuthenticationType.Sso;
oMCAuthorization.AuthenticateSSO(this, true,
HandleOMCAuthorizationAuthCompletionBlock);
```

SSO with a Third-Party Token

First, your app needs to get a token from the third-party token issuer. The way you can obtain the token varies by issuer. For detailed information on obtaining third-party tokens and configuring identity providers in MCS, see Third-Party SAML and JWT Tokens.

Once you have the token, use it to authenticate. The example below checks to see if the token is already stored in MCS before logging in again.

> **Note:**
>
> The default expiration time for storing a third-party token in MCS is 6 hours. You can adjust this time by changing the `Security_TokenExchangeTimeoutSecs` policy.

```
OMCAuthorization oMCAuthorization = oMCMobileBackend.Authorization;
oMCAuthorization.AuthenticationType =
```

```
OMCAuthenticationType.SSOTokenExchange;
NSError nSError =  oMCAuthorization.AuthenticateSSOTokenExchange(Token);

oMCAuthorization.AuthenticateSSOTokenExchange(Token,
HandleOMCAuthorizationAuthCompletionBlock);

oMCAuthorization.AuthenticateSSOTokenExchange(Token, true,
HandleOMCAuthorizationAuthCompletionBlock);

oMCAuthorization.AuthenticateSSOTokenExchange(Token, true);


bool iSLoaded = oMCAuthorization.LoadSSOTokenExchange;

oMCAuthorization.ClearSSOTokenExchange();
```

Facebook

For apps that allow login through Facebook, use:

```
oMCAuthorization.AuthenticationType = OMCAuthenticationType.Facebook;
oMCAuthorization.AuthenticateSocial(HandleOMCAuthorizationAuthCompletionBlo
ck);
```

If you haven't already set up the app and its mobile backend to use Facebook as the identity provider, see Facebook Login in MCS.

Calling Platform APIs Using the SDK for Xamarin iOS

Once the mobile backend's configuration info is loaded into the app and you have made a call to get the mobile backend, you can make calls to SDK classes to access platform features.

Here are some code snippets that illustrate how to access these APIs with the SDK.

User Management

Getting a User

```
OMCAuthorization oMCAuthorization = oMCMobileBackend.Authorization;
oMCAuthorization.GetCurrentUser(HandleOMCUserRegistrationCompletionBlockWit
hUser);
void HandleOMCUserRegistrationCompletionBlockWithUser(NSError nSError,
OMCUser oMCUser)
{
    if(nSError == null){
        output.Text = user.FirstName + " User details have been fetched
successfully";
    }
}
```

Updating a User

```
user.SetValueForKey(new NSNumber(26),new NSString("age"));
user.SetValueForKey(new NSString("address"), new NSString("india"));
oMCAuthorization.UpdateCurrentUser(user,HandleOMCUserRegistrationCompletion
Block);

void HandleOMCUserRegistrationCompletionBlock(NSError nSError)
{
    if (nSError == null)
    {
        //user = oMCUser;
        if (user != null)
        {
            if (username.Text == null)
            {
                username.Text = "Welcome " + user.FirstName;
            }
            else output.Text = user.FirstName + " User details have been
fetched successfully";
        }
    }
    else
    {
        output.Text = nSError.ToString();
    }
}
```

Location

Initialization

```
OMCLocation oMCLocation = oMCMobileBackend.Location;
```

Queries for Places, Devices, and Assets

```
private static OMCLocation oMCLocation;
private static OMCLocationPlace oMCLocationPlace;
private static OMCLocationDevice oMCLocationDevice;
private static OMCLocationAsset oMCLocationAsset;

oMCLocation = oMCMobileBackend.Location;
OMCLocationPlaceQuery oMCLocationPlaceQuery = oMCLocation.BuildPlaceQuery;
oMCLocationPlaceQuery.Name = "West";
oMCLocationPlaceQuery.ExecuteWithCompletionHandler(completionHandler);
OMCLocationAssetQuery oMCLocationAssetQuery = oMCLocation.BuildAssetQuery;
```

```
oMCLocationAssetQuery.Name = "joe";
oMCLocationAssetQuery.ExecuteWithCompletionHandler(completionHandler);

OMCLocationDeviceQuery oMCLocationDeviceQuery =
oMCLocation.BuildDeviceQuery;
oMCLocationDeviceQuery.Name = "Beacon";
oMCLocationDeviceQuery.ExecuteWithCompletionHandler(completionHandler);
```

Fetching

```
Action<OMCLocationObjectQueryResult, NSError> completionHandler = new
Action<OMCLocationObjectQueryResult,
NSError>((OMCLocationObjectQueryResult arg1, NSError arg2) =>
{
    if (arg2 == null)
    {
        OMCLocationObject[] LocationObjects = arg1.Items;
        OMCLocationPlace oMCLocationPlace;
        OMCLocationDevice oMCLocationDevice;
        OMCLocationAsset oMCLocationAsset;

        foreach (OMCLocationObject locationObject in LocationObjects)
        {
            Console.WriteLine("Location Object " +
locationObject.GetType() + "--> " + i + " is: " +
locationObject.ToString());

            if(locationObject.GetType().Equals(typeof(OMCLocationPlace))){

                oMCLocationPlace = (OMCLocationPlace)locationObject;

                oMCLocation.PlaceWithID(oMCLocationPlace.Id_,
placeCompletionHandler);
            }
            else if
(locationObject.GetType().Equals(typeof(OMCLocationDevice)))
            {

                oMCLocationDevice = (OMCLocationDevice)locationObject;

                oMCLocation.DeviceWithID(oMCLocationDevice.Id_,
deviceCompletionHandler);
            }
            else if
(locationObject.GetType().Equals(typeof(OMCLocationAsset)))
            {

                oMCLocationAsset = (OMCLocationAsset)locationObject;

                oMCLocation.AssetWithID(oMCLocationAsset.Id_,
assetCompletionHandler);
            }
        }
    }
```

```
});

private static void assetCompletionHandler(OMCLocationAsset arg0, NSError
arg1)
{
    if (arg1 == null)
    {
        Console.WriteLine("Location Asset " + arg0.ToString());
    }
}

private static void deviceCompletionHandler(OMCLocationDevice arg0,
NSError arg1)
{
    if (arg1 == null)
    {
        Console.WriteLine("Location Device " + arg0.ToString() );

    }
}

private static void placeCompletionHandler(OMCLocationPlace arg0, NSError
arg1)
{
    if(arg1 == null){
        Console.WriteLine("Location Place " + arg0.ToString());
    }
}
```

Refreshing

```
Action<OMCLocationObjectQueryResult, NSError> completionHandler = new
Action<OMCLocationObjectQueryResult,
NSError>((OMCLocationObjectQueryResult arg1, NSError arg2) =>
{
    if (arg2 == null)
    {
        OMCLocationObject[] LocationObjects = arg1.Items;


        foreach (OMCLocationObject locationObject in LocationObjects)
        {
            Console.WriteLine("Location Object " +
locationObject.GetType() + "--> " + i + " is: " +
locationObject.ToString());

            if(locationObject.GetType().Equals(typeof(OMCLocationPlace))){

                oMCLocationPlace = (OMCLocationPlace)locationObject;


oMCLocationPlace.RefreshWithCompletionHandler(placeCompletionHandler);
            }
            else if
```

```
(locationObject.GetType().Equals(typeof(OMCLocationDevice)))
            {

                oMCLocationDevice = (OMCLocationDevice)locationObject;


oMCLocationDevice.RefreshWithCompletionHandler(deviceCompletionHandler);
            }
            else if
(locationObject.GetType().Equals(typeof(OMCLocationAsset)))
            {

                oMCLocationAsset = (OMCLocationAsset)locationObject;


oMCLocationAsset.RefreshWithCompletionHandler(assetCompletionHandler);

            }
        }
    }
});

private static void placeCompletionHandler(NSError arg0)
{
    if (arg0 == null)
    {
        Console.WriteLine("Location Place " + oMCLocationPlace.ToString());
    }
}

private static void deviceCompletionHandler(NSError arg0)
{
    if (arg0 == null)
    {
        Console.WriteLine("Location Device " +
oMCLocationDevice.ToString());

    }
}

private static void assetCompletionHandler(NSError arg0)
{
    if (arg0 == null)
    {
        Console.WriteLine("Location Asset " + oMCLocationAsset.ToString());
    }
}
```

Storage

Initialization

```
OMCStorage oMCStorage = oMCMobileBackend.Storage;
```

Getting a Collection

```
OMCStorageCollection oMCStorageCollection =
oMCStorage.GetCollection("SharedCollection");
```

Getting an Object

```
oMCStorageObject =  collection.Get("Object Id");

System.Console.WriteLine("Storage Object1:  " +
oMCStorageObject.ToString());
```

Getting All Objects from a Collection

```
NSMutableArray nSMutableArray = collection.Get(0, 100, true);
OMCStorageObject oMCStorageObject;
if (nSMutableArray != null && nSMutableArray.Count > 0)
{
    for (uint i = 0; i < nSMutableArray.Count; i++){
        oMCStorageObject = nSMutableArray.GetItem<OMCStorageObject>(i);
        System.Console.WriteLine("Storage Object1:  " +
oMCStorageObject.ToString());
    }
}
```

Uploading a Text File

```
NSData text = "This is a sample Text file";
OMCStorageObject txtFile = new OMCStorageObject("Mytext.txt", text, "text/
plain");

collection.Put(txtFile);
```

Uploading an Image File

```
UIImage image = new UIImage("MyImage.png");
NSData data = image.AsPNG();
OMCStorageObject imageFile = new OMCStorageObject("MyImage", data, "image/
png");
collection.Put(imageFile);
```

Notifications

Initialization

```
OMCNotifications oMCNotifications = oMCMobileBackend.Notifications;
```

Registering for Notifications

```
oMCNotifications.RegisterForNotifications(appDelegate.DeviceToken,
HandleOMC_Notifications_SuccessBlock, HandleOMC_Notifications_ErrorBlock);


void HandleOMC_Notifications_SuccessBlock(NSHttpUrlResponse
nSHttpUrlResponse)
{
    if (nSHttpUrlResponse != null)
    {
        Console.WriteLine("Response from notification Server:  " +
nSHttpUrlResponse.StatusCode);

    }
}

void HandleOMC_Notifications_ErrorBlock(NSError nSError)
{
    if (nSError != null)
    {
        Console.WriteLine("Error in fetching mobiel file:  " +
nSError.LocalizedDescription);
    }
}
```

AppDelegate code

```
public NSData DeviceToken = string.Empty;

public override void RegisteredForRemoteNotifications(UIApplication
application, NSData deviceToken)
{
    DeviceToken = deviceToken; // Do something to storage deviceToken.

    Console.WriteLine("Device Token:  " + DeviceToken.ToString());
}

public override void FailedToRegisterForRemoteNotifications(UIApplication
application, NSError error)
{
    Console.WriteLine("FailedToRegisterForRemoteNotifications.. :(");
}

public override void DidReceiveRemoteNotification(UIApplication
application, NSDictionary userInfo, Action<UIBackgroundFetchResult>
completionHandler)
{
    ProcessNotification(userInfo, false);

}

void ProcessNotification(NSDictionary options, bool fromFinishedLaunching)
```

```
{
    // Check to see if the dictionary has the aps key.  This is the
notification payload you would have sent
    if (null != options && options.ContainsKey(new NSString("aps")))
    {
        //Get the aps dictionary
        NSDictionary aps = options.ObjectForKey(new NSString("aps")) as
NSDictionary;
        string alertTitle = string.Empty;
        string alert = string.Empty;
        string sound = string.Empty;
        int badge = -1;

        //Extract the alert text
        // NOTE: If you're using the simple alert by just specifying
        // "  aps:{alert:"alert msg here"}  ", this will work fine.
        // But if you're using a complex alert with Localization keys,
etc.,
        // your "alert" object from the aps dictionary will be another
NSDictionary.
        // Basically the JSON gets dumped right into a NSDictionary,
        // so keep that in mind.
        if (aps.ContainsKey(new NSString("alert")))
            alert = (aps[new NSString("alert")] as NSString).ToString();
        if (aps.ContainsKey(new NSString("alert")))
              alert = (aps[new NSString("alert")] as NSString).ToString();

          if (options.ContainsKey(new NSString("alertTitle")))
              alertTitle = (options[new NSString("alertTitle")] as
NSString).ToString();

          //Extract the sound string
          if (aps.ContainsKey(new NSString("sound")))
              sound = (aps[new NSString("sound")] as NSString).ToString();

          //Extract the badge
          if (aps.ContainsKey(new NSString("badge")))
          {
              string badgeStr = (aps[new NSString("badge")] as
NSObject).ToString();
              int.TryParse(badgeStr, out badge);
          }

        if (!fromFinishedLaunching)
        {
            //Manually show an alert
            if (!string.IsNullOrEmpty(alert))
            {
                UIAlertView avAlert = new UIAlertView("Notification",
alert, null, "OK", null);
                avAlert.Show();
            }
        }
    }
}
```

```
public override void ReceivedRemoteNotification(UIApplication application,
NSDictionary userInfo)
{
    ProcessNotification(userInfo, false);
}

public override bool FinishedLaunching(UIApplication application,
NSDictionary launchOptions)
{

    Window = new UIWindow(UIScreen.MainScreen.Bounds);

    ViewController viewController = new ViewController("LoginScreen",
null);
    Window.RootViewController = viewController;
    Window.MakeKeyAndVisible();

    if (UIDevice.CurrentDevice.CheckSystemVersion(8, 0))
    {
        var notificationSettings =
UIUserNotificationSettings.GetSettingsForTypes(
                                      UIUserNotificationType.Alert |
UIUserNotificationType.Badge | UIUserNotificationType.Sound, null
                                    );


UIApplication.SharedApplication.RegisterUserNotificationSettings(notificati
onSettings);
        UIApplication.SharedApplication.RegisterForRemoteNotifications();
    }
    else
    {
        //==== register for remote notifications and get the device token
        // set what kind of notification types we want
        UIRemoteNotificationType notificationTypes =
UIRemoteNotificationType.Alert | UIRemoteNotificationType.Badge;
        // register for remote notifications

UIApplication.SharedApplication.RegisterForRemoteNotificationTypes(notifica
tionTypes);
    }

    return true;

}
```

Analytics

Initialization

```
OMCAnalytics oMCAnalytics = oMCMobileBackend.Analytics;
```

Logging an Event

```
oMCAnalytics.LogEvent("this is test event "+ i +" from xamarin");
```

Setting Context Location

```
oMCAnalytics.SetContextLocationCountry("india", "Telangana", "Hyderabad",
"500081");
```

Flushing an Event

```
oMCAnalytics.Flush();
```

App Policies

Loading the App Config and Getting Policies

```
oMCMobileBackend.AppConfigWithCompletionHandler(HandleOMCAppConfigCompletio
nBlock);

lock(obj){
 Monitor.Wait(obj);
}

OMCAppConfig oMCAppConfig = oMCMobileBackend.AppConfig;


//Getting String

String str = oMCAppConfig.StringForProperty("Test_String", "No value
configured");

Console.WriteLine("oMCAppConfig: String:  " + str);


//Getting Number
NSNumber number = oMCAppConfig.NumberForProperty("Test_number", -1);

Console.WriteLine("oMCAppConfig: Number:  " + number);


//Getting Boolean
Boolean boolean = oMCAppConfig.BooleanForProperty("Test_Boolean", false);

Console.WriteLine("oMCAppConfig: Boolean:  " + boolean.ToString());



void HandleOMCAppConfigCompletionBlock(OMCAppConfig oMCAppConfig, NSError
arg1)
{
```

```
  if(arg1 == null){
      Console.WriteLine("oMCAppConfig: " + oMCAppConfig.ToString());
  }
}
```

Calling Custom APIs Using the SDK for Xamarin iOS

The SDK provides the `CustomCodeClient` class to simplify the calling of custom APIs in MCS. You can call a REST method (GET, PUT, POST, or DELETE) on an endpoint where the request payload is JSON or empty and the response payload is JSON or empty.

Using this class, you invoke a REST method (GET, PUT, POST, or DELETE) on an endpoint where the request payload is JSON or empty and the response payload is JSON or empty.

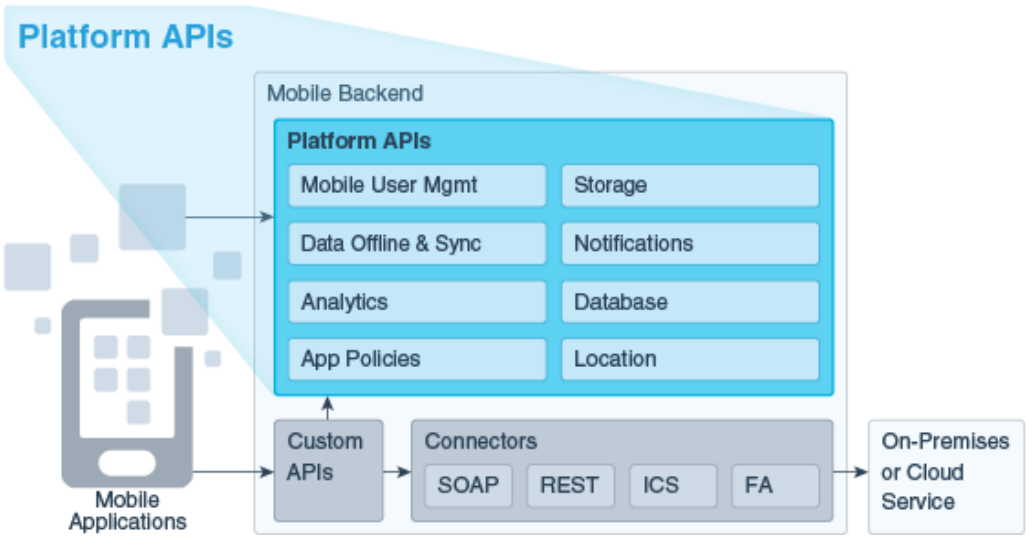
In addition you can provide a completion handler to be called when the method invocation is complete (meaning that the handler runs asynchronously).

Use of `CustomCodeClient` might look something like this:

```
oMCMobileBackend.CustomCodeClient.InvokeCustomRequest("mcs_examples_sync_sa
lesplus/reminders", "get", null, HandleOMCCustomRequestCompletionHandler);

void HandleOMCCustomRequestCompletionHandler(NSError arg0,
NSHttpUrlResponse arg1, NSObject nSObject)
{
    if (nSObject != null)
    {
        System.Console.WriteLine("response object:  " +
nSObject.ToString());
    }
}
```

Part III
Platform APIs

Oracle Mobile Cloud Service (MCS) comes with platform APIs built-in to provide functionality that is commonly required in mobile apps. You can configure these services directly within the MCS web interface and have your apps call those services using REST APIs. Continue reading to learn how to use these services and the APIs that access them.

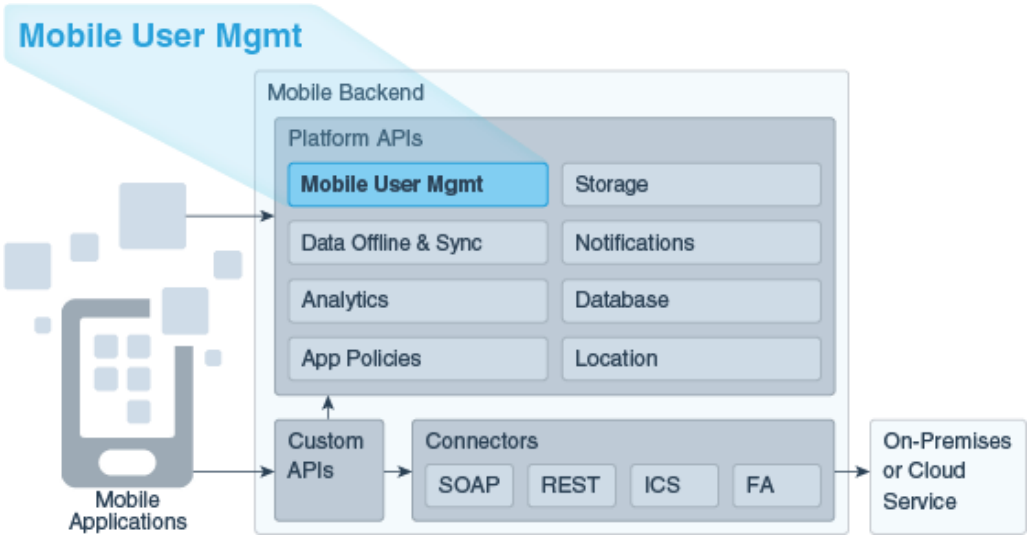


- Mobile User Management
- Location
- Storage
- Data Offline and Sync
- Notifications
- Analytics
- Database
- App Policies

12

Mobile User Management

As a mobile app developer, you can use the Mobile Users API to get information about the currently authenticated mobile, virtual, or social user. You also can use this API to update the current mobile user's custom properties. These are the properties that you've have added to the realm that the member belongs to. In addition, you can use the Mobile Users Extended API to retrieve the currently authenticated mobile or virtual user's roles.



We'll show how to make direct REST calls to these APIs. You can learn more about the APIs at REST APIs for Oracle Mobile Cloud Service.

You also can call this API from custom code, as shown in Accessing the Mobile Users API from Custom Code.

User Types

The information that the API returns depends on what type of user you are inquiring about. Here are the types of users:

- **Mobile User**: A member who's been added to the realm that's associated with the backend, as described in Set Up Mobile Users, Realms and Roles.
- **Virtual User**: These users pass a third-party token for authorization as described in Enterprise Single Sign-On in MCS.
- **Social User**: These users have logged into the app from Facebook, as described in Facebook Login in MCS.

Getting User Information

If your app needs user information, such as user name or first and last name, you can call the Mobile Users API to get that information.

You have two options for getting a user's profile:

- You can make a direct REST call as described in this topic and detailed in REST APIs for Oracle Mobile Cloud Service.
- You can call the ums.getUser(options, httpOptions) method from a custom API implementation.

To get the currently authorized user's profile via a direct REST call, send a `GET` request to `/mobile/platform/users/~`. Here's an example of using cURL to send the request:

```
curl -i \
-X GET \
-u joe.doe@example.com:mypass \
-H "Oracle-Mobile-Backend-ID: ABCD9278-091f-41aa-9cb2-184bd0586fce" \
https://fif.cloud.oracle.com/mobile/platform/users/~
```

The contents of the response body depends on the user type:

- When the user is a mobile user, the response contains the user name, first name, last name, and email address as well as the custom properties that were added to the realm that the user belongs to.
- When the user is a virtual user, the response contains the user name.
- When the user is a social user, the response contains the user's ID, identity provider, and access token.

Here's an example of a response for a mobile user:

```
{
  "username": "joe.doe@example.com",
  "firstName": "Joe",
  "lastName": "Doe",
  "email": "joe.doe@example.com",
  "locale": "en",
  "age": "39",
  "workPhone": "+19195550100",
  "mobilePhone": "+19195550101",
  "otherPhone": "+19195550102",
  "avatar": "DERFSKJAKJLSAJFLKASJDFLKADJF",
  "links": {
    { "rel": "canonical",
      "href": "/mobile/platform/users/~"
    }
  }
}
```

Here's an example of a response for a virtual user:

```
{
  "username": "username"
}
```

Here's an example of a response for a social (Facebook) user:

```
{
  "username": "1 :623:165",
  "mobileExtended": {
      "identityProvider": {
          "facebook": {
              "accessToken":"CAAI...YZD"
          }
      }
  }
}
```

For mobile users, you can limit the response to specific properties by adding a query string to the endpoint, such as `fields=firstName,lastName`. This argument is ignored if the user is a virtual or social user. For example, this command requests the `locale` property:

```
curl -i \
-X GET \
-u joe.doe@example.com:mypass \
-H "Oracle-Mobile-Backend-ID: ABCD9278-091f-41aa-9cb2-184bd0586fce" \
https://fif.cloud.oracle.com/mobile/platform/users/~?fields=locale
```

The response includes only the requested properties. For example:

```
{
  "locale": "en"
}
```

Getting User Roles

The Mobile Users Extended API lets you get a mobile or virtual user's roles in addition to the same information that you can get from the Mobile Users API. You can't use this API to get social user roles.

To learn how to get a user's roles using custom code, see ums.getUserExtended(options, httpOptions).

To get the roles via a direct Mobile Users Extended REST call, you make the same request as you would with the Mobile Users API, but you use the `/mobile/platform/extended` endpoint instead. For example:

```
curl -i \
-X GET \
-u joe.doe@example.com:mypass \
```

```
-H "Oracle-Mobile-Backend-ID: ABCD9278-091f-41aa-9cb2-184bd0586fce" \
https://fif.cloud.oracle.com/mobile/platform/extended/users/~
```

Here's an example of a response for a mobile user:

```
{
   "lastName":"Doe",
   "username":"joe.doe@example.com",
   "email":"joe.doe@example.com",
   "roles":[
      "Customer",
      "Trial"
   ],
  "links":[
        {
           "rel":"canonical",
           "href":"/mobile/extended/platform/users/joe"
        },
        {
           "rel":"self",
           "href":"/mobile/extended/platform/users/joe"
        }
  ],
   "firstName":"Joe"
}
```

Updating Mobile User Custom Properties

You can update a mobile user's custom properties. These are the properties that have been added to the user schema for the user's realm. You can't update the standard identity properties (`username`, `firstName`, `lastName`, and `email`).

To learn how to update a mobile user's custom properties from custom code, see ums.updateUser(fields, options, httpOptions).

To update a mobile user's custom properties via a direct REST call, send a `PATCH` or `PUT` to `/mobile/platform/users/~`. Include the properties with their new values in the body of the request. For example:

```
curl -i \
-X PUT \
-u joe.doe@example.com:mypass \
-d users.json \
-H "Content-Type: application/json; charset=utf-8" \
-H "Oracle-Mobile-Backend-ID: ABCD9278-091f-41aa-9cb2-184bd0586fce" \
https://fif.cloud.oracle.com/mobile/platform/users/~
```

Here's an example of the request body:

```
{
  "locale": "en_US",
```

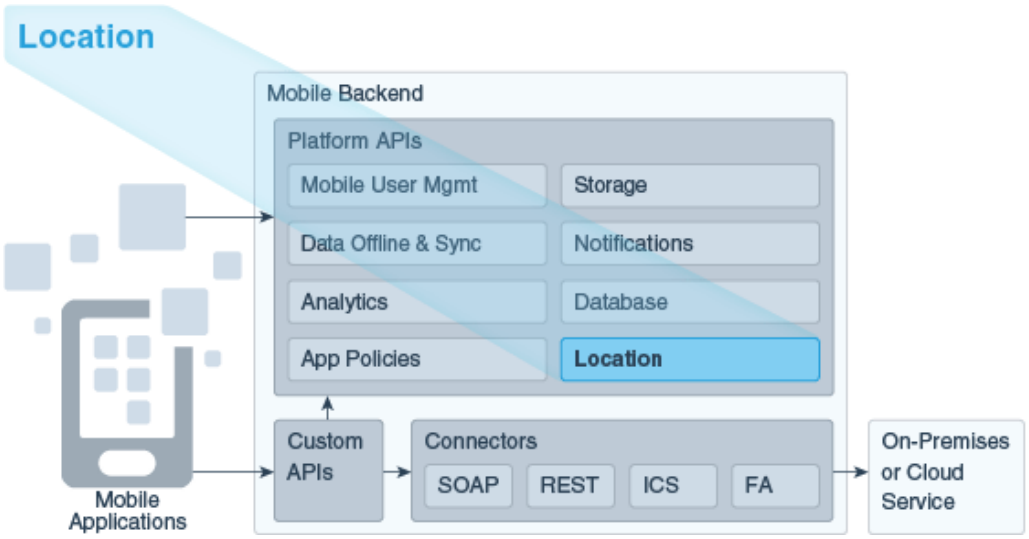
```
  "age": "40"
}
```

The response includes all the properties. For example:

```
{
  "username": "joe.doe@example.com",
  "firstName": "Joe",
  "lastName": "Doe",
  "email": "joe.doe@example.com",
  "locale": "en_US",
  "age": "40",
  "workPhone": "+19195550100",
  "mobilePhone": "+19195550101",
  "otherPhone": "+19195550102",
  "avatar": "DERFSKJAKJLSAJFLKASJDFLKADJF",
  "links": {
    { "rel": "canonical",
      "href": "/mobile/platform/users/~"
    }
  }
}
```

13
Location

As a mobile app developer, you can use the Oracle Mobile Cloud Service (MCS) Location API to access details about location devices, places and assets that have been registered in MCS.



What Can I Do With Location?

Users today expect information to be presented based on their current situation and individual needs and preferences. One of the most important contextual data points is location. The impact of location-aware mobile apps on users and businesses is growing faster every day.

- Everyone uses navigation apps for location data, including getting directions to restaurants, airports, hospitals, and just about anything else needed in a geographic area.
- You can implement location-based functionality in a wide range of apps, like focused queries and location-aware history.
- Your apps can use location data to send notifications targeted to mobile devices in a geographic area or a certain mobile user or asset only in a specific geographic area.
- Location-aware applications can also contribute a lot to business intelligence and analytics, including customer profiling and demographics, competitive analysis and supply chain tracking.

This chapter discusses how to use these Location APIs to perform common tasks. For more details on using the platform APIs, see REST APIs for Oracle Mobile Cloud Service.

Setting Up Location Devices, Places and Assets

Location devices, places and assets provide the tools you need to create location-aware mobile apps.

- A **location device** is any device that provides location services, like a Bluetooth proximity beacon. The following location protocols are currently supported:
  - *AltBeacon* is an open source protocol for Bluetooth proximity beacons. For more information and the full specification, see altbeacon.org and https://github.com/AltBeacon/spec.
  - *Eddystone* is Google's open protocol for Bluetooth proximity beacons. For details, see https://github.com/google/eddystone.
  - *iBeacon* is the Apple protocol for Bluetooth proximity beacons. For details, see https://developer.apple.com/ibeacon/.
- A **place** is a physical location associated with one or more location devices.
- An **asset** is a mobile physical object that's associated with one or more location devices.

To set up a location in MCS, define the related places and/or assets and register the associated location devices in the MCS UI under **Applications** > **Location**. You can also use the Location Management API to create, update and delete location devices, places and assets from custom code. For details, see Accessing the Location Management API from Custom Code.

Defining Places

A **place** is a physical location associated with one or more location devices. You can define places through the UI individually or by uploading a CSV file. You can also use the Location Management API to create, update and delete places from custom code. For details, see Accessing the Location Management API from Custom Code.

> **Note:**
>
> To manage places in the MCS UI, you need to be assigned the `MobileEnvironment_LocationMgmt` MCS team member role in the environment.

1. Click  to open the side menu and select **Applications > Location**.
2. From the **Places** tab, click **New Place** to define a place using the UI. This tab shows all the places defined. To edit an existing place, select it in the list and click **Edit** .
3. If you are creating a new place, enter a name, and an optional label and description. If you enter a new label, it will be saved and can be used to categorize other places, location devices and assets. Click **Create**.
4. On the **Overview** tab of the new Location Place Editor, enter the GPS coordinates for the place. You can also define a geofence by radius or polygon. To associate the place with another existing place, select that place from the **Parent** dropdown.

5. Click the **Attributes** tab to define custom attributes for the place. Create new attributes or copy them from an existing place. You can use attributes to associate a content URI with the place, for example a coupon or flier that a mobile app downloads when the user is nearby. Attributes can also be used to filter results in queries that use the Location Platform API.
6. Click the **Devices** tab to associate location devices with the place. You can register a new device from this page (Registering Location Devices) or select from location devices already registered. A device can be associated with a single place or asset, not both. By default, only the devices for the current place are displayed, but you can expand the list by checking the box *Show all devices associated with children of this place*.
7. When you are done configuring the place, click **Save**.

If a place has descendants, click **>** at the end of the table row to navigate to them.

Uploading Places Using a CSV File

You can upload multiple places using a CSV file.

1. From the Location : Places page, click **Upload Places**.
2. Browse to the .csv file and click **Upload**.

   The CSV file for uploading places must follow this format:

```
#version=1.0
#name,#label,#description,#GPSPoint,#GPSCircle,#GPSPolygon,#list of
Attributes
name,label,description,lat:lon,lat:lon:radius,lat1:lon1;lat2:lon2;lat3:l
on3,key1=val1,key2=val2
```

   The first line specifies the version, and the second line is for usability. Any line that starts with # is considered a comment line and is ignored.

   The data starts on line 3. For each line of data, you can define one type of place:

   - For specific GPS coordinates (`GPSPoint`), include the latitude and longitude.
   - For a circle geofence (`GPSCircle`), include the latitude and longitude of the center point, and the radius. In Oracle Spatial, GPS circles are converted to polygons, which might cause the radius to be recalculated.
   - For a polygon geofence (`GPSPolygon`), include the latitude and longitude for each corner of the polygon.

   Make sure to include commas for any empty properties to define the entry correctly. For example, the CSV file below defines a `GPSPoint`.

```
#version=1.0
#name,#label,#description,#GPSPoint,#GPSCircle,#GPSPolygon,#list of
Attributes
FixitFast Redwood City Warehouse,Warehouse,FixitFast Warehouse in
Redwood City,37.8453:-121.7845,,,key1=val1,prop2=val2,prop3=val3
```

> **Note:**
>
> The expected encoding for the CSV file is Unicode UTF-8, so it's best to use a text editor to edit CSV files. Opening a CSV file in Excel or another spreadsheet application can corrupt the encoding or add extra lines. If you use another application to edit your CSV files, confirm that the encoding is correct in a text editor before uploading the file.

Defining Location Assets

An **asset** is a physical object that's associated with one or more location devices, typically something mobile and valuable like a forklift or hospital bed. You can define location assets through the UI individually or by uploading a CSV file. You can also use the Location Management API to create, update and delete location assets from custom code. For details, see Accessing the Location Management API from Custom Code.

> **Note:**
>
> To manage location assets in the MCS UI, you need to be assigned the `MobileEnvironment_LocationMgmt` MCS team member role in the environment.

1. Click  to open the side menu and select **Applications > Location**.
2. From the **Assets** tab, click **New Asset** to define a location asset using the UI. This tab shows all the assets defined. To edit an existing asset, select it in the list and click **Edit Asset**.
3. If you are creating a new asset, enter a name, and a label and description if you choose. Labels will be saved and can be used to categorize other location assets. If the device(s) you want to associate with the asset are already registered, you can select them on this page. (A device can be associated with a single place or asset, not both.) Click **Create**.
4. On the **Overview** tab of the Location Asset Editor, you can update your entries.
5. Click the **Attributes** tab to define custom attributes for the asset. Create new attributes or copy them from an existing asset. You can use attributes to associate a content URI with the asset, for example a coupon or flier that a mobile app downloads when the user is nearby. Attributes can also be used to filter results in queries that use the Location Platform API.
6. When you are done configuring the asset, click **Save**.

Uploading Assets Using a CSV File

You can upload multiple assets using a CSV file.

1. From the Location : Assets page, click **Upload asset file**.
2. Browse to the .csv file and click **Upload**.

The CSV file for uploading assets must follow the following format:

```
#version=1.0
#name,#description,#label,#list of Attributes
Name,Description,label,key1=val1,key2=val2
```

The first line specifies the version, and the second line is for usability. Any line that starts with # is considered a comment line and is ignored.

The data starts on line 3, as shown in the example below. Make sure to include commas for any empty properties to define the entry correctly.

```
#version=1.0
#name,#description,#label,#list of Attributes
RC_WH_01_F01_B023,Beacon #23 in the FixItFast Warehouse in Redwood
City,beacon,
FiF Warehouse Forklift #6,MyMed DA332
forklift,forklift,EquipmentManufacturer=MyMed,MyMed serial
number=OU812-9845873
Hospital Bed #233,MyMed model 1225 hospital bed,hospital
bed,EquipmentManufacturer=MedBed,SJId=6754843090
```

> **Note:**
>
> The expected encoding for the CSV file is Unicode UTF-8, so it's best to use a text editor to edit CSV files. Opening a CSV file in Excel or another spreadsheet application can corrupt the encoding or add extra lines. If you use another application to edit your CSV files, confirm that the encoding is correct in a text editor before uploading the file.

Registering Location Devices

A **location device** is any device that provides location services, like a Bluetooth proximity beacon. You can define location devices through the UI or by uploading a CSV file.

1. Click  to open the side menu and select **Applications > Location**.
2. From the **Devices** tab, click **New Device** to register a location device using the UI. This tab shows all the location devices defined. To edit an existing device, select it in the list and click **Edit**. (You can also register devices from the Devices tab in the Location Places Editor.)
3. If you are creating a new location device, enter a name and a description. Select the **Protocol**:
   - **altBeacon**
   - **Eddystone**
   - **iBeacon**

> **Note:**
>
> The protocol can't be changed after a device is registered.

Click **Create**.

4. On the **Overview** tab of the Location Device Editor, enter the identifying information for the location device. The required values depend on the selected protocol:
   - For iBeacon, enter the **UUID**, **Minor** and **Major** values.
   - For altBeacon, enter **ID1**, **ID2** and **ID3**.
   - For Eddystone, enter the **Namespace**, **Instance** and **URL**.

   If the place and/or asset you want to associate with the device is already defined, select it from the dropdown list. A device can be associated with a single place or asset, not both.
5. Click the **Attributes** tab to define custom properties for the device. Create new attributes or copy them from an existing device. You can use attributes to associate a content URI with the device, for example a coupon or flier that a mobile app downloads when the user is nearby. Attributes can also be used to filter results in queries that use the Location Platform API.
6. When you are done configuring the device, click **Save**.

Uploading Location Devices Using a CSV File

You can upload multiple location devices using a CSV file.

1. From the Location > Devices page, click **Upload Devices**.
2. Browse to the .csv file and click **Upload**.

   The CSV file for uploading devices must follow the following format:

```
#version=1.0
#name,#description,#uuid,#major,#minor,#id1,#id2,#id3,#namespace,#instan
ce,#url,#list of Attributes
Name,Description,uuid,major,minor,id1,id2,id3,namespace,instance,url,key
1=val1,key2=val2
```

   The first line specifies the version, and the second line is for usability. Any line that starts with # is considered a comment line and is ignored.

   The data starts on line 3. For each line of data, you can define one protocol type. The required properties depend on the protocol type:
   - For iBeacon, include `uuid`, `major` and `minor` properties.
   - For altBeacon, include `id1`, `id2` and `id3` properties.
   - For Eddystone, include the `namespace`, `instance` and `URL`.

Make sure to include commas for any empty properties to define the entry correctly. For example, the CSV file below registers an iBeacon location device by defining values for the `uuid`, `major` and `minor` properties.

```
#version=1.0
#name,#description,#uuid,#major,#minor,#id1,#id2,#id3,#namespace,#instan
ce,#url,#list of Attributes
RC_WH_01_F01_B001,Beacon on 1st Floor in FixitFast Warehouse in Redwood
City,B9407F30-F5F8-466E-AFF9-25556B57FE6D,
1.0,1.1,,,,,,,key1=val1,key2=val2,key3=val3
```

> **Note:**
>
> The expected encoding for the CSV file is Unicode UTF-8, so it's best to use a text editor to edit CSV files. Opening a CSV file in Excel or another spreadsheet application can corrupt the encoding or add extra lines. If you use another application to edit your CSV files, confirm that the encoding is correct in a text editor before uploading the file.

Calling the Location API from Your App

Make your mobile apps location-aware by querying for and retrieving location devices, places and assets using the Location API. You can use the client SDK for your platform or access the API directly through REST endpoints.
Team members with the `MobileEnvironment_System` role can use the Location Management REST API to add and maintain places, devices, and assets.

Querying for Location Devices, Places and Assets

The Location API allows you to write complex queries for location devices, places and assets. You can call the REST endpoint directly or use the client SDK to construct a query.

The available query parameters depend on the object type.

Querying for Location Devices

Query for location devices using the following REST endpoints:

- `GET {baseUri}/mobile/platform/location/devices?name={name}` to query by the device name.
- `POST {baseUri}/mobile/platform/location/devices/query` to query using parameters in a JSON payload as described below.

To define your query, include a JSON payload with the following options:

Parameter	Description
`name`	Filters results by a partial match of this string with the name defined for the device in the UI. Not case sensitive.

Parameter	Description
`description`	Filters results by a partial match of this string with the description defined for the device in the UI. Not case sensitive.
`search`	Filters results by a partial match of this string with the name or description defined for the device in the UI. Not case sensitive.
`attributes`	Filters results by a match of the name-value pairs in the `Attributes` object, using the attributes defined for the device in the UI.
`protocol`	Filters results by device protocol type(s): • `iBeacon` • `altBeacon` • `eddystone`
`associatedAssetId`	The asset ID to search for. (Returns location devices associated with the specified asset.)
`listOfDevices`	An array of device IDs to search for.
`iBeacon_uuid`	The UUID of the iBeacon device(s) to search for.
`iBeacon_major`	The major version of the iBeacon device to search for.
`iBeacon_minor`	The minor version of the iBeacon device to search for.
`altBeacon_id1`	ID1 of the altBeacon to search for.
`altBeacon_id2`	ID2 of the altBeacon to search for.
`altBeacon_id3`	ID3 of the altBeacon to search for.
`eddystone_namespace`	The namespace of the Eddystone device to search for.
`eddystone_instance`	The instance of the Eddystone device to search for.
`eddystone_url`	The URL of the Eddystone device to search for.
`orderBy`	An enumeration of the field(s) to order results by. Can include any top-level attribute. Append the direction to order results by: • `:asc` for ascending • `:desc` for descending For example, `name:asc`.
`offset`	By default, 0 to start results at the first item. Specify an offset number to start results in a different place.
`limit`	By default, 40 items are returned. You can specify a different maximum number of results, up to 500. Generally meant to be used with `offset` for pagination.

Parameter	Description
format	By default, the response is in long format and results include the device id, name, description, attributes, createdOn and createdBy, as well as the place ID and identifying details about the device. Specify short to return only the device id, name, description and protocol.

iBeacon

```
{
  "protocol":"iBeacon",
  "iBeacon_major": "2.0",
  "iBeacon_minor": "2.2",
  "iBeacon_uuid": "B9407F30-F5F8-466E-AFF9-25556B57FE6D"
}
```

If the query is successful, the response will be 200, and the body will include the matching location device and its associated place or asset if it has one. For example:

```
{
  "items": [
    {
      "id": 15,
      "createdOn": "2015-11-11T21:15:34.341+0000",
      "createdBy": "thomas.smith@fif.com",
      "modifiedOn": "2015-11-11T21:15:34.341+0000",
      "modifiedBy": "thomas.smith@fif.com",
      "name": "RC_WH_01_F01_B003",
      "description": "Beacon on 1st Floor in FixItFast Warehouse in
Redwood City",
          "place": {
            "name": "FixitFast Redwood City Warehouse",
          "label": "FixitFast Warehouse",
          "description": "FixitFast Warehouse in Redwood City",
        "address" : {
          "gpsPoint" : {
            "latitude": 37.5548,
            "longitude": -121.1566
          }
        },
        "attributes" : {
          "EquipmentManufacturer": "Abc Corp"
        },
        "links": [
           {
             "rel": "canonical",
             "href": "/internal-tools/1.0/envs/dev/location/places/9876"
           },
           {
             "rel": "self",
             "href": "/internal-tools/1.0/envs/dev/location/places/9876"
```

```
            }
          ]
          },
          "beacon": {
         "iBeacon": {
           "major": "2.0",
           "minor": "2.2",
           "uuid": "B9407F30-F5F8-466E-AFF9-25556B57FE6D"
         }
       },
       "attributes": {
         "manufacturer": "Gimbal",
         "status": "Active",
         "manufacturerId": "10D39AE7-020E-4467-9CB2-DD36366F899D",
         "visibility": "Public"
       },
     },
  "totalResults": 1,
  "offset": 0,
  "limit": 20,
  "count": 1,
  "hasMore": false
}
```

The example below queries for `altBeacon` devices with "`Warehouse`" in the name or description and specifies the `short` response format, ordered by name, with a limit of 5 items.

```
{
    "protocol":"altBeacon",
    "orderBy":"name",
    "limit":"5",
    "format":"short",
    "search":"Warehouse"
}
```

If the query is successful, the response is 200 and the body contains just the id, name, description and protocol for the 5 returned devices.

```
{
    "items":[
        {
            "id":33,
            "name":"RC_WH_01_B09_C004",
            "description":"Beacon on 2nd Floor in FixItFast Warehouse in
Redwood City",
            "protocol":"altBeacon"
        },
        {
            "id":12,
            "name":"RC_WH_01_F01_B001",
            "description":"Beacon on 1st Floor in FixItFast Warehouse in
Redwood City",
            "protocol":"altBeacon"
```

```
        },
        {
            "id":61,
            "name":"RC_WH_01_F01_B008",
            "description":"Beacon on 2nd Floor in Fix*tFast Warehouse in
Redwood City",
            "protocol":"altBeacon"
        },
        {
            "id":58,
            "name":"RC_WH_02_F01_B011",
            "description":"Beacon on 1st Floor in FixitFast Warehouse in
Redwood City",
            "protocol":"altBeacon"
        },
        {
            "id":114,
            "name":"RC_WH_01_K22_A999",
            "description":"Beacon on 3rd Floor in FixitFast Warehouse in
Redwood City",
            "protocol":"altBeacon"
        }
    ],
    "totalResults":5,
    "offset":0,
    "limit":5,
    "count":5,
    "hasMore":false
}
```


Querying for Places

Query for places with specific parameters using the following REST endpoints:

- `GET {baseUri}/mobile/platform/location/places?name={name}` to query by the place name.
- `POST {baseUri}/mobile/platform/location/places/query` to query using parameters in a JSON payload as described below.

To define your query, include a JSON payload with the following options:

Parameter	Description
`name`	Filters results by a partial match of this string with the name defined for the place in the UI. Not case sensitive.
`description`	Filters results by a partial match of this string with the description defined for the place in the UI. Not case sensitive.
`search`	Filters results by a partial match of this string with the name, label or description defined for the place in the UI. Not case sensitive.

Parameter	Description
`attributes`	Filters results by a match of the name-value pairs in the `Attributes` object, using the attributes defined for the place in the UI.
`label`	Filters results by a partial match of this string with the label specified for the place in the UI. Not case sensitive.
`listOfPlaces`	An array of place IDs to search for.
`descendantOf`	Specify a place ID to search for direct descendants.
`includeDescendantsInResult: all`	Entire Place descendant hierarchy is returned in the results.
`includeDescendantsInResult: direct`	Only direct (first level) descendants are returned in the results.
`includeDescendantsInResult: none`	No descendants are returned in the results.
`nearestTo`	Specify a `gpsPoint` (latitude, longitude) to return the closest place. This parameter can't be combined with other query parameters.
`inGeoFence`	Specify a `gpsCircle` (latitude, longitude, radius) to return all places within that geofence.
`descendantDevices`	Set to `true` to include the `descendantDevices` property in the results, which lists the devices associated with this place and all its child places. These results are always in `short` format.
`orderBy`	An enumeration of the field(s) to order results by. Can include any top-level attribute. Append the direction to order results by: • `:asc` for ascending • `:desc` for descending For example, `name:asc`.
`offset`	By default, 0 to start results at the first item. Specify an offset number to start results in a different place.
`limit`	By default, 40 items are returned. You can specify a different maximum number of results, up to 500. Generally meant to be used with `offset` for pagination.
`format`	By default, the response is in `long` format and results include the place id, name, description, attributes, label, creation and modification data, as well as the place address, and a list of the devices within the place and the place's parent. Specify `short` to return only the place id, name, description and label.

```
{
  "label":"block 1",
  "inGeoFence": {
```

```
    "gpsCircle": {
      "latitude": 37.488179,
      "longitude": -122.229011,
      "radius": 32186
    }
   },
  "orderBy":"name:asc",
  "limit":100
}
```

If the query is successful, the response will be 200, and the body will include an array of matching places. In this example, only two places matched the query:

```
{
  "items": [
    {
      "id": 16,
      "createdOn": "2016-03-08T22:09:19.968+0000",
      "createdBy": "joe",
      "modifiedOn": "2016-03-08T22:09:19.968+0000",
      "modifiedBy": "joe",
      "name": "l1b1",
      "label": "lot 1 block 1",
      "parentPlace": 15,
      "description": "Lot 1 block 1 New City",
      "hasChildren": false,
      "address": {
        "gpsCircle": {
          "longitude": -120.87449998,
          "latitude": 37.98560003,
          "radius": 29999.99999997
        }
      },
      "links": [
        {
          "rel": "canonical",
          "href": "/mobile/platform/location/places/16"
        },
        {
          "rel": "self",
          "href": "/mobile/platform/location/places/16"
        }
      ]
    },
    {
      "id": 17,
      "createdOn": "2016-03-08T22:09:20.065+0000",
      "createdBy": "joe",
      "modifiedOn": "2016-03-08T22:09:20.065+0000",
      "modifiedBy": "joe",
      "name": "l2b1",
      "label": "lot2 block 1",
      "parentPlace": 15,
      "description": "Lot 2 block 1 New City",
```

```
      "hasChildren": false,
      "address": {
        "gpsPolygon": {
          "vertices": [
            {
              "longitude": -121.7845,
              "latitude": 37.8453
            },
            {
              "longitude": -120.9853,
              "latitude": 37.1248
            },
            {
              "longitude": -121.7758,
              "latitude": 37.6983
            }
          ]
        }
      },
      "links": [
        {
          "rel": "canonical",
          "href": "/mobile/platform/location/places/17"
        },
        {
          "rel": "self",
          "href": "/mobile/platform/location/places/17"
        }
      ]
    }
  ],
  "totalResults": 2,
  "offset": 0,
  "limit": 100,
  "count": 2,
  "hasMore": false
}
```

```
{
    "includeDescendantsInResult": "direct",
    "orderBy" : "name",
    "offset" : 0,
    "limit" : 10,
    "format" : "short"
}
```

If the query is successful, the response will be 200, and the body will include only the first level descendants. In this example, only three descendants matched the query:

```
{
  "places": [
        {
            "id": 3331,
```

```
            "name": "FixitFast Redwood City HQ Campus",
            "label": "campus",
            "description": "1st Floor in FixitFast Warehouse in Redwood
City"
            "children": [
                {
                    "id": 3334,
                    "name": "Building #1 FixitFast Redwood City HQ Campus",
                    "description": "Building #1 on FixitFast Redwood City
Headquarters Campus",
                    "label": "building",
                    "children": []
                },
                {
                    "id": 3335,
                    "name": "Building #2 FixitFast Redwood City HQ Campus",
                    "description": "Building #2 on FixitFast Redwood City
Headquarters Campus",
                    "label": "building",
                    "children": []
                },
                {
                    "id": 3336,
                    "name": "Building #3 FixitFast Redwood City HQ Campus",
                    "description": "Building #3 on FixitFast Redwood City
Headquarters Campus",
                    "label": "building",
                    "children": []
                }
        }
    ]
}
```

Querying for Assets

Query for assets with specific parameters using the following REST endpoints:

- `GET {baseUri}/mobile/platform/location/assets?name={name}` to query by the asset name.
- `POST {baseUri}/mobile/platform/location/assets/query` to query using parameters in a JSON payload as described below.

To define your query, include a JSON payload with the following options:

Parameter	Description
`name`	Filters results by a partial match of this string with the name defined for the asset in the UI. Not case sensitive.
`description`	Filters results by a partial match of this string with the description defined for the asset in the UI. Not case sensitive.
`search`	Filters results by a partial match of this string with the name, label or description defined for the asset in the UI. Not case sensitive.

Parameter	Description
`attributes`	Filters results by a match of the name-value pairs in the `Attributes` object, using the attributes defined for the asset in the UI.
`label`	Filters results by a partial match of this string with the label specified for the asset in the UI.
`listOfAssets`	An array of asset IDs to search for.
`associatedDeviceId`	A device ID to search for. Returns the asset associated with this device ID. When you use this query parameter, don't combine it with other parameters.
`nearestTo`	Specify a `gpsPoint` (latitude, longitude) to return the closest asset. Can't be combined with other parameters.
`inGeoFence`	Specify a `gpsCircle` (latitude, longitude, radius) to return all assets within that geofence.
`orderBy`	An enumeration of the field(s) to order results by. Can include any top-level attribute. Append the direction to order results by: • `:asc` for ascending • `:desc` for descending For example, `name:asc`.
`offset`	By default, 0 to start results at the first item. Specify an offset number to start results in a different place.
`limit`	By default, 40 items are returned. You can specify a different maximum number of results, up to 500. Generally meant to be used with `offset` for pagination.
`format`	By default, the response is in `long` format and results include the asset id, name, description, attributes, label, creation and modification data, as well as the associated place, and the IDs of associated devices. Specify `short` to return only the asset id, name, description and label.

```
{
    "label":"bed",
    "attributes":{
        "EquipmentManufacturer":"Example Company"
    },
    "orderBy":"createdOn:asc",
    "format":"long"
}
```

If the query is successful, the response will be 200, and the body will include an array of matching assets:

```
{
    "items":[
```

```
            {
                "id":333,
                "createdBy":"jdoe",
                "createdOn":"2015-08-06T18:37:59.424Z",
                "modifiedOn":"2015-08-06T18:37:59.424Z",
                "modifiedBy":"jdoe",
                "name":"hospital bed #233",
                "label":"hospital bed",
                "description":"model 1225 hospital bed",
                "lastKnownLocation":{
                    "placeId":244
                },
                "devices":[
                    3409
                ],
                "attributes":{
                    "EquipmentManufacturer": "Example Company",
                    "SJId": "6754843090"
                }
            },
            {
                "id":888,
                "createdBy":"jdoe",
                "createdOn":"2015-10-16T09:24:41.354Z",
                "modifiedOn":"2015-10-16T09:24:41.354Z",
                "modifiedBy":"jdoe",
                "name":"hospital bed #233",
                "label":"hospital bed",
                "description":"model 1225 hospital bed",
                "lastKnownLocation":{
                    "placeId":360
                },
                "devices":[
                    658
                ],
                 "attributes":{
                    "EquipmentManufacturer": "Example Company",
                    "SJId": "6754843090"
                }
            }
        ],
        "totalResults":2,
        "offset":0,
        "limit":100,
        "count":2,
        "hasMore":false
    }
```


Using the SDK to Query for Location Objects: iOS

The `OMCLocationQuery` class in the iOS client SDK allows you to construct queries for location devices, places and assets.

To access the Location API through the iOS SDK, use `[[OMCMobileBackendManager sharedManager].defaultMobileBackend` as described in Calling Platform APIs Using the SDK for iOS.

Below is an example of using the iOS SDK to query for a place by name.

```
OMCLocation* location = [[OMCMobileBackendManager
sharedManager].defaultMobileBackend location];

NSString* searchString = @"store";

// search by name
// sort results by name, in ascending order
// results will be in "short" format
OMCLocationPlaceQuery* query = [location buildPlaceQuery];
query.name = searchString;
query.orderByAttribute =
OMCLocationDeviceContainerQueryOrderByAttributeTypeName;
query.format = OMCLocationObjectQueryFormatTypeShort;

__block OMCLocationPlaceQueryResult* result;
do {
    result = nil;
    __block NSError* error = nil;
    __block BOOL executing = YES;
    [query executeWithCompletionHandler:^(OMCLocationPlaceQueryResult*
result_, NSError* error_) {
        result = result_;
        error = error_;
        executing = NO;
    }];

    while (executing) {
        [[NSRunLoop currentRunLoop] runUntilDate:[NSDate
dateWithTimeInterval:0.5 sinceDate:[NSDate date]]];
    }

    if (error) {
        // handle error...
    } else {
        for (OMCLocationPlace* place in result.items) {
            // process each place...
            NSLog(@"place name: %@", place.name);
        }
    }
    query = result.nextQuery;
} while ((result != nil) && result.hasMore);
```

For more information on place queries, see Querying for Places.

Using the SDK to Query for Location Objects: Android

The `LocationQuery` class in the Android client SDK allows you to construct queries for location devices, places and assets.

To access the Location API through the Android SDK, use the `MobileBackendManager` class as described in Calling Platform APIs Using the SDK for Android.

Below is an example of using the Android SDK to query for a place by name:

```
Location location =
MobileBackendManager.getManager().getDefaultMobileBackend(mContext).getServ
iceProxy(Location.class);
Object lock = new Object();

String searchString = "store";
final AtomicReference<String> searchString = "store";
final AtomicReference<LocationObjectQueryResult> mResult = new
AtomicReference<LocationObjectQueryResult>();
final AtomicReference<ServiceProxyException> mError = new
AtomicReference<ServiceProxyException>();

// search by name
// sort results by name, in ascending order
// results will be in "short" format
LocationPlaceQuery query = location.buildPlaceQuery();
query.setName(searchString);
query.setOrderByAttributeType(LocationDeviceContainerQuery.LocationDeviceCo
ntainerQueryOrderByAttributeType
    .LocationDeviceContainerQueryOrderByAttributeTypeName);
query.setFormat(LocationObjectQuery.LocationObjectQueryFormatType.LocationO
bjectQueryFormatTypeShort);

do{
    query.execute(new LocationObjectsQueryCallback(){
        @Override
        void onComplete(LocationObjectQueryResult result,
ServiceProxyException exception){
                mError.set(exception);
                mResult.set(result);

                synchronized(lock){
                    lock.notifyAll();
                }
        }
    });

    synchronized(lock) {
        lock.wait();
    }

    if(mError.get() != null){
        //handle error
    }
```

```
    else{
        for(LocationObject object : mResult.get().getItems()){
            LocationPlace place = (LocationPlace) object;
            // process each place...
        }
    }

    query = mResult().get().getNextQuery();

} while(mResult.get() != null && mResult.get().hasMore());
```

For more information on place queries, see Querying for Places.

Retrieving Location Objects and Properties

Use the Location API to retrieve location devices, places and assets and their associated properties.

The following REST endpoints allow you to retrieve location objects:

- Location devices: `GET {baseUri}/mobile/platform/location/devices`
- Assets: `GET {baseUri}/mobile/platform/location/assets`
- Places: `GET {baseUri}/mobile/platform/location/places`

You can retrieve an object by ID or by name:

- To retrieve an object by ID, include the ID in the path, for example: `GET {baseUri}/mobile/platform/location/devices/12345`.
- To retrieve an object by name, pass the name of an existing object to the endpoint in the `name` query parameter, for example `GET {baseUri}/mobile/platform/location/devices?name=RC_WH_01_F01_B001`.

Using the SDK to Retrieve a Location Object: iOS

The examples below show how to use the client SDK to retrieve a place and its properties by ID.

To access the Location API through the SDK, use the `OMCMobileBackendManager` class as described in Calling Platform APIs Using the SDK for iOS.

The example below uses the place ID to retrieve the properties for the place:

```
OMCLocation* location = [[OMCMobileBackendManager
sharedManager].defaultMobileBackend location];

// query for all places
// sort results by name, in ascending order
// results will be in "short" format
OMCLocationPlaceQuery* query = [location buildPlaceQuery];
query.orderByAttribute =
OMCLocationDeviceContainerQueryOrderByAttributeTypeName;
query.format = OMCLocationObjectQueryFormatTypeShort;

__block OMCLocationPlaceQueryResult* result = nil;
__block NSError* error = nil;
```

```
__block BOOL executing = YES;
[query executeWithCompletionHandler:^(OMCLocationPlaceQueryResult*
result_, NSError* error_) {
    result = result_;
    error = error_;
    executing = NO;
}];

while (executing) {
    [[NSRunLoop currentRunLoop] runUntilDate:[NSDate dateWithTimeInterval:
0.5 sinceDate:[NSDate date]]];
}

// take the first item from the results
// it will be in "short" format...
OMCLocationPlace* shortPlace = result.items.firstObject;

// ...now, fetch the "entire" place directly
__block OMCLocationPlace* place = nil;
error = nil;
executing = YES;
[location placeWithID: shortPlace.id_
completionHandler:^(OMCLocationPlace* place_, NSError* error_) {
    place = place_;
    error = error_;
    executing = NO;
}];

while (executing) {
    [[NSRunLoop currentRunLoop] runUntilDate:[NSDate dateWithTimeInterval:
0.5 sinceDate:[NSDate date]]];
}

// process place...
NSLog(@"place name: %@", place.name);
```

If you've already retrieved an object, you can use an SDK refresh method to get the latest properties. The code below uses refresh to retrieve the latest properties for a place:

```
...
// take the first item from the results
// it will be in "short" format...
OMCLocationPlace* place = result.items.firstObject;

// ...now, refresh the place
error = nil;
executing = YES;
[place refreshWithCompletionHandler:^(NSError* error_) {
    error = error_;
    executing = NO;
}];

while (executing) {
```

```
    [[NSRunLoop currentRunLoop] runUntilDate:[NSDate dateWithTimeInterval:
0.5 sinceDate:[NSDate date]]];
}

// process place...
NSLog(@"place name: %@", place.name);
```

Using the SDK to Retrieve iBeacon Identifiers: iOS

The first step to monitoring a place that uses beacons is to retrieve the beacon identifiers, as shown in the iOS client SDK example below.

```
CLLocationManager *locationManager = [[CLLocationManager alloc] init]; //
iOS CoreLocation object

OMCLocation* location = [[OMCMobileBackendManager
sharedManager].defaultMobileBackend location];

OMCLocationPlaceQuery *queryPlace = [location buildPlaceQuery];

queryPlace.name = @"Chris's Emporium";
queryPlace.limit = @1;
// Order-bys are required as name is search by wildcard, not exact match
queryPlace.orderByAttribute =
OMCLocationDeviceContainerQueryOrderByAttributeTypeName;
queryPlace.orderByOrder = OMCLocationObjectQueryOrderByOrderTypeAscending;

[queryPlace
executeWithCompletionHandler:^(OMCLocationObjectQueryResult<OMCLocationPlac
eQuery *,OMCLocationPlace *>* queryResult, NSError * _Nullable queryError)
{
    OMCLocationPlace *place = queryResult.items.firstObject;

    [place devicesWithCompletionHandler:^(NSArray<OMCLocationDevice *>
*locationDevices, NSError * error) {
        // Following code assumes 1 device for place
         OMCLocationDevice *device = [locationDevices firstObject];
         OMCLocationIBeacon *beacon = (OMCLocationIBeacon*)device.beacon;
         NSUUID *beaconUuid = beacon.uuid;
         CLBeaconMajorValue beaconMajor =
(CLBeaconMajorValue)beacon.major.integerValue;
         CLBeaconMinorValue beaconMinor =
(CLBeaconMinorValue)beacon.minor.integerValue;

         CLBeaconRegion *beaconRegion = [[CLBeaconRegion
alloc]initWithProximityUUID:beaconUuid major:beaconMajor minor:beaconMinor
identifier:@"MyBeaconRegion"];
         beaconRegion.notifyOnEntry = YES;
         beaconRegion.notifyOnExit = YES;

         beaconRegion.delegate = // Assign instance of
CLLocationManagerDelegate to handle beacon events
```

```
        [locationManager startMonitoringForRegion:beaconRegion]; //
Invokes CLLocationManagerDelegate didEnterRegion/didExitRegion
        [locationManager startRangingBeaconsInRegion:beaconRegion]; //
Invokes CLLocationManagerDelegate inRegion
    }];
}];
```

Using the SDK to Define a Geofence: iOS

You can use a geofence to define a monitoring area as a place, as shown in the iOS client SDK example below.

```
CLLocationManager *locationManager = [[CLLocationManager alloc] init]; //
iOS CoreLocation object

OMCLocation* location = [[OMCMobileBackendManager
sharedManager].defaultMobileBackend location];

OMCLocationPlaceQuery *queryPlace = [location buildPlaceQuery];

queryPlace.name = @"Chris's Emporium";
queryPlace.limit = @1;
// Order-bys are required as name is search by wildcard, not exact match
queryPlace.orderByAttribute =
OMCLocationDeviceContainerQueryOrderByAttributeTypeName;
queryPlace.orderByOrder = OMCLocationObjectQueryOrderByOrderTypeAscending;

[queryPlace
executeWithCompletionHandler:^(OMCLocationObjectQueryResult<OMCLocationPlac
eQuery *,OMCLocationPlace *>* queryResult, NSError * queryError) {
        OMCLocationPlace *place = queryResult.items.firstObject;

        OMCLocationGeoCircle *geocircle = (OMCLocationGeoCircle *)[place
address];
        OMCLocationGeoPoint *geopoint = [geocircle center];

        CLLocationDegrees latitude = [[geopoint latitude]doubleValue];
        CLLocationDegrees longitude = [[geopoint longitude]doubleValue];
        CLLocationDistance radius = [[geocircle radius]doubleValue];
        CLLocationCoordinate2D coordinate =
CLLocationCoordinate2DMake(latitude, longitude);

        CLCircularRegion *circularRegion = [[CLCircularRegion
alloc]initWithCenter:coordinate radius:radius
identifier:@"MyGeofenceRegion"];
        circularRegion.notifyOnEntry = YES;
        circularRegion.notifyOnExit = YES;

        circularRegion.delegate = // Assign instance of
CLLocationManagerDelegate to handle events

        [locationManager startMonitoringForRegion:circularRegion]; //
Invokes CLLocationManagerDelegate didEnterRegion/didExitRegion
```

```
    }];
}];
```

Using the SDK to Retrieve Custom Attributes: iOS

Many location objects use custom attributes. The iOS client SDK makes it easy to access these properties, as shown in the examples below.

Retrieving a Custom Attribute for a Place

The SDK example below retrieves a custom attribute for a place:

```
CLLocationManager *locationManager = [[CLLocationManager alloc] init]; //
iOS CoreLocation object

OMCLocation* location = [[OMCMobileBackendManager
sharedManager].defaultMobileBackend location];

OMCLocationPlaceQuery *queryPlace = [location buildPlaceQuery];
queryPlace.name = @"Chris's Emporium";
queryPlace.limit = @1;
// Order-bys are required as name is search by wildcard, not exact match
queryPlace.orderByAttribute =
OMCLocationDeviceContainerQueryOrderByAttributeTypeName;
queryPlace.orderByOrder = OMCLocationObjectQueryOrderByOrderTypeAscending;

[queryPlace
executeWithCompletionHandler:^(OMCLocationObjectQueryResult<OMCLocationPlac
eQuery *,OMCLocationPlace *>* queryResult, NSError * queryError) {
        OMCLocationPlace *place = queryResult.items.firstObject;

        NSString *myCustomProperty = [place
attributeForKey:@"MyCustomProperty"];
        NSLog(@"My Custom Property = %@", myCustomProperty);
}];
```

Retrieving a Custom Attribute for a Location Device

The SDK example below is very similar to the one above, but uses `OMCLocationDevice` to retrieve a custom attribute for a beacon:

```
OMCLocation* location = [[OMCMobileBackendManager
sharedManager].defaultMobileBackend location];

// Query iBeacon
OMCLocationDeviceQuery *queryDevice = [location buildDeviceQuery];
NSUUID *uuid = [[NSUUID alloc] initWithUUIDString:@"0AC59CA4-
DFA6-442C-8C65-22247851344C"];
NSNumber *major = @4;
NSNumber *minor = @200;
queryDevice.beacon = [OMCLocationIBeacon iBeaconWithUUID:uuid major:major
minor:minor];
```

```
[queryDevice
executeWithCompletionHandler:^(OMCLocationObjectQueryResult<OMCLocationDevi
ceQuery *,OMCLocationDevice *>* queryResult, NSError * queryError) {
        OMCLocationDevice *device = queryResult.items.firstObject;

        // Retrieve device/beacon custom property
        NSString *customProperty = (NSString *) [device
attributeForKey:@"MyCustomProperty"];
}];
```

Using the SDK to Retrieve a Location Object: Android

To access the Location API through the Android client SDK, use the `MobileBackendManager` class as described in Calling Platform APIs Using the SDK for Android.

The example below uses the place ID to retrieve the properties for the place:

```
Location location =
MobileBackendManager.getManager().getDefaultMobileBackend(mContext).getServ
iceProxy(Location.class);
Object lock = new Object();

final AtomicReference<LocationObjectQueryResult> mResult = new
AtomicReference<LocationObjectQueryResult>();
final AtomicReference<LocationPlace> mError = new
AtomicReference<LocationPlace>();

// query for all places
// sort results by name, in ascending order
// results will be in "short" format
LocationPlaceQuery query = location.buildPlaceQuery();
query.setName(searchString);
query.setOrderByAttributeType(LocationDeviceContainerQuery.LocationDeviceCo
ntainerQueryOrderByAttributeType
    .LocationDeviceContainerQueryOrderByAttributeTypeName);
query.setFormat(LocationObjectQuery.LocationObjectQueryFormatType.LocationO
bjectQueryFormatTypeShort);

query.execute(new LocationObjectsQueryCallback(){
    @Override
    void onComplete(LocationObjectQueryResult result,
ServiceProxyException exception){
            mResult.set(result);

            synchronized(lock){
                lock.notifyAll();
            }
    }
});

synchronized(lock){
    lock.wait();
}
```

```
// take the first item from the results
// it will be in "short" format...
LocationPlace place = (LocationPlace) mResult.get().getItems().get(0);

// ...now, fetch the "entire" place directly
location.fetchPlace(place.getID(), new LocationObjectQueryCallback(){
    @Override
    void onComplete(LocationObject object, ServiceProxyException exception)
{
        LocationPlace detailedPlace = (LocationPlace) object;
        mPlace.set(detailedPlace);

        synchronized(lock){
            lock.notifyAll();
        }
    }
});

synchronized(lock){
    lock.wait();
}
// process place...
Log.i(TAG, "place name is " + mPlace.get().getName());
```

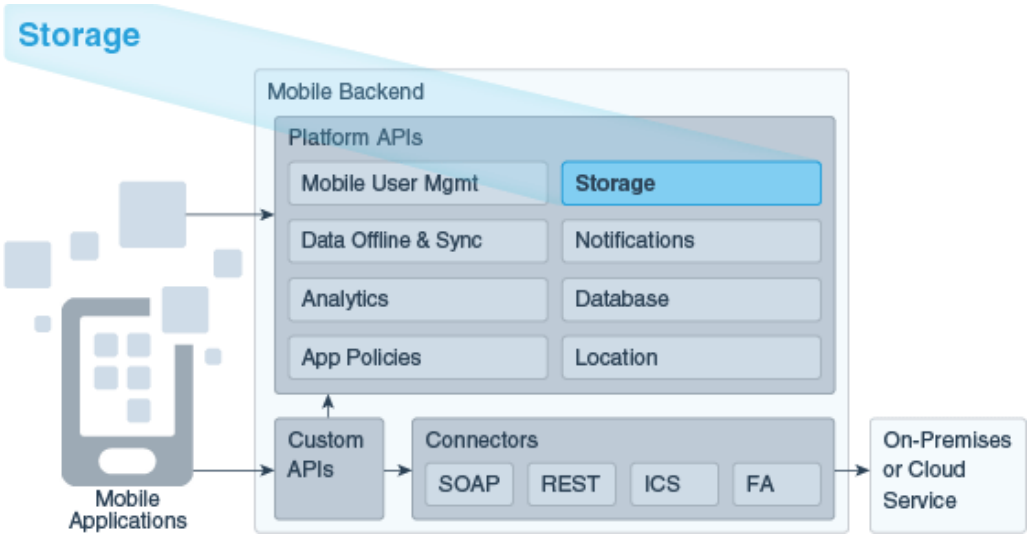
If you've already retrieved an object, you can use an SDK refresh method to get the latest properties. The code below uses refresh to retrieve the latest properties for a place:

```
...
// take the first item from the results
// it will be in "short" format...
LocationPlace place = (LocationPlace) mResult.get().getItems().get(0);

// ...now, refresh the place
place.refresh(new LocationObjectFetchCallback(){
    @Override
    void onComplete(LocationObject object, ServiceProxyException exception)
{
        if(exception != null)
            //handle error

        synchronized(lock) {
            lock.notifyAll();
        }
    }
});

synchronized(lock){
    lock.wait();
}
// process place...
Log.i(TAG, "place name is " + place.getName());
```

14

Storage

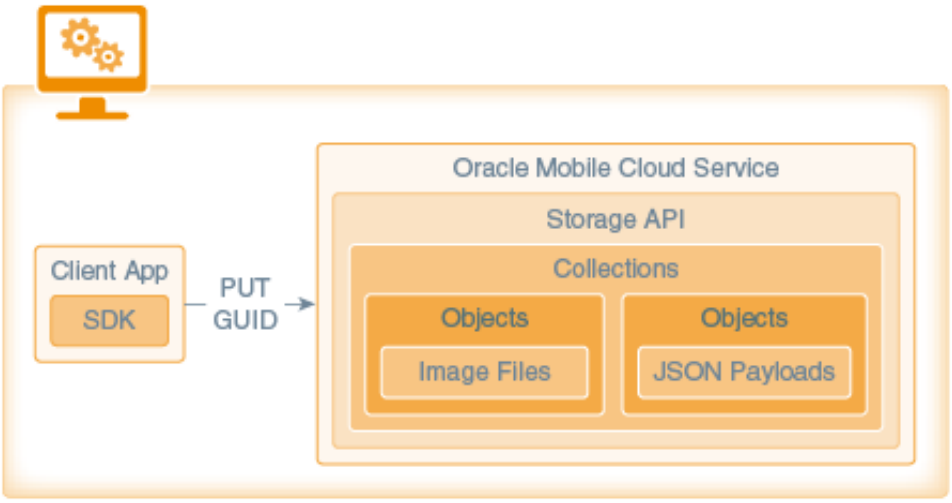
Oracle Mobile Cloud Service (MCS) provides a Storage API for storing media in the cloud. As a mobile app developer, you can use this API in your mobile app to store and retrieve objects, such as files, text, images, and JSON objects.



What is the Storage API?

The Storage API enables your mobile app to store, update, retrieve, and delete media, such as JSON objects, text files, and images, in collections in your MCS environment. Storage is key based, and you can use roles to restrict access to a collection. You can also grant anonymous access to shared collections to anyone who also has backend access by adding the collection name to the Security_CollectionsAnonymousAccess environment policy.

Note that this API isn't intended to act as a database-as-a-service (DBaaS) solution by storing business data used by external systems, nor is it intended to host HTML 5 applications as a content management system (CMS) would.



How Mobile Applications Access Collections

Mobile applications access collections through the Storage API. As a mobile developer, you can access this API through the mobile client SDK, or directly through REST calls. As a service developer, you can call the Storage API from the code that you write to implement a custom API.

To access the Storage API through the mobile client SDK, you use a backend manager class.

- For Android apps, you use the `MobileBackendManager` class as described below and in Calling Platform APIs Using the SDK for Android.
- For iOS apps, you use the `OMCMobileBackendManager` class as described in Calling Platform APIs Using the SDK for iOS.
- For JavaScript apps, you use the `MobileBackendManager` class as described in Loading a Mobile Backend's Configuration into a JavaScript App.

Here is an example of using the backend manager class in an Android app to access a collection.

```
try {
    Storage storage =

MobileBackendManager.getManager().getDefaultMobileBackend(this).getServiceP
roxy(Storage.class);
    StorageCollection imagesCollection =
        storage.getStorageCollection("FIF_Images");
    StorageObject imageToUpload =
        new StorageObject(null, imageBytes, "image/jpeg");
    StorageObject uploadedImage = imagesCollection.post(imageToUpload);
    } catch(ServiceProxyException e) {int errorCode = e.getErrorCode();
            ...
}
```

To call a Storage endpoint from custom code, you invoke the custom code SDK method that calls the appropriate Storage API operation, as shown in the following example:

```
// Get metadata about the objects in the attachments collection.
// List most recently modified first.
service.get('/mobile/custom/incidentreport/attachments',
  function (req, res) {
    req.oracleMobile.storage.getAll('attachments',
    {orderBy: 'modifiedOn:desc', sync: true}).then(
      function (result) {
        res.send(result.statusCode, result.result);
      },
      function (error) {
        res.send(error.statusCode, error.error);
      }
    );
  });
```

For more information on how custom code can retrieve collection information and store and retrieve objects, see Accessing the Storage API from Custom Code.

Shared and User Isolated Collections

A collection is either shared or user isolated.

When a collection is shared, no one owns the collection or an object, and the objects are kept in a shared space. Those with certain mobile user roles, permissions, and access to the backend, or anonymous access to the backend associated with the collection, can update an object. Note that in both shared and user isolated collections, each object has an ID that is unique to the collection.

When a collection is user isolated, users who have `Read-Only (All Users)` access can read objects in other users' spaces. Users with `Read-Write (All Users)` access can both read and write objects in other users' spaces. Anonymous access is not permitted in user isolated collections.

Let's look at some examples of this behavior using the following scenarios:

Shared Collection

An online magazine is leveraging the Storage API as a way for authors to submit, change, or read, articles. They've provisioned a shared collection called articles, as shown in the figure below.

- Ben has contributed articles on bugs and bats, while Art has written about cows and dogs.
- The dogs article is shared, allowing both Ben and Art to collaborate on it.
- Art and Ben are able to modify any article regardless of who originally submitted it.
- Dee can read all the articles, but she can't make changes.

However, if this shared collection is added to the `Security_CollectionsAnonymousAccess` environment policy, then Ben, Art, Dee or anyone who has access to the backend can submit, change, or read articles.

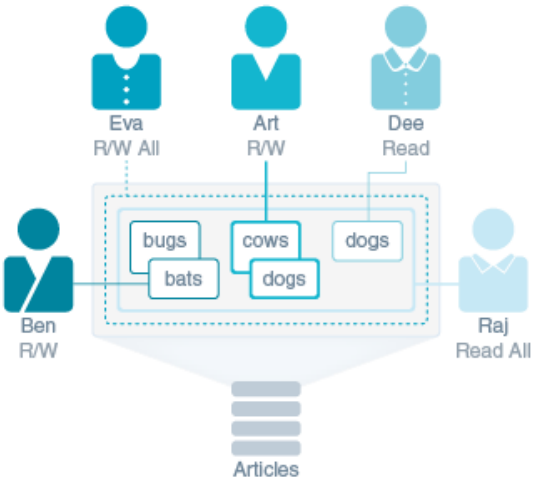
User	Role	Permission
Ben	Author	R/W
Art	Author	R/W
Dee	InactiveAuthor	READ

User Isolated Collection

An online magazine has provisioned a user isolated collection called Articles, as shown in the following figure.

- Ben and Art can read and edit their articles, and upload new articles as well. They can't read or write each other's files.
- Dee can read only her article. Because her role is `InactiveAuthor`, which gives her `Read-Only` permission, she can't upload any new articles.
- Eva, the editor, can make changes to any file and return it to the author's isolated space.
- Raj, the publisher, can view all the articles, but he can't make changes.
- Because users are isolated, the authors don't have to worry about naming conflicts with others. Objects in different isolation spaces can have the same name (as is the case for the "dogs" articles by Dee and Art).
- Eva and Raj can access Ben, Art, and Dee's objects only by specifying a user qualification parameter. When Eva wants to make changes to Art's article, the call that enables her to write to Art's user space must include Art's ID.

Anonymous users don't have access to user isolated collections. If a user isolated collection is added to the `Security_CollectionsAnonymousAccess` environment policy, it's just ignored.



User	Role	Permission
Ben	Author	R/W
Eva	Editor	R/W-ALL
Art	Author	R/W
Dee	InactiveAuthor	READ
Raj	Publisher	READ-ALL

Permissions in Shared and User Isolated Collections

You can designate who can access and update objects in a collection by attaching access permissions to mobile user roles, or for anonymous access, by adding the shared collection name to the `Security_CollectionsAnonymousAccess` environment policy.

For example, to include the `Articles` collection use `Security_CollectionsAnonymousAccess=Articles`.

If the collection does not, or cannot permit anonymous access:

- Art and Ben's `Author` mobile user role is associated with the `Read-Write` permission.
  - In the shared collection, they can read and update any article within the shared collection.
  - In the user isolated collection, they can read and update their own articles.
- In contrast, Dee has the `InactiveAuthor` mobile user role, which gives her `Read-Only` permission.
  - In the shared collection, Dee can read Art's article about dogs, as well as various articles from either Art or Ben about bugs, cows, and bats. Unlike Ben or Art, she can't delete articles or add new ones.
  - In the user isolated collection, she can read her own article about dogs, but she can't read Art's article about dogs.
- For user isolated collections, mobile user roles that are associated with the `Read-Only (All Users)` permission can view any object. The `Read-Write (All Users)` permission allows users to view and update objects in other users' spaces. Because her role as `Editor` has a `Read-Write (All Users)` permission, Eva can read and edit various authors' files, such as those authored by Ben and Art.

> **Note:**
>
> Although different mobile user roles can grant access to the same objects in a collection, such as Eva (`Editor`), Ben (`Author`), and Art (also `Author`), in the user isolated collection, the objects remain in their respective isolated spaces.
> When anonymous access is allowed on a shared collection, access and the ability to update an object is granted to any authenticated user as well, regardless of role. This means adding a collection name to the `Security_CollectionsAnonymousAccess` environment policy overrides permissions given through roles. Take care when allowing anonymous access to a collection. Security is more limited than with role-based permissions.

Working with Collections

Mobile apps can use only the collections that are associated with a mobile backend.

You can add existing collections to a mobile backend. You can also create new collections as part of the process of creating a mobile backend. There's a page for each approach:

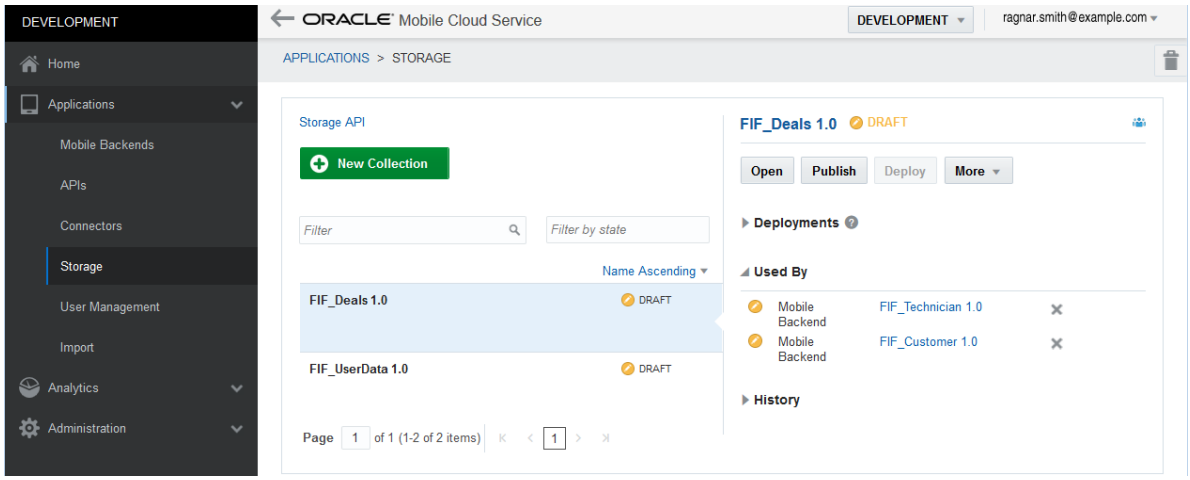
- The Storage page that you access by clicking ☰ > **Applications** > **Storage** can be used to create collections and view a master list of all collections. To associate one of the collections with a mobile backend, select the collection, click **More**, and then select **Associate Mobile Backends**.
- The Storage page that you access from the **Storage** tab on a mobile backend page lets you associate a collection with the mobile backend as well as create a new collection that is associated with that mobile backend.

Using the Storage Configuration Pages

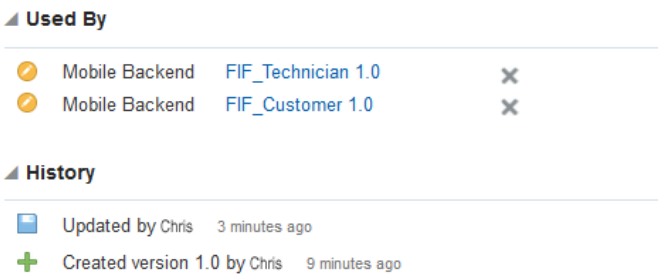
You can use the Storage pages to perform tasks such as create and configure a collection, configure whether the collection is shared or user isolated, and associate a collection with a mobile backend.

To open the Storage page for all collections, click ☰ to open the side menu. Next, click **Applications** and then click **Storage**.

Using this page, you can create collections, edit existing ones, associate them with mobile backends, and publish them. To find out more about collections, policies, and other artifacts, see Lifecycle.



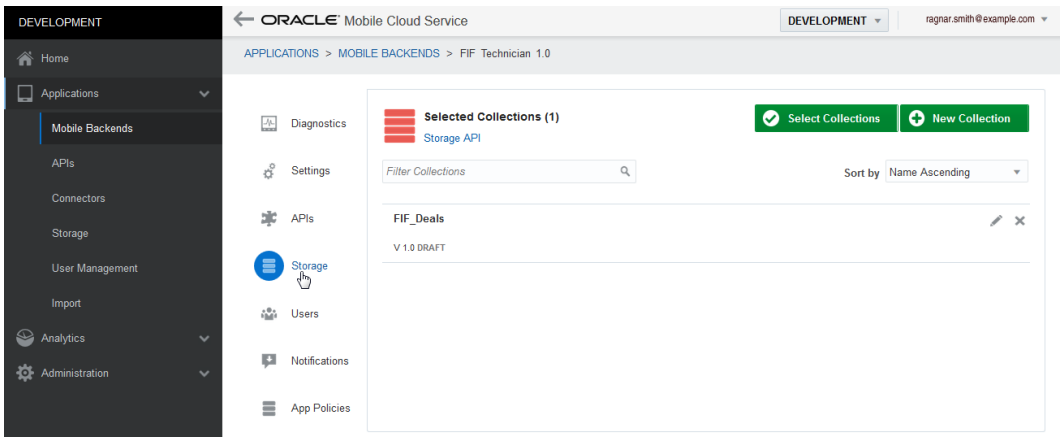
You can find out when the collections listed were created or updated and which mobile backends are using them by first selecting a collection and then expanding **Used By** and **History**.



To associate a collection with a mobile backend, select the collection, click **More**, and then select **Associate Mobile Backends**.

To create or update collections for a specific mobile backend, click  to open the side menu. Next, click **Applications** and then **Mobile Backends**.

To find out if a mobile backend has collections assigned to it, click **Storage** on the mobile backend page.



In addition to the tasks described here, you can also do the following tasks:

Task	Description
Set Permissions	Configure who can access the collection and how. See Adding Access Permissions to a Collection.
Maintain Locally Stored Objects	Set how long before the data stored locally on the device becomes stale and needs to be refreshed. See Offline Data Storage.
Test	Test the endpoint operations that manage collections and their objects. See Testing Runtime Operations Using the Endpoints Page.
Deploy	Deploy collections. See Collection Lifecycle.

Creating Collections

The following tasks enable you to create and update collections:

1. Defining a Collection
2. Adding Access Permissions to a Collection
3. Updating the Collection
4. Adding Objects to a Collection

Defining a Collection

The New Collection dialog lets you name a collection so that it can be identified in REST calls and designate it as shared or user isolated.

1. Open the Storage page either from a mobile backend or by clicking **Storage** in the side menu, and click **New Collection**.
2. Complete the New Collection dialog:
   a. Enter a name for your collection. This name is used to form the Universal Resource Identifier (URI) for the collection. Within the context of the API call, the collection name is referred to as the collection ID:

      ```
      {baseUri}/mobile/platform/storage/collections/{collection ID}
      ```

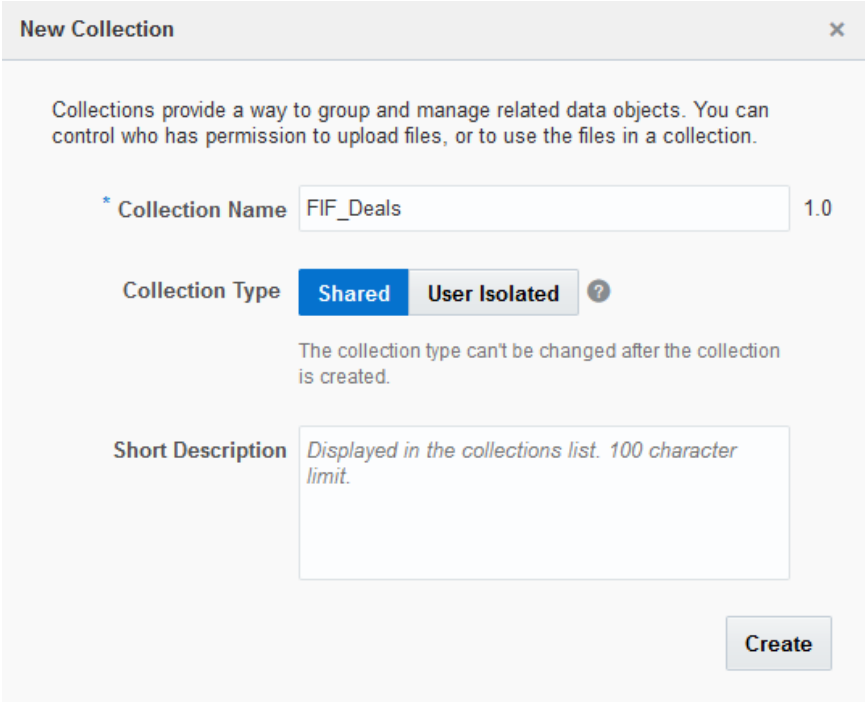
      For example, for a collection named *FiF_UploadedImages* (cloud storage of images uploaded from mobile apps), the URI call would look like this:

      ```
      {baseUri}/mobile/platform/storage/collections/FiF_UploadedImages
      ```

      For a closer look at Storage API syntax, see Storage API Endpoints.

   b. Choose the collection type: `Shared` or `User Isolated`. You can't change the scope of the collection after you've set it. For details and examples, see Shared and User Isolated Collections.

   c. If needed, enter a short description for the purpose of the collection, to be displayed in the list of collections.
3. Click **Create**.



Note:

When you initially create a collection, it's in a draft state, in version 1.0.

- You can modify the collection name, access permissions, and its contents. Remember, you can't change the collection type after it's created.
- You can version a collection. You might want to increment a collection's major and minor version numbers when you publish it or when you add new objects.
- While in the draft state, a collection can be moved to the trash from the **More** menu.

Collection Metadata

In addition to the basic properties like size (in bytes), and description, the collection metadata includes the collection name that identifies it for REST calls.

When you create a collection, the Storage API defines it using the following metadata:

Property	Value Type	Description
`description`	string	The short description. This is an optional value.

Property	Value Type	Description
`id`	string	The collection name, which is used in the uniform resource identifier (URI). For example: `{baseURI}/mobile/platform/storage/collections/{collection}` The collection name is case-sensitive, meaning that `mycollection` and `Mycollection` are two different collections.

Adding Access Permissions to a Collection

Collection access is granted through an anonymous user setting in the environment policy file, or managed by mobile user roles. Once a mobile user role is defined, you can also grant which roles can read and write objects in the collection. To see what mobile user roles are available, go to the Mobile User Management UI and click **Roles**. To learn more about roles and mobile users, see Creating and Managing Mobile User Roles and Creating Mobile Users and Assigning Roles.

Anonymous Access to Collections

Anonymous access is often given to users who just want to check information on an app without logging in or needing an assigned role. Weather apps, where a user can check their local weather, are a good example of this.

Likewise, you can grant anonymous access to your shared collection. Once a shared collection is created, the administrator adds its name to the Security_CollectionsAnonymousAccess policy. You can then read and write objects to the shared collection via the REST API or the SDKs using anonymous access. To read and write objects to the shared collection from the UI, grant `Read-Write`permission to any role on the collection's properties page. For environment policies, see Environment Policies and Their Values.

Keep in mind that when you add a shared collection to the policy, both anonymous and named users have access and read/write privileges to the collection.

> **Note:**
>
> If you try to upload an object to a shared collection which allows anonymous access, an error dialog appears. To work around this issue, in the Properties page, specify any mobile user role for the collection's `Read-Write` permission type.

Role-Based Access to Collections

To define which mobile user roles can read and write objects in a collection:

1. In the Storage page, select a collection and then click **Open**.
2. In the Properties page, specify one or more mobile user roles for each permission type.
   - `Read-Only` and `Read-Write` access apply to all collections (shared or user isolated).

- You can specify `Read-Only (All Users)` and `Read-Write (All Users)` permissions only if the collection type is user-isolated.

Permission	Shared	User Isolated
Read-Only	Read-only access to all of the objects in a collection. For example, both a field technician and a customer can read promotional material like coupons, but they can't update them.	Read-only access to a user isolated collection. When the `Read-Only` permission is applied to user isolated collections, for example, a customer can view images (like a coupon), but he can't update them, or submit additional ones (only a user with `Read-Write (All Users)` privileges can add an object to the customer's user space). Because this is a user isolated collection, the customer can view only his images (or other customer-specific objects that are intended only for him). The `Read-Only` permission also prevents him from adding additional work orders or deleting them.
Read-Write	A user can override any object in the collection.	A user can override the objects in his isolated space. For example, a customer can update the images of broken appliances that he's submitted. Because this is a user isolated collection, the images that he can add (and update) are intended only for him. Because these images exist in his isolated space, he can update these objects, but no one else's. Likewise, he can add or delete images, but can't do this in anyone else's isolated space.
Read-Only (All Users)	NA	A user can read objects in all spaces. For example, a field technician can see the images updated by any customer, but she can't update them, delete them, or add new ones.
Read-Write (All Users)	NA	A user can override objects in all spaces. If a field technician has `Read-Write (All Users)` permission, then she can update work orders submitted by any customer.

> **Note:**
>
> By default, mobile users can't access a collection until they've been assigned mobile user roles that are associated with the `Read-Write`, `Read-Only`, `Read-Write (All Users)` or `Read-Only (All Users)` permissions. Anonymous users can't access a shared collection until the collection has been added to the Security_CollectionsAnonymousAccess environment policy. Anonymous users are automatically granted `Read-Write` permissions.

Updating the Collection

You can update the name, description and access to a collection. You can't however, change the collection type.

1. On the Storage page, select a collection and then click **Open**.
2. Click **Properties**. (The Properties page opens by default when you first create a collection. On subsequent visits, the Content page opens by default.)
3. Change the name, description or access as needed.
4. Click **Save**.

Offline Data Storage

The client SDK's Sync Client library, in conjunction with the Storage library, enables mobile apps to cache a collection's objects for offline use and performance improvement. The apps can then use the cached objects instead of re-retrieving them from Storage, as described in How Synchronization Works with the Storage APIs. If a collection's content changes infrequently, then consider enabling those mobile apps to cache the collection's objects by selecting **Enable the mobile client SDK to cache collection data locally for offline use**.

When **Enable the mobile client SDK to cache collection data locally for offline use** is selected, the objects that a mobile app retrieves can remain in the cache for the period set in the `Sync_CollectionTimeToLive` policy. This value is conveyed to the app through the `Oracle-Mobile-Sync-Expires` response header. By default, the timeout period is set for 24 hours (86,400 seconds).

To learn how to configure the timeout period, see Environment Policies.

Don't select this option for time-critical data, where a cached value might be misleading. For example, if the collection contains current stock prices, you shouldn't select this option, because users expect the latest value (or no value at all).

If your mobile app isn't using the client SDK's Storage library, and your app is caching Storage objects, then you can take advantage of the following request and response headers:

Type	Header	Description
Request	`Oracle-Mobile-Sync-Agent`	When this header is set to `true` in the request, then the response includes either `Oracle-Mobile-Sync-Expires` or `Oracle-Mobile-Sync-No-Store`.
Response	`Oracle-Mobile-Sync-Expires`	Specifies when the returned resource must be marked as expired. Uses RFC 1123 format, for example `EEE, dd MMM yyyy HH:mm:ss z` for `SimpleDateFormat`. This value is determined by the `Sync_CollectionTimeToLive` policy.
Response	`Oracle-Mobile-Sync-No-Store`	When set to `true`, the client mustn't cache the returned resource.

To learn more about data caching, see Data Offline and Sync.

Adding Objects to a Collection

You can populate a collection with objects.

These steps show how to add an object using the UI. When you add an object from the UI, the ID is generated automatically. If you want to assign a specific ID to an object, use the Storage API, the custom code SDK, or the client SDK for your mobile platform. For details, see Storing an Object.

1. On the Storage page, select a collection and click **Open**.
   - If this collection has no objects, click **Upload Files** and then browse to and retrieve the object. Click **Open**.
   - If this collection already has objects, click **Upload** in the Content page. Browse to and retrieve the object. Click **Open**.
2. If the collection is shared, click **Add**. If you have the identity domain administrator role, you can also upload to user isolated collections. Add the user realm and user name to the User Name Required dialog, and click **Ok**. You can only select from users whose roles have been granted permission to the collection. (Assign these roles in the Properties page.)
3. To view the object data, select it from the list.

> **Tip:**
>
> To permanently remove an object from a collection, select it and click **Delete**.

Object Metadata

When you upload an object, the Content page displays basic metadata, such as size, content type, version information, and who uploaded it. Using this page, you can also delete unneeded objects, or filter them. Some functions in user isolated collections are only available if you have the identity domain administrator role.

Property	Value Type	Description/Usage
`ID`	string	The object name, which is used for operations on a single object. It is the last value specified in the URI.
`Content Length`	integer	The size, in bytes.
`Content Type`	media type	The media type for the data, such as `image/jpeg` for a JPEG image, or `application/json` for JSON.
`ETag`	string (an integer in quotes, for example, `"17"`)	A value that represents the version of the object. It's used with the `If-Match` and `If-None-Match` HTTP request headers.
`Created By`	user name	The name of the user who uploaded the data.
`Created On`	time stamp (In ISO 8601)	The time that the object was most recently stored on the server. Time stamps are stored in UTC.
`Modify By`	user name	The name of the user who modified the object.
`Modified On`	time stamp (in ISO 8601)	The time when the server received a request for an object. Time stamps are stored in UTC.
`User ID`	string	For a user isolated collection, the ID of the user whose space the object is in.

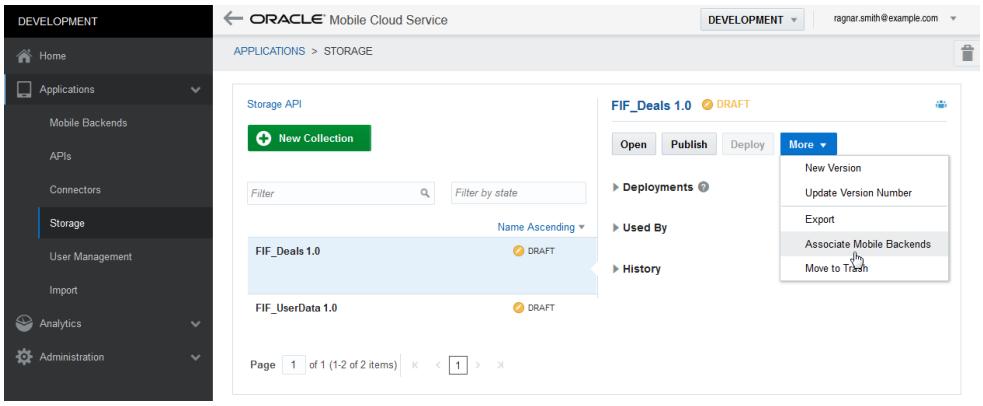
Managing Collections

You can update collections in terms of their contents, but you can’t change the type of the collection. That is, if the collection is a user isolated collection, you can’t change it to a shared collection.

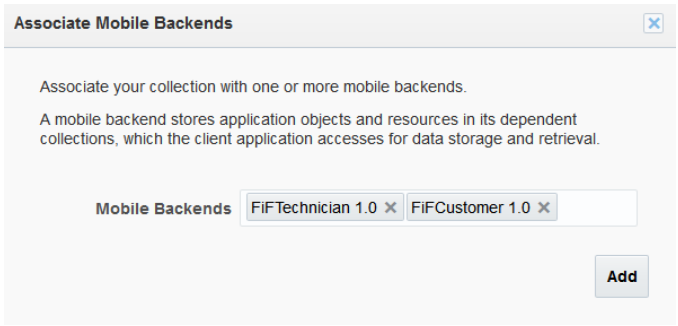
Associating a Collection with a Backend

Associating a collection makes its contents available to a specific backend. The associated collection is a dependency.

1. In the **Storage** page, select a collection.
2. Click **More** and then select **Associate Mobile Backends**.



3. In the Associate Backends dialog, select one or more backends from the list.



4. Click **Add**.

In the details pane, you can see any associated backends by expanding **Used By**.

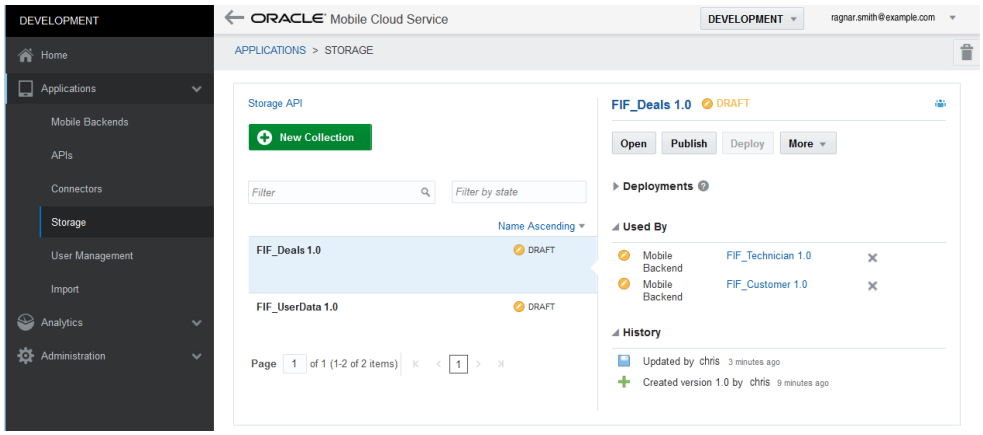
You can also associate a collection with a backend this way:

1. Open the backend.
2. Click the **Storage** tab and then choose **Select Collections**.
3. Choose one or more collections from the Select Collections dialog, and then click **Select**.

Removing a Collection from a Backend

You might want to disassociate a collection from a backend so that you can change the backend's state without affecting the collection. Or you might want to disassociate the collection and associate a different one.

1. In the **Storage** page, select a collection.
2. In the Details section on the right, view the **Used By** list.



3. To delete the association, click the **X** that follows the backend version number.

4. You'll be prompted to remove the dependency. Click **Remove**.

To remove a collection from a backend:

1. Open the backend.
2. Open the **Storage** page.
3. Click the **X** adjacent to the collection that you want to remove.
4. In the Confirm Remove Dependency dialog, click **Remove**.

Calling the Storage API from Your App

To access the Storage API from your app code, you can use the SDK for your platform.

For info on setting up the SDKs, see Connecting Your Application to a Mobile Backend. For complete reference documentation of the SDKs, see Oracle Mobile Cloud Service Help Center.

Here are some code snippets that you can use in your apps once you have your SDK set up.

iOS

The code to retrieve an object might look like this:

```
- (void) downloadData{

    //fill in IDs for collection and object
    NSString* collection_Id = @"";
    NSString* object_Id = @"";
```

```
    // Get storage object
    AppDelegate* appDelegate = [[UIApplication sharedApplication]
delegate];
    OMCMobileBackend* mbe = [appDelegate myMobileBackend];
    OMCStorage* storage = [mbe storage];

    // Get your collection
    OMCStorageCollection* aCollection = [storage
getCollection:collection_Id];

    // Get your object from your collection
    OMCStorageObject* aObject = [aCollection get:object_Id];

    // Get the data from payload of your object
    NSData* data = [aObject getPayloadData];

    lblDownloadStatus.text = @"Download finished";

}
```

Here's code you can use to add an object:

```
- (void) uploadData{

    //Specify a text object to be added to a collection called
"myCollection"
    NSString* collection_Id = @"myCollection";
    NSString* payload = @"This is a simple text object";
    NSString* contentType = @"text/plain";

    if ( payload == nil || [payload isEqualToString:@""]) {

        lblUploadStatus.text = @"There is nothing to upload";
    }
    else{

        // Get storage object
        AppDelegate* appDelegate = [[UIApplication sharedApplication]
delegate];
        OMCMobileBackend* mbe = [appDelegate myMobileBackend];
        OMCStorage* storage = [mbe storage];

        // Get collection where you want to upload new data
        OMCStorageCollection* aCollection = [storage
getCollection:collection_Id];

        // Create new data from payload
        NSData* payloadData = [payload
dataUsingEncoding:NSUTF8StringEncoding];
        OMCStorageObject* aObject = [[OMCStorageObject alloc]
setPayloadFromData:payloadData

withContentType:contentType];
```

```
        // Post data to collection
        [aCollection post:aObject];

        lblUploadStatus.text = @"Upload finished";
    }
}
```

Android

The code to retrieve an object might look like this:

```
private Storage mStorage;
private String collectionID = "YOUR_COLLECTION_ID";
private String objectID = "YOUR_OBJECT_ID";

...

try {
    //Initialize and obtain the storage client
    mStorage =
MobileBackendManager.getManager().getDefaultMobileBackend(this).getServiceP
roxy(Storage.class);
    //Fetch the collection
    StorageCollection collection =
mStorage.getStorageCollection(collectionID);
    //Fetch the object
    StorageObject object = collection.get(objectID);
    //Get the payload
    InputStream payload = object.getPayloadStream();
    //Display the image
    ImageView imageView = (ImageView) findViewById(R.id.imageView);
    imageView.setImageBitmap(BitmapFactory.decodeStream(payload));

} catch (ServiceProxyException e) {
    e.printStackTrace();
}
```

Here's code you can use to add an object:

```
private Storage mStorage;
private String collectionID = "YOUR_COLLECTION_ID";
private String mPayload = "YOUR_PAYLOAD";
private String mContentType = "YOUR_CONTENT_TYPE";

...
//Create or upload an object with specified ID, payload and content-Type
private void uploadObject(String id, String payload, String contentType){
    try {
        //Initialize and obtain the storage client
        mStorage =
MobileBackendManager.getManager().getDefaultMobileBackend(this).getServiceP
roxy(Storage.class);
        //Fetch the collection
        StorageCollection collection =
```

```
mStorage.getStorageCollection(collectionID);
        //Create an object with id, payload and content-Type explicitly
specified
        StorageObject object = new StorageObject(null, payload.getBytes(),
contentType);
        //Upload the object
        collection.post(object);
    } catch (ServiceProxyException e) {
        e.printStackTrace();
    }
}
```

Cordova and JS

Here's code to retrieve an object:

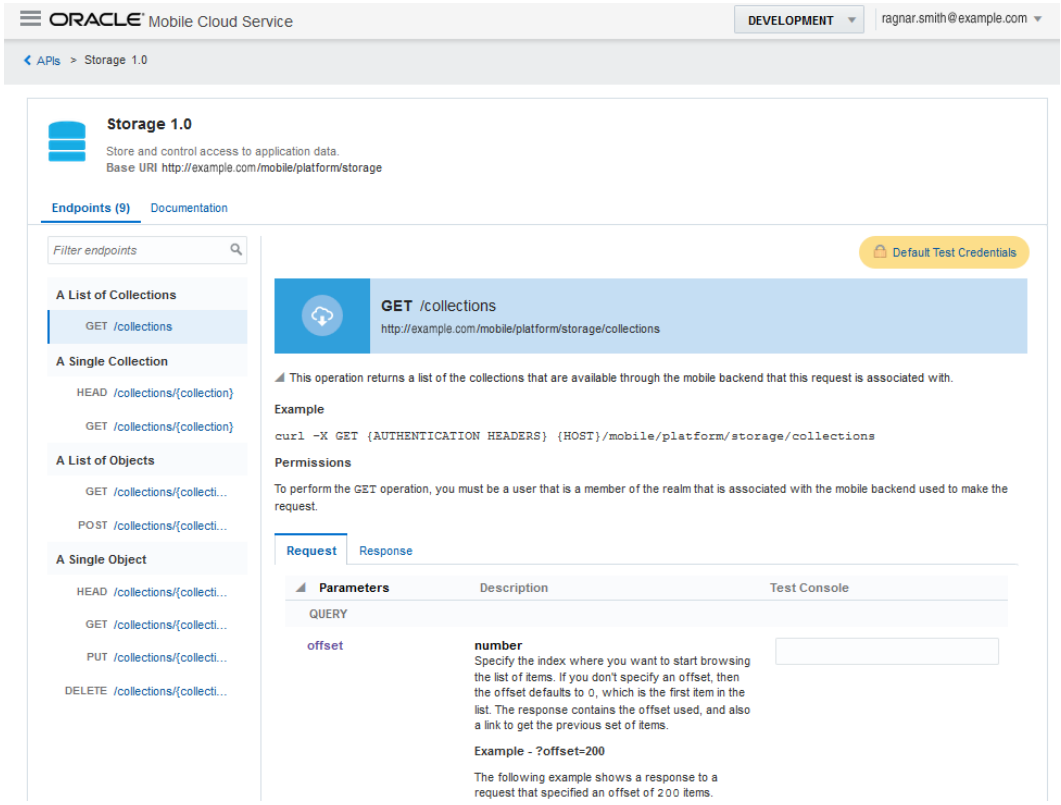
```
function downloadData(){
            varcollectiondId = 'COLLECTION_ID';
            var objectId = 'OBJECT_ID';

      mcs
        .mobileBackend
        .storage
        .getCollection(collectionId, null)
        .then(getCollectionSuccess)
        .then(getObjectSuccess)
        .catch(error);

      function getCollectionSuccess(storageCollection)
{returnstorageCollection.getObject(objectId, 'json');
      }

      function getObjectSuccess(storageObject){
        console.log(storageObject.name);
      }

      function error(error){
        console.log(error.statusCode);
      }
    }

    function uploadData(){
            varcollectiondId = 'COLLECTION_ID';
            varobjectId = 'OBJECT_ID';
            varfileName = 'YOUR_FILE_NAME';
            varpayload = 'YOUR_PAYLOAD';
            var contentType = 'text/plain';

      mcs
        .mobileBackend
        .storage
        .getCollection(collectionId, null)
        .then(getCollectionSuccess)
        .then(getObjectSuccess)
        .catch(error);
```

```
      function getCollectionSuccess(storageCollection){var storageObject =
newmcs.StorageObject(collection);
        storageObject.setDisplayName(fileName);
        storageObject.loadPayload(payload, contentType);return
storageCollection.postObject(storageObject);
      }

      function postObjectSuccess(response){
        console.log(response.storageObject.id);
      }

      function error(error){
        console.log(error.statusCode);
      }
    }
```

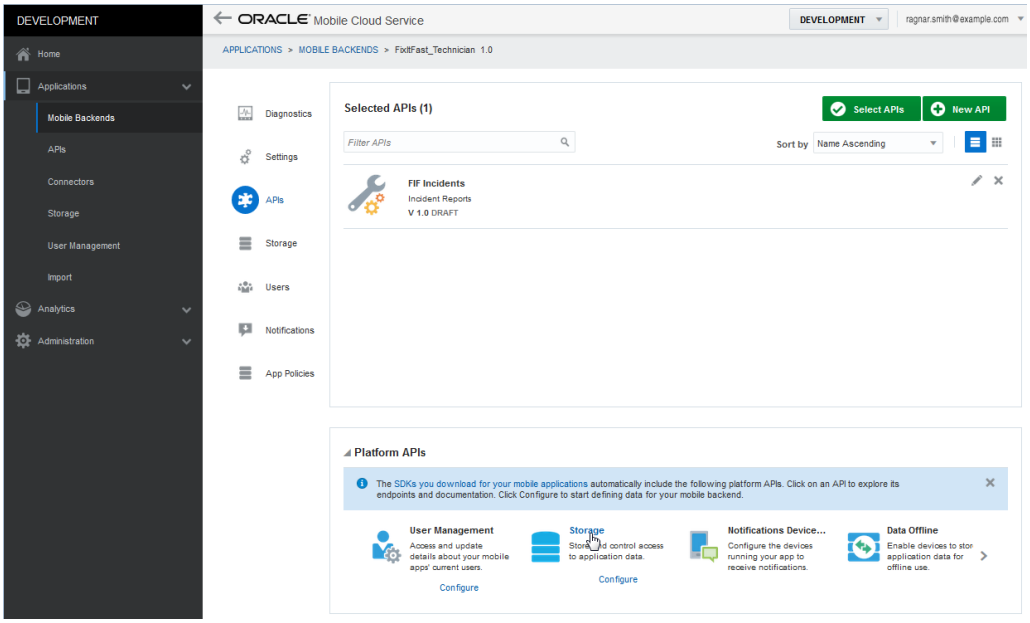
Testing Runtime Operations Using the Endpoints Page

You can test client REST calls for collections manually through a command line tool or utility, from a mobile app running on a device or simulator, or you can use the **Endpoints** page to test various operations.

Using the **Endpoints** page for the Storage API, you can try out basic collection calls, which would typically be exercised by a mobile app. These endpoints would be called directly by calling REST APIs, indirectly (by calling the client SDK), or through custom code. Instead of configuring a device or simulator, or entering the command manually, you can test the API by first entering mobile app user credentials and parameters appropriate to the call and then by clicking **Test Endpoint**. The page displays the payload and the status code.

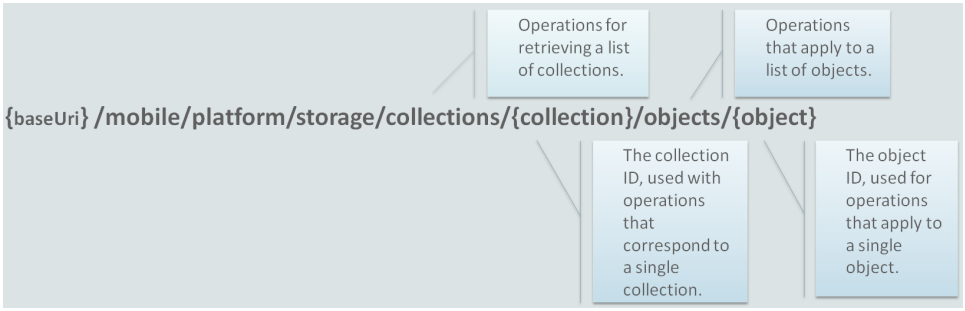


You can access the Endpoints page by clicking **Storage** in Platform APIs section that is located at the bottom of the APIs page for a mobile backend. You can also open the page by clicking **Storage** in the Platform APIs section at the bottom of the APIs page. (You open this page by clicking ☰ to open the side menu. You then click **Applications> Mobile Backends** and then **APIs**).



Storage API Endpoints

The Storage API has endpoints for retrieving, paginating, and ordering collections and also for retrieving, updating, and removing objects.



Here, we give a brief overview of the Storage API endpoints. For detailed information, see REST APIs for Oracle Mobile Cloud Service.

Getting a Single Collection

To get the metadata about a collection, such as ID, description, and whether it is user isolated, call the `GET` operation on the `{collection}` endpoint as follows:

```
GET {baseUri}/mobile/platform/storage/collections/{collection}
```

For example, for a collection named `images`:

```
GET {baseUri}/mobile/platform/storage/collections/images
```

Getting All Collections Associated with a Mobile Backend

To get a list of the collections that are associated with a mobile backend, call the `GET` operation on the `collections` endpoint as follows:

```
GET {baseUri}/mobile/platform/storage/collections
```

Storing an Object

The Storage API has two operations for creating objects. The operation that you use depends on if you want to specify the object's ID or you want the ID to be generated automatically.

- To specify the ID, use `PUT`, and put the ID in the URI as described in Specifying the Object Identifier. Note that you can use the `If-None-Match` header to ensure that you don't overwrite an object that has the same ID, as described in Creating an Object (If One Doesn't Already Exist).
- To generate an ID, use `POST` as described in Generating an Object Identifier.

When you create an object using your own ID, remember that, for shared collections, the ID must be unique to the collection. For user isolated collections, the ID must be unique to the user's space.

Always include the `Content-Type` header to specify the media type of the object being stored. This property also specifies the media type to return when the object is requested. If you don't include this header, then the content type defaults to `application/octet-stream`.

Note that Storage doesn't transform or encode an object. Storage stores the exact bytes that you send in the request. For example, you can't send a Base-64 encoded image and store it as a binary image by including a `Content-Type` header set to `image/jpeg` and a `Content-Encoding` header set to `base64`. You can use a custom API to perform the transformation for you, as shown in the code examples in storage.store(collectionId, object, options, httpOptions).

Specifying the Object Identifier

When performing a `PUT` operation, the identifier of the object corresponds to the last value specified in the URI. For example, to store an object with an ID called `part1524`:

```
PUT {baseUri}/mobile/platform/storage/collections/images/objects/part1524
```

Creating an Object (If One Doesn't Already Exist)

Put the wildcard (`*`) character in the request's `If-None-Match` header to force the `PUT` operation to create the object with the specified object ID only if no other object exists with that ID. Specifying the wildcard causes the call to fail if another object already exists with the same ID. For example:

```
PUT {baseUri}/mobile/platform/storage/collections/images/objects/part1542

Headers:
  If-None-Match: *
```

Generating an Object Identifier

To generate the identifier for an object and then store the object, use the `POST` operation. Unlike the `PUT` operation, there's no identifier specified at the end of the URI for a `POST` operation. For example:

```
POST {baseUri}/mobile/platform/storage/collections/images/objects
```

The URI that accesses the newly created object is returned through the `Location` header in the response, and the `ID` attribute is included in the response body.

What Happens When an Object is Created?

When an object is created:

- The content is stored.

- The value of the `Content-Type` field in the request is stored. (This becomes the `Content-Type` field definition returned when the object is requested using a `GET` operation.)
- An entity tag (ETag) value is assigned.
- The `createdBy` value is set to the user ID of the user who performed the create operation.
- The `createdOn` value is set to the time the object was stored on the server.

Updating an Object

Objects are updated using the `PUT` operation. For the `PUT` call, specify the same identifier that was specified or generated when the object was created. Because objects are opaque, updating an object completely replaces the previous contents.

What Happens When an Object Is Updated?

When a `PUT` is performed on an object, the following occurs:

- The content is completely replaced.
- The value of the ETag changes.
- The `modifiedBy` value is set to the user ID for whom the mobile app performed the `PUT` operation.
- The `modifiedOn` value is set to the time the object was stored on the server.

Optimistic Locking

Optimistic locking is a strategy to use when you want to update an object only if object was not updated by someone else after you originally retrieved it. To implement this strategy, do one of the following:

- Put the timestamp of when you last retrieved the object in the `If-Unmodified-Since` header.
- Put the object's ETag in the `If-None-Match` header.

For example, if the ETag value from the previous call is `2`, then the `PUT` operation in the following example is performed only when the `If-None-Match` value of `"2"` matches the ETag of the object (`part1524`). If the versions don't match, then the call's `PUT` operation isn't performed and `part1524` remains unchanged.

```
PUT{baseUri}/mobile/platform/storage/collections/images/objects/part1524

Headers:
  If-None-Match: \"2\"
```

You can get a similar result using `If-Unmodified-Since`:

```
PUT {baseUri}/mobile/platform/storage/collections/images/objects/part1524

Headers:
  If-Unmodified-Since: Mon,30 Jun 2014 19:43:31 GMT
```

Retrieving Objects

You can get a list of the objects in a collection, and you can get an object.

Retrieving a List of Objects

To get the metadata about a set of objects in a collection, use the `GET` operation on the `/collections/{collection}/objects` endpoint. This metadata includes the object's ID, its name, and size. The metadata also includes the canonical link and self links. For a full list of properties, see Taking a Look at Object Metadata.

In this example, `images` is the name of a shared collection.

```
GET {baseURI}/mobile/platform/storage/collections/images/objects
```

If the collection is user isolated and you have `READ_ALL` or `READ_WRITE_ALL` access, then you must include the `user` query parameter and specify which user's objects you want listed, even if you want to see your own objects (use `*` to list all user's objects). Note that you provide the user's ID, not the user name. For example:

```
GET {baseURI}/mobile/platform/storage/collections/images/objects?
user=0cea04ee-9e26-4de3-ad6b-00a66c8d3b96
```

Paging Through a List of Objects

If you don't want to see all the results, or if you want to get the results in small blocks, use the `limit` and `offset` query parameters to request a subset of items.

Use the `limit` parameter to restrict the number of items returned. The default is `100`. Define `offset` as the zero-based starting point for the returned items. The returned JSON body contains links for retrieving both the next and previous sets of items.

The following example gets the metadata for 50 objects, starting with the 201st object.

```
Get {baseUri}/mobile/platform/storage/collections/images/objects?
offset=200&limit=50
```

Ordering

Use the `orderBy` parameter to control the order of the returned items. You can specify which property to order on and specify whether to put the items in ascending (`asc`) or descending (`desc`) order:

```
Get {baseUri}/mobile/platform/storage/collections/images/objects?
orderBy=contentLength:desc
```

You can sort by the `name`, `modifiedBy`, `modifiedOn`, `createdBy`, `createdOn`, or `contentLength` property.

> **Note:**
>
> You can order by one property only (either `asc` or `desc`).

Querying

Use the `q` query parameter to restrict the list of returned objects to the value specified for the `id`, `name`, `createdBy`, or `modifiedBy` attributes.

```
Get {baseUri}/mobile/platform/storage/collections/images/objects?q=part
```

The objects returned are based on a case-sensitive, partial match of the `id`, `name`, `createdBy`, and `modifiedBy` attributes. With this example, the results might include an item with an ID of `part1524` and an item modified by `bonapart`.

Retrieving an Object

Use the `GET` operation to retrieve the entire object. When performing the `GET` operation, the identifier (such as `part1524` in the following example) is specified at the end of the URI.

Storage always returns the exact bytes that were stored. If the `Accepts` header doesn't match the `Content-Type` that the object was stored with, then it returns a `406` status code.

In this example, the object is returned only if the Etag does not match. You can use this strategy prevent re-fetching an object if it hasn't changed.

```
Get {baseUri}/mobile/platform/storage/collections/images/objects/part1524

Headers:
  If-None-Match: \"2\"
```

Deleting an Object

To remove an object from a collection, call the `DELETE` operation. Deleting an object is permanent. There's no way to restore an object after you call this operation.

```
DELETE {baseUri}/mobile/platform/storage/collections/images/objects/
part1524
```

To safely remove an object, use the `If-None-Match` header with the object's ETag, or the `If-Unmodified-Since` header with the timestamp of when you last retrieved the object:

```
DELETE {baseUri}/mobile/platform/storage/collections/images/objects/
part1524

Headers:
  If-None-Match: \"2\"
```

As described in Updating an Object, you can use these headers to prevent overriding a change that another user made after you originally retrieved the object.

Optimizing Performance

You can use these strategies to optimize performance when you retrieve an object:

- Check If Exists
- Get If Newer
- Reading Part of an Object (Chunking Data)

Check If Exists

To check if an object exists, use the `HEAD` operation instead of a `GET` operation. The `HEAD` operation returns the same information except for the actual object value.

Put If Absent

You can use the `If-None-Match` header with a wildcard (`*`) value in a `PUT` operation to store an object only when (or if) it isn't already included in the collection.

When you use this strategy, the call executes only when the ETag is absent, which is true only if the object does not exist.

```
PUT {baseUri}/mobile/platform/storage/collections/profiles/objects/uprofile

Headers:
  If-None-Match: *
```

In this example, if the `uprofile` object doesn't have an ETag, then `myProfile.txt` is stored as the `uprofile` object.

Get If Newer

If you have already retrieved an object, and you want to re-fetch it only if it has changed, use the `GET` operation with the `If-None-Match` or `If-Modified-Since` header to retrieve the object only if there has been a change since the last time the object was fetched.

- **If-None-Match**

  This example re-fetches the object only if the ETag is not `2`.

  ```
  GET {baseUri}/mobile/platform/storage/collections/images/objects/
  part1542

  Headers:
    If-None-Match: \"2\"
  ```

- **If-Modified-Since**

This example re-fetches the object only if it was modified after the date and time specified. Otherwise, the response status is `304 not modified`.

```
GET {baseUri}/mobile/platform/storage/collections/images/objects/
part1542

Headers:
  If-Modified-Since: Mon, 30 Jun 2014 19:43:31 GMT
```

Reading Part of an Object (Chunking Data)

If the mobile app needs to get a large object like a video file, you can use the `Range` header to retrieve a subset of the object. This field lets the mobile app retrieve the data in chunks, rather than all at once, by requesting a subset of bytes. Using this strategy, you can start streaming a video, or start displaying the contents of a long list before you fetch the whole object.

Here are examples of byte-range specifier values:

- First 100 bytes: `bytes=0-99`
- Second 100 bytes: `bytes=100-199`
- Last 100 bytes: `bytes=-100`
- First 100 and last 100 bytes: `bytes=0-99,-100`

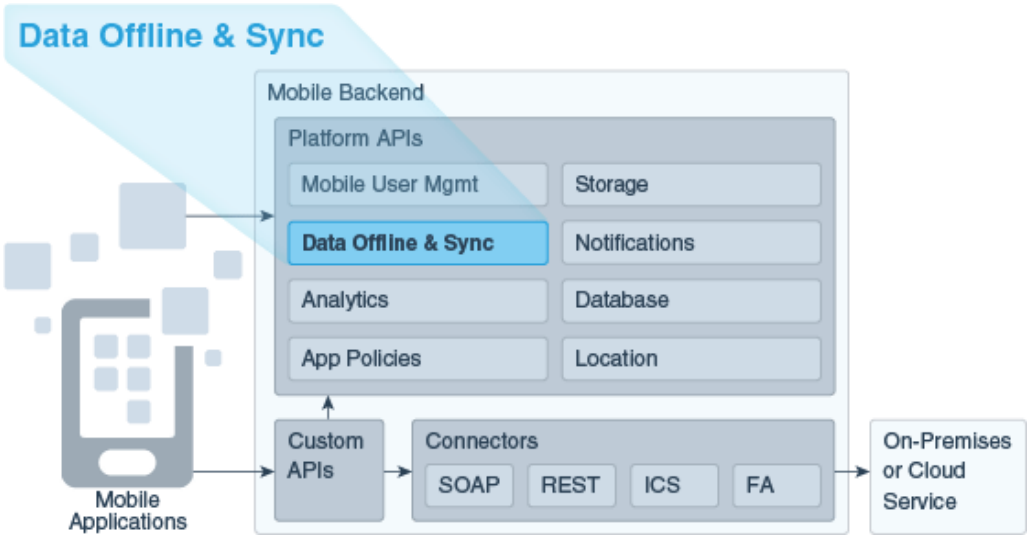
This example gets the first 100 and last 100 bytes of a profile to display a preview of the object's contents:

```
GET {baseUri}mobile/platform/storage/collections/profiles/objects/uprofile

Headers:
  Range: bytes=0-99,-100
```

15

Data Offline and Sync

Mobile app developers can use the Data Offline and Sync features to build a client app that enables the users to perform critical tasks when offline.



You can use the following APIs to build applications that cache REST resources for offline use and then synchronize all offline changes with the server when the device goes online again.

API	Platforms	Features
Sync Express	• Cordova • JavaScript	• Basic synchronization. • Easy to use. • Works with any REST API where the resource name alternates between plural nouns and singular resource identifiers (rid), such as `/items/{rid}/subitems/{rid}`. • Requires minimal changes to existing code. • Works with any JavaScript framework. • When device reconnects, sends change requests one resource object at a time. • Always overwrites the server version of the object.
Synchronization	• Android • iOS	• Robust synchronization. • Works with synchronization-compliant custom APIs. • When device reconnects, sends all changes in one request. • Provides choices for what to do if the server version of an object changes while edits were made offline (server wins, client wins, preserve conflict). • Provides choices for how long to store resource objects on the device, when to refresh data from the server, and which resources can be edited when offline. • Automatically synchronizes with the Storage platform.

Building Apps that Work Offline Using Sync Express

The Javascript and Cordova client SDKs feature Sync Express, which enables you to easily and quickly make your application work offline using your existing REST requests. You can use this library for REST APIs where the resource name alternates

between plural nouns and singular resource identifiers (rid), such as `/items/{rid}/subitems/{rid}`.

Adding Sync Express to Your App

To use Sync Express in your app, you must complete the following tasks.

- Copy both `mcs.sync.min.js` and `mcs.min.js` from the SDK into the directory where you keep your JavaScript libraries.
- Use a `script` element to load `mcs.sync.min.js`. This must be the first script that the app fetches and loads unless you add `loki-cordova-fs-adapters.js`, which is explained next.
- Use either RequireJS or a `script` element to load `mcs.min.js`.
- From the command line, enter the following to add the cordova-plugin-network-information plugin. This plugin enables Sync Express to detect if the device is online or offline.

  ```
  cordova plugin add cordova-plugin-network-information
  ```

When an application attempts to store more REST resources than the device's cache size allows, Sync Express throws a `QUOTA_EXCEEDED_ERR` exception. With Cordova apps, you can install the cordova-plugin-file to increase the device's cache size. This plugin isn't available for JavaScript web apps.

1. To install and use the cordova-plugin-file.

   ```
   cordova plugin add cordova-plugin-file
   ```

2. Copy `loki-cordova-fs-adapters.js` from the SDK into the directory where you keep your JavaScript libraries.
3. Add a script element to load `loki-cordova-fs-adapter.js`. This must be the first script that the app fetches and loads. Then the app can load `mcs.sync.min.js` and `mcs.min.js` as described above.

Configuring Your App to Use Sync Express

To enable Sync Express, add a `syncExpress` entry to `oracle_mobile_cloud_config.js`, and use `path` elements in the `policies` array to identify the endpoints that you want to activate Sync Express for. The name that you use for a path parameter must exactly match the name of the property that uniquely identifies a returned object. Use a colon to identify the path parameter, such as `:deptId`.

> **Note:**
>
> The configuration file can have a `syncExpress` entry for Sync Express or a `sync` entry for the Synchronization library, but it can't have both.

Let's say, for example, that you want to activate Sync Express for all calls to these endpoints:

- `/departments`

- `/departments/{deptId}`

The department database object has these properties:

```
deptId: number
name: string
```

The response object for a department collection looks like this:

```
[
  {
    "deptId": 1,
    "name": "Department 1"
  },
  {
    "deptId": 2,
    "name": "Department 2"
  }
]
```

The corresponding `syncExpress` entry would look like this. Notice that you need only one entry in the configuration file to activate Sync Express for both endpoints.

```
var mcs_config = {
  "logLevel": mcs.LOG_LEVEL.INFO,
  "mobileBackends": {
    "myBackend": {
    ...
    }
  }
  "syncExpress": {
    "policies": [
      {
        "path": '/mobile/custom/myApi/departments/:deptId(\\d+)?'
      }
    ]
  }
};
```

Now let's say, for example, that you want to include calls to endpoints with subcollections (nested entities), such as an employees within a department:

- `/departments`
- `/departments/{deptId}`
- `/departments/{deptId}/employees`
- `/departments/{deptId}/employees/{empId}`

The employee database object has these properties:

```
deptId: number
empId: number
name: string
```

The response object for an employee collection looks like this:

```
[
  {
    "empId": 1,
    "name": "John Doe"
  },
  {
    "empId": 2,
    "name": "Jane Doe"
  }
]
```

The corresponding `syncExpress` entry would look like this. Notice that you need only one entry in the configuration file to activate Sync Express for all the endpoints.

```
var mcs_config = {
  "logLevel": mcs.LOG_LEVEL.INFO,
  "mobileBackends": {
    "myBackend": {
    ...
    }
  }
  "syncExpress": {
    "policies": [
      {
        "path": '/mobile/custom/myApi/departments/:deptId(\\d
+)/:_employees?/:empId(\\d+)?'
      }
    ]
  }
};
```

Sync Express provides some regular expressions for formulating the path specification:

- Use a colon (:) plus the property name to indicate either a path parameter or the name of the property that uniquely identifies each returned object (or both). For example, for the `/departments` endpoint, you must include `:deptId(\\d+)` in the path specification to indicate the unique identifier for a department resource, even if the API didn't have a `/mobile/custom/myAPI/departments/{deptId}` endpoint.
- Use a question mark (?) to indicate that the path parameter is optional.
- When a path segment represents a collection of children resources (a subcollection), then you must precede the parameter name with a colon and an underscore (`:_`) so that Sync Express stores the response objects in the client cache as children objects that are associated with the parent object.
- By default, Sync Express assumes that the path parameter is a string. Use `(\\d+)` to indicate that the path parameter must be a numeric value.

For example, given the `/mobile/custom/myApi/departments/:deptId(\\d+)/:_employees?/:empId(\\d+)?` path specification:

- `:deptId` specifies a path parameter and also provides the name of the property in the department object that uniquely identifies a department.
- The `?` after `:deptId(\\d+)` indicates that this and subsequent parameters are not required. Thus, the path specification applies to these endpoints:
  - `/mobile/custom/myApi/departments`
  - `/mobile/custom/myApi/departments/{deptId}`
  - `/mobile/custom/myApi/departments/{deptId}/employees`
  - `/mobile/custom/myApi/departments/{deptId}/employees/{empId}`
- `(\\d+)` indicates that the path parameter value must be numeric. If the object's `deptId` property is a string, then you'd use `/mobile/custom/myApi/departments/:deptId?` instead.
- `(:_employees)` identifies a subcollection and indicates that all response objects must be stored in the client cache as children of the specified `deptId`.

Configuring Your App to Handle items Arrays

If any response bodies wrap a collection in an `items` property, such as `"items":[{"id:":33},{"id:":34}]`, then you must add the Oracle REST handler to the `syncExpress` entry in the configuration file, as shown in the following example:

```
var mcs_config = {
  "logLevel": mcs.LOG_LEVEL.INFO,
  "mobileBackends": {
    "myBackend": {
    ...
    }
  }
  "syncExpress": {
    "handler": "OracleRestHandler",
    "policies": [
      {
        "path": '/mobile/custom/myApi/departments/:deptId(\\d+)?'
      }
    ]
  }
};
```

Making Your App Synchronize Offline Changes Automatically

To make an app synchronize offline changes with the server automatically, add code to refresh the user interface when the device re-connects (goes online) by making explicit REST calls, which then flush pending changes automatically.

Building Apps that Work Offline Using the Synchronization Library

Use the Synchronization library from Android and iOS mobile apps to enable the app users to continue to use the app when offline.

What Can I Do with the Synchronization Library?

When developing Android and iOS client apps, you, as a mobile app developer, might often take these goals into consideration:

- Enable updates to app data on mobile devices when connectivity is intermittent or non-existent.
- Improve performance by minimizing the amount of calls and data transported over the wire.

The client SDK's Synchronization library, with its data caching, support for offline operations, and automated synchronization, enables you to achieve these goals when you access custom API resources. In addition, through declarative policies, you can design caching and synchronization policies for your custom APIs that you can apply across your apps, and adjust without having to modify code.

Using the Synchronization Library to Enable Edits to App Data When the Mobile Device Is Offline

As an example of how you can use the Synchronization library to enable app users to read, create, update, and delete data when the mobile device is offline, consider some apps that are designed for the Fix it Fast (FiF) company, which maintains in-house appliances. The mobile app developer wants to ensure that the apps continue to work even when there is no internet connection. For example:

- A customer uses an FiF mobile app to fill out the details for an incident report regarding a basement furnace. She then goes to the basement to take a picture of the furnace's barcode, attaches it to the report, and taps Send. Even though there's no internet connection in the basement, the app should enable the customer to access, change, and send the incident report. As soon as the device reconnects to the internet, the app should transmit the report and the attached photo to the server.
- During the day, a technician reviews her job list, sorts the jobs by priority, driving distance, and issue type, and adjusts the priorities as needed. As she completes a job, she attaches notes to the incident report, and she updates the job status. She expects to be able to do all these tasks even when she doesn't have access to the internet. When her device is connected, she expects the app to synchronize her offline modifications with the server, first synchronizing the essential information, such as job status, and then synchronizing the less essential information, such as her notes.
- After an unexpectedly long repair, the technician lowers the priority for customer that is the furthest away, John Doe. Because she is offline, her modifications are stored in the offline edits in the local cache. During the time she was offline, John Doe called the office to report that his water heater was now leaking, and the office changed his priority to high. When the technician goes back on line, the app synchronizes the updates, and sees that there is a conflict. The app pops up a notice about the conflict and asks the technician if she still wants to lower the priority.

To implement these data offline requirements, the mobile app developer uses the Synchronization library to fetch and update data, and sets the appropriate fetch, update, and conflict resolution policies in the configuration file.

- To ensure that incident reports from the `/incidents` resource are always available, that they can be modified while offline, and that the server is updated with queued offline modifications as soon as the device resumes access, the mobile app developer sets the following policies for the resource:
  - Fetch policy: Fetch resources from the server when the client application is online, and fetch them from the local cache when the app is offline (`FETCH_FROM_SERVICE_IF_ONLINE`).
  - Update policy: Queue updates if offline and synchronize automatically when the client app is back online (`QUEUE_IF_OFFLINE`)).
- To ensure that two technicians don't inadvertently update the same status or priority for an `/incidentstatus` resource due to queued offline updates, the mobile app developer sets the following policy:
  - Conflict resolution policy: Don’t overwrite the server’s version with the local version if there’s a conflict. The edited local version is kept in the offline edits in the local cache, and the mobile app handles the conflict (`PRESERVE_CONFLICT`).

> **Note:**
>
> This assumes that the code for this custom API returns the correct information, such as the ETag that is used to detect conflicts, as described in Returning Cacheable Data.

To learn about all the data offline policy options, see Synchronization Policies.

Using the Synchronization Library to Improve Performance

As an example of how you can use the Synchronization library to improve performance, consider the FiF apps that we discussed previously.

- Before leaving the office every morning, the technicians start an FiF app on their tablets, and pull a list of their jobs for the day. Because the customer information such as name, phone, and address is static, the app can cache that data upon startup and not re-retrieve it during the day to improve performance. Other information, such as incident status and priority, must be kept current.
- Expired data needs to be cleared whenever the app is restarted.
- The finance department designed an API that supplies a customer's default credit card information. Because the information is fairly static, mobile apps might consider caching that information to improve performance. However, the finance department wants to ensure that mobile apps never cache that information.

To implement these performance requirements, the mobile app developer uses the Synchronization library to fetch and update data, and sets the appropriate fetch, expiration, and eviction policies in the configuration file.

- To cache the information from the `/customer` resource so that it's retrieved from the server on startup, and, after that from the local cache only, the mobile app developer sets the following policies:
  - Expiration policy: Mark resources as expired when the client application restarts (`EXPIRE_ON_RESTART`).

  - Eviction policy: Delete expired resources from the local cache when the client application restarts (`EVICT_ON_EXPIRY_AT_STARTUP`).
  - Fetch policy: Fetch resource from the server only if it isn't in the local cache or is expired (`FETCH_FROM_SERVICE_ON_CACHE_MISS_OR_EXPIRY`).
- To ensure that the priority and status from the `/incidentstatus` resource is always available, but stays as current as possible:
  - Fetch policy: Fetch resources from the server when the client application is online, and fetch them from the local cache when the app is offline (`FETCH_FROM_SERVICE_IF_ONLINE`).
  - Eviction policy: Delete expired resources from the local cache when the client application restarts (`EVICT_ON_EXPIRY_AT_STARTUP`).
  - Expiration policy: Mark a resource as expired when the client application restarts. Update the local cache with the latest version from the server the next time the client application calls the resource (`EXPIRE_ON_RESTART`).
- To ensure that none of the information from the `/creditcards` resource is cached, the custom code that implements this API makes sure that all HTTP responses include the `Oracle-Mobile-Sync-No-Store` header set to `true`.

To learn about all the data caching policy options, see Synchronization Policies. To learn about the synchronization headers, see Defining Synchronization Policies and Cache Settings in a Response Header.

Synchronization Library Process Flow

To help you understand how the parts fit together, here's an explanation of how the Synchronization library does the following:

- Manages objects in the local cache
- Uses synchronization policies to retrieve resources from either the local cache or the server
- Handles object updates

When the mobile app makes a request through the Synchronization library to get data from a custom API, the Synchronization library looks at the fetch policy setting to determine whether to get the objects from the server or the local cache. Whenever the Synchronization library fetches objects from the server, it refreshes the local cache with the newly fetched objects.

Depending on the policy settings, the Synchronization library might also periodically refresh expired items in the local cache using a background process.

When the user edits an object, the following occurs depending on whether the mobile device is online or offline:

- Online edit: An update request is sent to the server.
- Offline edit: The edited object is stored in the offline edits in the local cache. When the app goes online, a background process sends a request to update the resource on the server.

If the conflict resolution policy is `CLIENT_WINS`, the update request includes an `If-Match` header of `*` so that the server updates the resource without conflict. Otherwise the request includes an `If-Match` header that is set to the ETag that was last returned by the server.

To learn more about the synchronization policy types and options and how to set them, see Synchronization Policies.

Video: Overview of the Data Offline & Synchronization API

To learn more about how the Synchronization library uses caching to enable a client app to work offline as well as improve performance, take a look at this video:

Video

Android Synchronization Library

This section shows how to use the Synchronization library to implement several of the common data offline tasks for working with a custom API's resources.

For detailed information about the library, see Oracle Mobile Cloud Service Android SDK Reference.

> **Tip:**
>
> The client SDK ZIP file contains an `examples` folder, which contains the source code for the SalesPlus app. This app illustrates many of the synchronization features that are described in this section.

Setting Up Your Mobile App for the Android Synchronization Library

1. Ensure that the `AndroidManifest.xml` file contains the following entries. `WRITE_EXTERNAL_STORAGE` lets the Synchronization library maintain the local cache. `ACCESS_NETWORK_STATE` lets the Synchronization library determine the connection status.

   ```
   <uses-permission
   android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
   <uses-permission
   android:name="android.permission.ACCESS_NETWORK_STATE" />
   ```

2. Ensure that the correct policies are in place for the mobile backend and API endpoints as described in Synchronization Policy Levels and Precedence and Defining Synchronization Policies Using a Configuration File.
3. As with all mobile apps, instantiate `MobileBackendManager`, and then instantiate `MobileBackend` to manage connectivity, authentication, and other transactions between your application and its associated mobile backend, including calls to platform and custom APIs.
4. To access the custom APIs from the Synchronization library, get the mobile backend's synchronization service.

   ```
   try {
       Synchronization synchronization =
               MobileBackendManager.getManager().
               getDefaultMobileBackend(this).
               getServiceProxy(Synchronization.class);
   ```

```
} catch (ServiceProxyException e) {
    e.printStackTrace();
}
```

Fetching Resources

After you set up your app to work with data offline, you use the mobile endpoint class to open endpoints to custom code API resources, and you use fetch builders to synchronize data retrieval and modifications with the local cache automatically. A fetch builder enables you to specify how to fetch the data, and then enables you to execute the fetch.

1. To access an endpoint, instantiate `MobileEndpoint` for that endpoint. This example instantiates an endpoint for `/mobile/custom/incidentreport/incidents`.

```
// open Endpoint
MobileEndpoint endpoint =
  synchronization.openMobileEndpoint(
    "incidentreport",
    "incidents",
    MobileObject.class);
```

2. (Optional) Add objects or files to the collection. This example adds an object.

```
MobileObject newObject = endpoint.createObject();
JSONObject payload = new JSONObject();
// Set properties
try {
    payload.put("title", "incident 213");
    ...
} catch (JSONException e) {
    ...
}
newObject.initialize(null, endpoint, payload);
// Add incident
newObject.saveResource(new MobileEndpointCallback() {
    @Override
    public void onComplete(Exception exception, MobileResource
mobileResource) {
        //This function is called when the request completes
        ...
    }
});
```

3. Use a fetch builder to specify how to fetch the objects from the endpoint. The fetch builder method that you use depends on whether you want to retrieve an object, a collection, or a file:
   - `FetchObjectBuilder`
   - `FetchCollectionBuilder`
   - `FetchFileBuilder`

Here's an example of creating a fetch builder for a collection.

```
FetchCollectionBuilder fetchCollectionBuilder = endpoint.fetchObjects();
```

In this example, we want to filter all the incidents for the signed-in technician (which is the same as the user name). The API provides a query parameter for technician, so we can tell the builder to add that query parameter to the request:

```
fetchCollectionBuilder =
fetchCollectionBuilder.withQueryParameter("technician", username);
```

> **Tip:**
>
> You can call `withQueryParameter` as many times as you need to specify all the query parameters.

4. Add necessary headers.

   In this example, to enable easy searching for all diagnostic log entries associated with this fetch builder, the request includes the `Oracle-Mobile-Diagnostic-Session-ID` header. The `mDiagLogFilterTag` string variable has been set to a value that uniquely identifies requests that are made using this fetch builder.

   ```
   fetchCollectionBuilder.withHeader("Oracle-Mobile-Diagnostic-Session-
   ID", mDiagLogFilterTag);
   ```

5. Use the builder to execute the fetch.

   ```
   fetchCollectionBuilder.execute(new MobileEndpointCallback(){
       @Override
       public void onComplete(Exception exception, MobileResource
   mobileResource) {
           //This function is called when the request completes
           ...
           MobileObjectCollection collection = (MobileObjectCollection)
   mobileResource;
       }
   });
   ```

   If the fetch policy is to fetch the data from the local cache, such as `FETCH_FROM_SERVICE_ON_CACHE_MISS`, then it's fetched from the local cache if available. In all other cases, the collection is fetched from the server if the policy allows. If the `noCache` setting is false, then the results are saved to a local cache.

6. The raw downloaded JSON object is exposed through the `JsonObject` property. Use this property to set the appropriate values.

   ```
   List objectsList = collection.getObjectsList();
   MobileObject incidentMobileObject = (MobileObject)
   objectsList.get(index);
   JSONObject json = incidentMobileObject.getJsonObject();
   ```

```
// This updates incidentMobileObject
json.put("status", "completed");
```

7. Use one of the `MobileObject` save methods to save the changes on the server.

```
incident.saveResource(new MobileEndpointCallback(){
    @Override
    public void onComplete(Exception exception, MobileResource
mobileResource) {
        ...
    }
});
```

If the device isn't connected to the internet, and the update policy is `UPDATE_IF_OFFLINE`, then the library saves the changes to the local cache. The Synchronization library sends the changes to the server automatically when the device reconnects with the internet.

8. Use one of the `MobileObject` delete methods to delete an object.

```
incident.deleteResource(new MobileEndpointCallback(){
    @Override
    public void onComplete(Exception exception, MobileResource
mobileResource) {
        ...
    }
});
```

If the client is offline, then the library deletes the object in the local cache. It deletes the object on the server when the client is online again.

Fetching Filtered Resources

You might have an app that filters which items it displays. For example, an FiF app might want to display all incidents with a status of `new`. When the device is online, your code can fetch the items as `mobileResource` objects, convert the objects to JSON objects, and then filter the items. However, when the device is offline, your app can't filter the `mobileResource` objects in the local cache because the objects are just blobs of data. The solution is to use a custom `MobileObject`. When you do this, the local cache stores the data in a table with a column for each of the custom object's fields, which enables your mobile app to query data in the local cache based on field values. We'll use the incident list in the FiF example to illustrate how to do this. In this example, the users must be able to filter the incident list by status.

When you open a mobile endpoint on a custom `MobileObject` class, you can use the fetch builder's `queryFor` method to specify the filter to use in the local cache. Note that this method is for filtering JSON objects from the local cache. It doesn't affect the way that the Synchronization library retrieves results from the server. Whenever you execute the fetch builder, the library first looks at the fetch policy setting to determine whether to refresh the local cache. If the policy specifies that it must refresh the local cache from the server, then it retrieves all the objects, regardless of the filter that you specify using the `queryFor` method. Regardless of the fetch policy and whether it refreshed the local cache, the library then uses the `queryFor` method to filter the data in the local cache, and return the filtered results. That is, regardless of whether the

device is online or offline, and regardless of whether the library fetches data from the server or uses the local cache, the `queryFor` method filters the results based on the query property and value.

1. Create a class that extends `MobileObject`. Add a property for every field that you'll use in the app. Then override `onDataLoad()` and `getPropertyNames()` and create getters and setters for the fields. Here's an example of creating an `IncidentCustomMobileObject` class.

```
public class IncidentCustomMobileObject extends MobileObject {
    private int id;
    private String title;
    private String technician;
    private String customer;
    private String status;
    private String priority;
    private String createdBy;
    private String createdOn;
    private String modifiedBy;
    private String modifiedOn;

    // This method tells the Synchronization library how to get the
values from the JSON object.
    @Override
    protected void onDataLoad(){
        try{
            if(jsonObject != null){
                title = jsonObject.has("title") ?
jsonObject.getString("title") : "";
                technician = jsonObject.has("technician") ?
jsonObject.getString("technician") : "";
                customer = jsonObject.has("customer") ?
jsonObject.getString("customer") : "";
                status = jsonObject.has("status") ?
jsonObject.getString("status") : "";
                createdBy = jsonObject.has("createdBy") ?
jsonObject.getString("createdBy") : "";
                createdOn = jsonObject.has("createdOn") ?
jsonObject.getString("createdOn") : "";
                modifiedBy = jsonObject.has("modifiedBy") ?
jsonObject.getString("modifiedBy") : "";
                modifiedOn = jsonObject.has("modifiedOn") ?
jsonObject.getString("modifiedOn") : "";
                priority = jsonObject.has("priority") ?
jsonObject.getString("priority") : "";
            }
        } catch (Exception e){
            e.printStackTrace();
        }
    }

    // The Synchronization library uses this method to determine the
column names and data
    // types for the database table for the local cache.
    @Override
```

```
    public void getPropertyNames(Map<String,PropertyType> properties,
List<List<String>> indexes){
        properties.put("title", PropertyType.String);
        properties.put("technician", PropertyType.String);
        properties.put("customer", PropertyType.String);
        properties.put("status", PropertyType.String);
        properties.put("createdBy", PropertyType.String);
        properties.put("createdOn", PropertyType.String);
        properties.put("modifiedBy", PropertyType.String);
        properties.put("modifiedOn", PropertyType.String);
        properties.put("priority", PropertyType.String);
    }

    //Getters and Setters

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getTechnician() {
        return technician;
    }

    public void setTechnician(String technician) {
        this.technician = technician;
    }

    public String getCustomer() {
        return customer;
    }

    public void setCustomer(String customer) {
        this.customer = customer;
    }

    public String getStatus() {
        return status;
    }

    public void setStatus(String status) {
        this.status = status;
    }
```

```
    public String getPriority() {
        return priority;
    }

    public void setPriority(String priority) {
        this.priority = priority;
    }

    public String getCreatedBy() {
        return createdBy;
    }

    public void setCreatedBy(String createdBy) {
        this.createdBy = createdBy;
    }

    public String getCreatedOn() {
        return createdOn;
    }

    public void setCreatedOn(String createdOn) {
        this.createdOn = createdOn;
    }

    public String getModifiedBy() {
        return modifiedBy;
    }

    public void setModifiedBy(String modifiedBy) {
        this.modifiedBy = modifiedBy;
    }

    public String getModifiedOn() {
        return modifiedOn;
    }

    public void setModifiedOn(String modifiedOn) {
        this.modifiedOn = modifiedOn;
    }


}
```

2. Open the endpoint for the custom class.

```
MobileEndpoint endpoint =
  synchronization.openMobileEndpoint(
    "incidentreport",
    "incidents",
    IncidentCustomMobileObject.class);
```

3. When you create the fetch builder, use the `queryFor` method to add a query to filter the results by status.

```
FetchCollectionBuilder fetchCollectionBuilder = endpoint.fetchObjects();
fetchCollectionBuilder = fetchCollectionBuilder.queryFor(
  "status",
  Comparison.Equals,
  "pending");
```

4. Fetch the data.

```
fetchCollectionBuilder.execute(new MobileEndpointCallback(){
  @Override
   public void onComplete(Exception exception, MobileResource
mobileResource){
      MobileObjectCollection collection = (MobileObjectCollection)
mobileResource
   }
})
```

5. The raw downloaded JSON object is exposed through the `JsonObject` property. Use this property to access the appropriate values.

```
Incident incident = (Incident) collection.getObjectsList().get(index);
JSONObject json = incident.getJsonObject();
json.put("status", "completed");
```

6. Save and delete objects the same way you save and delete `OMCMobileObject` objects.

```
//Save the object
incident.saveResource (new MobileEndpointCallback(){
});
...
// Delete the object
incident.deleteResource (new MobileEndpointCallback(){
});
```

Specifying Which Resources to Synchronize First

When a mobile app reconnects with the internet, the library synchronizes the local cache with the server. If you want the library to synchronize some resources before others, such as statuses before images, then **pin** the resources with the applicable priorities.

When you fetch the resource, you use the `MobileResource` class' `pinResource` method to set a resource's priority (`MobileFile`, `MobileObject`, and `MobileObjectCollection` inherit from this class).

```
builder.execute(new MobileEndpointCallback(){
  @Override
  public void onComplete(Exception exception, MobileResource
mobileResource) {
     mobileResource.pinResource(PinPriority.High);
```

```
  }
});
```

Setting a Resource's Synchronization Policies Programmatically

When you fetch a resource, the Synchronization library saves with the resource object the synchronization policies that are specified in the configuration file. These saved policies are associated with that resource object for its lifetime. You can change these saved policies when you fetch the data and before you add, update, or delete a resource.

Setting a Fetch Builder's Synchronization Policy

You can use the fetch builder's synchronization policy to override an endpoint's configured policies. When the library fetches the resource from the server, it saves the fetch builder's policy settings with the resource.

1. Create the fetch builder.

   ```
   FetchCollectionBuilder fetchCollectionBuilder = endpoint.fetchObjects();
   ```

2. Create a `SyncPolicy` object and set the policies to override. This example overrides all the policies:

   ```
   SyncPolicy policy = new SyncPolicy();
   policy.setFetchPolicy(SyncPolicy.FETCH_POLICY_FETCH_FROM_SERVICE_IF_ONLI
   NE);
   policy.setExpirationPolicy(SyncPolicy.EXPIRATION_POLICY_EXPIRE_ON_RESTAR
   T);
   policy.setEvictionPolicy(SyncPolicy.EVICTION_POLICY_EVICT_ON_EXPIRY_AT_S
   TARTUP);
   policy.setUpdatePolicy(SyncPolicy.UPDATE_POLICY_QUEUE_IF_OFFLINE);
   policy.setConflictResolutionPolicy(SyncPolicy.CONFLICT_RESOLUTION_POLICY
   _CLIENT_WINS);
   policy.setNoCache(false);
   ```

3. Set the builder's synchronization policy.

   ```
   fetchCollectionBuilder = fetchCollectionBuilder.withPolicy(policy);
   ```

Changing a Resource Object's Synchronization Policy

Sometimes, you'll need to change the synchronization policy for a mobile resource object (such as a mobile object, mobile collection, or mobile file) before you send an add, update, or delete to the server. This example sets the mobile resource object's conflict resolution policy to `CONFLICT_RESOLUTION_POLICY_CLIENT_WINS`.

1. Get the synchronization policy for the mobile resource object.

   ```
   SyncPolicy policy = mIncidentMobileObject.getCurrentSyncPolicy();
   ```

2. Set the conflict resolution policy to `CONFLICT_RESOLUTION_POLICY_CLIENT_WINS`. All other policies remain as is.

```
policy.setConflictResolutionPolicy(SyncPolicy.CONFLICT_RESOLUTION_POLICY
_CLIENT_WINS);
```

3. Set the mobile resource object's synchronization policy. This change doesn't take affect until you call `saveResource` (to perform an add or update). For a delete, you must call `reloadResource` for the policy change to take affect before you call `deleteResource`.

```
mIncidentMobileObject.setSyncPolicy(policy);
```

Detecting and Handling Conflicts

In Conflict Resolution Policies, you learn how to set the conflict resolution policy for the custom API resources that your mobile app accesses. When the conflict resolution policy that is in affect for a resource is `PRESERVE_CONFLICT`, the Synchronization library doesn't overwrite the server's version with the local version if there's a conflict. Instead, an edited version is kept in the offline edits in the local cache, and the mobile app is responsible for handling the conflict, such as programmatically merging the two versions.

A conflict occurs when the object on the server was updated after you retrieved it, and thus is no longer the version that you tried to update. For example, Mary uses her app to change an incident status at 4:00 p.m. However, her device is offline, so the change is stored in the offline edits in the local cache. At 4:30, Tom updates the same incident. At 5:00, Mary's device reconnects with the internet, and the Synchronization library automatically sends Mary's offline edit to the server. The server responds with a `412 Precondition Failed` status to indicate the conflict.

When a conflict happens, the library marks the modified object as having conflicts, and it makes available both the modified object (from the offline edits in the local cache), and the current server version to enable you to handle the conflict in your code.

If the device is online when the library sends an update or delete to the server, then the mobile app can handle the conflict as soon as it receives the response. However, when the user makes edits when the device is offline, there's no way to know if there are conflicts. You can't check for conflicts until the device reconnects and the library synchronizes the offline edits with the server. You have two options for detecting and handling conflicts that occur when a device reconnects:

- To detect and handle conflicts after the library finishes synchronizing offline edits with the server, use the `offlineResourceSynchronized` method, as shown in the first example. After the library finishes synchronizing all offline edits, it calls this method for each offline edit that it synchronized.
- To check whether a conflict occurs at the time that the library sends the offline edit to the server (when the device is online), use the `cacheResourceChanged` method to listen for online updates and deletes, as shown in the second example. The callback for this method is called for each resource that the library updates or deletes. Typically, you use this method to detect any resource change during a background cache refresh so that you can refresh the UI with the change. However, you also can use this method to detect and handle conflicts when the

library synchronizes the offline edits. Note that the callback is not called when the library adds a new resource to the local cache.

Don't initialize `CachedResourceChanged` more than once during the lifetime of the application.

Detecting Conflicts When the Library Completes Synchronization

Here's an example of using the `Synchronization` `offlineResourceSynchronized` method to detect conflicts after the Synchronization library has finished synchronizing the cache. In this example, the only mobile endpoint that the mobile app accesses is the `incidents` endpoint. This example shows how to handle both custom and generic `MobileObject` objects.

```
synchronization.offlineResourceSynchronized(new
SyncResourceUpdatedCallback() {
    @Override
    public void onResourceUpdated(String uri, MobileResource
mobileResource) {
        if (mobileResource == null) {
            Log.i("offlineResourceSync", "Resource for " + uri +
                    "deleted from cache after offline synchronization");
            return;
        }

        String result = null;
        if (mobileResource.hasConflict()) {
            result = "with conflicts";
        } else if (mobileResource.hasOfflineUpdates()) {
            result = "with offline update";
        } else if (mobileResource.hasOfflineCommitError()) {
            result = "with error";
        } else {
            result = "successfully";
        }

        // If you created a custom MobileObject class, you can access
properties directly
        if (mobileResource instanceof IncidentCustomMobileObject) {

            IncidentCustomMobileObject anIncident =
(IncidentCustomMobileObject) mobileResource;

            Log.i("offlineResourceSync", "Offline edits for " +
anIncident.getTitle()
                    + " finished with result :" + result);

            // Incident has been synchronized with the service object.
            // You can show a pop up or reload the resources in the UI,
            // such as in the main thread.

        } else {

            // Process has finished.
            // MobileObject/MobileFile has been synchronized with the
service object.
```

```
            // You can show a pop up or reload the resources in the UI,
            // such as in the main thread.

        }
    }
});
```

Detecting Conflicts When the Library Updates the Cache

Here's an example of using the `Synchronization` `cachedResourceChanged` method to detect conflicts whenever a cached resource is updated either from new data from the service or an update or delete from the mobile app. In this example, the only mobile endpoint that the mobile app accesses is the `incidents` endpoint. This example shows how to handle both custom and generic `MobileObject` objects.

```
synchronization.cachedResourceChanged(new SyncResourceUpdatedCallback() {
    @Override
    public void onResourceUpdated(String uri, MobileResource
mobileResource) {
        if (mobileResource == null) {
            Log.i("cachedResourceChanged", "Resource for " + uri +
"deleted from cache");
            return;
        }

        String result = null;
        if (mobileResource.hasConflict()) {
            result = "with conflicts";
        } else if (mobileResource.hasOfflineUpdates()) {
            result = "with offline update";
        } else if (mobileResource.hasOfflineCommitError()) {
            result = "with error";
        } else {
            result = "successfully";
        }

        // If you created a custom MobileObject class, you can access
properties directly
        if (mobileResource instanceof IncidentCustomMobileObject) {

            IncidentCustomMobileObject anIncident =
(IncidentCustomMobileObject) mobileResource;

            Log.i("cachedResourceChanged", "Cache changes for " +
anIncident.getTitle()
                    + " finished with result :" + result);

            // Custom object changed in local cache. You can show a pop up
            // or reload the resources in the UI, such as in the main
thread.
        } else {

            Log.i("cachedResourceChanged", "Cache changes finished with
result :" + result);
```

```
            // OMCMobileObject, OMCMobileFile, or OMCMobileObjectCollection
            // object changed in local cache.
            // You can show a pop up or reload the resources in the UI,
            // such as in the main thread.

        }
    }
});
```

Reviewing and Discarding Offline Edits

You might want to enable a mobile user to work offline while they make their changes, and then switch back to working online when the user has completed making changes, is satisfied with the end result, and is ready for the Synchronization library to synchronize with the server. The code examples in this section show how to:

- Switch the app to work-offline mode and switch back to work-online mode.
- List the resources that have been changed while offline.
- Discard all offline edits.
- Discard a resource's offline edits.

The `Synchronization` class provides the methods for reviewing and discarding offline edits. As shown in the following steps, you use its `getNetworkStatus` and `setOfflineMode` methods, along with the `SyncNetworkStatus` enumeration to switch the work-offline mode on and off. You use its `loadOfflineResources` method to get all the offline edits that haven't been synchronized with the server, and its `discardOfflineUpdates` method to discard all offline edits.

1. At application start-up, instantiate `Synchronization` and open the mobile endpoint.

```
try {
    synchronization =

MobileBackendManager.getManager().getDefaultMobileBackend(this).getServi
ceProxy(Synchronization.class);
    } catch(ServiceProxyException e) {
        e.printStackTrace();
    }
incidentsEndpoint = synchronization.openMobileEndpoint(
    "incidentreport",
    "incidents",
    MobileObject.class);
```

2. Add a `Switch` component to the layout.

```
<Switch
    android:id="@+id/workOfflineSwitch"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignParentBottom="true"
    ...
    android:onClick="changeWorkOfflineMode"
    android:text="Work Offline" />
```

3. Add the `changeWorkOfflineMode` function, which is called when `workOfflineSwitch` is clicked. This method uses the `Synchronization` `getNetworkStatus` method to determine the current network status, and the `setOfflineMode` method to switch the work-offline mode on and off. When it calls `setOfflineMode`, the library synchronizes all offline edits with the server automatically. Note that calling `setOfflineMode(true)` when the device isn't connected to the internet has no effect.

```
public void changeWorkOfflineMode(View view) {
    SyncNetworkStatus syncNetworkStatus =
synchronization.getNetworkStatus();
    try {
        if (syncNetworkStatus == SyncNetworkStatus.SyncOffline) {
            // Because setOfflineMode() is a no-op when the device
            // is offline, don't allow user to switch modes when
offline.
            Toast.makeText(MainActivity.this,
                    "No internet connection. " +
                            "You can't switch the Work Offline mode on
or off when " +
                            "there isn't an internet connection.",
                    Toast.LENGTH_SHORT).show();
        } else {
            // Device is not in "real" offline mode.
            // Switch from work online to work offline, or switch from
work offline to work online
            // setOfflineMode(true) sets SyncNetworkStatus to
SyncOfflineTest
            // setOfflineMode(false) sets SyncNetworkStatus to
SyncOnline
            // (if the device is actually online)
            synchronization.setOfflineMode(syncNetworkStatus ==
SyncNetworkStatus.SyncOnline);
        }
    } catch (Exception e) {
        // Handle error
    }
}
```

4. Add code to the `onCreate` method to set the switch according to the current mode.

```
Switch workOfflineSwitch = (Switch)
findViewById(R.id.workOfflineSwitch);

workOfflineSwitch.setChecked(
        synchronization.getNetworkStatus() ==
SyncNetworkStatus.SyncOfflineTest);
```

5. Add code to display a list of the offline edits. You use the `Synchronization` `loadOfflineResources` method to get the list. In this example, the mobile app

accesses only the incidents endpoint, and all the items in the offline edits list are of type `MobileObject`.

```
//Display a list of offline edits
synchronization.loadOfflineResources(new SyncLocalLoadingCallback() {
    @Override
    public void onSuccess(List<MobileResource> resources) {
        // This list contains all the MobileResource objects in the
local edit cache
        // In this app, the only mobile endpoint is for incidents
        // So, only MobileObjects are in the edit cache
        for (MobileResource resource : resources) {
            // Put your code to add the incident to the display list
here
        }
    }

    @Override
    public void onError(String errorMessage) {
        //Handle the error
    }
});
```

6. Add a button to discard all offline edits. Use code like the following to discard the edits.

```
final Button mDiscardEdits = (Button)
findViewById(R.id.buttonDiscardOfflineEdits);

mDiscardEdits.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        //Discard all offline edits:
        //Deletes all resources in the edit cache,
        //but keeps all resources in the local cache as is
        synchronization.discardOfflineUpdates(new
SyncDiscardOfflineResourceCallback() {
            @Override
            public void onError(String errorMessage) {
                //Handle the error
            }
        });
    }

});
```

7. The previous step shows how to discard all offline updates. You also can discard offline updates for a specific resource. You call the resource's `reloadResource` method with the `discardOfflineUpdates` parameter set to `true` and the `reloadFromService` parameter set to `false`.

In the following code example, `arraySelectedResourcesToDiscardOfflineEdits` is a list of resources that were edited while offline and were selected for discarding the edits.

```
try {
    for (int index = 0; index <
arraySelectedResourcesToDiscardOfflineEdits.length; index++) {

        MobileResource mobileResource =
arraySelectedResourcesToDiscardOfflineEdits[index];
        mobileResource.reloadResource(true, false, new
MobileEndpointCallback() {
            @Override
            public void onComplete(Exception exception, MobileResource
mobileResource) {
                if (exception != null) {
                    // handle exception here
                } else {
                    // handle success here
                }
            }
        });

    }
} catch (Exception ex) {
    // handle exception here
}
```

iOS Synchronization Library

This section shows how to use the Synchronization library to implement several of the common data offline tasks for working with a custom API's resources.

For detailed information about the library, see Oracle Mobile Cloud Service iOS SDK Reference.

> **Tip:**
>
> The client SDK ZIP file contains an `examples` folder, which contains the source code for the SalesPlus app. This app illustrates many of the synchronization features that are described in this section.

Setting Up Your Mobile App for the iOS Synchronization Library

1. Ensure that the correct policies are in place for the mobile backend and API endpoints as described in Synchronization Policy Levels and Precedence and Defining Synchronization Policies Using a Configuration File.
2. As with all mobile apps, instantiate `OMCMobileBackendManager`, and then instantiate `OMCMobileBackend` to manage connectivity, authentication, and other transactions between your application and its associated mobile backend, including calls to platform and custom APIs.

3. To access the custom APIs from the Synchronization library, get the mobile backend's synchronization service.

```
OMCSynchronization* synchronization = [mbe synchronization];
[synchronization initialize];
```

Fetching Resources

After you set up your app to work with data offline, you use the mobile endpoint class to open endpoints to custom code API resources, and you use fetch builders to synchronize data retrieval and modifications with the local cache automatically. A fetch builder enables you to specify how to fetch the data, and then enables you to execute the fetch.

1. To access an endpoint, instantiate `OMCMobileEndpoint` for that endpoint. This example instantiates an endpoint for `/mobile/custom/incidentreport/incidents`.

```
// open Endpoint
OMCMobileEndpoint* endpoint = [
  synchronization openEndpoint:OMCMobileObject.class
  apiName:@"incidentreport"
  endpointPath:@"incidents"
];
```

2. (Optional) Add objects or files to the collection. This example adds an object.

```
OMCMobileObject* newObject = [mobileEndpoint createObject];
    // Set properties
    [newObject addOrUpdateJsonProperty:@"title"
propertyValue:@"incident 213"];
    ....
    [newObject saveResourceOnSuccess:^(id mobileObject) {

       } OnError:^(NSError *error) {

       }];
```

3. Use a fetch builder to specify how to fetch the objects from the endpoint. The fetch builder method that you use depends on whether you want to retrieve an object, a collection, or a file:

   - `OMCFetchObjectBuilder`
   - `OMCFetchObjectCollectionBuilder`
   - `OMCFetchFileBuilder`

   Here's an example of creating a fetch builder for a collection.

```
OMCFetchObjectCollectionBuilder* builder = [endpoint
fetchObjectCollectionBuilder];
```

In this example, we want to get all the incidents for the signed-in technician (which is the same as the user name). The API provides a query parameter for technician, so we can tell the builder to add that query parameter to the request:

```
[builder withParamName:@"technician" paramValue:username];
```

You can call `withParamName` as many times as you need to specify all the query parameters.

4. Add necessary headers.

   In this example, to enable easy searching for all diagnostic log entries associated with this fetch builder, the request includes the `Oracle-Mobile-Diagnostic-Session-ID` header. The `diagLogFilterTag` string variable has been set to a value that uniquely identifies requests that are made using this fetch builder.

   ```
   [builder setRequestHeaders:[NSDictionary dictionaryWithObjectsAndKeys:
   diagLogFilterTag, @"Oracle-Mobile-Diagnostic-Session-ID", nil]];
   ```

5. Use the builder to execute the fetch.

   ```
   [builder executeFetchOnSuccess:^(OMCMobileObjectCollection
   *mobileObjectCollection) {
     // This function is called when the request finishes successfully.
     // Get all the objects from the collection.
     NSArray* collection = [mobileObjectCollection getMobileObjects];
   } OnError:^(NSError *error) {
     // This function is called when the request finishes with an error
   }];
   ```

   If the fetch policy is to fetch the data from the local cache, such as `FETCH_FROM_SERVICE_ON_CACHE_MISS`, then it's fetched from the local cache if available. In all other cases, the collection is fetched from the server if the policy allows. If the `noCache` setting is false, then the results are saved to a local cache.

6. The raw downloaded JSON object is exposed through the `jsonObject` property. You can use this property to set the appropriate values, or use `addOrUpdateJsonProperty`.

   ```
   OMCMobileObject* incident = [collection objectAtIndex:index];
   // You can access raw JSON
   NSDictionary* json = [incident jsonObject];
   // Or use the addOrUpdateJsonProperty method
   [incident addOrUpdateJsonProperty:@"status" propertyValue:@"completed"];
   ```

7. Use one of the `OMCMobileObject` save methods to save the changes on the server.

   ```
   [incident saveResourceOnSuccess:^(id object){
     // Block that is called after the request finishes successfully
     ...
   }OnError:^(NSError *error){
     // Block that is called after the request finishes with an error
   ```

```
  ...
}];
```

If the device isn't connected to the internet, and the update policy is `UPDATE_IF_OFFLINE`, then the library saves the changes to the local cache. The changes are sent to the server automatically when the device reconnects with the internet.

8. Use one of the `OMCMobileObject` delete methods to delete an object.

```
[incident deleteResourceOnError:^(NSError *error) {

}];
```

If the device isn't connected to the internet, and the update policy is `UPDATE_IF_OFFLINE`, then the library saves the changes to the local cache. The changes are sent to the server automatically when the device reconnects with the internet.

Fetching Filtered Resources

You might have an app that filters which items it displays. For example, an FiF app might want to display all incidents with a status of `new`. When the device is online, your code can fetch the items as `mobileResource` objects, convert the objects to JSON objects, and then filter the items. However, when the device is offline, your app can't filter the `mobileResource` objects in the local cache because the objects are just blobs of data. The solution is to use a custom `MobileObject`. When you do this, the local cache stores the data in a table with a column for each of the custom object's fields, which enables your mobile app to query data in the local cache based on field values. We'll use the incident list in the FiF example to illustrate how to do this. In this example, the users must be able to filter the incident list by status.

When you open a mobile endpoint on a custom `MobileObject` class, you can use the fetch builder's `queryForProperty` method to specify the filter to use in the local cache. Note that this method is for filtering JSON objects from the local cache. It doesn't affect the way that the Synchronization library retrieves results from the server. Whenever you execute the fetch builder, the library first looks at the fetch policy setting to determine whether to refresh the local cache. If the policy specifies that it must refresh the local cache from the server, then it retrieves all the objects, regardless of the filter that you specify using the `queryForProperty` method. Regardless of the fetch policy and whether it refreshed the local cache, the library then uses the `queryForProperty` method to filter the data in the local cache, and return the filtered results. That is, regardless of whether the device is online or offline, and regardless of whether the library fetches data from the server or uses the local cache, the `queryForProperty` method filters the results based on the query property and value.

1. Create a custom mobile object class that extends `OMCMobileObject`, define all the properties that you need for your custom mobile object, and synthesize those properties. Here's an example of the `incident.h` header file for an `Incident` class.

```
#import <Foundation/Foundation.h>
#import "OMCMobileObject.h"

@interface Incident : OMCMobileObject {
```

```
}

// Properties
@property (nonatomic, retain) NSNumber* id
@property (nonatomic, retain) NSString* title;
@property (nonatomic, retain) NSString* customer;
@property (nonatomic, retain) NSString* status;
@property (nonatomic, retain) NSString* priority;
@end
```

2. When you initialize the mobile backend's synchronization service, use the `initializeWithMobileObjectEntities` method to create database entities for the `Incident` custom class.

```
NSArray* entities = [NSArray arrayWithObjects:[Incident class], nil];
[synchronization initializeWithMobileObjectEntities:entities];
```

   You can include more than one custom object in the initialization.

3. Open the endpoint for the custom class.

```
OMMobileEndpoint* endpoint = [
  synchronization openEndpoint:Incident.class
  apiName:@"incidentreport"
  endpointPath:@"incidents"
];
```

4. When you create the fetch builder, use the `queryForProperty` method to add a query to filter the results by status.

```
OMCFetchObjectCollectionBuilder* builder = [endpoint
fetchObjectCollectionBuilder];

[builder queryForProperty:@"status"
  comparision:Equals
  compareWith:@"pending"];
```

5. Fetch the data.

```
[builder executeFetchOnSuccess:^(OMCMobileObjectCollection
*mobileObjectCollection) {
  // This function is called when the request finishes successfully.
  // Get all the objects from the collection.
  NSArray* collection = [mobileObjectCollection getMobileObjects];
} OnError:^(NSError *error) {
  // This function is called when the request finishes with an error
}];
```

6. The raw downloaded JSON object is exposed through the `jsonObject` property. You can use this property to set the appropriate values, or you can access the properties directly.

```
Incident* incident = [collection objectAtIndex:index];
// You can access raw JSON
NSDictionary* json = [incident jsonObject];
// Or you can access the property directly
incident.status = @"completed";
```

7. Save and delete objects the same way you save and delete `OMCMobileObject` objects.

```
//Save the object
[incident saveResourceOnSuccess:^(id object){

}OnError:^(NSError *error) {

}];
...
// Delete the object
[incident deleteResourceOnError:^(NSError *error) {

}];
```

Specifying Which Resources To Synchronize First

When a mobile app reconnects with the internet, the library synchronizes the local cache with the server. If you want the library to synchronize some resources before others, such as statuses before images, then **pin** the resources with the applicable priorities.

When you fetch the resource, you use the `OMCMobileResource` class' `pinResource` method to set a resource's priority (`OMCMobileFile`, `OMCMobileObject`, and `OMCMobileObjectCollection` inherit from this class).

```
[builder executeFetchOnSuccess:^(OMCMobileObjectCollection
*mobileObjectCollection) {
   [mobileObjectCollection pinResource:High];
  // Get all the objects from the collection
  NSArray* objects = [mobileObjectCollection getMobileObjects];
} OnError:^(NSError *error) {
  // This function is called when the request finishes with an error
}];
```

Setting a Resource's Synchronization Policies Programmatically

When you fetch a resource, the Synchronization library saves with the resource object the synchronization policies that are specified in the configuration file. These saved policies are associated with that resource object for its lifetime. You can change these

saved policies when you fetch the data and before you add, update, or delete a resource.

Changing a Fetch Builder's Synchronization Policy

You can use the fetch builder's synchronization policy to override an endpoint's configured policies. When the library fetches the resource from the server, it saves the fetch builder's policy settings with the resource.

1. Create the fetch builder.

```
OMCFetchObjectCollectionBuilder* builder = [endpoint
fetchObjectCollectionBuilder];
```

2. Create an `OMCSyncPolicy` object, and then set the policies that you want to override. This example overrides all the policies:

```
OMCSyncPolicy* policy = [[OMCSyncPolicy alloc] init];
policy.fetch_Policy = FETCH_POLICY_FETCH_FROM_SERVICE_IF_ONLINE;
policy.expiration_Policy = EXPIRATION_POLICY_EXPIRE_ON_RESTART;
policy.eviction_Policy = EVICTION_POLICY_EVICT_ON_EXPIRY_AT_STARTUP;
policy.update_Policy = UPDATE_POLICY_QUEUE_IF_OFFLINE;
policy.conflictResolution_policy =
CONFLICT_RESOLUTION_POLICY_CLIENT_WINS;
policy.no_cache = false;
```

3. Set the builder's synchronization policy.

```
[builder setSyncPolicy:policy];
```

Changing a Resource Object's Synchronization Policy

Sometimes, you'll need to change the synchronization policy for a mobile resource object (such as a mobile object, mobile collection, or mobile file) before you send an add, update, or delete to the server. This example sets the mobile resource object's conflict resolution policy to `CONFLICT_RESOLUTION_POLICY_CLIENT_WINS`.

1. Get the synchronization policy for the mobile resource object. In this example, `anIncident` is an `OMCMobileObject`.

```
OMCSyncPolicy* policy = [anIncident getCurrentSyncPolicy];
```

2. Set the conflict resolution policy to `CONFLICT_RESOLUTION_POLICY_CLIENT_WINS`. All other policies remain as is.

```
policy.conflictResolution_policy =
CONFLICT_RESOLUTION_POLICY_CLIENT_WINS;
```

3. Set the mobile resource object's synchronization policy. This change doesn't take affect until you call `saveResource` (to perform an add or update). For a delete, you must call `reloadResource` for the policy change to take affect before you call `deleteResource`.

```
[anIncident setSyncPolicy:policy];
```

Detecting and Handling Conflicts

In Conflict Resolution Policies, you learn how to set the conflict resolution policy for the custom API resources that your mobile app accesses. When the conflict resolution policy that is in affect for a resource is `PRESERVE_CONFLICT`, the Synchronization library doesn't overwrite the server's version with the local version if there's a conflict. Instead, an edited version is kept in the offline edits in the local cache, and the mobile app is responsible for handling the conflict, such as programmatically merging the two versions.

A conflict occurs when the object on the server was updated after you retrieved it, and thus is no longer the version that you tried to update. For example, Mary uses her app to change an incident status at 4:00 p.m. However, her device is offline, so the change is stored in the offline edits in the local cache. At 4:30, Tom updates the same incident. At 5:00, Mary's device reconnects with the internet, and the library automatically sends Mary's offline edit to the server. The server responds with a `412 Precondition Failed` status to indicate the conflict.

When a conflict happens, the library marks the modified object as having conflicts, and the library makes available both the modified object (from the offline edits in the local cache), and the current server version to enable you to handle the conflict in your code.

If the device is online when the library sends an update or delete to the server, then the mobile app can handle the conflict as soon as it receives the response. However, when the user makes edits when the device is offline, there's no way to know if there are conflicts. You can't check for conflicts until the device reconnects and the library synchronizes the offline edits with the server. You have two options for detecting and handling conflicts that occur when a device reconnects:

- To detect and handle conflicts after the library finishes synchronizing offline edits with the server, use the `offlineResourceSynchronized` method, as shown in the first example. After the library finishes synchronizing all offline edits, it calls this method for each offline edit that it synchronized.
- To check whether a conflict occurs at the time that the library sends the offline edit to the server (when the device is online), use the `cacheResourceChanged` method to listen for online updates and deletes, as shown in the second example. The callback for this method is called for each resource that the library updates or deletes. Typically, you use this method to detect any resource change during a background cache refresh so that you can refresh the UI with the change. However, you also can use this method to detect and handle conflicts when the library synchronizes the offline edits. Note that the callback is not called when the library adds a new resource to the local cache.

  Don't initialize `CachedResourceChanged` more than once during the lifetime of the application.

Detecting Conflicts When the Library Completes Synchronization

Here's an example of using the `OMCSynchronization offlineResourceSynchronized` method to detect conflicts after the library has finished synchronizing the cache. In this example, the only mobile endpoint that the mobile app accesses is the `incidents`

endpoint. This example shows how to handle both custom and generic `MobileObject` objects.

```
    [sync offlineResourceSynchronized:^(NSString *uri, id mobileResource) {

        if ( !mobileResource ) {
            NSLog(@"Resource for %@ deleted from cache after offline
synchronization ", uri);
            return;
        }

        NSString* result = nil;
        if ( ((OMCMobileResource*) mobileResource).hasConflicts ) {
            result = @"with conflicts";
        }
        else if ( ((OMCMobileResource*)
mobileResource).hasOfflineCommitError ) {
            result = @"with error";
        }
        else {
            result = @"successfully";
        }

        // If you created a custom MobileObject class, you can access
properties directly
        if([mobileResource isKindOfClass:[Incident class]]) {

            Incident* anIncident = mobileResource;

            NSLog(@"Offline edits for %@ finished %@.", anIncident.title,
result);

            // Incident has been synchronized with the service object.
            // You can show a pop up or reload the resources in the UI,
            // such as in the main thread.

            // When mobileResource is a custom MobileObject class,
            // and hasConflicts is true,
            // then both the MobileObject class and its jsonObject
property
            // contain the local edited copy and the
            // jsonObjectPersistentState property contains the server copy
        }
        else {

            OMCMobileResource* aMobileResource = mobileResource;
            NSLog(@"Offline edits for resource %@ finished %@",
                aMobileResource.uri, result)

            // OMCMobileObject or OMCMobileFile has been synchronized
            // with the service object.

            // You can show a pop up or reload the resources in the UI,
            // such as in the main thread.
```

```
            // When mobileResource is an OMCMobileObject,
            // and hasConflicts is true,
            // then its jsonObject property contains the local edited copy
and
            // its jsonObjectPersistentState property contains the server
copy
        }
    }];
```

Detecting Conflicts When the Library Updates the Cache

Here's an example of using the `OMCSynchronization` `cachedResourceChanged` method to detect conflicts whenever a cached resource is updated either from new data from the service or an update or delete from the mobile app. In this example, the only mobile endpoint that the mobile app accesses is the `incidents` endpoint. This example shows how to handle both custom and generic `MobileObject` objects.

```
    [sync cachedResourceChanged:^(NSString *uri, id mobileResource) {

        if ( !mobileResource ) {
            NSLog(@"Resource for %@ deleted from cache ", uri);
            return;
        }

        NSString* result = nil;
        if ( ((OMCMobileResource*) mobileResource).hasConflicts ) {
            result = @"with conflicts";
        }
        else if ( ((OMCMobileResource*)
mobileResource).hasOfflineUpdates ) {
            result = @"with offline update";
        }
        else if ( ((OMCMobileResource*)
mobileResource).hasOfflineCommitError ) {
            result = @"with error";
        }
        else {
            result = @"successfully";
        }

        // If you created a custom MobileObject class, you can access
properties directly
        if([mobileResource isKindOfClass:[Incident class]]) {

            Incident* anIncident = mobileResource;

            NSLog(@"Cache changes for %@ finished %@.", anIncident.title,
result);

            // Custom object changed in local cache. You can show a pop up
            // or reload the resources in the UI, such as in the main
thread.
        }
        else {
```

```
            OMCMobileResource* aMobileResource = mobileResource;
            NSLog(@"Cache changes for %@ finished %@.",
                aMobileResource.uri, result);
            // OMCMobileObject, OMCMobileFile, or
OMCMobileObjectCollection
            // object changed in local cache.
            // You can show a pop up or reload the resources in the UI,
            // such as in the main thread.

        }
    }];
```

Reviewing and Discarding Offline Edits

You might want to enable a mobile user to work offline while they make their changes, and then switch back to working online when the user has completed making changes, is satisfied with the end result, and is ready for the Synchronization library to synchronize with the server. The code examples in this section show how to:

- Switch the app to work-offline mode and switch back to work-online mode.
- List the resources that have been changed while offline.
- Discard all offline edits.
- Discard a resource's offline edits.

The `OMCSynchronization` class provides the methods for working offline, and for reviewing and discarding offline edits. As shown in the following steps, you use its `GetNetworkStatus` and `setOfflineMode` methods, along with the `SyncNetworkStatus` constants to switch the work-offline mode on and off. You use its `loadOfflineResourcesOnSuccess` method to get all the offline edits that haven't been synchronized with the server, and its `discardOfflineUpdatesOnError` method to discard all offline edits. You also can discard a specific resource's offline updates by calling the resource's `reloadResource` method.

1. Add a button to switch between work-online mode and work-offline mode. Use code like the following to switch modes when the user clicks the button. You use the `OMCSynchronization GetNetworkStatus` method to determine the current network status, and the `setOfflineMode` method to switch the work-offline mode on and off. When you call `setOfflineMode(false)`, the library synchronizes all offline edits with the server automatically. Note that calling `setOfflineMode` when the device isn't connected to the internet has no effect.

   ```
   - (IBAction) switchOfflineMode:(id)sender {

       // Get current status
       SyncNetworkStatus networkStatus = [synchronization
   getNetworkStatus];

       if ( networkStatus == SyncOffline) {

           UIAlertController *myAlertController = [UIAlertController
   alertControllerWithTitle:@"Sorry!"
               message:@"You can't switch to Work Offline mode when there
   isn't an internet connection."
   ```

```
            preferredStyle:UIAlertControllerStyleAlert ];
        UIAlertAction* okBtn = [UIAlertAction
                                    actionWithTitle:@"OK"
                                    style:UIAlertActionStyleDefault
                                    handler:^(UIAlertAction * action)
                                    {
                                        [myAlertController
dismissViewControllerAnimated:YES

        completion:nil];
                                    }];
        [myAlertController addAction: okBtn];
        [self presentViewController:myAlertController
                           animated:YES
                         completion:nil];
    }
    else {

        [omcSynchronization setOfflineMode:(networkStatus ==
SyncOnline)];

        // Get updated status
        networkStatus = [omcSynchronization getNetworkStatus];

        if ( networkStatus == SyncOfflineTest ) {

            lblNetworkStatus.text = @"Working offline.";

        }
        else {

            lblNetworkStatus.text = @"";
        }
    }
}
```

2. Add code to display a list of the offline edits. You use the `OMCSynchronization LoadOfflineResourcesAsync()` method to get the list. In this example, the mobile app accesses only the incidents endpoint and all items in the offline edits list are of type `MobileObject`.

```
[omcSynchronization loadOfflineResourcesOnSuccess:^(NSArray
*mobileResources) {

    for ( OMCMobileResource* aResource in mobileResources ) {
            // Put your code to add the incident to the display list
here
    }

} onError:^(NSError *error) {

     // Handle error here.

}];
```

3. Add a button to discard all offline edits. Use code like the following to discard the edits.

```
// Discard all offline edits only.
// Resources remain in the cache with their persistent state (that is,
the server version).
[omcSynchronization discardOfflineUpdatesOnError:^(NSError *error) {
    // Handle error here
}
```

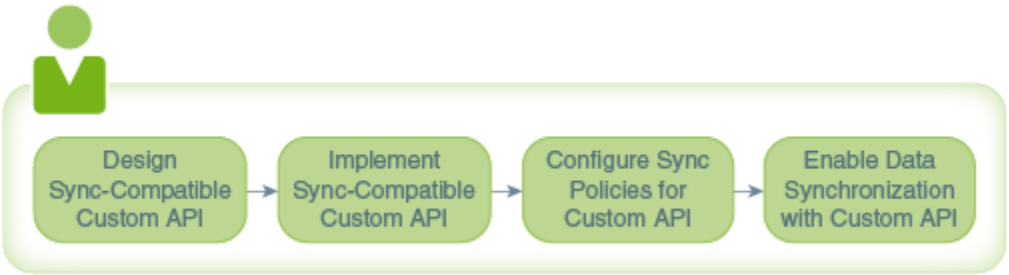
4. The previous step shows how to discard all offline updates. You also can discard offline updates for a specific resource. You call the resource's `reloadResource` method with the `discardOfflineUpdates` parameter set to `YES` and the `reloadFromService` parameter set to `NO`.

   In the following code example, `arraySelectedResourcesToDiscardOfflineEdits` is a list of resources that were edited while offline and were selected for discarding the edits.

```
for ( int index = 0; index <
arraySelectedResourcesToDiscardOfflineEdits.count; index++ ) {

    OMCMobileResource* aResource =
[arraySelectedResourcesToDiscardOfflineEdits objectAtIndex:index];

    [aResource reloadResource:YES
            reloadFromService:NO
                    onSuccess:^(id mobileResource) {

                        // Offline edits succesfully discarded from a
resource.
                    }];
}
```

Making Custom APIs Synchronizable

If your mobile app uses the Synchronization library to access a custom API offline, then that API should follow the sync-compatibility guidelines and should return data in a sync-compatible format. You also need to consider whether to configure synchronization policies for some or all of its resources.



Designing a Synchronization-Compatible API

As described in API Design Considerations, the custom API should follow these guidelines to be synchronization compatible:

- The resource name should alternate between plural nouns and singular resource identifiers (rid). For example: `/items/{rid}/subitems/{rid}/`.
- For pagination, use the `limit` and `offset` query parameters so that the Synchronization library uses paged downloads correctly. If you don't need to support pagination, then you don't need to specify these parameters.
- Use the `orderBy` query parameter to specify sorting. For example: `orderBy=propA,propB:desc,propC:asc`.
- The API must contain all the necessary endpoints to support data synchronization. For example, if you have an endpoint that returns a collection, then you must also have an endpoint that returns a specific item in the collection. See Endpoint Requirements for Sync Compatibility.

Implementing a Sync-Compatible API

As detailed in Implementing Synchronization-Compatible APIs, the custom API implementation should follow these guidelines:

- For `GET` requests, use the custom code SDK's `setItem` and `addItem` methods in your API's custom code to return data in a format that enables the Synchronization library to more easily cache and synchronize the data in the client's local cache. Responses must include the `Oracle-Mobile-Sync-Resource-Type` header, and, for single items, the `ETag` header.
- For `PUT` and `DELETE` requests, your code must honor the `If-Match` header as follows:
  - If the header contains an ETag value, and that value doesn't match the ETag on the server, then the code must not update or delete the item and must return a `412` HTTP response status (precondition failed) to indicate that the ETag does not match the server-side object's ETag.
  - If the header contains a value of * (asterisk), then the server-side's object must be replaced by the request object (or deleted for a `DELETE` request).
- For `PUT` requests, responses must include the `Oracle-Mobile-Sync-Resource-Type` and `ETag` headers. If the item was added, then it must include the `Location` header. For example `Location: /mobile/custom/incidentreport/incidents/1`.
- For `POST` requests, responses must include the `Oracle-Mobile-Sync-Resource-Type`, `Location`, and `ETag` headers.
- When you need to control data caching from the server side, use the `Oracle-Mobile-Sync-Evict`, `Oracle-Mobile-Sync-Expires`, and `Oracle-Mobile-Sync-No-Store` headers to override client side configuration.

Configuring Synchronization Policies for a Custom API

As described in Defining Synchronization Policies Using a Configuration File, you use the configuration file to set the synchronization policies for each mobile backend that your mobile app accesses. In addition to setting the overall (default) synchronization policies for each mobile backend, consider the custom API's resources that you'll access, and determine which, if any, need special synchronization policy configuration.

Say, for example, that your default fetch policy is `FETCH_FROM_SERVICE_ON_CACHE_MISS`. The custom API might have a resource for which the mobile app always needs the most current data. In that case, you can use the configuration file to specify the `FETCH_FROM_SERVICE_IF_ONLINE` fetch policy for that specific resource. To learn about configuring synchronization policies on a resource basis, see the Resource-Level Configuration section in Synchronization Configuration File Structure. Note that you can define synchronization policies at the default level and the resource level, and that you can override these programmatically, To learn more, see Synchronization Policy Levels and Precedence.

Synchronization Policies

The Synchronization library uses several types of synchronization policies:

- Conflict Resolution Policies define how to handle offline edits if the server's version changed after the initial data was fetched from the server. For example, if another client updated a resource, you might want the app's updates to overwrite the other client's update.
- Eviction Policies designate when to delete expired resources in the local cache. For example, you might want the app to delete all expired resources when the app starts. Expiration and eviction policies work together to keep stale resources from cluttering the cache. You can also use them to prevent users seeing out-of-date data and, by inference, potentially harmful data. Note that these policies apply only to resources in the local cache, not to server-side resources.
- Expiration Policies define how and when the Synchronization library marks resources stored in the local cache as out-dated or stale. For example, you might want all the resources to expire when the app is restarted so that the app fetches the latest version of a resource from the server the first time the app uses it in that session. The expiration policy only marks data, allowing you the option to display stale data if the app is offline. To delete data, use the eviction policy.
- Fetch Policies define how the Synchronization library determines whether to retrieve resources from the local cache or from the server. For example, if the resource changes frequently, you might choose to always retrieve it from the server unless the client is offline.
- Update Policies define what to do if the app modifies resources when the device is offline. For example, you might want the app to put all changes that are made while the device is offline in a queue and then synchronize the changes with the server when the device goes online again.

In addition to configuring the synchronization policies, you also can configure the cache settings for a mobile backend. You can configure the maximum size of the cache and you can specify when and how to perform background cache refreshes. See Synchronization Configuration File Structure.

You can specify synchronization policies for custom API resources at several levels:

- In the app's configuration file, you can specify default synchronization policies for all custom API endpoints that the library accesses through a specific mobile backend.
- In the app's configuration file, you can specify synchronization policies for specific custom API endpoints.

- In the custom API implementation, you can specify a resource's synchronization policies in a response header.
- In the app, you can specify a resource's synchronization policies when you fetch the data.
- In the app, you can specify a resource's synchronization policies when you add, update, or delete the resource.

When the Synchronization library fetches a resource from the server, it sets the resource's synchronization policies according to your configuration, and then saves those policies with the resource. When you configure a policy at more than one level, the library uses precedence rules to determine which policy level to use. For example, a response-header policy setting takes precedence over a fetch builder's policy setting. If a policy isn't set at the response header or fetch builder level, then the library uses the policy's setting from the configuration file. First, the library looks for the policy setting for the path that matches the fetch builder's endpoint. When there isn't a policy for the endpoint, then it uses the configuration file's default policy. If a policy isn't specified at any level, then the Synchronization library's hard-coded default policy is used. The actual rules are somewhat more complex than summarized here. For complete details see Synchronization Policy Levels and Precedence.

When the library does an automatic refresh, it always uses the `FETCH_POLICY_FETCH_FROM_SERVICE` fetch policy. For all other policies, the refresh process honors the response header values, if present, and, when not present, it uses the policies that were saved with the resource.

When you fetch a resource and the library uses the resource from the cache instead of from the server, then the resource's policies are not necessarily the policies that you configured for the object's endpoint. For example, if the resource was fetched using a fetch collection builder, then the resource's policies are the collection endpoint's policies and not the object's endpoint policies. Thus, you can't be sure what the resource's policies are. A cached resource's policies depend on whether it was originally fetched from the server as part of a collection, as an object, or as part of a refresh.

Defining Synchronization Policies Using a Configuration File shows how to configure default policies for the mobile backend and for endpoints (paths). Defining Synchronization Policies and Cache Settings in a Response Header shows how a custom API can use headers to control whether the response is cached, when it should expire in the local cache, and when it should be evicted. The following platform-specific topics show how to get and change a fetch builder's policies and get and change a mobile resource's policies programmatically:

- Android: Setting a Resource's Synchronization Policies Programmatically
- iOS: Setting a Resource's Synchronization Policies Programmatically

Video: Introduction to the Data Offline & Sync Policies

If you want a high-level understanding of how to use synchronization policies to drive data offline and synchronization capabilities, take a look at this video:

Video

Synchronization Policy Options

Here are the Synchronization library's policy options for each policy type.

Conflict Resolution Policies

Conflict resolution policies define what to do if, when updating a resource, it's discovered that the server version was updated after it was last requested. Say, for example, that the client app retrieved a resource on startup. Soon after, someone else updated the resource on the server. If the resource is then updated on the client app, you might want the client updates to overwrite the updates made by someone else.

Policy	Description
`CLIENT_WINS`	Instructs the Synchronization library to overwrite the server's version with the local version regardless of whether there is a conflict.
`PRESERVE_CONFLICT`	Instructs the Synchronization library to not overwrite the server's version with the local version if there's a conflict. The edited version is kept in the offline edits in the local cache, and the mobile app is responsible for handling the conflict, such as programmatically merging the two versions.
`SERVER_WINS`	Instructs the Synchronization library to not overwrite the server's version with the local version if there's a conflict. The edited version is removed from the offline edits in the local cache.

Eviction Policies

Eviction policies designate when expired resources in the local cache will be deleted. For example, you could set the eviction policy to `EVICT_ON_EXPIRY_AT_STARTUP` so expired items are deleted when the app starts. Keep in mind that if a user didn't use the app for several days and it's offline when it starts, the local cache could get cleared.

These policies apply to resources in the local cache only, not to server-side resources.

Policy	Description
`EVICT_ON_EXPIRY_AT_STARTUP`	Instructs the Synchronization library to delete expired resources from the local cache when the client application restarts, and update the local cache with the server copy the next time it's called by the client application. This can result in an empty cache, but this is appropriate if the latest resource is required.
`MANUAL_EVICTION`	Instructs the Synchronization library that resources can't be deleted from the local cache automatically. To evict resources manually, use an API.

Expiration Policies

Expiration policies define how and when the Synchronization library marks resources stored in the local cache as out-dated or stale. For example, if your resources change

frequently, then you can set the policy to `EXPIRE_ON_RESTART` to ensure that the local cache gets cleared periodically, and thus does not become too large.

Policy	Description
`EXPIRE_ON_RESTART`	Instructs the Synchronization library to mark a resource as expired when the client application restarts. The Synchronization library updates the local cache with the latest version from the server the next time it's called by the client application.
`EXPIRE_AFTER`	Instructs the Synchronization library to mark resources as expired after the specified time (in seconds) set for the `expireAfter` parameter. When you use the `EXPIRE_AFTER` policy, you must set a value for the `expireAfter` property.
`NEVER_EXPIRE`	Instructs the Synchronization library that resources in the local cache can't be marked as expired.

Fetch Policies

Fetch policies define how the Synchronization library determines whether to retrieve resources from the local cache or from the server. For example:

- If your data doesn't change often, like a contact's photo, then a good choice for the fetch policy is `FETCH_FROM_SERVICE_ON_CACHE_MISS_OR_EXPIRY` with an `EXPIRE_AFTER` expiration policy set to a suitable timeout.
- If data will change very frequently and you always want the most current data, but cached data is acceptable if the user is offline, then use `FETCH_FROM_SERVICE_IF_ONLINE`.

Note that setting the `noCache` property to `true` in the configuration file, as described in Synchronization Configuration File Structure, tells the Synchronization library to ignore fetch policies and to not add data to the local cache.

Policy	Description
`FETCH_FROM_CACHE`	Instructs the Synchronization library to fetch resources from the local cache only, not from the server. Because the Synchronization library retrieves resources directly from the cache, this policy can be carried out whether the client application is online or offline. If a resource is not in the local cache, then the Synchronization library returns null.
`FETCH_FROM_SERVICE`	Instructs the Synchronization library to always fetch resources directly from the server, not from the local cache. The library can only apply this policy when the client application is online. If the app is offline, the Synchronization library returns null.

Policy	Description
FETCH_FROM_SERVICE_IF_ONLINE	Instructs the Synchronization library to fetch resources from the server when the client application is online, and to fetch them from the local cache when the app is offline.
FETCH_FROM_SERVICE_ON_CACHE_MISS	Instructs the Synchronization library to fetch resources from the local cache if it is present. If a collection is empty, or if the requested object isn't in the local cache, then the Synchronization library fetches it from the server. If the app is offline, then the Synchronization library returns null.
FETCH_FROM_SERVICE_ON_CACHE_MISS_OR_EXPIRY	Instructs the Synchronization library to fetch resources from the local cache if they are present and not expired. Make sure to set `expireAfter` parameter to a suitable time period. If a collection is empty or has expired, or if the resource isn't in the local cache or has expired, then the Synchronization library fetches it from the server. If the app is offline, then it returns null.
FETCH_FROM_CACHE_SCHEDULE_REFRESH	Instructs the Synchronization library to fetch resources from the local cache and schedule a background refresh to update the cache with the latest version from the server. If a resource is not in the local cache, then the Synchronization library returns null.
FETCH_WITH_REFRESH	Instructs the Synchronization library to fetch resources from the local cache if they exist and are not expired, and schedule a background refresh to update the cache with the latest version from the server. If a resource is not in the local cache or has expired, then the Synchronization library fetches it directly from the server. If the app is offline, then it returns null.

Update Policies

Update policies define what the app should do if a resource is updated when the client app is offline.

Policy	Description
UPDATE_IF_ONLINE	If the client app is offline when the update request is sent, then the Synchronization library returns an error.
QUEUE_IF_OFFLINE	If the client app is offline when the update request is sent, then the Synchronization library queues the operation and updates the local cache when the client app is back online.

Video: Deep-Dive into the Data Offline & Sync Policies

If you want an overview of the ways you can configure synchronization policies, which methods take precedence, and the outcomes of the various policies, take a look at this video:

Video

Synchronization Policy Levels and Precedence

As described in Synchronization Policy Options, there are several policy types that you can configure for custom APIs. You can configure these at the following levels, which are listed in order of precedence, from highest to lowest. Note that the order of precedence applies to both fetch and save calls to a mobile endpoint and `requestWithURI` calls to a synchronization object.

- Response-level policies: The server can use HTTP response headers to transmit expiration and eviction policies, as described in Defining Synchronization Policies and Cache Settings in a Response Header. The server also can use a header to instruct the client to not cache a response. These policies take precedence over policies set for all other levels.
- Request-level policies: For requests made through an `OMCMobileEndpoint`, you can call the fetch builder's `setPolicy` method to set a policy at the request level. For requests made using the `requestWithURI` method, you can use the `SyncPolicy` object to set policies. Request-level policies take precedence over policies set at the resource and mobile-backend levels.
- Resource-level policies: In the configuration file, you can define a set of policies and associate the set with a resource path (URL). You can associate the set with a specific endpoint, or you can use wildcard characters to associate the set with a resource hierarchy (`/*` applies to all resources at the same level, and `/**` applies to all resources at the same level and any nested levels), as described later in this section. These policies take precedence over policies that are set at the mobile-backend level.

  When a policy type is defined for more than one resource level, then the precedence is:

  - A synchronization policy type that is defined for a specific endpoint takes precedence over the same policy type setting for a path that has wildcard characters. For example, if the URL is `www.baseuri.com/mobile/custom/incidentreport/incidents`, and an eviction policy is set for both `/mobile/custom/incidentreport/incidents` and `/mobile/custom/incidentreport/incidents/*`, then the eviction policy for `/mobile/custom/incidentreport/incidents` takes precedence.
  - Policies that are defined for a path that has the `/*` wildcard take precedence over policies for a path with the `/**` wildcard. For example, if the URL is `/mobile/custom/incidentreport/incidents/1`, and an eviction policy is set for both `/mobile/custom/incidentreport/incidents/*` and `/mobile/custom/incidentreport/incidents/**`, then the eviction policy for `/mobile/custom/incidentreport/incidents/*` takes precedence.

    For information about setting resource-level policies, see Synchronization Configuration File Structure.

- Mobile backend-level default policies. You can override the default policies at the request, response, and resource levels. These settings take precedence over the Synchronization library default settings. For information about setting mobile backend-level default policies, see Synchronization Configuration File Structure.
- Synchronization library default settings: For custom APIs, if a policy is not set at the request, resource, or mobile-backend level, then the Synchronization library default setting is used.

  Here are the default policy settings:

Setting	Synchronization Library Default Value
`conflictResolutionPolicy`	`PRESERVE_CONFLICT`
`evictionPolicy`	`MANUAL_EVICTION`
`expirationPolicy`	`EXPIRE_ON_RESTART`
`expireAfter`	Maximum integer value
`fetchPolicy`	`FETCH_FROM_SERVICE_IF_ONLINE`
`noCache`	`false`
`updatePolicy`	`QUEUE_IF_OFFLINE`

Defining Synchronization Policies Using a Configuration File

You can define the synchronization policies for a custom API's resource programmatically, and you can use a configuration file to define the synchronization policies for a mobile backend and the custom API resources that it uses. You typically define the policies in the configuration file for the following reasons:

- You can change a policy without needing to change code.
- You can view all your policies in one place.
- If you access the same resource from several places in your code, you can ensure that all accesses use the same policies.

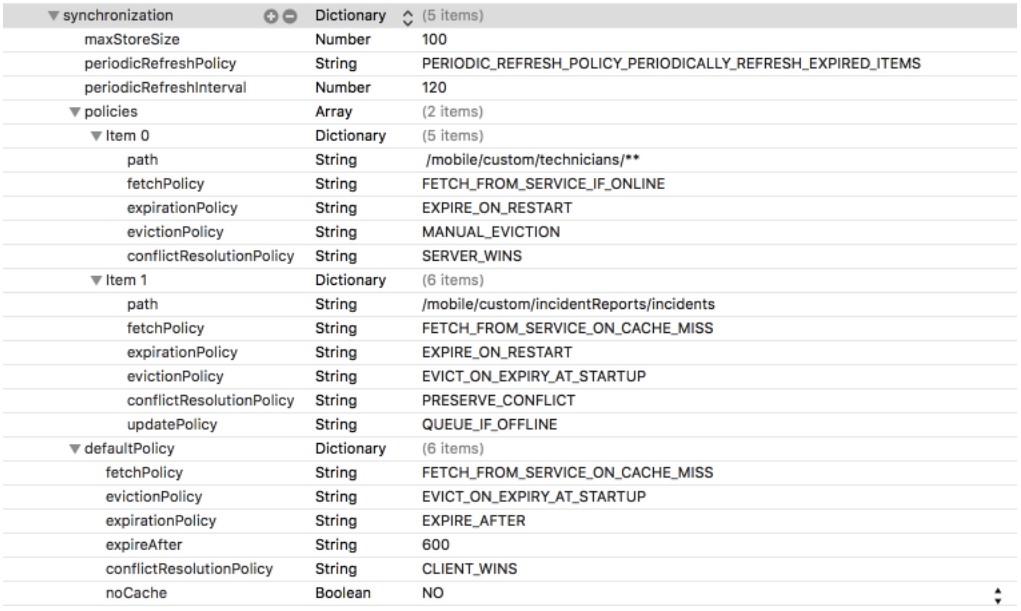
The name of the configuration file differs by platform:

- Android: `/assets/oracle_mobile_cloud_config.xml`
- iOS: `OMC.plist`

Synchronization Configuration File Structure

To configure the Synchronization library for the custom API resources that are accessed by a mobile backend, add the elements described in this section to its `synchronization` element in the configuration file.

The following illustration shows the synchronization section from an `OMC.plist` file for iOS.



▼ synchronization	Dictionary	(5 items)
maxStoreSize	Number	100
periodicRefreshPolicy	String	PERIODIC_REFRESH_POLICY_PERIODICALLY_REFRESH_EXPIRED_ITEMS
periodicRefreshInterval	Number	120
▼ policies	Array	(2 items)
▼ Item 0	Dictionary	(5 items)
path	String	/mobile/custom/technicians/**
fetchPolicy	String	FETCH_FROM_SERVICE_IF_ONLINE
expirationPolicy	String	EXPIRE_ON_RESTART
evictionPolicy	String	MANUAL_EVICTION
conflictResolutionPolicy	String	SERVER_WINS
▼ Item 1	Dictionary	(6 items)
path	String	/mobile/custom/incidentReports/incidents
fetchPolicy	String	FETCH_FROM_SERVICE_ON_CACHE_MISS
expirationPolicy	String	EXPIRE_ON_RESTART
evictionPolicy	String	EVICT_ON_EXPIRY_AT_STARTUP
conflictResolutionPolicy	String	PRESERVE_CONFLICT
updatePolicy	String	QUEUE_IF_OFFLINE
▼ defaultPolicy	Dictionary	(6 items)
fetchPolicy	String	FETCH_FROM_SERVICE_ON_CACHE_MISS
evictionPolicy	String	EVICT_ON_EXPIRY_AT_STARTUP
expirationPolicy	String	EXPIRE_AFTER
expireAfter	String	600
conflictResolutionPolicy	String	CLIENT_WINS
noCache	Boolean	NO

Cache Settings

To configure the cache settings for the mobile backend, add these elements in any order directly under the mobile backend's `synchronization` element. These settings affect both custom API and storage resources.

Key	Description	Default
`maxStoreSize`	The maximum size of the local cache in megabytes. The Synchronization library stops storing resources when it reaches this limit.	`100`
`periodicRefreshPolicy`	Names the policy that instructs the Synchronization library when to refresh cached resources. Use this attribute for background refreshes. You can set this to one of the following options: • `PERIODIC_REFRESH_POLICY_REFRESH_NONE` • `PERIODIC_REFRESH_POLICY_REFRESH_EXPIRED_ITEM_ON_STARTUP` • `PERIODIC_REFRESH_POLICY_PERIODICALLY_REFRESH_EXPIRED_ITEMS`	`PERIODIC_REFRESH_POLICY_REFRESH_NONE`

Key	Description	Default
periodicRefreshInterval	Sets the interval, in seconds, for refreshing cached resources in the background. The interval should be appropriate to the policy named by the periodicRefreshPolicy attribute.	When the periodicRefreshPolicy is PERIODIC_REFRESH_POLICY_PERIODICALLY_REFRESH_EXPIRED_ITEMS, then the default is 120.

Here's an example of adding cache settings to an OMC.plist file.

```
<key>synchronization</key>
<dict>
    <key>maxStoreSize</key>
    <integer>100</integer>
    <key>periodicRefreshPolicy</key>
    <string>PERIODIC_REFRESH_POLICY_PERIODICALLY_REFRESH_EXPIRED_ITEMS</
string>
    <key>periodicRefreshInterval</key>
    <integer>120</integer>
    ....
```

Synchronization Policy Settings

You can add the following settings at the resource and mobile-backend default levels. These are explained in Synchronization Policy Options.

- conflictResolutionPolicy
- expirationPolicy
- expireAfter
- evictionPolicy
- fetchPolicy
- noCache

Resource-Level Configuration

To configure resource-level synchronization policies for custom APIs, first add a policies node to the sychronization element.

Next, configure the policies for the specific resources:

- IOS: Add dictionary items to the policies array.
- Android: Add policy elements under policies.

You use the path element to identify the resource to associate the policy set with. You can use the path to specify a policy set for a specific endpoint, or you can use wildcard characters to associate the policy set with a hierarchy of resources:

> **Note:**
>
> You can begin your path with or without the forward slash (/).

- If there are no wildcard characters, then the request URL must match the string exactly. For example, if `<path>` is set to `/mobile/custom/incidentreport/incident` then `www.baseuri.com/mobile/custom/incidentreport/incident` matches, but `www.baseuri.com/mobile/custom/incidentreport/incidents` does not.
- `/*` matches 0 or more characters after the value in `<Path>` but does not include lower resources in the hierarchy in the wildcard matching. For example, if `<Path>` is set to `/mobile/custom/incidentreport/incidents/*` then both `www.baseuri.com/mobile/custom/incidentreport/incidents/report` and `www.baseuri.com/mobile/custom/incidentreport/incidents/id` match, but `www.baseuri.com/incidentreport/incidents/id/attachments` does not.
- /** matches 0 or more characters after the value in `<Path>` including resources lower in the hierarchy. For example, if `<Path>` is set to `/mobile/custom/incidentreport/incidents/**`, then the following match:
  - `www.baseuri.com/mobile/custom/incidentreport/incidents`
  - `www.baseuri.com/mobile/custom/incidentreport/incidents/id`
  - `www.baseuri.com/mobile/custom/incidentreport/incidents/id/attachments`

Here's an example of setting resource-level policies in an `OMC.plist` file.

```
<key>synchronization</key>
<dict>
  ...
  <key>policies</key>
  <array>
    <dict>
      <key>path</key>
      <string>/mobile/custom/incidentreport/technicians/**</string>
      <key>fetchPolicy</key>
      <string>FETCH_FROM_SERVICE_IF_ONLINE</string>
      <key>expirationPolicy</key>
      <string>EXPIRE_ON_RESTART</string>
      <key>evictionPolicy</key>
      <string>MANUAL_EVICTION</string>
      <key>conflictResolutionPolicy</key>
      <string>SERVER_WINS</string>
    </dict>
  ...
</dict>
```

Mobile Backend-Level Configuration

To define mobile backend-level synchronization policies, add a `defaultPolicy` element. Then, for each type you want to configure, add a dictionary item for iOS, and add a child element for Android.

The next sections show examples for each platform.

Android Example Configuration File

The following example for Android is an excerpt from the `oracle_mobile_cloud_config.xml` file.

```
<mobileBackends>
    <mobileBackend>
      ...
      <synchronization>
          <maxStoreSize>100</maxStoreSize>

<periodicRefreshPolicy>PERIODIC_REFRESH_POLICY_PERIODICALLY_REFRESH_EXPIRED
_ITEMS</periodicRefreshPolicy>
          <periodicRefreshInterval>120</periodicRefreshInterval>
          <policies>
              <policy>
                  <path>/mobile/custom/incidentreport/technicians/**</path>
                  <fetchPolicy>FETCH_FROM_SERVICE_IF_ONLINE</fetchPolicy>
                  <expirationPolicy>EXPIRE_ON_RESTART</expirationPolicy>
                  <evictionPolicy>MANUAL_EVICTION</evictionPolicy>
                  <conflictResolutionPolicy>SERVER_WINS</
conflictResolutionPolicy>
              </policy>
              <policy>
                  <path>/mobile/custom/incidentreport/incidents</path>
                  <fetchPolicy>FETCH_FROM_SERVICE_ON_CACHE_MISS_OR_EXPIRY</
fetchPolicy>
                  <expirationPolicy>EXPIRE_ON_RESTART</expirationPolicy>
                  <evictionPolicy>EVICT_ON_EXPIRY_AT_STARTUP</
evictionPolicy>
                  <conflictResolutionPolicy>SERVER_WINS</
conflictResolutionPolicy>
                  <updatePolicy>QUEUE_IF_OFFLINE</updatePolicy>
                  <expireAfter>300</expireAfter>
              </policy>
          </policies>
          <defaultPolicy>
              <fetchPolicy>FETCH_FROM_SERVICE_ON_CACHE_MISS</fetchPolicy>
              <evictionPolicy>EVICT_ON_EXPIRY_AT_STARTUP</evictionPolicy>
              <expirationPolicy>EXPIRE_AFTER</expirationPolicy>
              <expireAfter>600</expireAfter>
              <conflictResolutionPolicy>CLIENT_WINS</
conflictResolutionPolicy>
              <noCache>false</noCache>
          </defaultPolicy>
      </synchronization>
    </mobileBackend>
</mobileBackends>
```

iOS Example Configuration File

The following example XML for iOS is an excerpt from the `OMC.plist` file.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/
DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>mobileBackends</key>
  <dict>
    <key>myBackend/1.0</key>
    <dict>
      <key>synchronization</key>
      <dict>
        <key>maxStoreSize</key>
        <integer>100</integer>
        <key>periodicRefreshPolicy</key>

<string>PERIODIC_REFRESH_POLICY_PERIODICALLY_REFRESH_EXPIRED_ITEMS</string>
        <key>periodicRefreshInterval</key>
        <integer>120</integer>
        <key>policies</key>
        <array>
          <dict>
            <key>path</key>
            <string>/mobile/custom/incidentreport/technicians/**</string>
            <key>fetchPolicy</key>
            <string>FETCH_FROM_SERVICE_IF_ONLINE</string>
            <key>expirationPolicy</key>
            <string>EXPIRE_ON_RESTART</string>
            <key>evictionPolicy</key>
            <string>MANUAL_EVICTION</string>
            <key>conflictResolutionPolicy</key>
            <string>SERVER_WINS</string>
          </dict>
          <dict>
            <key>path</key>
            <string>/mobile/custom/incidentreport/incidents</string>
            <key>fetchPolicy</key>
            <string>FETCH_FROM_SERVICE_ON_CACHE_MISS_OR_EXPIRY</string>
            <key>expirationPolicy</key>
            <string>EXPIRE_ON_RESTART</string>
            <key>evictionPolicy</key>
            <string>EVICT_ON_EXPIRY_AT_STARTUP</string>
            <key>conflictResolutionPolicy</key>
            <string>PRESERVE_CONFLICT</string>
            <key>updatePolicy</key>
            <string>QUEUE_IF_OFFLINE</string>
          </dict>
        </array>
        <key>defaultPolicy</key>
        <dict>
          <key>fetchPolicy</key>
```

```
            <string>FETCH_FROM_SERVICE_ON_CACHE_MISS</string>
            <key>evictionPolicy</key>
            <string>EVICT_ON_EXPIRY_AT_STARTUP</string>
            <key>expirationPolicy</key>
            <string>EXPIRE_AFTER</string>
            <key>expireAfter</key>
            <integer>600</integer>
            <key>conflictResolutionPolicy</key>
            <string>CLIENT_WINS</string>
            <key>updatePolicy</key>
            <false/>
          </dict>
        </dict>
        ...
</dict>
</plist>
```

Defining Synchronization Policies and Cache Settings in a Response Header

When you implement a custom API, you can fine tune caching for a response by defining synchronization policies or basic cache settings in response headers.

To specify the basic synchronization and cache settings for a REST resource use the following optional HTTP Headers:

Header	Description
`Oracle-Mobile-Sync-No-Store`	If set to `true`, the client does not cache the returned resource.
`Oracle-Mobile-Sync-Evict`	Specifies the date and time after which the expired resource should be deleted from the local cache. Uses RFC 1123 format, for example `EEE, dd MMM yyyyy HH:mm:ss z` for `SimpleDateFormat`. The following synchronization policies are set for the resource object that is created from the response: • Eviction policy: `EVICT_ON_EXPIRY_AT_STARTUP` • Expiration policy: `EXPIRE_AFTER` with the `expireAfter` property set to date and time provided in the header value .
`Oracle-Mobile-Sync-Expires`	Specifies when the returned resource will be marked as expired. Uses RFC 1123 format, for example `EEE, dd MMM yyyyy HH:mm:ss z` for `SimpleDateFormat`.

Tracking Cache Hits with the Synchronization Library

The Synchronization library tracks cache hits and detects if the returned result came from the cache. Use these `OMCSynchronization` methods to get data about cache hits and misses:

- `cacheHitCount`: Returns the number of cache hits.
- `cacheMissCount`: Returns the number of cache misses.

How Synchronization Works with the Storage APIs

When your mobile app accesses the Storage APIs, the client SDK automatically works with the Storage library to refresh and synchronize the storage objects in the local cache. You don't need to add any code to enable synchronization with storage.

The client SDK enforces the following synchronization policies for the Storage APIs:

- Conflict resolution policy: `SERVER_WINS`
- Eviction policy: `EVICT_ON_EXPIRY_AT_STARTUP`
- Expiration policy: `EXPIRE_AFTER` `86400` seconds (24 hours).

  You can use the `Sync_CollectionTimeToLive` environment policy to override the number of seconds after which a Storage object expires. This value is conveyed to the Storage library through the `Oracle-Mobile-Sync-Expires` response header. See Offline Data Storage.
- Fetch policy: `FETCH_FROM_SERVICE_IF_ONLINE`
- Update policy: `QUEUE_IF_OFFLINE`

See Synchronization Policy Options for detailed descriptions of these synchronization policies.

Just as with the custom API resources, you can use the configuration file to override the default cache settings for storage resources on a mobile backend basis.

The default cache settings are:

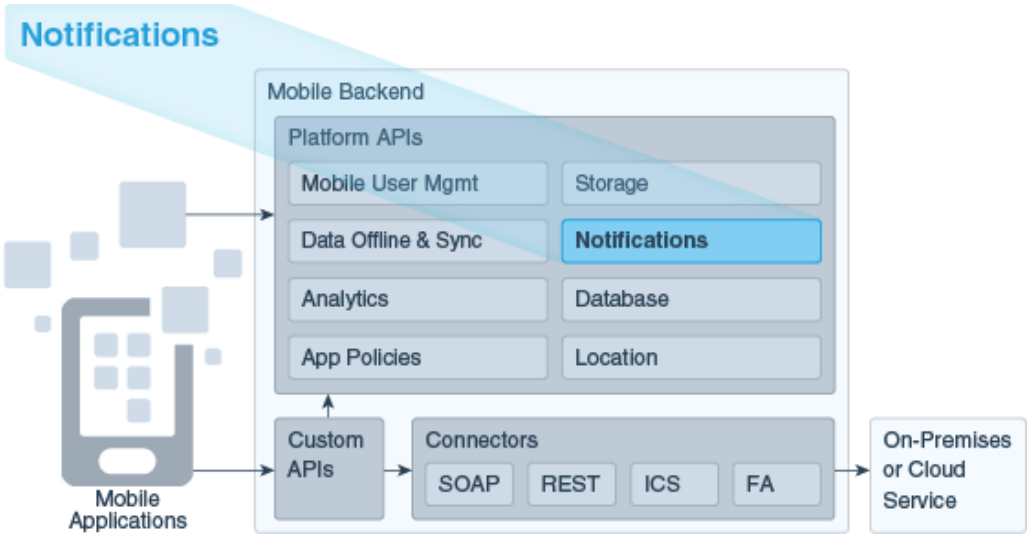
- Maximum storage size in the local cache: `100` MB
- Periodic refresh policy: Don't automatically refresh cached resources periodically

To learn how to configure the cache settings, see the Cache Settings section in Synchronization Configuration File Structure.

16

Notifications

Oracle Mobile Cloud Service (MCS) provides a Notifications API to simplify sending notifications to devices running your mobile apps. As a mobile app developer, you can set up your mobile applications for notifications and use the Notifications API to send notifications. As a service developer, you can add implementation code to your custom APIs to trigger notifications.



What Can I Do with Notifications?

Notifications can provide the timely awareness of information and events that mobile users seek. Notifications are short, specific, targeted messages sent to a mobile application. The purpose of a notification is usually to tell users that there is new information available. For example, a user who is running a shopping app might get information about an upcoming sale.

You can send these targeted messages either on demand or on a predefined schedule to:

- a specific device ID or a collection of device IDs (mostly useful for testing)
- a specific user or a collection of users
- all users and devices associated with a specific mobile backend
- devices or users for a given operating system (iOS, Android or Windows)

> **Note:**
>
> Push notifications should not be used to send critical or emergency information, because network delays and other issues can make deliveries untimely. However, for everyday uses like sports scores and upcoming sales, notifications are great.

How Are Notifications Sent and Received?

As a mobile application developer, you configure your mobile app to receive notifications over the network. Once your mobile app is configured and installed on a device, it connects to its backend to receive notifications. The steps below summarize the path that a notification takes.

1. You compose a notification, for example, "Hi! Our storewide sale is tomorrow," and define a recipient for it. You can send the notification to a specific user or device or set of users or devices, to everyone in the backend, or to a specific device type (Android, iOS or Windows). You can send the notification immediately or schedule it to be sent at a later date and time. When you POST a notification, an ID is created for the message. You can use this ID to cancel a message if it hasn't been sent yet.
2. The notification is addressed to the associated device IDs and distributed to the appropriate push networks for delivery.
3. The notification is received by the mobile application, and the owner of the device gets it.

   The notification service providers and their payload limits are:

   - WNS: 5K
   - FCM: 4K
   - GCM: 4K
   - APNS: 4K
   - SMS: 1000 bytes

What is the Device ID or Notification Token?

The device ID, also known as the notification token, uniquely identifies the specific instance of a mobile application associated with a specific device. This ID is used to ensure that notifications are sent to the correct recipient.

A unique device ID is assigned when a mobile app registers a device during the device handshake. After that point, the ID can be used to identify that specific recipient. Multiple instances of the same mobile app on the same device have different device IDs. The device ID changes periodically, but this is handled internally and is transparent to the mobile app.

You can look up the device IDs registered with a mobile app in the Device Registry, from the Notifications page for the associated backend in the UI. To register a specific device ID to be used as a recipient address for notifications, you can use the REST API. Keep in mind that sending a notification directly to a device ID is only useful for testing. There are more efficient ways to send notifications to a specific group of users. For details and examples, see Sending Notifications to and from Your App.

Setting Up a Mobile App for Notifications

Before you begin, you can install the client SDK's Notifications library to simplify development. The Notifications SDK library can be individually installed into your app, or along with the other mobile client SDK libraries. For details on the SDKs, see The SDKs.

The set up process is different for each platform:

- Setting Up Android Notifications
- Setting Up iOS Notifications
- Setting Up Windows Notifications

After you complete the set up steps for your platform, you have a few options for sending notifications from MCS to your mobile app. See Sending Notifications to and from Your App.

Now that you have registered the app client in OMCe, you have a few options for sending notifications to your app, as shown in Sending Notifications to and from Your App.

Setting Up Android Notifications

To set up your Android app for notifications, follow the steps below:

1. First, get credentials from the notification provider to establish your mobile app as a known item on the network. See Android: Google API Key.
2. Create a client for your mobile app in MCS, and configure notifications profile(s) by entering the network credentials you got in step 1. See Client Management.
3. Set up the app to connect to the notification provider from the mobile device and establish rules for communication. See Setting Up a Device Handshake for Android (FCM).

After you complete these steps, you have a few options for sending notifications from MCS to your mobile app. See Sending Notifications to and from Your App.

Android: Google API Key

Configuring an Android mobile app for notifications requires Firebase Cloud Messaging (FCM), formerly Google Cloud Messaging (GCM). GCM is being phased out, so you should configure new apps with FCM. For information on migrating existing apps, see Migrate a GCM Client App for Android to Firebase Cloud Messaging on Google Developers.

For details on setting up your Android mobile application, see Set Up a Firebase Cloud Messaging Client App on Android on Google's developer site. This page includes detailed instructions and a link to generate the required configuration file for your project, as well as information on using the Instance ID API to create and update registration tokens.

> **Note:**
>
> When you generate the configuration file for your app, make sure you choose to enable the Cloud Messaging service.

FCM Notifications

For FCM notifications, in the Android app's AndroidManifest.xml file, within the `<application>` node, add the following entries:

```
<service
android:name="oracle.cloud.mobile.fcmnotifications.McsRegistrationIntentSer
vice" android:exported="false" />
<service
android:name="oracle.cloud.mobile.fcmnotifications.MCSFirebaseInstanceIDSer
vice">
    <intent-filter>
        <action android:name="com.google.firebase.INSTANCE_ID_EVENT"/>
    </intent-filter>
</service>
```

The FCM messaging library must be added as a dependent library in the application's build file as described in Set up Firebase and FCM SDK. When generation is complete, the Project Number (aka Sender ID) and server key are displayed. You need these credentials to register the mobile app for notifications in MCS. They are unique to the mobile app and can't be used to send notifications to any other app. You also need these values to get a registration token from FCM and set up the connection with MCS, as described in Setting Up a Device Handshake for Android (FCM).

GCM Notifications

For GCM notifications, in the Android app's AndroidManifest.xml file, within the `<application>` node, add the following entries:

```
<service
android:name="oracle.cloud.mobile.notifications.McsRegistrationIntentServic
```

```
e" android:exported="false" />
<service
android:name="oracle.cloud.mobile.notifications.GcmTokenRefreshListenerServ
ice" android:exported="false">
    <intent-filter>
        <action android:name="com.google.android.gms.iid.InstanceID" />
    </intent-filter>
</service>
```

Google Play Services must be added as a dependent library in the application's build file, or these services will be flagged in error.

When generation is complete, the **Project Number** (aka **Sender ID**) and **legacy server key** are displayed. You need these credentials to register the mobile app for notifications in MCS. They are unique to the mobile app and can't be used to send notifications to any other app. You also need these values to get a registration token from FCM and set up the connection with MCS, as described in Setting Up a Device Handshake for Android (FCM).

Setting Up a Device Handshake for Android (FCM)

This section assumes you have already generated a configuration file for your app. You will need the Sender ID (Project Number) you got when you configured your project, as described in Android: Google API Key.

For FCM Notifications, an Android app needs to extend `FirebaseMessagingService` to define a service for receiving Notifications. By overriding the `onMessageReceived` method, you can perform actions based on the incoming message. For more information on handling notifications in Android, see Receive Messages on Google FCM Developers.

In your app's `src/main/AndroidManifest.xml` file, just before the closing `</application>` tag, register for the Notifications service, as shown below.

```
<application> ...
<service

android:name="oracle.cloud.mobile.fcmnotifications.MCSFirebaseMessagingServ
ice">
 <intent-filter>
    <action android:name="com.google.firebase.MESSAGING_EVENT"/>
 </intent-filter>
</service>
</application>
```

Set permissions to receive and display notifications by inserting these entries in the Android manifest (somewhere above the `<application>` entry).

```
<uses-permission
android:name="android.permission.INTERNET"/>
<uses-permission
android:name="android.permission.ACCESS_NETWORK_STATE"/>
<uses-permission
android:name="android.permission.WRITE_INTERNAL_STORAGE"/>
```

```
<uses-permission
android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
<uses-permission
android:name="android.permission.ACCESS_FINE_LOCATION"/>
<uses-permission
android:name="android.permission.ACCESS_COARSE_LOCATION"/>
<application>
```

To establish communication and register for notifications, here's what the device handshake might look like in an Android app, using the client SDK:

```
...
import oracle.cloud.mobile.exception.ServiceProxyException;
import oracle.cloud.mobile.fcmnotifications.Notifications;
import oracle.cloud.mobile.mobilebackend.MobileBackendManager;

public class MainActivity extends Activity {
    private Notifications mNotification;

    @Override protected void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.activity_main);
            this.registerNotificationClient();
}
//method that initializes and returns the Notifications client
private void registerNotificationClient(){
try {
            mNotification =
MobileBackendManager.getManager().getDefaultMobileBackend(this).getServiceP
roxy(Notifications.class);
          mNotification.initialize(this);
      } catch (ServiceProxyException e) {
            e.printStackTrace();
      }
}
}
```

Getting a FCM Registration Token

You also need the Sender ID to register your app with FCM to get a registration token. The registration token is passed to OMCe, which packages it with the notification to tell Google that your app and the device it runs on are legitimate recipients on the network. Google provides the Instance ID API to handle registration tokens. See Set Up a Firebase Cloud Messaging Client App on Android on Google Developers.

To set up a callback on successful registration, you could add code like the example below:

```
public void onClick(View view) {
  try {
    //Registration process callback
    mRegistrationBroadcastReceiver = new BroadcastReceiver() {
      @Override
```

```
      public void onReceive(Context context, Intent intent) {
        SharedPreferences sharedPreferences =
                    PreferenceManager.getDefaultSharedPreferences(context);
        boolean sentToken = sharedPreferences
                    .getBoolean(NotificationsConfig.SENT_TOKEN_TO_SERVER,
false);
        if (sentToken) {
            Logger.debug(TAG, "Token retrieved and sent to server.");
        } else {
            Logger.debug(TAG, "An error occurred while registering the
device");
        }
      }
    };
    //call on successful registration
    LocalBroadcastManager.getInstance(mCtx).registerReceiver(
            mRegistrationBroadcastReceiver,
            new IntentFilter(NotificationsConfig.REGISTRATION_COMPLETE));
    //Initialization of MCS notifications service
    not = MobileBackendManager.getManager().getDefaultMobileBackend
            (mCtx).getServiceProxy(Notifications.class);
    boolean result = not.initialize(mCtx);
  } catch (ServiceProxyException e) {
    e.printStackTrace();
  }
}
```

After you've set up and registered your app, it can send and receive notifications. For details and sample code, see Sending Notifications to and from Your App.

De-Registering a Device

To de-register a device for notifications, here's what the code might look like in an Android app, using the client SDK:

```
//Initialization of MCS notifications service
Notifications notifications =
MobileManager.getManager().getDefaultMobileBackend(getApplicationContext())
.getServiceProxy(Notifications.class);
boolean result = notifications.deregisterDevice(view.getContext());

Logger.debug(TAG, "unregister " + result);
```

Setting Up a Device Handshake for Android (GCM)

This section assumes you have already generated a configuration file for your app. You will need the Sender ID (Project Number) you got when you configured your project, as described in Android: Google API Key.

In addition to the device handshake, for GCM notifications an Android app needs to extend `GcmListenerService` to define a receiver for the Notifications service. By overriding the `onMessageReceived` method in the Android SDK, you can perform actions based on the incoming message. See Simple Downstream Messaging on Google Developers.

In your app's `main/AndroidManifest.xml` file, just before the closing `</application>` tag, register service and broadcast receivers for the Notifications service, as shown below.

```
<application>
...
    <receiver
android:name="oracle.cloud.mobile.notifications.Mcs2GcmListenerService"
            android:permission="com.google.android.c2dm.permission.SEND">
        <intent-filter>
            <action android:name="com.google.android.c2dm.intent.RECEIVE"/>
            <action
android:name="com.google.android.c2dm.intent.REGISTRATION"/>
            <category android:name="YOUR.PACKAGE.NAME"/>
        </intent-filter>
    <receiver>
</application>
```

Set permissions to receive and display notifications by inserting these entries in the Android manifest (somewhere above the `<application>` entry).

```
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
<uses-permission android:name="android.permission.WRITE_INTERNAL_STORAGE"/>
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
<permission android:protectionLevel="signature"
            android:name=YOUR.PACKAGE.NAME.permission.C2D_MESSAGE"/>
<uses-permission android:name="YOUR.PACKAGE.NAME.permission.C2D_MESSAGE"/>
<application>
```

To establish communication and register for notifications, here's what the device handshake might look like in an Android app, using the SDK:

```
...
import oracle.cloud.mobile.exception.ServiceProxyException;
import oracle.cloud.mobile.mobilebackend.MobileBackendManager;
import oracle.cloud.mobile.notifications.Notifications;

public class MainActivity extends Activity {
    private Notifications mNotification;
    private final String PROJECT_ID =
"PROJECT_ID_COPIED_FROM_GOOGLE_API_CONSOLE";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        this.registerNotificationClient();
        ...
    }
```

```
//method that initializes and returns the Notifications client
    private void registerNotificationClient(){
        try {
            mNotification =
MobileBackendManager.getManager().getDefaultMobileBackend(this).getServiceP
roxy(Notifications.class);
            mNotification.initialize(this, PROJECT_ID);
        } catch (ServiceProxyException e) {
            e.printStackTrace();
        }
    }
...
}
```

Getting a GCM Registration Token

You also need the Sender ID to register your app with GCM to get a registration token. The registration token is passed to MCS, which packages it with the notification to tell Google that your app and the device it runs on are legitimate recipients on the network.

To set up a callback on successful registration, you could add code like the example below:

```
public void onClick(View view) {
    try {
        //Registration process callback
        mRegistrationBroadcastReceiver = new BroadcastReceiver() {
            @Override
            public void onReceive(Context context, Intent intent) {

                SharedPreferences sharedPreferences =

PreferenceManager.getDefaultSharedPreferences(context);
                boolean sentToken = sharedPreferences
                        .getBoolean(NotificationsConfig.SENT_TOKEN_TO_SERVE
R, false);
                if (sentToken) {
                    Logger.debug(TAG, "Token retrieved and sent to
server!");
                } else {
                    Logger.debug(TAG, "An error occurred while either
fetching the InstanceID");
                }
            }
        };

                //call on successful registration

LocalBroadcastManager.getInstance(getApplicationContext()).registerReceive
r(mRegistrationBroadcastReceiver,
                new
IntentFilter(NotificationsConfig.REGISTRATION_COMPLETE));

        //Initialization of notifications service
```

```
        not =
MobileBackendManager.getManager().getDefaultMobileBackend(view.getContext()
).getServiceProxy(Notifications.class);
        boolean result = not.initialize(view.getContext(), "714568881816");

    }
        catch (ServiceProxyException e) {
        e.printStackTrace();
    }
}
```

After you've set up and registered your app, it can send and receive notifications. For details and sample code, see Sending Notifications to and from Your App.

Setting Up iOS Notifications

To set up your iOS app for notifications, follow the steps below:

1. First, get credentials from the notification provider to establish your mobile app as a known item on the network. See iOS: Apple Secure Certificates.
2. Create a client for your mobile app in MCS, and configure notifications profile(s) by entering the network credentials you got in step 1. See Client Management.
3. Set up the app to connect to the notification provider from the mobile device and establish rules for communication. See Setting Up a Device Handshake for iOS.

After you complete these steps, you have a few options for sending notifications from MCS to your mobile app. See Sending Notifications to and from Your App.

iOS: Apple Secure Certificates

Notifications require additional secure certificates from Apple, in addition to the certificate used to set up your account. This section assumes you have an Apple Developer account. For information on using notifications in iOS, see the *Local and Remote Notification Programming Guide* on http://developer.apple.com.

Notifications make special use of Apple's network, so Apple wants extra security protections. You need one of the following secure certificates:

- **Apple Push Notification service SSL (Sandbox)** certificate for developing and testing your application with notifications while you do development. Sandbox certificates are intended for automated QA environments where devices don't change often. In most cases, spam filters should be disabled.
- **Apple Push Notification service SSL (Production)** certificate for releasing your application to Apple's App Store. Apple requires this certificate before you can ship your app to the public, but you can wait until your app is finished to get it.

The steps for getting a Sandbox or Production certificate are very similar to the steps you used to get the first secure certificate when you set up your app. This section assumes that you already set up your Apple developer account, got the required secure certificate, and set up an Application ID and a Provisioning Profile.

1. If you didn't enable notifications in your provisioning profile when you created your App ID, go back and enable it now.

2. Get your certificate(s) from the Apple Developer Center. Use the App ID you set up when you created your app.

> **Note:**
>
> Follow Apple's direction to create a Certificate Signing Request (CSR) file, then export it to a `.p12` file to upload it to MCS. Do not password protect the `.p12` secure certificate. (Leave the password field blank when you save the `.p12` file.)

You need your certificate to register the mobile app for notifications in MCS. It is unique to the mobile app and can't be used to send notifications to any other app. Once you have configured these extra certificates, you can get a device token from Apple and set up communication with MCS, described in Setting Up a Device Handshake for iOS.

Setting Up a Device Handshake for iOS

As an iOS developer, to make a device handshake happen you need to add this code to your Xcode project to get a device token, get a notifications object, and register your app for notifications:

Note that the registration code should be called each time the app starts.

1. Get a device token from Apple.

```
if([application
respondsToSelector:@selector(registerUserNotificationSettings:)]){
    //use registerUserNotificationSettings for iOS 8 and later
    UIUserNotificationSettings *settings=[UIUserNotificationSettings
settingsForType:(UIUserNotificationTypeBadge
                |UIUserNotificationTypeSound
                |UIUserNotificationTypeAlert) categories:nil];
    [application registerUserNotificationSettings:settings];
} else {
    //We expect deprecation warnings here - this is for iOS 7.1 or
before
    [[UIApplication sharedApplication]
registerForRemoteNotificationTypes:
    (UIRemoteNotificationTypeBadge | UIRemoteNotificationTypeSound |
UIRemoteNotificationTypeAlert)];
}
```

After calling the above lines of code, the Apple Push Notification Service (APNS) will call one of the delegate methods based on the success or failure to retrieve the device token. If successful, one of the following methods is called: `didRegisterUserNotificationSettings:` (iOS 8 or later) or `didRegisterForRemoteNotificationsWithDeviceToken:` (iOS 7.1). In case of an error, the `didFailToRegisterForRemoteNotificationsWithEffor:` method is called.

2. Get the Notifications SDK object.

```
(OMCNotifications *) getOMCNotifications{
   OMCAuthorization *auth = [[OMCMobileBackendManager sharedManager]
mobileBackendForName:
     <Name_of_Mobile_Backend_from_OMC.Plist>].authorization;
   OMCNotifications* omcNOtifications=nil;
   NSError* err = [auth authenticate:<Username> password:<Password>];
   if (!err){
      omcNotifications = [[[OMCMobileBackendManager sharedManager]
mobileBackendForName:
         <Name_of_Mobile_Backend_from_OMC.Plist>] notifications];
   }
   return omcNotifications;
}
```

3. Register for notifications using the Notifications SDK object. Note that `omcNotifications` is the object of `OMCNotifications.class`. .

```
[omcNotifications registerForNotifications:dataDeviceToken
    onSuccess:^(NSHTTPURLResponse *response){
        dispatch_asynch(dispatch_get_main_queue(),^{
            //Update UI here
        });
    } onError:^(NSError *error){
    dispatch_async(dispatch_get_main_queue(),^{
            //Update UI here
        });
    }];
```

Next, register your mobile app with the associated mobile backend, and enable notifications. SeeRegistering an App as a Client in MCS.

After you've registered your app, it can receive notifications from a range of sources. For details and sample code, see Sending Notifications to and from Your App.

Setting Up Windows Notifications

To set up your Windows app for notifications, follow the steps below:

1. First, get credentials from the notification provider to establish your mobile app as a known item on the network. See Windows: WNS Credentials or Syniverse: SMS Credentials.
2. Create a client for your mobile app in MCS, and configure notifications profile(s) by entering the network credentials you got in step 1. See Client Management.
3. Set up the app to connect to the notification provider from the mobile device and establish rules for communication. See Setting Up a Device Handshake for Windows.

After you complete these steps, you have a few options for sending notifications from MCS to your mobile app. See Sending Notifications to and from Your App.

Windows: WNS Credentials

Configuring a Windows mobile app for notifications requires a unique set of credentials for Windows Push Notification Service (WNS). This section assumes you have a Microsoft Developer account.

The following credentials are required to authenticate with WNS:

- **Client ID** (also called the Package SID)
- **Client Secret** (also called a secret key)

To get these credentials, register your mobile app in the Windows Store Dashboard, accessible from the Windows Dev Center. For details on WNS, see WNS Overview on MSDN.

You need these credentials to register the mobile app for notifications in MCS. They are unique to the mobile app and can't be used to send notifications to any other app.

Syniverse: SMS Credentials

To send Short Message Service (SMS) messages using the Syniverse Messaging Service, the first step is to establish a profile on the Syniverse Developer Community, where you subscribe to the service, register your app, and get credentials.

Creating a Profile on the Syniverse Developer Community

1. Go to the Syniverse Developer Community (developer.syniverse.com).
2. Click **Sign Up** in the top right corner of the site and enter the requested information.
3. If you have an invitation code from a company in the Syniverse Developer Community enter that into "Company invite code" field. If not, ignore this step.
4. Read and accept the Terms of Service.
5. Check the **Captcha** box and answer the challenges to prove you aren't a robot.
6. Click **Create profile**.
7. When the confirmation email arrives, click the link in the email and verify your user credentials.

Subscribing to the Syniverse Messaging Service

To use SMS in your apps using the SMS short code you got from Syniverse, you need to subscribe to the Syniverse Messaging Service.

1. Log in to the Syniverse Developer Community (developer.syniverse.com).
2. Click your user name in the top right corner and select **Company**. Verify that your accounts have a billing address associated with them.
3. Navigate to **Service Offerings** > **Messaging Offering** and click **Subscriptions**.
4. Click **Subscribe** and select "Initial account for [Your username]"
    a. Read and accept the Terms of Service.
    b. Select **Confirm**.
    c. Verify that your account is listed in **Subscriptions**.

5. If you're using a Syniverse-provisioned public channel to test messages, you also need to add test phone numbers to the associated whitelist. (Whitelisting is only necessary when testing SMS to U.S. or Canada phone numbers and isn't required for production apps.)
   a. Click your user name in the top right corner and select **Company**.
   b. On the Company page, click the **Whitelist** tab.
   c. Click **Add phone number** and enter your phone number in the ITU-T E.164 format (i.e., +11234567890).
   d. Click **Send confirmation code** to send a randomly generated number to the phone number in a text message.
   e. Retrieve the confirmation code from the text message and enter it in the **Confirmation code** field. Click **Add** to confirm the phone number whitelist.
   f. Verify that your phone number is included in the whitelist table with "Validated" status.

Register Your App and Get Credentials

Before messages can be sent through the Syniverse Messaging Service, there must be an application configured in the SDC platform. Once your app is registered, you can generate the required credentials.

1. Log in to the Syniverse Developer Community (developer.syniverse.com).
2. Click **Applications**.
3. Click **New application**.

   In the dialog:
   a. Give your application a name and description and click **Save**.
   b. Click the gear icon next to your app name and select **Edit**.
   c. Click **SDC Self Service** and make sure all the options are selected.
   d. Click **Account & APIs** and select the "Initial account for [Your username]" from the **Account** dropdown.

      Turn on the following services: **Messaging**, **SDC Gateway Services**, **Event Subscription Services**, **Voice & Messaging** and **Whitelisting Services**.
   e. Click **Save**.
4. Generate the required credentials:

Setting Up a Device Handshake for Windows

This section assumes you have already registered your mobile app with WNS, described in Windows: WNS Credentials.

Here's what a device handshake might look like in a Windows app, using the SDK:

```
...

using Oracle.Cloud.Mobile.Notifications;
using Windows.Networking.PushNotifications;

namespace MyWindowsApp
```

```
{
    public sealed partial class MainPage : Page
    {
        public MainPage()
        {
            this.InitializeComponent();

            //
            // First login to MCS
            //
            var loginDialog = new LoginDialog();
            loginDialog.ShowAsync().ContinueWith((task) =>
RegisterForNotificationsAsync());
        }

        private async Task RegisterForNotificationsAsync()
        {
            var backend = ((App)App.Current).Backend;

            // Register for Push Notifications
            PushNotificationChannel channel =
                await
PushNotificationChannelManager.CreatePushNotificationChannelForApplicationA
sync();

            await
backend.GetService<Notifications>().RegisterForNotificationsAsync(channel.U
ri);
        }

        ...
    }
}
```

For details on requesting a channel URI and constructing the notification payload, see Windows Push Notification Services (WNS) overview.

Next, register your mobile app with the associated MCS mobile backend, and enable notifications. For detailed instructions, see Registering an App as a Client in MCS in the Mobile Backends chapter.

After you've registered your app, it can receive notifications from a range of sources. For details, see Sending Notifications to and from Your App.

Sending Notifications to and from Your App

Once you've set up and registered your mobile app, you can start sending notifications and SMS messages.

- Send notifications and cancel scheduled notifications from the UI, which can be useful for development.
- Use the Notifications API to send notifications to and from apps and devices all over the place.

You can also check the status of your notifications in the UI or using the Notifications API. For details, see Troubleshooting Notifications.

Testing Notifications from the MCS UI

MCS provides a notifications testing UI that allows you to send scheduled notifications to a defined set of recipients.

1. Make sure you're in the environment where you want to create the notification.

   Click  to open the side menu and select **Applications > Mobile Backends**.
2. On the Mobile Backends page, select the mobile backend that includes your mobile app and click **Open**.
3. Click **Notifications**.
4. On the Notifications page, click the **Send** icon.
5. If your device isn't registered yet, you can access the Device Registry by clicking **Manage Devices**.

   To register a device for SMS through the UI, you must have consent management disabled in the associated MCS client profile as described in Client Management. If you register a device for SMS through the UI and it fails, it's probably a problem with your Syniverse Developer Community setup. Make sure you completed all the steps described in Syniverse: SMS Credentials.
6. Enter the notification message you want to send in plain text or a JSON payload. If you enter JSON, it must conform to the notification provider's requirements. If it is not valid JSON, it will be sent as a plain text message.
7. Choose when to send the message.
   - To send the notification immediately, leave the default **Now**.
   - To schedule the notification for a later date and time, choose **Later** and select the date and time for the notification to be sent.
8. Choose who to send the message to.
   - To send the notification to everyone in the mobile backend, leave the default **All notifications-enabled mobile apps that use this mobile backend**. A single mobile backend may contain more than one version of a mobile application, with implementations for different devices and networks. This option sends to all notification-enabled clients, regardless of the network or device.
   - To define a filter by user name, platform type, device ID, Facebook ID, or any combination, choose **Filtered set of recipients**. Under **Match all of the following**, select the filter type from the dropdown list:
     - **Device ID**: Send a notification to a single device ID or to multiple device IDs at the same time. The device ID is a unique number assigned to a mobile device during the device handshake. For SMS, the device ID is a phone number. In general, sending a notification to a device ID is useful for testing your application but not practical in bulk.
     - **Platform**: Send to all recipients running on iOS, Android, Windows or Web.
     - **Provider**: Send to all recipients receiving APNS, GCM, FCM, WNS or SMS notifications.

- **User**: Send a notification to a single user or to a list of users.
- **Facebook Unique ID**: Send a notification to a Facebook user, by ID.

If the list of recipients gets too long, click the **+** button to add another filter and continue your entries there. Filters can be mixed and matched for additional selectivity.

9. Click **Send**.

Once you click Send, you can monitor the status of your notifications in the **History** pane. For details, see Troubleshooting Notifications.

Cancelling a Scheduled Notification from the UI

The only notifications that can be cancelled are those that are scheduled for a future time.

To cancel a scheduled notification, go to the **Scheduled** tab in the History pane and click the **X** in the corner of the entry you want to remove. You will be prompted to confirm the cancellation.

Sending Notifications Using the Notifications API

You can send notifications to mobile devices from your apps using the Notifications API. Notifications have a maximum limit of 1,000 devices per call.
You can call Notifications REST API endpoints directly or use custom code in your mobile app. This section details the REST endpoints. For information on using custom code including examples and sample code, see Accessing the Notifications API from Custom Code in the Calling APIs from Custom Code chapter.

To register a device ID for notifications, you can use the UI or the Notifications Device Registration API as described in Registering a Device ID.

The `/mobile/system/notifications/notifications` endpoint allows you to send notifications, cancel scheduled notifications, and check the status of sent notifications.

> **Note:**
>
> Calls to this endpoint must include these headers:
>
> - `Authorization`: If you're using basic authentication, this header should include the name and password for a team member with the `MobileEnvironment_Notification` role, encoded in Base64. For OAuth, this header should include the access token. If you're using OAuth, you must also be a team member with the `MobileEnvironment_Notification` role.
> - `Oracle-Mobile-Backend-ID`: If you're using basic authentication, you must include this header. The mobile backend ID is listed on the Settings tab for the mobile backend. For OAuth, this information is included in the access token.

When you send a notification, you can specify any combination of the following for the payload:

- `{"payload":""}` A unified payload that includes well-formed JSON for each supported notification provider (Google, Apple, Windows and Syniverse). For details, see Sending a Notification Using a Unified Payload.
- `{"template":""}` A reusable payload template with defined parameters, used to create payloads for each supported notification provider. The payload template includes the following optional parameters: `title`, `body`, `badge`, `sound` and `custom`. For details, see Sending a Notification Using a Payload Template.
- `{"message":""}` A plain-text message string. For details, see Sending a Text Message Notification.

The unified payload is used if it exists, then the template, then the message, in that order.

To send notifications to specific recipients, add an argument after the content of the payload:

- To send to a user or a list of users, add the `users` argument. A user can be defined by `firstname:lastname` or email address. Multiple users are listed as tokens in an array, and there's no limit on the number. For example:

  ```
  -d '{"message": "Hi! Our storewide sale is tomorrow.", "users":
  ["bob@acme.com", "sjones@xyz.net", "banana@peelme.com"]}'
  ```

- To send to everyone on the same mobile platform, add the `platform` (IOS, ANDROID, WINDOWS or WEB). For example:

  ```
  -d '{"message": "Hi! Our storewide sale is tomorrow.", "platform":
  "IOS"}'
  ```

- To send to a specific notification provider, add the `provider` (APNS, GCM or FCM, WNS or SYNIVERSE). For example:

  ```
  -d '{"message": "Hi! Our storewide sale is tomorrow.", "provider":
  "APNS"}'
  ```

- To send to a specific device ID or a list of device IDs, add the `notificationTokens` argument. Multiple IDs are listed as tokens in an array, and there's no limit on the number. For example:

  ```
  -d '{"message": "Test of notifications feature.", "notificationTokens":
  ["2DD2D2-D2DDG44GD-GDGSDFZS3-3-3DFZSDFDS"]}'
  ```

To schedule a notification for a future date and time, add the `sendOn` argument. For example:

```
-d '{"message": "Come to our discount sale today!", "sendOn":
"2015-06-15T6:00Z"}'
```

For further details, including HTTP response status codes and full schemas for the request and response bodies, see the REST APIs for Oracle Mobile Cloud Service.

Registering a Device ID

The Notifications Device Registration API lets you register the device ID of your mobile app, which can then be used as a recipient address for sending notifications. This API can also associate a user with the device ID, so the user name can also be used as a target for notifications.

You can register a device ID (`notificationToken`) directly and send notifications directly to that ID. You can also use this API to associate any user with the device ID.

The Notifications Device Registration API includes the following endpoints:

- `POST /mobile/platform/devices/register`
- `POST /mobile/platform/devices/deregister`

When you register a device, include these parameters:

- The `mobileClient` parameter identifies the client in the backend with three properties:
  - `id`: The Application ID assigned by the Google or Apple app store. (This is different from the "App-Key".)
  - `version`: The version of the mobile client that will receive the notifications, currently 1.0.
  - `platform`: "IOS" or "ANDROID" or "WINDOWS" or "WEB" (all caps)
- The `notificationProvider` parameter defines the service the `notificationToken` is used for: "APNS" or "GCS" or “FCM” or "WNS" or "SYNIVERSE".
- The `notificationToken` parameter defines the token needed by the notification service for sending calls. This token uniquely identifies the specific instance of a mobile app associated with a specific device, and is used to ensure that notifications are sent to the correct recipient. Encode in hexadecimal if necessary.
- The optional `user` parameter associates the device ID with the user name provided. If the `user` parameter isn’t included, the device ID is associated with the user who is logged in during the registration call.

> **Note:**
>
> To specify a different user name, the logged in user must be a team member with the `MobileEnvironment_Notifications` role. Keep in mind that registering a user name this way doesn’t validate the entry in the Device Registry. If this results in duplicate user names, notifications could be sent to multiple users. It’s up to the app to ensure that user names are unique if that’s a requirement.

This example registers a device with the device ID `MyAppToken`:

```
curl -v
  -H "Authorization: Basic
VGVzdE1vYmlsZVVzZXIyYzE4YWRiZjMyMDg0ZWZkOWQyODM0NjA1OGNmExampleAuthString="

  -H "Oracle-Mobile-Backend-ID: 7cf06198-053e-4311-8186-cae145900d59"
```

```
  -H "Content-Type:application/json"
  -d '{"mobileClient": {"id":
"MyClientac3d8baf1aa348b48d80e9b7fd026067","version": "1.0","platform":
"IOS"},"notificationProvider":"APNS","notificationToken":"03767dea-29ac-444
0-b4f6-75a755845ade","user":"JoeSmith"}'
     http://www.fixitfast.com:8080/mobile/platform/devices/register
```

If the REST operation to register the device is successful, you can expect to get a response something like this:

```
Connected to fixitfast.com port (10.176.45.198) port 8080 (#0)
Server auth using Basic with user 'lucy'
POST /mobile/platform/devices/register/
Authorization: Basic
VGVzdE1vYmlsZVVzZXIyYzE4YWRiZjMyMDg0ZWZkOWQyODM0NjA1OGNmExampleAuthString=
User-Agent: curl/7.33.0
Host: fixitfast.com:8080
Accept: application/json
Content-Type: application/json
Oracle-Mobile-Backend-ID: 7cf06198-053e-4311-8186-cae145900d59
Content-Length: 32
upload completely sent off: 32 out of 32 bytes
HTTP/1.1 201 Created
```

The response includes a JSON payload that contains the device ID for the registered device.

```
{
  "id": "7cf06198-053e-4311-8186-cae145900d59",
  "user": "JoeSmith",
  "notificationProvider":"APNS",
  "notificationToken":"03767dea-29ac-4440-b4f6-75a755845ade",
  "mobileClient": {"id":
"MyClientac3d8baf1aa348b48d80e9b7fd026067","version": "1.0","platform":
"IOS"},
  "modifiedOn": "2016-05-25T14:58:16.373Z"
}
```

Sending a Text Message Notification

The example below uses the Notifications REST API to send a simple notification to everyone in the mobile backend. As noted above, the name and password sent in the `Authorization` header must be a team member with the necessary permissions.

```
curl -X POST
  -H "Authorization: basic bWNzOldlbGNvbWUxKg=="
  -H "Accept: application/json"
  -H "Content-Type: application/json; charset=UTF-8"
  -H "Oracle-Mobile-Backend-ID:1d97542d-51d6-4f18-897f-35053cfdfd2d"
  -d '{"message": "Hi! Our storewide sale is tomorrow."}'
    http://www.FixItFast.com:8080/mobile/system/notifications/
notifications/
```

If the notification is sent successfully, the response might look like the example below. The body will be the JSON for the created notification.

```
Connected to FixItFast.com port (10.176.45.198) port 8080 (#0)
Server auth using Basic with user 'lucy'
POST /mobile/system/notifications/notifications/ HTTP/1.1
Authorization: Basic bWNzOldlbGNvbWUxKg==
User-Agent: curl/7.33.0
Host: newclothes.com:8080
Accept: application/json
Content-Type: application/json; charset=UTF-8
Oracle-Mobile-Backend-ID:1d97542d-51d6-4f18-897f-35053cfdfd2d
HTTP/1.1 201 Created
```

You could also get a status code of `400` (bad request) or `401` (unauthorized).

Sending a Notification Using a Unified Payload

A unified payload allows you to specify a different payload for each supported notification provider using Notifications REST API. One or more of the following can be defined under the `services` property:

- The `apns` payload must conform to APNS requirements.
- The `gcm` or `fcm` payload can contain arbitrary JSON properties.
- The `wns` payload property must contain a well-formed `WNS payload`.
- The `syniverse` payload property should contain the string to send as a SMS message.

> **Note:**
>
> The payload template allows you to send provider-specific payloads without defining the code. For details, see Sending a Notification Using a Payload Template.

The following are simple examples that define payloads for FCM. An FCM object can contain either a notification object or a data object. A notification object has a predefined set of user-visible keys described in the FCM documentation. A data object has custom key-value pairs.

Notification object:

```
{"notificationTokens": [ "xxxxx"],"payload": {"services": {"fcm":
{"notification": {"title": "Sale On Now!","body": "50% off until Saturday"
        }
      }
    }
  }
}
```

Data object:

```
 "notificationTokens": [ "xxxxxx"],"payload": {"services": {"fcm":
{"data": {"acme1": "value1","acme2": "value2"
        }
      }
    }
  }
}
```

Sending a Notification Using a Payload Template

When you use a payload template with the Notifications REST API, the content you enter is used to create a driver-specific payload for each supported notification provider. The default payload template includes the following optional parameters.

Parameter	Description	Data Type	Example
`title`	The alert title. If a title is specified, the `body` parameter is also required.	string	`"Sale On Now!"`
`body`	The alert body. If only a `body` is specified, the content is used as the value for the `alert` property in the APNS and FCM payloads.	string	`"50% off until Saturday"`
`badge`	A number to badge the notification with. Android applications don't support badging, so the number is not passed in the payload. If there is a requirement to pass the "badge" value, it can be passed as part of a custom data payload.	number	`43`

Parameter	Description	Data Type	Example
sound	The sound file to play with the notification. Only .wav format is supported by APNS , WNS, and FCM. • For APNS, the file must be in the app bundle. • For WNS, the file must be in the app package (the "ms-appx:///" prefix is added automatically). • For FCM, the file can be anywhere.	string	"alert.wav"
custom	Any required custom data.	object	{ "acme1": "value1", "acme2": ["value2", "value3"] }

The example below shows a notification sent using FCM that includes all five parameters and the resulting payload. An FCM object can contain either a notification object or a data object. A notification object has a predefined set of user-visible keys described in the FCM documentation. A data object has custom key-value pairs.

This specifies the default template:

```
{
 "template": {
 "name" : "#default",
 "parameters": {
 "title":"this is the title",
 "body":"this is the body",
 "sound":"alert.wav",
 "badge": 5,
 "custom":
{ "key1": "value1", "key2": "value2", "key3": [ "value3.1", "value3.2"] }
}
 },
```

This payload is delivered in the same way as the following unified payload. As noted above, Android apps don't support badging, so your app can use the `badge` value in other ways. Note that in this example, value is a string, so the value for key3 is converted to a string.

FCM driver payload:

```
"fcm": {
 "notification":
{ "title": "this is the title", "body": "this is the body", "sound":
"alert.wav" }
"data":
{ "key1": "value1", "key2": "value2", "key3": "[ \"value3.1\",
\"value3.2\"]" }
}
```

Cancelling Scheduled Notifications

To cancel a scheduled notification, send DELETE to `/mobile/system/notifications/notifications/{id}` with the ID assigned to the notification you want to cancel. For this example, the notification ID is `113455`.

```
curl -X DELETE
  -H "Authorization: Basic bWNzOldlbGNvbWUxKg=="
  -H "Oracle-Mobile-Backend-ID:1d97542d-51d6-4f18-897f-35053cfdfd2d"
  -H "Accept: application/json"
  -H "Content-Type: application/json; charset=UTF-8"
    http://www.fixitfast.com:8080/mobile/system/notifications/
notifications/113455
```

Troubleshooting Notifications

Sending a notification is an asynchronous process. Once you send a notification, it can sit for minutes, hours, or maybe even days on an Apple, Google or Microsoft server before it gets delivered to the mobile device. Even if a notification can't be delivered, there might be no error message returned. You have no control over a notification once it gets sent, but these are some common notification problems:

- A secure certificate is missing, expired, or not located in the right place.
- The network credentials for the device don't match the credentials registered.
- A security identifier used in your code doesn't match the identifier registered with Google, Apple or Windows, or match what's defined in your Android manifest or iOS Xcode project.
- The wrong identifier has been entered into a form. For example, when you register for notifications in a backend and it asks you for an API Key, you entered the application key instead.
- An APNS mismatch between production/development flag and certificate, for example uploading a production certificate but configuring the client saying it's a development certificate.
- In FCM and GCM, the wrong API key or Project Number/Sender ID means the user might have disabled notifications on their device.

MCS will automatically unregister the device if a notification is sent to it and the notification provider reports the device ID as being bad. This can happen in a few ways:

- The most likely is that the token has expired. A device token lasts between 30 and 90 days depending on the provider. A mobile app should reregister the notifications token every time the app starts up with both MCS and the notifications provider to refresh it.
- The user deleted the app from their device
- The API key or certificate in MCS has gone bad by either expiring, or a new API key or certificate was requested from Google/Apple and not uploaded.
- The user has reinstalled/updated their OS and hasn't run the app since reloading the OS.
- The token was mangled somehow during registration.

Checking Notification Status in the UI

Check the **History** pane, accessible from the **Notifications** page for your mobile backend, to find out if your notifications were successfully sent.

Scheduled notifications are displayed in the **Scheduled** tab. To see a list of sent notifications, click the **Sent** tab. If you don't see the notifications you expect, click **Check for Updates**.

The status you see in the History pane reflects the success rate of the notifications that have been sent. You can quickly tell the status of each notification in the History pane by the color in the left column:

- **Green** means that more than 70% of individual notifications in the batch were accepted by the Apple and/or Google networks.
- **Yellow** means that less than 70% of individual notifications in the batch were accepted.
- **Red** means that the batch failed to send successfully from MCS. In most cases, there is a configuration error that needs to be fixed. See Troubleshooting Notifications.
- **Blue** means a batch of notifications is currently being sent. In most cases, a Blue indicator appears for only a few moments.

Given the large the number of recipients sent to a popular mobile application, there will never be 100% success. For example, if a notification is directed to a user that has recently lost her phone, the Apple or Google network won't accept the notification for delivery to the device. The default warning threshold is 70%, but you can change it in the `Notifications_DeviceCountWarningThreshold` environment policy.

The **Device Manager**, also accessible from the **Notifications** page for your mobile backend, lists all registered devices for the mobile backend with their device IDs/ notification tokens. If you don't see your device, the network provider might have specified that the device ID/notification token is invalid and should be deregistered. Also, if a device hasn't been reregistered in 60 days, it will be removed from the registry. You can click **Clear Registry** to remove all registered devices from a mobile backend to facilitate troubleshooting.

You can always look at the MCS logs to see if more information about a notification or batch of notifications is available. Click  to open the side menu and select **Administration > Logs**. For details on the diagnostics tools available through MCS, see Diagnostics.

Checking Notification Status with the Notifications REST API

You can use the Notifications API to check the status of notifications.

Send `GET` to `mobile/system/notifications/notifications` with the ID of the notification or using the `status=` query parameter. You can check for any notification status: `New`, `Scheduled`, `Sending`, `Error`, `Warning`, or `Sent`. (The notification must have been successfully sent.)

The example below checks for scheduled notifications.

```
curl -i
-X GET
-u team.user@example.com:Welcome1!
-H "Oracle-Mobile-Backend-ID: ABCD9278-091f-41aa-9cb2-184bd0586fce"
http://fif.cloud.oracle.com/mobile/system/notifications/notifications/?
status=Scheduled
```

If the query is successful, the response will be JSON listing the first 1000 notifications found. You can specify a range using limit and offset parameters, for example, `limit=100&offset=400` would return notifications 400-499.

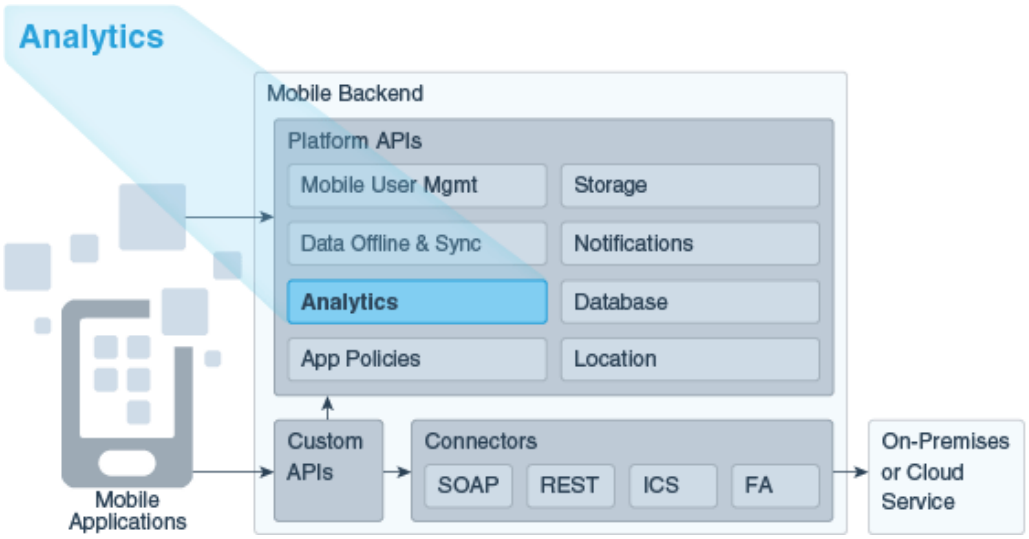
```
{
  "items": [
   {
    "id": 1234,
    "tag": "Marketing",
    "message": "This is the alert message.",
    "status": "Sent",
    "notificationTokens": ["APNSdeviceToken"],
    "createdOn": "2014-04-02T12:34:56.789Z",
    "platformCounts": [
     {
       "platform": "IOS",
       "deviceCount": 1,
       "successCount": 1
     }
  ],
  "links": [
   {
     "rel": "canonical",
     "href": "/notifications/1234"
   },
   {
     "rel": "self",
     "href": "/notifications/1234"
   }
  ]
 },
 {
   "id": 1235,
   "tag": "System",
   "message": "Update required.",
   "status": "Sent",
```

```
    "processedOn": "2014-04-01T12:34:56.789Z",
    "notificationTokens": ["APNSdeviceToken"],
    "platformCounts": [
     {
      "platform": "IOS",
      "deviceCount": 1,
      "successCount": 1
     }
    ],
    "createdOn": "2014-04-03T58:24:12.345Z",
    "links": [
     {
      "rel": "canonical",
      "href": "/notifications/1235"
     },
     {
      "rel": "self",
      "href": "/notifications/1235"
     }
    ]
   }
  ],
   "hasMore": false
   "links": [
    {
     "rel": "canonical",
     "href": "/notifications?offset=0&limit=2"
    },
    {
     "rel": "self",
     "href": "/notifications?offset=0&limit=1000"
    }
   ]
}
```

17

Analytics

Oracle Mobile Cloud Service (MCS) provides an Analytics API to help you measure patterns in app performance and usage. As a business development manager or mobile program manager, you can use analytics to find out how to improve your apps.



What Can I Do With Analytics?

Use Analytics to gain insight into how (and how often) users use a mobile app at any given time. The analytics reports generated enable you to see an application's adoption rate, and find out which functions are used the most (or the least).

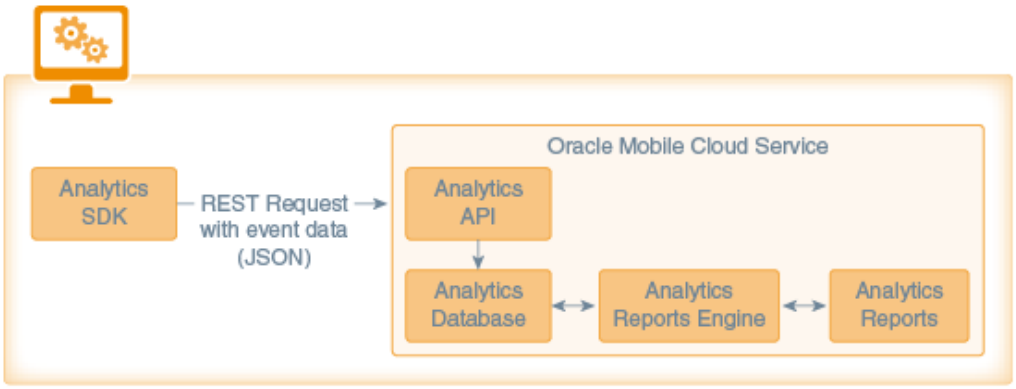
How Does MCS Create Analytics Reports?

MCS creates analytics reports from events, which describe how users interact with the mobile app.

A mobile app developer can track the mobile app's entire usage by raising events in the mobile app code. For example, a mobile app for repair technicians might track events like *Work Order Dispatched*, *Work Order Accepted*, *Work Order Resolved*, and *Work Order on Hold*. To add further detail to an event, you can define properties that describe an event's characteristics. For the *Work Order on Hold* event, for example, you might add properties for *Customer Not Home* or *Parts on Order*.

> **Tip:**
>
> Mobile program managers should decide which aspects of an app to track by events early in the app development process.

Mobile backends receive events from the REST calls made from mobile apps. A mobile app makes a single call, which includes a JSON payload that describes the events along with such contextual information like a user's location, the start and end of a user session, and details about the user's mobile device. You can craft the payload yourself if you use straight REST calls, or use the mobile client SDK to construct one for you. The SDK defines the user session and automatically applies the user and system context that allows MCS to generate reports that describe the number of users of the app, and how (and from where) they're using it.



> **Note:**
>
> While the SDK enables Analytics to automatically generate reports that tell you how many users your app has, or how much time they're spending on it, you must define events in the mobile app's code if you want to see these reports.

Enabling Your Mobile Apps to Report Event Data

MCS creates analytics reports from information conveyed in JSON payloads. The calls that deliver the JSON payload to the Analytics API, which records event data, can be either straight REST calls or REST calls made through the mobile client SDK. In either case, MCS uploads and stores the JSON payload and then graphs it in a report.

Describing Analytics Events in JSON

The JSON payload describes the context for mobile app users in terms of both their mobile devices and the events that track user interactions. These types of events are known as custom events. A JSON payload has one or more of these custom events, and is also constructed from a context event that provides user and system details, a start session event, and end session event. The custom events are grouped within the session events to describe an analytic session.

Within the mobile app code, developers can determine the point at which the app flushes the custom events that have accumulated on the mobile device to the MCS server. This content is considered to be a session that can be logged. Theoretically, an analytic session can remain open for longer than a single batch update to the MCS server. In other words, sessions can vary in length according to your event logging use case: a session might be created to track event data for a single action or a set of

actions that comprise a task. You can also use a session to log the entire span of user interactions within a user session. That said, the length of an analytic session generally does not, and should not, equal that of a user session. Instead, create analytic sessions that are short and concise. By keeping these sessions crisp, you'll maintain system performance and accurate event reporting.

> **Note:**
>
> The mobile client SDK tracks analytic sessions on a file system, which means that a file grows as you add more events to a session. The MAFMCS Utility, which allows mobile apps built using Oracle Mobile Application Framework (MAF) to access MCS , enables sessions to be saved in memory. However, saving sessions in memory might degrade memory consumption when there are a large number of custom events (say, more than 1000). Consequently, you might lose some event logging, because the mobile app may crash before it can post events to MCS . See MAF Utility Developer Guide.

Taking a Look at the JSON Payload

Within a JSON payload, events have the following properties:

- A `name` of fewer than 100 characters.
- A unique string defined for the `sessionID` property, which associates an event with a particular session. If you create your own JSON, you must assign a unique string to this property. The mobile client SDK ensures uniqueness by adding a text string punctuated by hyphens known as a Universally Unique Identifier (UUID).
- A time stamp: Events are ordered by time stamp (though not strictly, because events can share the same time stamp). The mobile client SDK generates the time stamp automatically.

A JSON payload posted toMCS may look something like this:

```
[
    {
        "name":"context",
        "type":"system",
        "timestamp":"2013-04-12T23:20:54.345Z",
        "properties":{
            "userName":"jimSmith",
            "model":"iPhone5,1",
            "longitude":"-122.11663",
            "latitude":"37.35687",
            "timezone":"-14400",
            "manufacturer":"Apple",
            "osName":"iPhone OS",
            "osVersion":"7.1",
            "osBuild":"13E28",
            "carrier":"AT&T"
                    }
    },
    {
        "name":"sessionStart",
```

```
        "type":"system",
        "timestamp":"2013-04-12T23:20:55.052Z",
        "sessionID":"2d64d3ff-25c7-4b92-8e49-21884b3495ce"
    },
    {
        "name":"PurchaseFailed",
        "type":"custom",
        "timestamp":"2013-04-12T23:20:56.523Z",
        "sessionID":"2d64d3ff-25c7-4b92-8e49-21884b3495ce",
        "properties":{
            "cartContent":"WIDGET",
            "cartPrice":"$50,000"
     }

     {
        "name":"sessionEnd",
        "type":"system",
        "timestamp":"2013-04-12T23:25:55.052Z",
        "sessionID":"2d64d3ff-25c7-4b92-8e49-21884b3495ce"
     }

]
```

Every JSON payload must begin with a context event. In the preceding example, this event is indicated by `"name":"context"` and includes properties that describe the current context of the mobile app, such as user name and the longitude and latitude. The context event is associated with each event that follows it, such as the session start and end events that demarcate a session. It is also associated with events raised in the mobile app code, such as `PurchaseFailed` in the preceding example.

> **Note:**
>
> Although you can add this context to events using straight REST calls, the mobile client SDK adds both session and device context information to the payload automatically.

Creating Your Own JSON Payload

If you don't use the mobile client SDK, keep these tips in mind when composing the JSON payload:

- Start each payload with a context event (indicated by `"name":"context"`).
- Add a context event whenever the device's context changes — typically when the `longitude`, `latitude`, or `username` properties need to change.
- You can randomly add the events within the payloads, but you must associate every event raised in the mobile app code with `sessionStart` and `sessionEnd` events just like `PurchaseFailed` in the preceding example, as noted by `“type”:”custom”`.

> **Note:**
>
> Ensure that these events share the same `sessionID` value. When events have the same `sessionID` value, theMCS server can approximate the session even if part of the payload (like the `endSession` definition) isn't recorded by the database.

MCS responds with a 202 status code (Accepted) when it receives a complete and syntactically correct REST call. Otherwise, it returns 400 (Bad Request) or 405 (Method not Allowed) responses.

Why Should I Use the Mobile Client SDK?

The mobile client SDK:

- Automatically defines the start and end of sessions and manages them using the UUIDs that it assigns to the `sessionID` property.
- Adds the context event at the beginning of each payload.
- Adds such device properties as the `username`, `latitude`, and `longitude` for context events.

> **Note:**
>
> On the server, the `longitude` and `latitude` values are translated into city, country, postal code, and street. See Integrating Analytics into a Mobile App Using the Mobile Client SDK.

- Marks events raised in mobile app code as `custom` (which is described in Tracking Sessions and Logging Events for Mobile Apps) or `system` for session or context events. The SDK also adds a `timeStamp` to each event.

Adding Location Properties to the `context` Event

The Oracle eLocation Service (maps.oracle.com) derives location from the `longitude` and `latitude` properties in the JSON request body. These properties only work if your mobile apps are used in countries where Oracle eLocation Service is available. For countries where Oracle eLocation Services is unavailable, you can still enable MCS to record the location data that allows countries to display in the Dashboard map by adding location-related properties to the `context` event.

To enable requests to support country data, add any combination for the following properties to the `context` event:

- `locality` — The mobile device's locality, such as city, township, or village.
- `region` — The mobile device's region, such as state, canton, or province.
- `postalCode` — The mobile device's postal code.
- `country` — The mobile device's GPS country. For some countries in the Asia-Pacific region, you can use a two-letter identifier, such as JP (Japan), CN (China), or KR (South Korea).

> **Note:**
>
> Do not include `longitude` and `latitude` in the `context` event if you define any of these properties.

For example:

```
{
     "name":"context",
         "type":"system",
         "timestamp":"2013-04-12T23:23:34.345Z",
         "properties":{
         "userName":"GDoe321",
         "locality":"Aomi",
         "region":"Kanto",
         "postalCode":"135-0064",
         "country":"JP",
         "timezone":"-14400",
         "carrier":"AT&T",
         "model":"iPhone5,1",
         "manufacturer":"Apple",
         "osName":"iPhone OS",
         "osVersion":"7.1",
         "osBuild":"13E28"
         }
```

Integrating Analytics into a Mobile App Using the Mobile Client SDK

The `oracle-cloud-mobile-analytics.jar` and the `libOMCAnalytics.a` libraries included in the mobile client SDK enable mobile apps deployed on Android and iOS devices to post events to the Analytics API. These libraries become available to your mobile app when you download the mobile client SDK and integrate it into the mobile app. See Connecting Your Application to a Mobile Backend.

Understanding Different Types of Analytics Reports

The Analytics reports plot the frequency of incoming events against specified time periods. These reports enable you to spot patterns in mobile app usage and performance.

Reports	Uses
Events	Find out how users use your app. You select which events you want to track in the mobile app code. You can add additional events by calling the Analytics API from custom code. See Tracking Sessions and Logging Events for Mobile Apps. You can build conversion funnels that let you trace user participation through a workflow path. See Improving User Retention with Funnel Analysis.
API Calls Count API Calls Response Time	Track app usage and performance, as well as how apps use MCS .
New Users Active Users Session Count Session Duration	Enables you to answer such questions about engagement as: • Is the app gaining or losing users? • How often do users use the app and how long are the user sessions?
My Reports	Stores your saved report definitions.

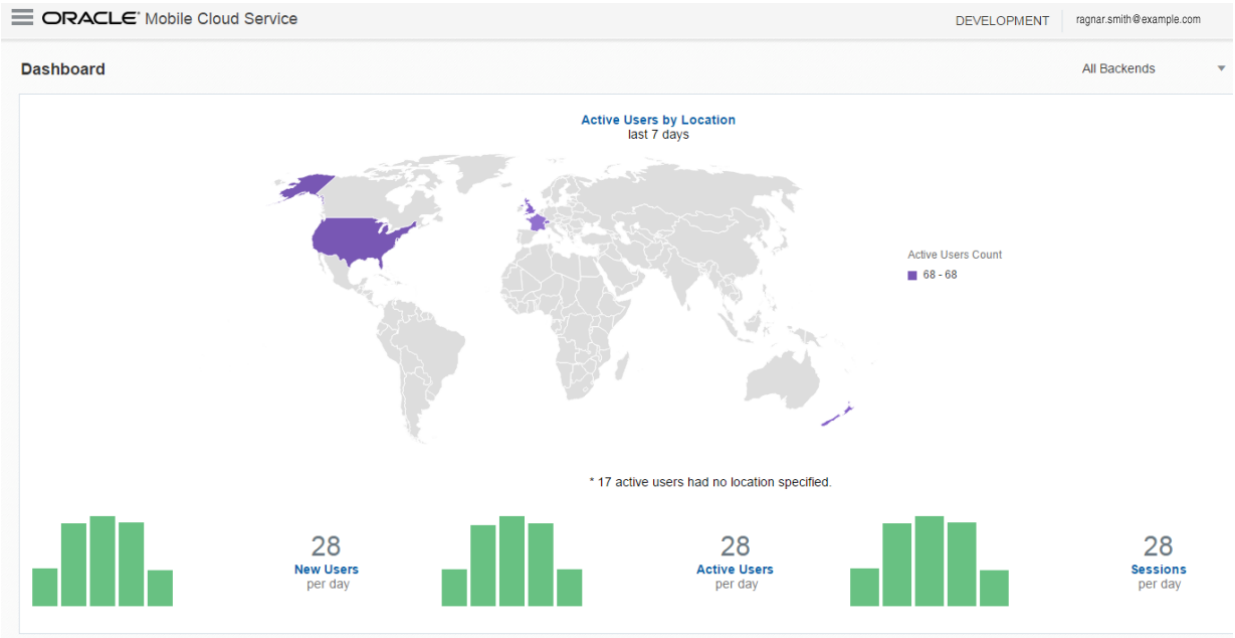
Accessing the Analytics Reports

Click  to open the side menu and then to open the dashboard. From here, you can quickly explore the following:

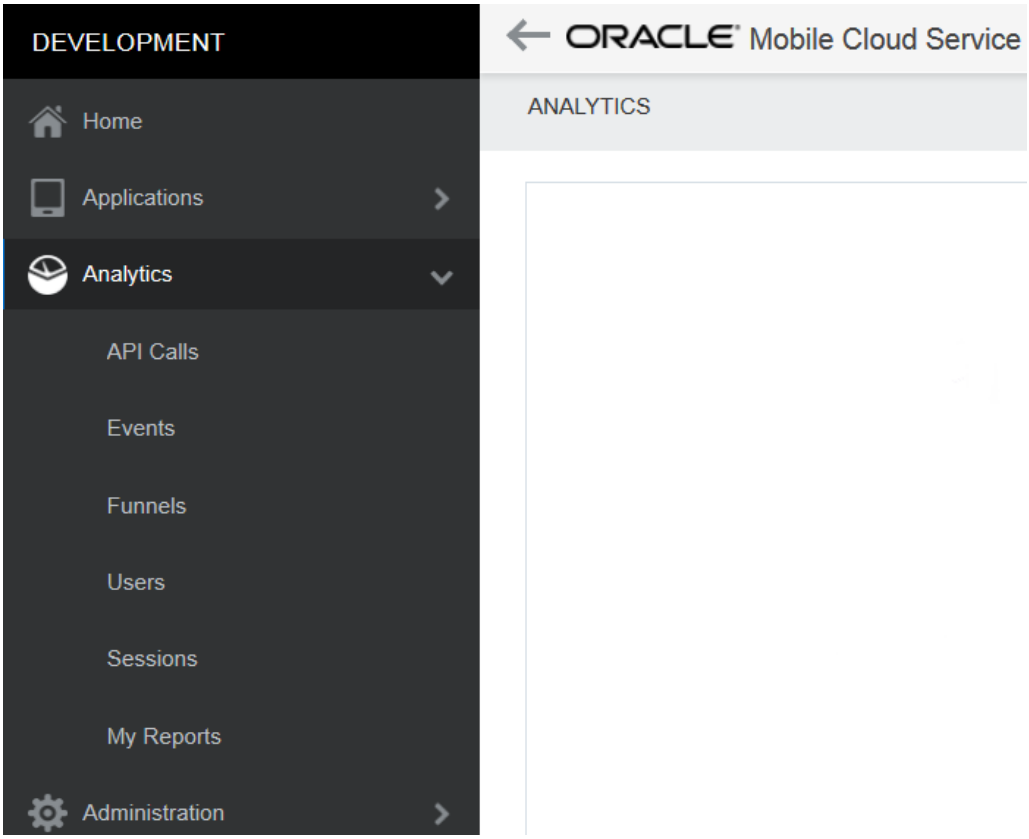
- API Calls: See the number of calls to your app and the response times over various time periods.
- Events: Track customer events, like putting an item into a cart and checking out.
- Funnels: Discover what events customers complete on their way to a goal, like making a purchase or signing up for promos.
- Users: Find out who your new and active users are, and group them by properties such as their zip code or country.
- Sessions: Learn how often and how long users are in your app.
- My Reports: Save analytics data insights as a report so you can return to it later.

While MCS displays data gathered from all of the mobile backends by default, you can use the menus to isolate the activity for all versions of a selected mobile backend or API.

> **Note:**
>
> Unless you have mobile cloud administrator privileges, you can see only the analytics reports generated in environments to which you have been granted access. For example, if you have access to only the Development environment, then you can't see the usage and traffic data for mobile backends in the Production environment.



The dashboard summarizes the user base and activity on a per-day basis. Click the bar charts to access more detailed reports that help you draw conclusions about the API traffic or app adoption rate. Accessing these reports from the menu lets you also view detailed data on app use.



Tip:

For each report type (except funnels), you can view data plotted as a line graph or as a grouped bar chart by toggling the display options at the bottom of the page.

API Calls Reports

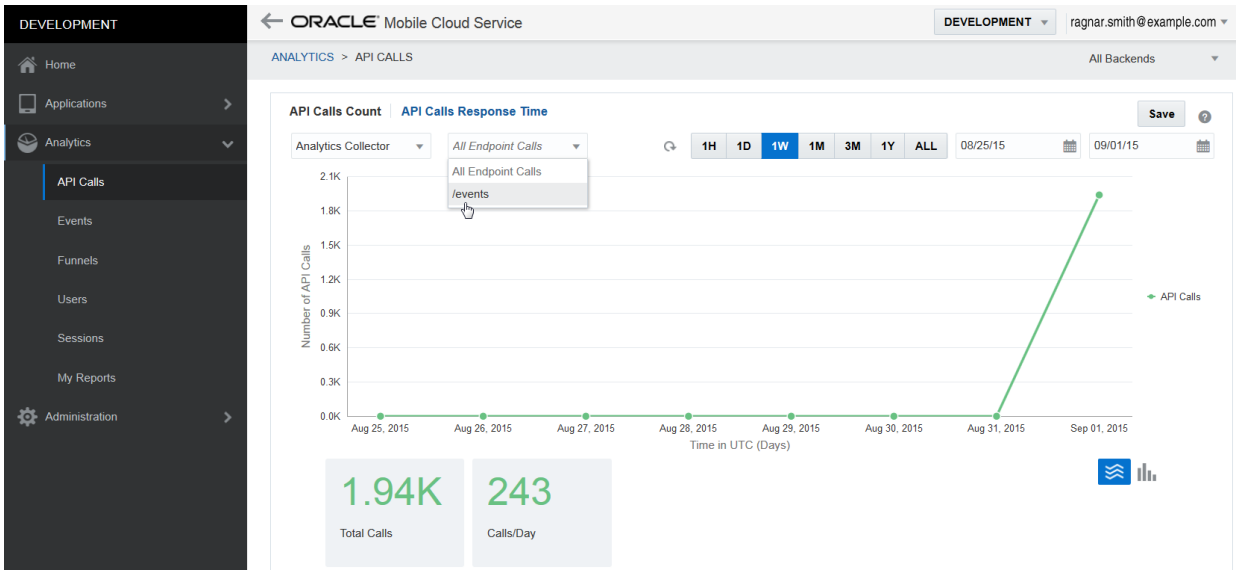
Click the API Calls menu to access the API endpoint reports. The API endpoint reports track all of the APIs, or a particular API, over a specified period of time. The volume of received calls indicates how much an app utilizes the MCS platform (and consequently, how widely used the app may be). Selecting a time frame allows you to see when traffic picks up or drops off. You can also drill down to specific endpoints to see how frequently they have been called and the corresponding response times.

Note:

MCS automatically generates analytics data whenever an API is called. This data accumulates over time and can use up all the available database space, which can severely affect service. To mitigate the issue, have your mobile cloud administrator modify the `Analytics_ApiCallEventCollectionEnabled` policy in the `policies.properties` in your environment. Setting this policy to `false` turns off automatic event generation. To avoid losing service, set this policy to `false` before your storage capacity reaches full.

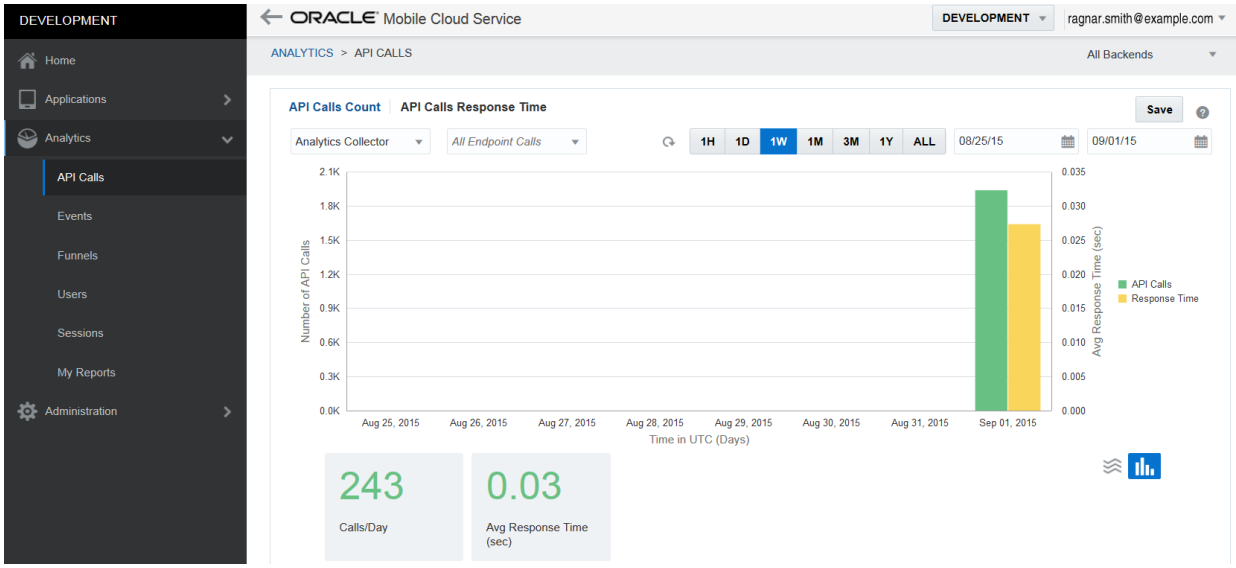
API Calls Count

The reports for API Calls Count let you view the traffic for one, or many, APIs for a selected period of time. The report includes both successful and failed calls.



API Calls Response Time

MCS measures the response time (in milliseconds) for an API call as starting when the server receives the request and ending when the call returns the data to the mobile app. The response time includes the time dispatching the call. You can compare the response time for one (or all), APIs for a selected period of time. The bar graph compares the response time against the number of calls.

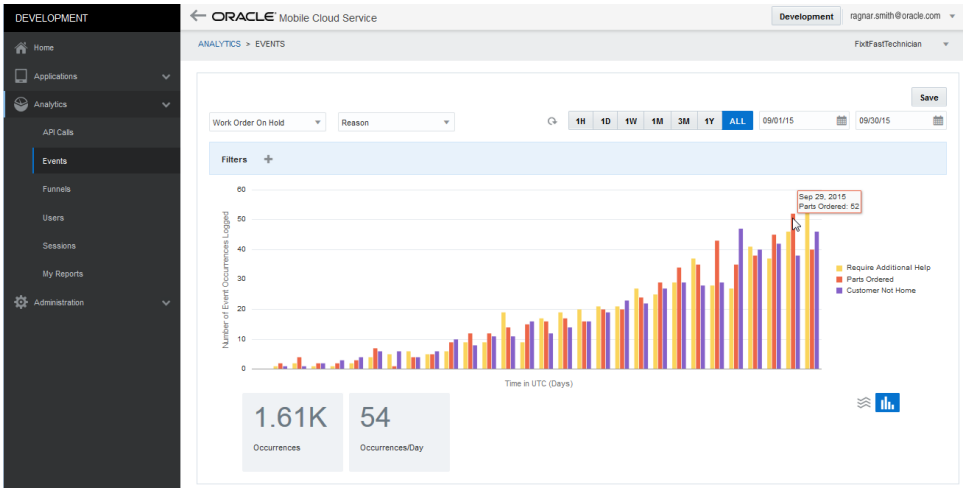


Events Report

The Events report lets you focus on how to improve the mobile app user experience and how to explore business opportunities. For example, the Events report can show you not only how frequently users use an app's search function, but how frequently users perform searches on specific devices and operating systems. By filtering an

event with the device and operating system properties, you can see when usage of the app on a specific platform or device has outpaced usage on another. Declining usage across an entire platform may indicate that the application requires optimization for that platform.

Events reports aren't limited to assessments of how a mobile app performs. You can also use them to spot inefficiencies in your company's processes, such as its supply chain orchestration. To get an idea, say you are part of a mobile project team for an appliance repair company. Its repair technicians use mobile apps to accept or reject the work orders dispatched to them. They can also update the status of the work order from open to closed, or from open to on hold. You need to investigate why orders are left pending, or closed by frustrated customers. To do this, you can draw some conclusions by viewing an event and filtering it by its properties. Because your mobile app developer raised events in the app's code to reflect the workflow outlined by your work order processing use case, MCS can graph the occurrence of work orders on hold. From this report, you can also see the reasons that prevent the fulfillment of an order from the properties defined for this event, such as *Parts Ordered*.



To gain business intelligence, you can filter a report using the properties specific to an event to discover user behavior and trends. For example, filters let you find out which products customers search for most often. By cross-referencing this against the location-specific reports for a mobile app, you can target your workforce, training, or marketing efforts accordingly.

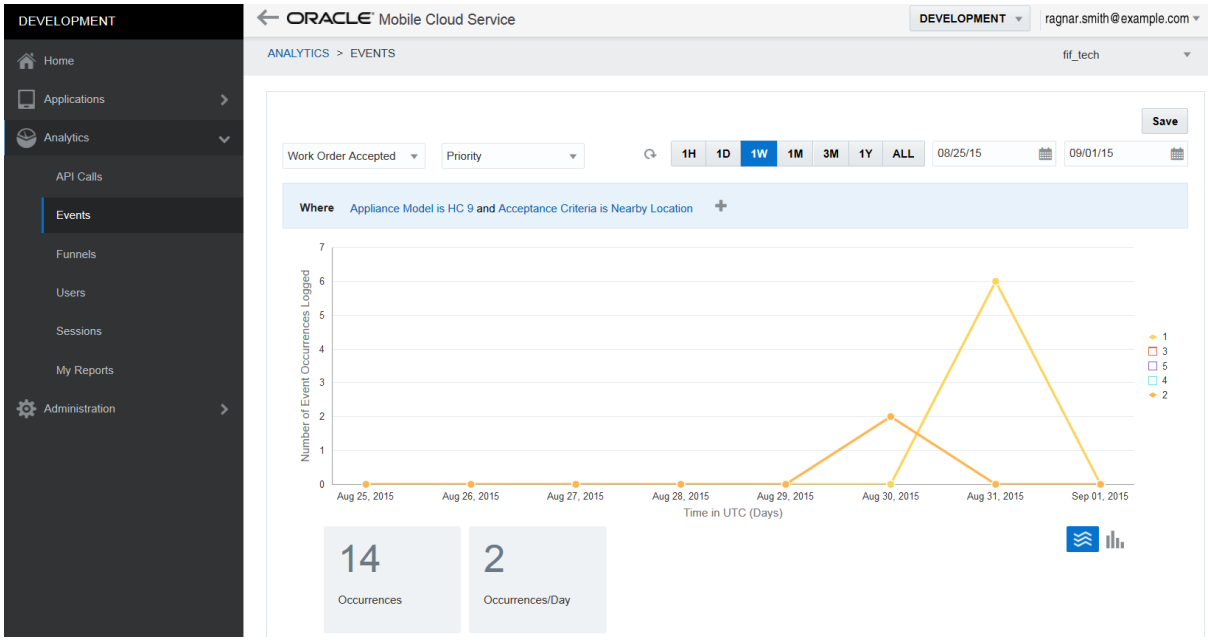
Events

MCS plots the events against time. You can select from among events and then segment your search reports by creating filters. After you create a filter, click **Done**.

MCS applies the event name and the time-stamp properties as the default properties for the Events report. You can filter events by the following properties, which MCS applies automatically to events that are raised in the mobile app code:

- Device-related properties
  - Operating system
  - Operating system name

  - Operating system version
  - Operating system build
  - Device model
  - Device manufacturer
  - Carrier
- Location Properties:
  - Country
  - State
  - City
  - Postal code



User and Session Reports

The user and session reports not only show you how many customers use a mobile app, but also how long they use it. Are applications gaining users? Does the time users spend using the app reflect its purpose?

When users authenticate, MCS gathers the events and plots the data points for these reports that you use to spot trends both over time and by location. You can group the data in these reports using the following properties:

- Client application
- Country
- State
- City

- Postal code

> **Note:**
>
> Analytics creates reports for users only when sessions are used in the mobile app code.

User Reports

If you have a number of mobile apps deployed in the field, you can use the New User and Active User reports to find out which ones are gaining traction with new users, which are sustaining their user base, and which are losing users.

Why User Counts Can Vary

MCS approximates user counts through user IDs and device IDs.

For the events sent from mobile apps, MCS identifies a user through a user ID (a property provided by the OAuth token) or the mobile client SDK's Device ID header (`Oracle-Mobile-DEVICE-ID`) when the user ID is not known (for example, for mobile apps that do not require authentication). Although the Device ID reflects a user (not the device manufacturer) it isn't always interchangeable with the user ID: a single user might access the same app using different devices (that is, a single user logging in from two devices will be counted as two users). Because MCS uses the Device ID in the absence of a user ID, the user counts are an approximation for mobile apps that allow both authenticated and unauthenticated (that is, anonymous) users.

User Session Reports

The Session Count and Session Duration reports describe user engagement.

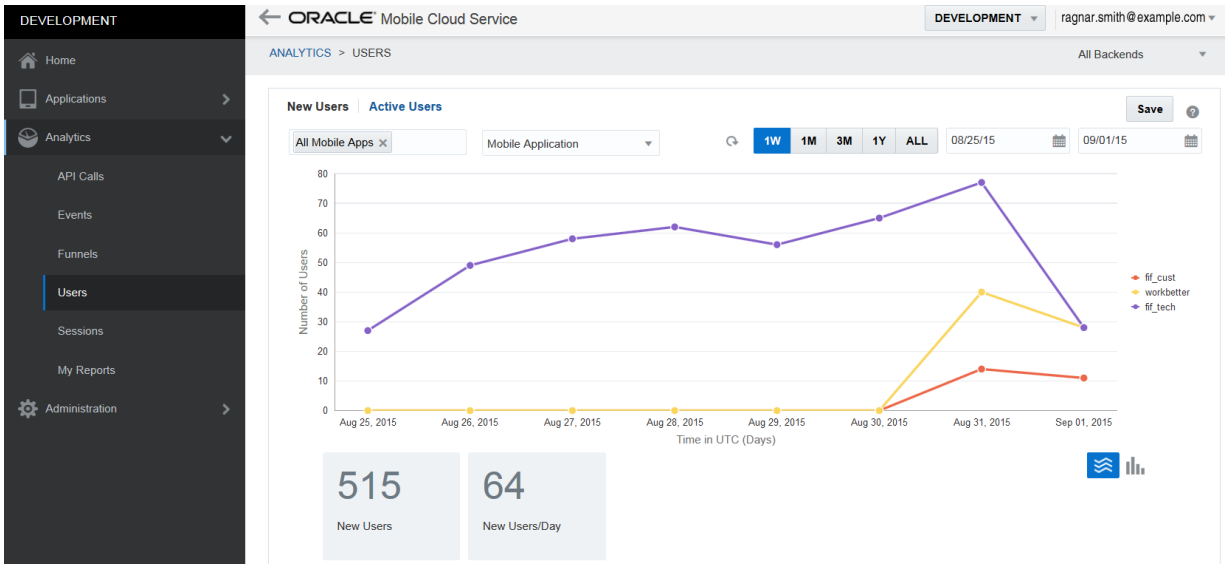
These reports reveal the time users spend on a mobile app, not only in terms of the number of sessions, but also how much time users spend on the app. Although a session may be seen as starting when a user brings an app to the front on the device's springboard and ending when it's sent to the back, the concept of session may differ in terms of platform and implementation, as described in Defining Sessions.

The user session reports let you assess if the app elicits the appropriate level of user interaction. In other words, are user sessions intended to be short, as they are for apps giving time and weather updates, or long, as they would be for shopping apps?

New Users

The New Users report lets you see the number of users (authenticated or anonymous) for any (or all) mobile apps over a selected period of time.

When MCS receives an event from a previously unknown user (or from a device if the user is anonymous), it notes the existence of a new user. Keep in mind that the New Users report may not reflect the exact number of users if it includes both mobile apps that require user authentication along with ones that allow users to access services anonymously. See Why User Counts Can Vary.

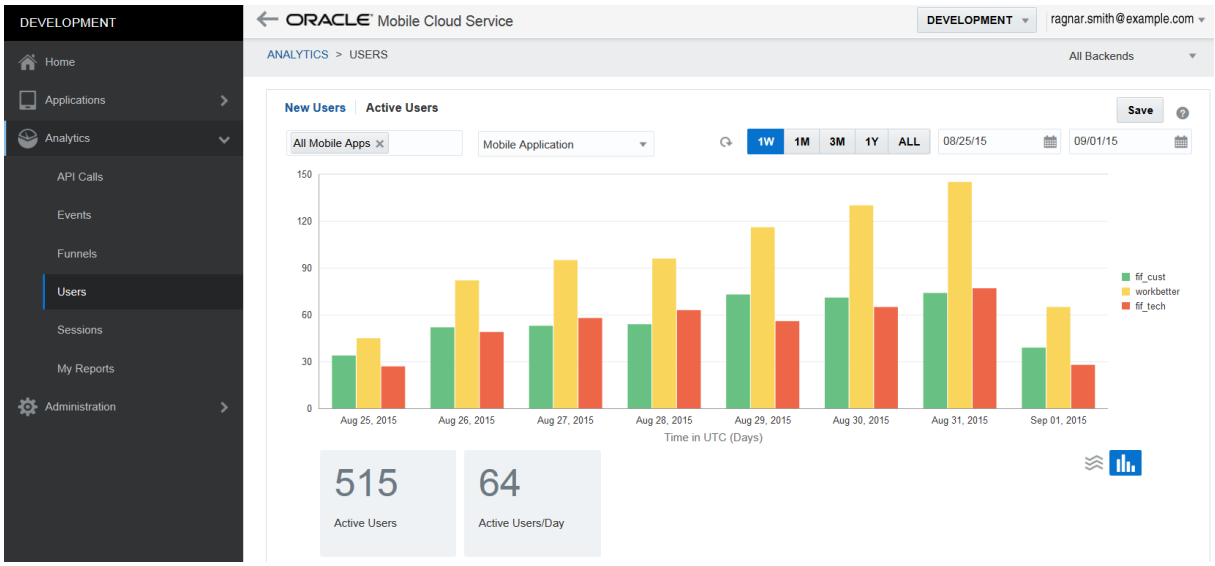


Active Users

MCS considers a user as active if it has previously received event data from the user or the device.

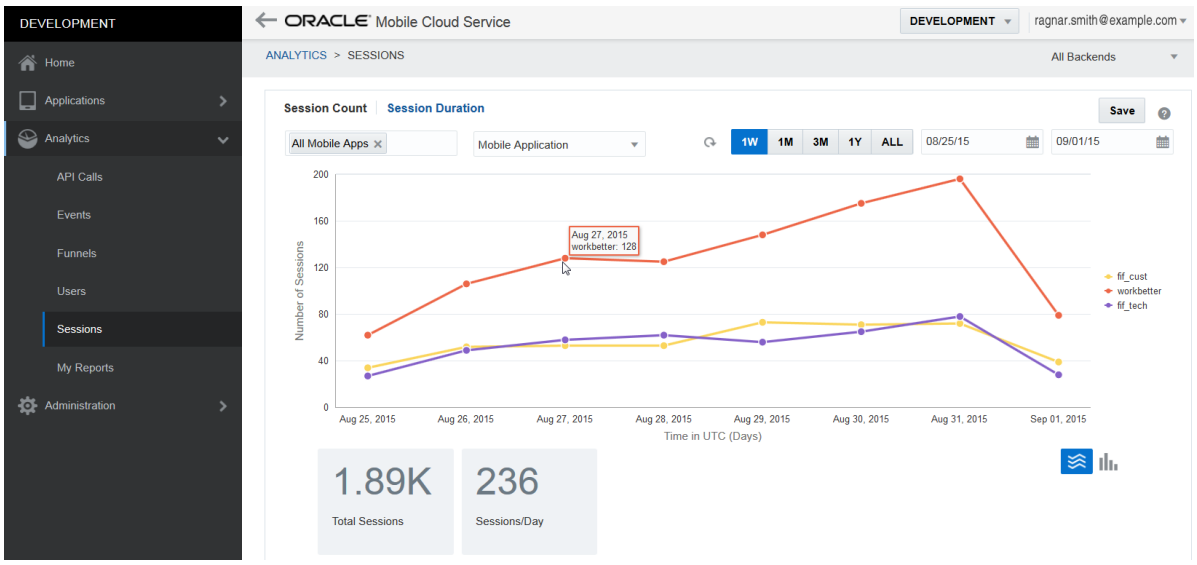
The Active Users report lets you see the number of active users for any (or all) mobile apps over a selected period of time. To find out the usage rate for a mobile app, you first select it from the dropdown list and then select the **Mobile Application** property. This property lets you compare the usage of two or more mobile apps. For reports that include mobile apps which require authentication along with those that don't, the number of actual users may not be accurate. See Why User Counts Can Vary.

> **Tip:**
>
> You can use the **Mobile Application** property to compare the adoption rate for different versions of a mobile app.



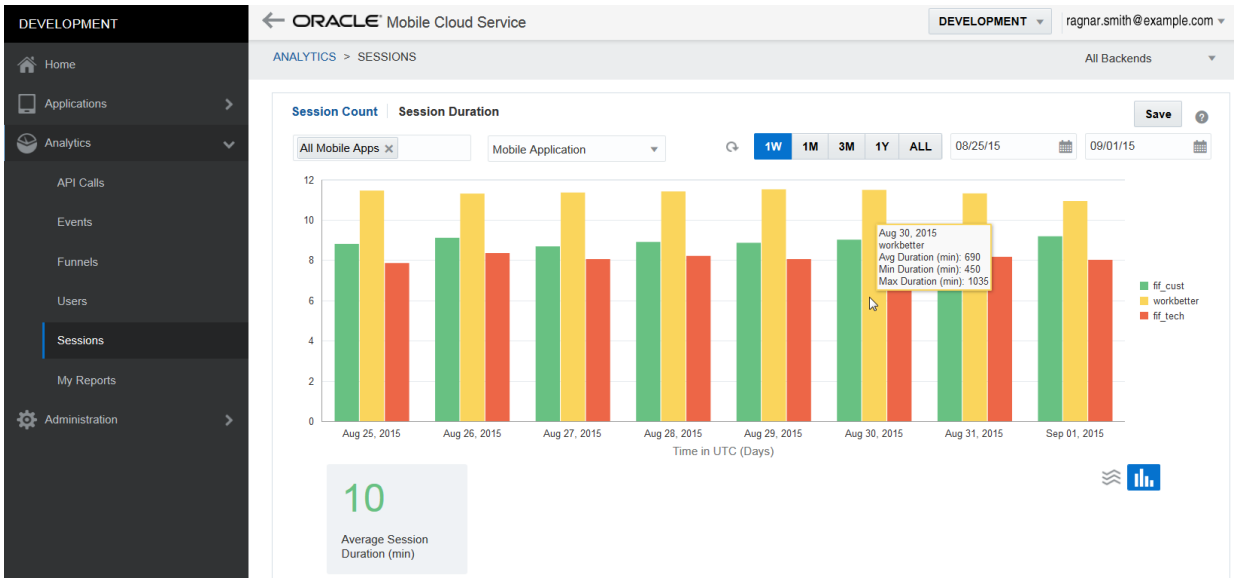
Session Count

The Session Count report lets you see how many times a mobile app has been used over a selected period of time and location.



Session Duration

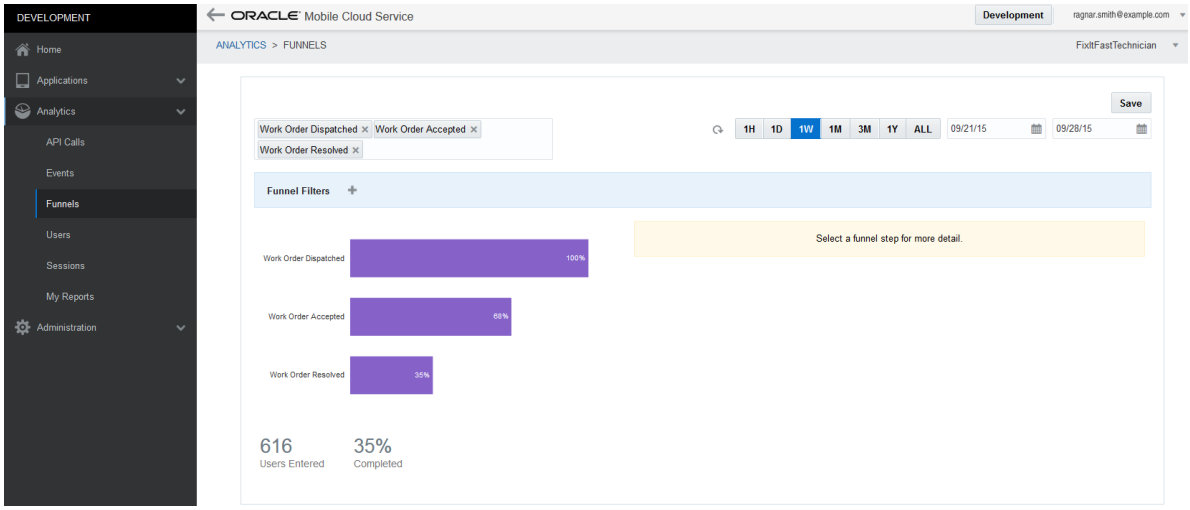
The Session Duration report lets you see the minimum, maximum, and average session times for one, or all, mobile apps over a selected period of time and location.



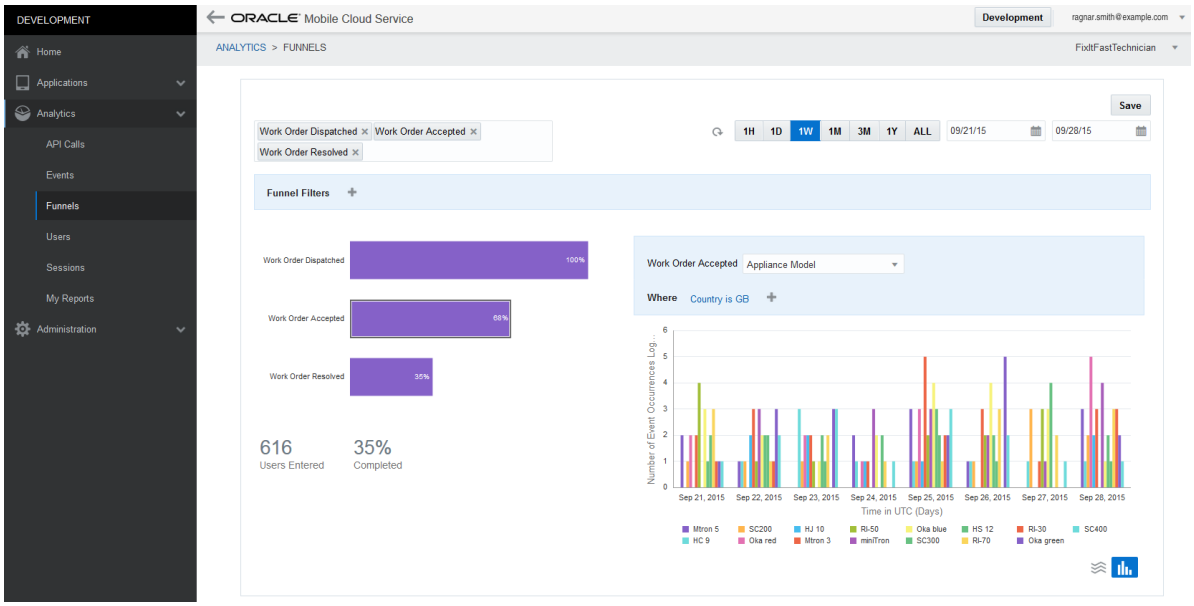
Improving User Retention with Funnel Analysis

Conversion funnels let you compare how many users start a workflow (say, a checkout process, a user registration process, or a lead generation) against how many actually complete it. A funnel segments a workflow into a sequence of steps designed to guide users to some goal (or conversion). Typically, users drop off at each step of a workflow; many may begin a checkout process, for example, but comparatively few complete it. Funnel analytics show you the conversion rate for a workflow by showing the number of users who drop off at various points.

> **Note:**
>
> For funnel analysis to be meaningful, you need to think about how events can be assembled into work flows early in the development process. Defining the appropriate events allows the right data to be collected.

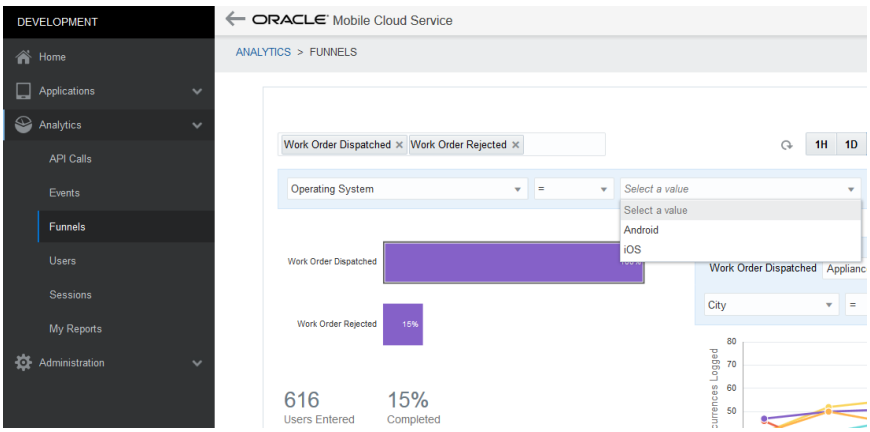


Because funnels show you where users lose interest, you can use them to improve a process or identify bugs in a workflow. Further, because you construct funnels from the events raised for a mobile app, you can see where and why users are dropping off by analyzing the event properties.



Creating a Funnel

To create a funnel, you first select a mobile app that has been released long enough so that a meaningful amount of event data can be collected. After you select the app, the events defined for it become available so that you can build a funnel from them.

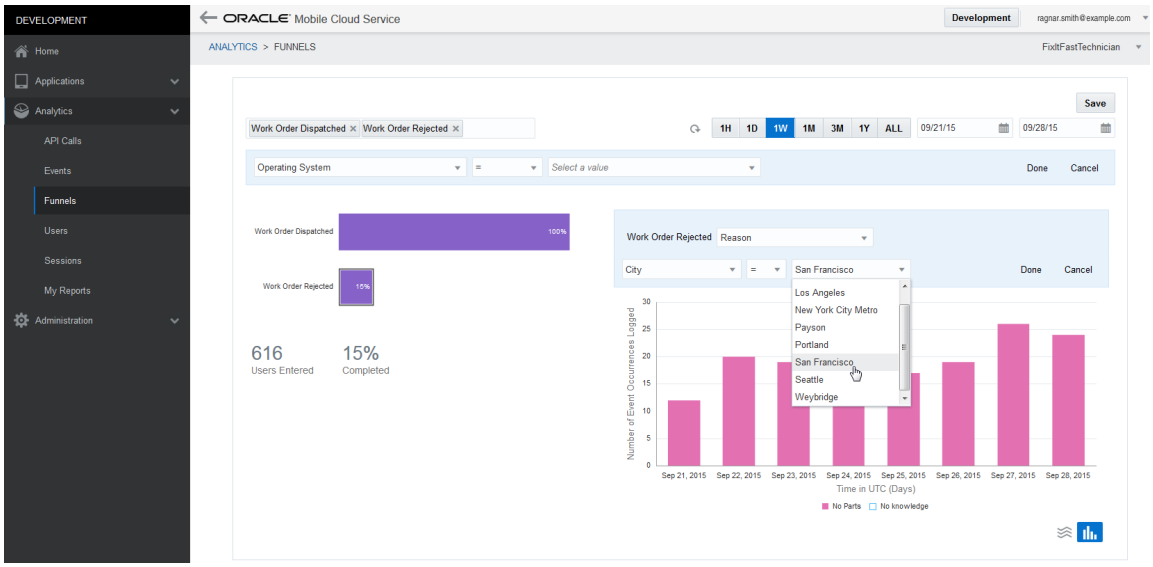


1. Select a date range.
2. Add the events in sequential order to form the funnel steps.

> **Note:**
>
> MCS automatically displays the conversion rate for each event that you select. By selecting these events, you can view their properties. Use the filter and group by functions to analyze these properties.

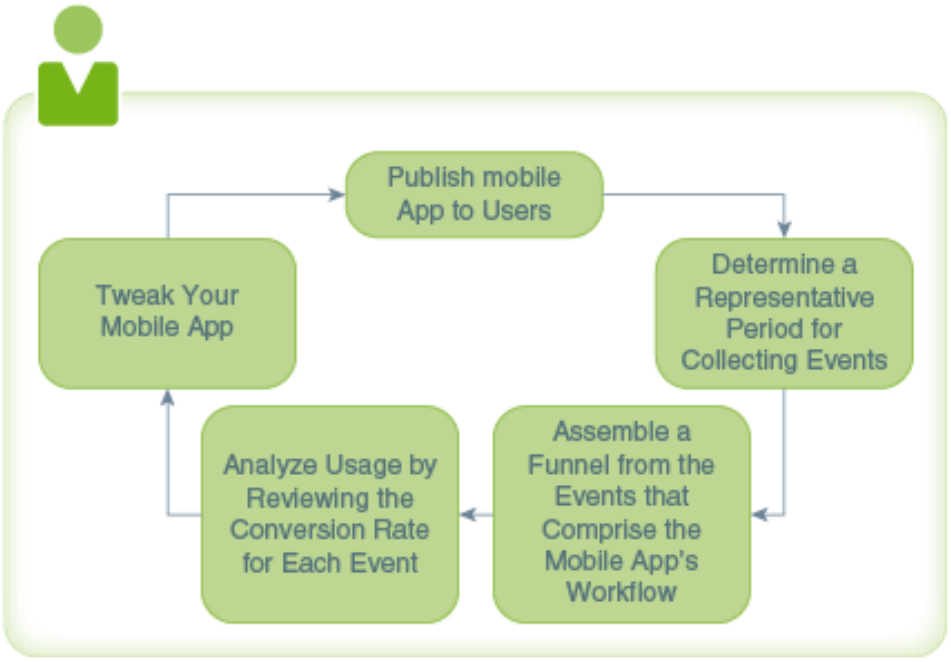
Analyzing Funnels

After you've selected all of events for the process, take a look at the conversion rate. You can select an event and then drill down on the property.



> **Tip:**
>
> If the conversion rates indicate a large decrease, select more events to find out why.
> For example, for a user registration workflow, select a *signup failed* event and then select the detail view for the event. Use the filter and group by options in the detailed report. You can group by system and custom properties. For example, grouping the data by the property, *reason*, lets you to see event data sorted by the attributes defined for this property, *Duplicate User ID* and *Incomplete Data*.

You can take an iterative approach to refining your funnel. For example, deploy your app long enough to collect a significant amount of data and then tweak the app accordingly. Redeploy for a second round of adjustments and then select another date range.



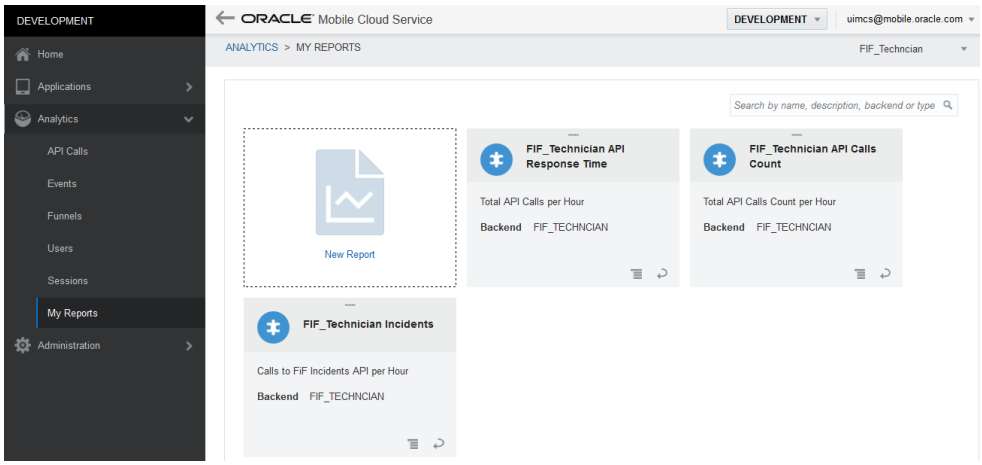
Creating Custom Analytics Reports

As a mobile program manager, you can keep an eye on the usage and health of your mobile app on an ongoing basis by creating a suite of custom reports that you can run whenever you want. MCS .

Say that you've launched the Fix-It-Fast (FiF) app in three cities. For each locale, you also want to find out daily peak usage times and also segment the user data by age group. To do this, you'd create a set of reports for the FiF app that include a New Users report, an Active Users report as well as a daily Session Duration report and an API calls report. MCS enables you to keep these reports on hand, organize them, update them, or delete reports that you no longer need. And you can create new reports as needed.

MCS organizes your custom reports into My Reports. To open My Reports:

1. Select the environment for your reports and then click ☰ to open the side menu.
2. Click **Analytics** to open the reports drawer.
3. Select **My Reports**.



The My Reports page lists all of the reports that you've created for a particular environment. That is, this page shows only the reports that you've created, not those created by someone else.

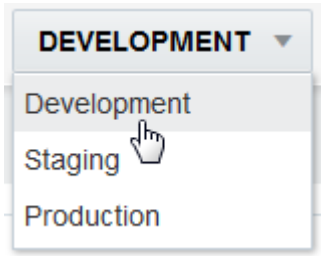
> **Note:**
>
> Not only do your reports belong only to you, but they also belong to the environment in which you created them. You can't share a report across environments. Instead, you have to replicate a report for each environment.

How Do I Create a Custom Analytics Report?

You can save your custom report definitions while you're looking at an analytics page, or from the My Reports page.

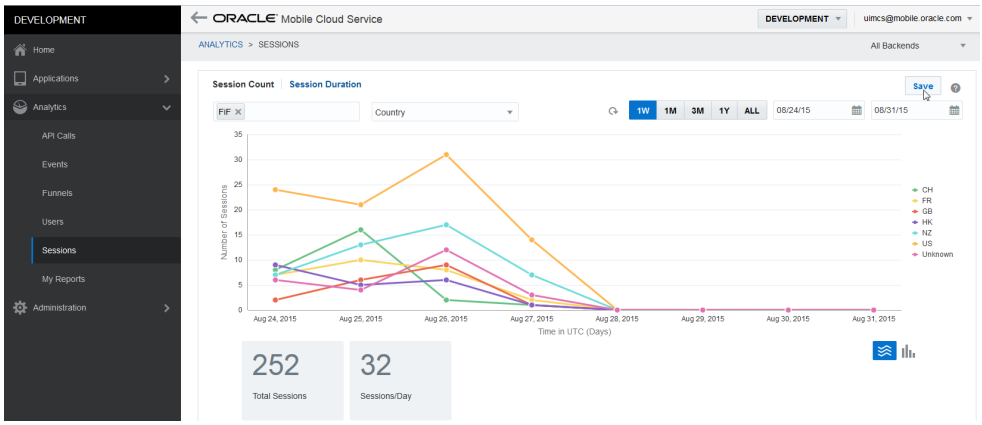
To create a report directly from a report page:

1. Select the environment.

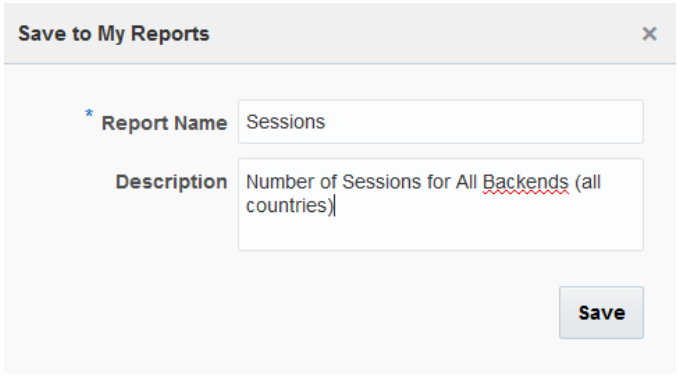


> **Note:**
>
> You need to create separate sets of reports for each environment.

2. If needed, open Reports by clicking ☰ and then **Analytics**.
3. Choose the report type and apply any filters you need. Click **Save**.



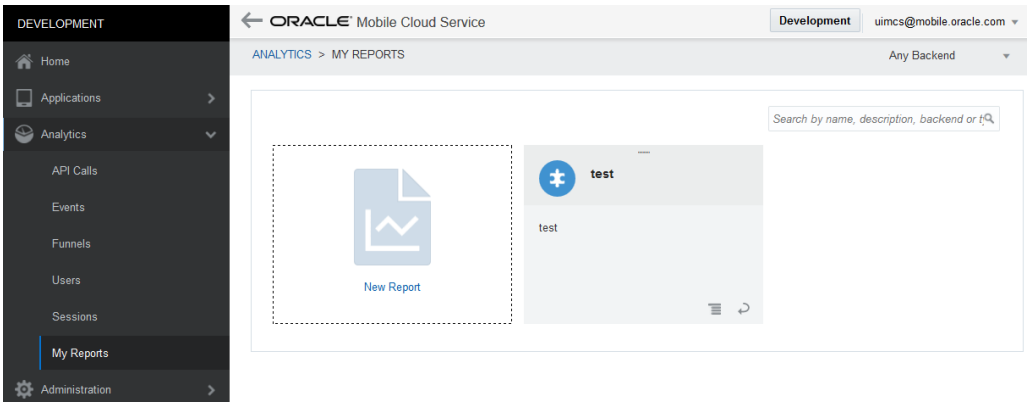
4. Complete the Save to My Reports dialog by entering a name and optionally, a description. Click **Save**.



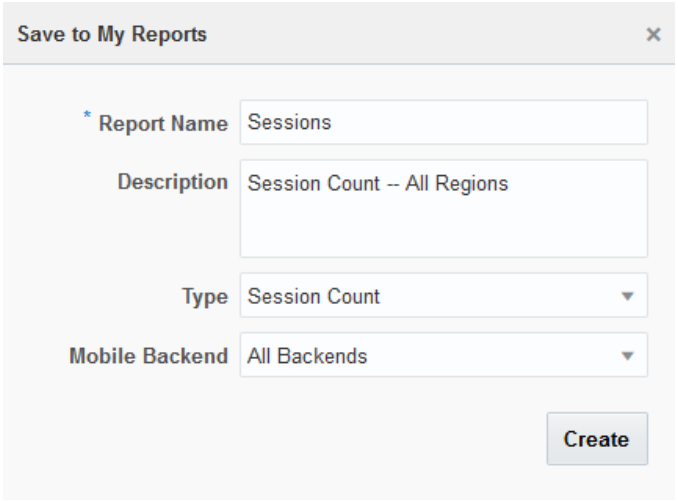
> **Note:**
>
> Keep in mind that you're saving everything in page; not just the filter criteria, but also the environment (Development, Staging, Production) and also the chart style (line or bar).

You can run, edit, or delete the report from the My Reports page.

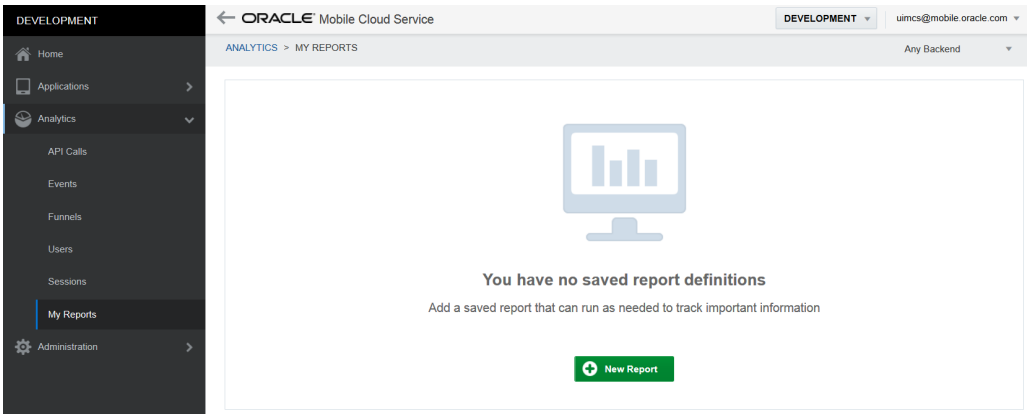
You can also create a report from the My Reports page. Click **New Report** and then complete the dialog.



Complete the dialog by giving the report definition a name and an optional description. You also need to choose the type of report and the mobile backend (either **All Backends** or a specific mobile backend).



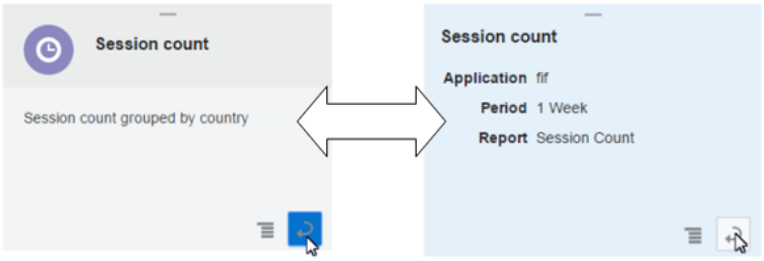
If you have no report definitions saved in My Reports, you can use this same dialog to create one. To access this dialog, click **New Report**.



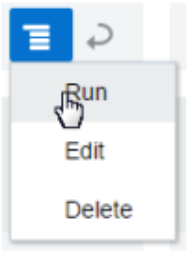
My Reports

The My Reports page displays all the report definitions (reports) that you've created for a specific environment. From this page, you can organize your reports, run them, update them, and delete them.

MCS creates a tile for each report definition. The front of each tile lists the information that you provided when you created the report definition. Clicking  flips the tile over to reveal some additional information provided by MCS , such as the name of the mobile app, the reporting period, and the type of report. The information on the back of the tile varies depending on the type of report that you've defined. For example, an event report includes not only the mobile app name and the reporting period, but the selected endpoint as well.

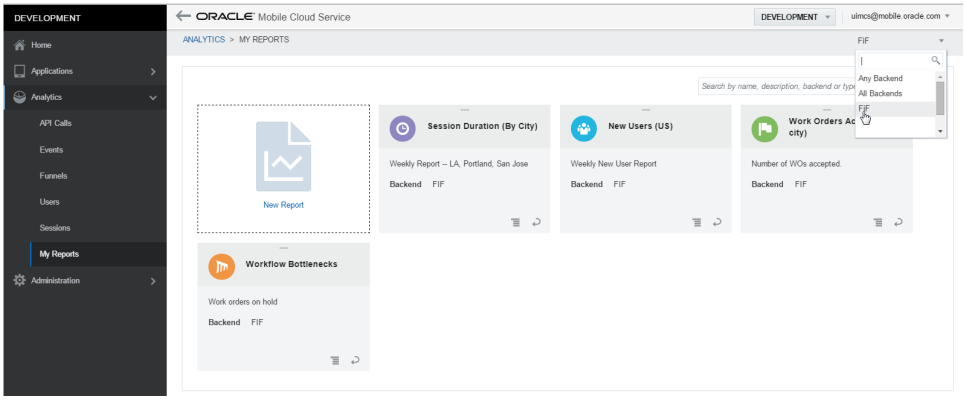


Each tile has a menu which enables you to run, edit, or delete a report.



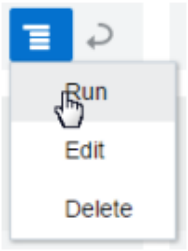
You can rearrange the reports to suit your needs by dragging and dropping the tiles. For example, if you have a report you want to run first thing every morning, you can drag it to the first position (right next to the New Report tile). You can also adjust the display using these options:

- **Any Backend**—Displays all of the reports that you've created (including those created using the **All Backends** filter).
- **All Backends**—Narrows the display to only the reports created using the **All Backends** filter.
- By mobile backend—Displays only the reports created for a specific mobile backend.

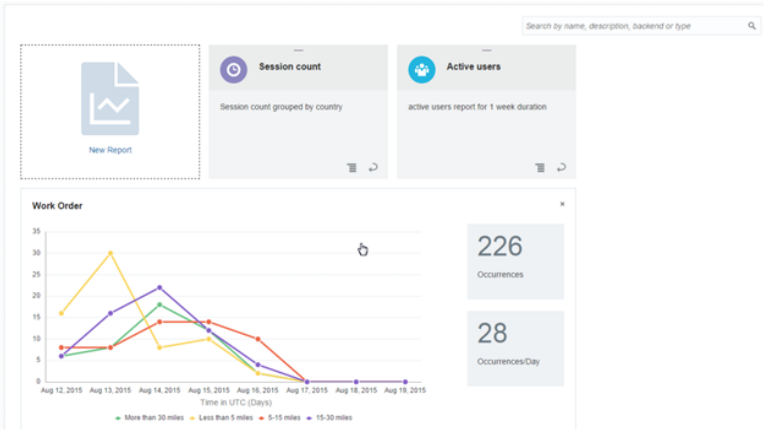


How Do I Run a Custom Report?

To run a report, first click  and then click **Run**.

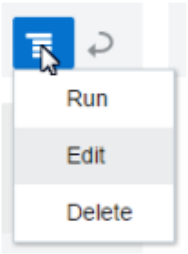


The report opens on the My Reports page. Depending on the number of tiles in the page, you may need to scroll down to see it.

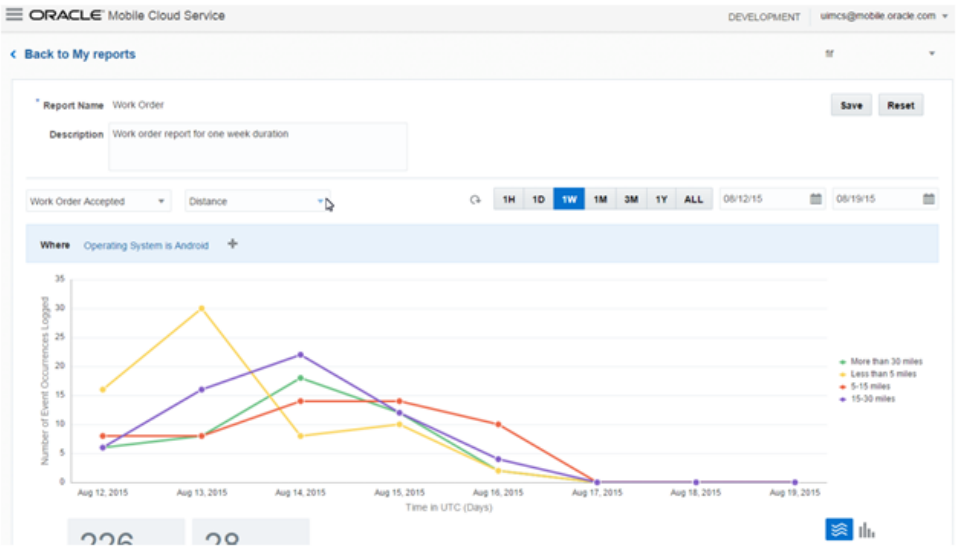


How Do I Edit a Custom Report?

If you need to change something about your report, first click  and then click **Edit**.



You can’t change the report type, or the environment, but you can change the report name, description, reporting period and any filtering criteria. When you’ve made your changes, click **Save**. Click **Reset** to revert the report definition to its original state.



You can’t change the report type, but you can change the report name, description, reporting period and any filtering criteria. When you’ve made your changes, click **Save**. Click **Reset** to revert the report definition to its original state.

Tracking Sessions and Logging Events for Mobile Apps

The analytics libraries of the mobile client SDK enable the monitoring and measuring of any event that has been defined for the mobile app.

Knowing which item the user added to a shopping cart is better than just knowing that a user put some unnamed item in the cart. Likewise, you'd want to know which products users search for rather than just knowing that they've performed a search. To add this level of detail to your analysis, you can create events. You can further segment reports by adding properties that describe these events in terms of something that characterizes the event itself or an activity related to the event. Here’s an example of what an event looks like in JSON form (which is the payload format for the underlying REST calls that send event data to the service).

```
[
  ...
    {
```

```
        "name":"PurchaseFailed",
        "type":"custom",
        "timestamp":"2013-04-12T23:20:56.523Z",
        "sessionID":"2d64d3ff-25c7-4b92-8e49-21884b3495ce",
        "properties":{
            "cartContent":"WIDGET",
            "cartPrice":"$50,000"
        }
    ...

]
```

The mobile client SDK provides you with a shorter route to generating analytics reports than does writing a straight REST call. After you've linked the mobile app to the platform-specific SDK, you can enable analytics reporting for your app by adding code that calls the analytics library and designates the beginning of the session before flushing the events to the server.

Creating Events and Sessions Using the iOS Library

The `libOMCAnalytics.a` library includes classes for logging the events and sessions from an iOS mobile app. You can also use the classes from this library to track successful and failed posts.

Sessions provide a means of grouping the events raised in the mobile app code, as the events logged between the start and end of a particular session belong to that session. As discussed further in Designating Sessions, you can call instance methods for starting and ending sessions as well as logging events to the service's server.

Calling the Analytics Service

To call the Analytics service, import the `OMCMobileBackend+OMC_Analytics.h` and the `OMCAnalytics.h` header files in addition to the ones noted in Calling APIs Using the iOS SDK.

Designating Sessions

You can designate sessions by calling the `startSession` and `endSession` methods on the `OMCAnalytics` object. The `endSession` method automatically flushes events, but you can flush events explicitly by calling the `flush` method, which will post the currently outstanding events to the service's server.

> **Tip:**
>
> Use the `flush` method if you have a large number of events that are logged between the start and end of a session.

The signatures for the `startSession` and `endSession` methods are as follows:

```
@interface OMCAnalytics : OMCServiceProxy

/** The Analytics service's delegate. */
```

```
@property (nonatomic, weak) id<OMCAnalyticsDelegate> delegate;

/**
 Starts an Analytics service session. If a session is already in progress,
then it continues. A new session is not created.
  */
[analytics startSession];

/**
 Stops the current Analytics service session. Does nothing if a session is
not in progress.
 */
 [analytics endSession];
/**
 Logs an Analytics service event with the specified name. If a session is
not in progress,
it starts a session. Copies the specified name before returning.
 @param name the event name
 */
- (void)logEvent:(NSString*)name;

/**
 Logs an Analytics service event with the specified name and properties.
 If a session is not in progress, then it starts a session.
 Copies the specified name and properties before returning.
 @param name the event name
 @param properties a dictionary of arbitrarily named properties for the
event
 */
- (void)logEvent:(NSString*)name properties:(NSDictionary*)properties;

/**
 Uploads all the posted events to the OMC Mobile Analytics REST service.
 */
- (void)flush;

@end
```

> **Note:**
>
> The `startSession` begins with the first logged event (even if no session has been started). The `endSession` method flushes events to the service's server.

Associating a Session With Your Mobile App Being in the Foreground

A user session might correspond to the length of time that a user spends on the mobile app when it runs in the foreground. To associate Analytics sessions when your mobile app is running in the foreground, your app delegate should subclass `OMCAnalyticsApplicationDelegate`. Doing this will automatically log the start of a session when the mobile app moves the foreground and log the end of the session

when the mobile app moves to the background. In general, you don't need to call either `startSession` or `endSession`, as these are added automatically.

Adding Custom Properties to Events

You can describe an event more fully by adding one or more custom properties as key-value pairs.

> **Note:**
>
> Both the key and the value must be strings.

You can add custom properties to an event by calling the `logEvent:properties:` method and by passing a dictionary of property key-value pairs. For example:

```
[Analytics logEvent:@"Event name"];
properties:@{"customProp1":"value1", @"customProp2":@"value2"}];
```

> **Caution:**
>
> The following custom property names are reserved and can't be used for your event property names.
>
> - `Carrier`
> - `Count`
> - `Country`
> - `Day`
> - `Hour`
> - `Locality`
> - `Manufacturer`
> - `Minutes`
> - `MobileAppKey`
> - `Model`
> - `Month`
> - `OS`
> - `OSVersion`
> - `OSBuild`
> - `PostalCode`
> - `Region`
> - `Week`
> - `Year`

Receiving the Status of Event Posts

As an optional feature, you can implement the `OMCAnalyticsDelegate` protocol to receive notifications when the `OMCAnalytics` object posts events to the Analytics REST service successfully or encounters errors. To do this, you must register a delegate, an object that implements `OMCAnalyticsDelegate` with the `OMCAnalytics` object. For example:

```
OMCAnalytics* analytics = [[OMCMobileBackendManager
sharedManager].defaultMobileBackend analytics];

analytics.delegate = myDelegate;
```

You can implement one or both of the following instance methods to receive event status:

- Notifies the delegate that the `OMCAnalytics` object successfully posted events to the Analytics REST service.

  ```
  (void)analytics:(OMCAnalytics*)analytics
    didPostEvents:(NSURLRequest*)request
         response:(NSHTTPURLResponse*)response
     responseData:(NSData*)responseData;
  ```

- Notifies the delegate that the `OMCAnalytics` object encountered a specified error.

  ```
   (void)analytics:(OMCAnalytics*)analytics
  didFailWithError:(NSError*)error
  ```

Creating Events and Sessions Using the Android Library

The `oracle-cloud-mobile-analytics.jar` library includes the `Analytics` and `Event` classes that enable mobile apps to post events.

To enable the MCS to record your mobile app's event data, this JAR must be placed in the `libs` directory of your project.

The `Analytics` class is a singleton client object that exposes the Analytics API and has the `startSession`, `endSession`, `logEvent`, and `flush` methods. To start and end the sessions, which group events, and upload the events to the server, call these methods in your mobile app's code.

Method	Description
`startSession (Context context)`	Creates a new session.
`logEvent (String name)`	Adds a new event. The `logEvent` method starts a session automatically if one doesn't already exist.
`logEvent (Event Event)`	Adds an existing event.
`endSession (Context context)`	Ends the current session.

Method	Description
`flush`	Uploads events to the MCS 's server. Calling the `endSession` or `flush` methods uploads all of the buffered events to the MCS server. All event data is stored locally in JSON file until one of these methods is called. If a mobile app is offline, then it posts this file when it reconnects with the service's server.

The `Context` parameter in the `startSession` and `endSession` methods is the Android `Context` class. See the Android Developers website `http://developer.android.com`.

> **Tip:**
>
> For long-running mobile apps, calling the `flush` method periodically not only reduces the size of the payload in the JSON file posted to the MCS server, but also keeps the MCS server up to date.

The following code snippet shows how to call the Analytics class methods.

```
public final class Analytics extends ServiceProxy {
    // Creates a new session and generates an $sessionStart Event.
    public void startSession(Context context);

    // Ends the current session and generates a $sessionEnd Event.
    public void endSession(Context context);

    // Adds a new Event object.
    public Event logEvent(String name);

    // Adds an existing Event object.
    public Event logEvent(Event event);

    // Forces the upload of buffered Events.
    public void flush();
}
```

The Event class' methods create new events and their properties. As listed in the following table, this class also has events for returning information about an event, like its timestamp or its name.

Method	Description
`Event(String name, Date timestamp, Map<String, String> properties)`	Creates the new event.
`addProperty (String name, String value)`	The key-value pairs are managed by the `HashMap` interface. Call this method to add the key-value for a property to the existing map of event properties.
`getProperty`	Returns the properties associated with the event.

Method	Description
getProperties	Returns a property.
getName	Returns the name of the event.
getTimeStamp	Returns the date on which the event was recorded.

The following code snippet shows how to call these methods.

```
public final class Event {
    // Creates a new Event. Time stamp and properties can be null.
    public Event(String name, Date timestamp,  Map<String, String>
properties)

    // Sets a key/value property for the Event.
    public Event addProperty(String name, String value);

    // Returns the Event's name.
    public String getName();

    // Returns the Event's properties.
    public HashMap<String, String> getProperties();

    // Returns the timestamp of the Event.
    public Date getTimestamp();
```

See Taking a Look at Events and Sessions in Android Apps for examples of using these methods, as well as guidelines on how the mobile app code can reference a mobile backend and the Analytics service.

Taking a Look at Events and Sessions in Android Apps

The following code samples show how to call the Analytics class' method to add an event called "ShoppingCartCancelled" to your mobile app code:

```
...

Analytics analytics = mbe.getServiceProxy(Analytics.class);

client.startSession(this); // "this" is the Android View.
//...
client.logEvent("ShoppingCartCanceled");
//...
client.endSession(this);

...
```

Instead of adding a series of lines, you can add an event as well as properties as a single, fluent line of code:

```
...
mbe.getServiceProxy(Analytics.class).
```

```
    logEvent(new Event(this, "ShoppingCartCanceled").
         addProperty("cartSize", "2").
         addProperty("cartValue", "$50,000"));
...
```

Some general steps to follow when adding events to your Android app:

1. Add a reference to the `MobileBackendManager` class to access to the default moblie backend (which is specified in the `oracle_mobile_cloud_config.xml` file):

   ```
   try {

        MobileBackendManager mbem = mobileBackendManager.getmanager();
        MobileBackend mbe = mbem.getDefaultMobileBackend(this);
   ```

2. Because you need to log a custom event, you must reference the Analytics service:

   ```
   Analytics analytics = mbe.getServiceProxy(Analytics.class);
   ```

3. Create the event by calling the event constructor and pass in the name of the event, such as `"Work Order on Hold"`:
4. Call the `logEvent` method and pass the event:
5. Call the `flush` method to post events to the MCS server.

> **Tip:**
>
> To end the session and post all of the events to the MCS server, call `endSession` instead.

Defining Sessions

Sessions, which can group events together, can vary in length: a session may represent the entire lifespan of an application, or a function within the application. Within your code, you can specify the start and end of sessions, as illustrated by the `ShoppingCartCancelled` event shown in Creating Events and Sessions Using the Android Library and the `Purchase Start` and `Purchase Failed` events in Creating Events and Sessions Using the iOS Library. If you don't specify the start of a session, the analytics libraries in the mobile client SDK create an implied session.

Exporting Event Data

The Analytics Export API lets you return event data or API call metadata as a JSON object, which you can then import into a third-party tool. You may also want to export event data before you permanently delete it by purging it from one or more mobile backends.

For details on how to query the API and information about the responses, see REST APIs for Oracle Mobile Cloud Service.

For details on how to query the API and information about the responses, see REST APIs for Oracle Autonomous Mobile Cloud

For example, say you want to export data for a custom event called `MeanTimeResolution`, which measures performance data for your team. To find out how your team stacks up against an industry benchmark, you'd post a call to the Analytics Export API to return the custom event data. Using a third-party tool, you can mash up the `MeanTimeResolution` event data with the benchmark data and create reports.

How Do I Request Event and API Logging Data?

You can return data as a JSON object by issuing a POST call to *{baseUri}*`/mobile/system/analyticsExport`.

- If you are using basic authentication to connect to the mobile backend, see Authenticating with HTTP Basic in Direct REST Calls.
- If you have OAuth enabled as the authentication mechanism for the mobile backend, see Authenticating with OAuth in Direct REST Calls.

If you plan to use a third-party tool to pull the analytic data on a nightly basis, you could use cURL or some other tool to run the automated job. Here's some example cURL code for basic authentication:

```
curl -i
-X POST
-u team.user@example.com:Welcome123
-d @export.json
-H "Content-Type: application/json; charset=utf-8"
-H "Oracle-Mobile-Backend-ID: ABCD9278-091f-41aa-9cb2-184bd0586fce"
http://fif.cloud.oracle.com/mobile/system/analyticsExport
```

In the request body, you can specify whether you want to return custom events, or data from the API History Log by defining the required parameter, `exportType`. You can also limit the number of items returned in the JSON object and set the date range for the reporting period. A request body might look like this:

```
{
    "startDate": "2015-04-12",
    "endDate": "2015-05-12",
    "exportType": "Events",
    "name": "IncidentRaised",
    "offset": 0,
    "limit": 1000
}
```

Here are the request body properties.

Request Body Parameters	Mandatory?	Description	Example
`startDate` and `endDate`	No. If you don't define these values, then MCS applies these default values: • `startDate` — The first timestamp of the first event or API call record. • `endDate` —The current system time and date.	The start and end of the reporting period, expressed as `YYYY-MM-DD`.	`"startDate": "2015-04-12", "endDate": "2015-05-12",`
`exportType`	Yes	The type of data that you want to export: API calls or custom events.	`"exportType": "APICalls", "exportType": "Events",`
`name`	No	Depends on the value for `exportType`: • For `APICalls`, refer to the names listed in the menus for the API Endpoint reports. • For `Events`, the name of a custom event. Use the menus in Events report.	• For an API: `"name": "Analytics Collector",` • For a custom event: `"name": "IncidentRaised",`
`offset`	No	The zero-based index of the first item that's returned. The default value is zero (`0`).	`"offset": 0`
`limit`	No	The maximum number of items returned by your call. If you set a limit that's too high, then MCS substitutes a limit of 1000 (the default value).	`"limit": 500`

A portion of the thousand items in the JSON payload that's returned by the call in the preceding example (a request for a custom event called `IncidentsRaised`) might look like this:

```
{
   "items":[
      {
         "name":"IncidentRaised",
         "type":"custom",
```

```
            "timestamp":"2013-04-12T23:20:56.523Z",
            "sessionID":"2d64d3ff-25c7-4b92-8e49-21884b3495ce",
            "component":"Incidents",
            "mobileApplicationKey":"cd4b13b5-608c-4a18-9ef4-341fe4873063",
            "deviceId":"cd4b13b5-608c-4a18-9ef4-asdfasd",
            "backendName":"FixitFastCustomer",
            "backendversion":"1.0",
            "userName":"JDoe123",
            "locality":"San Francisco",
            "region":"CA",
            "country":"US",
            "postalCode":"95549",
            "timezone":"-14400",
            "carrier":"Verizon",
            "model":"iPhone5,1",
            "manufacturer":"Apple",
            "osName":"iPhone OS",
            "osVersion":"7.1",
            "osBuild":"13E28",
            "customProperties":{
               "Appliance Manufacturer":"Abc Corp",
               "Model Number":"M1234"
            }
         },
         ...
   ],
   "hasMore":true
}
```

The request body for a platform API (the Analytics Collector) might look like this:

```
{
    "startDate": "2015-04-12T01:20:55.052Z",
    "endDate": "2015-05-12T01:20:55.052Z",
    "exportType": "APICalls",
    "name": "Analytics Collector",
    "offset": 0,
    "limit" : 1000
}
```

A portion of the returned JSON payload might look like this:

```
{
  "items": [
    {
        "backendName": "FixItFastTechnician",
        "backendVersion": "1.0",
        "apiName":"analytics",
        "apiVersion": "1.0",
        "apiImplementationName": "analytics",
        "apiImplementationVersion": "1.0",
        "resourcePath": "/events",
        "requestMethod": "POST",
        "requestTime": "2013-04-12T23:20:56.523Z",
```

```
        "executionTime": 20,
        "responseCode": "202",
        "responseMessage": "",
        "responseErrorId": "",
        "responseErrorMessage": "",
        "type": "ServiceableREST",
        "ecid": "f2cd201e-535b-48d7-afe2-e85a1f30406b-00007b19",
        "rid": "0",
        "parameters" : {
            "x": "x1",
            "y": "y1"
        }
    }
    ...
  ],
  "hasMore": true
}
```


Purging Analytics Data

You can purge analytics data through either the MCS UI or your application. Whichever method you choose, note that purging analytics data permanently removes it from the selected mobile backend(s). Data that has been purged can not be restored.

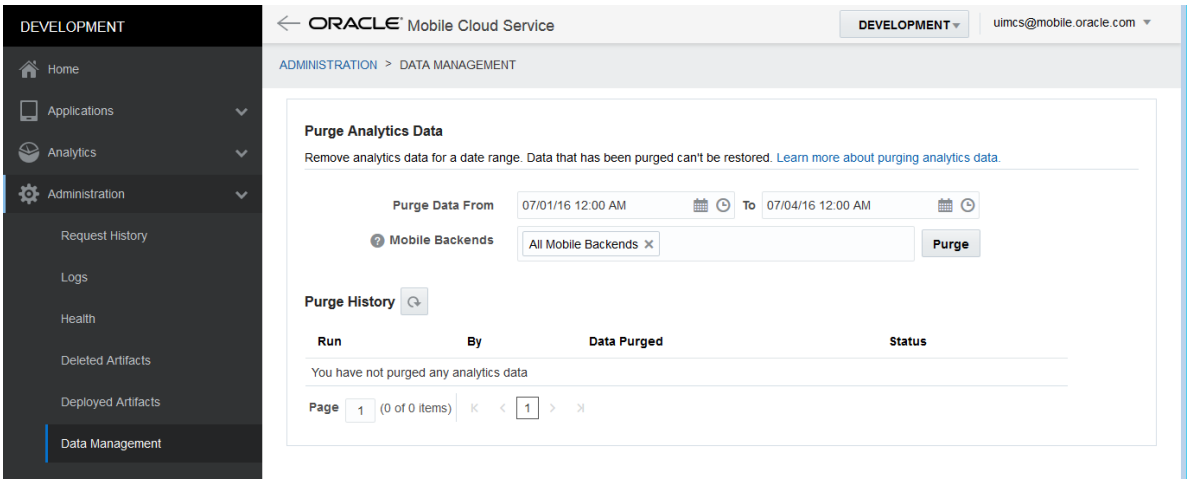
Purging Data through the MCS UI

First, export your data using the Export Data API before performing the purge. See Exporting Event Data for details.

Administrators must be assigned to the Mobile Deploy role to purge analytics data through the UI. To avoid server conflicts, you can run only one purge job at a time. Any purge request made while a purge is in progress is ignored.

1. On the side menu, select **Administration** , then **Data Management**.
2. Select the date range to purge analytics data.

   Click the calendar icon to select the date. The default time is 12 am, but you can change it by clicking the clock icon.
3. Select **All mobile backends**, or individual backends.



4. Once you enter a date range and at least one mobile backend, click **Purge**.

Look for the results of the purge action under Purge History.

> **Note:**
>
> The time to complete a purge action varies depending on the size of the purge job. Large purge jobs can take some time to complete. Wait at least five to ten minutes before refreshing ( ) the purge history to see the latest purge information.

Purging Data from an Application
Use the Analytics Data Management (ADM) API to make calls from your application to permanently purge data from selected backends. Purge means this API both deletes the data and “shrinks” the database that stores it in order to free up more space.

Like the purging feature you can use from the UI, the ADM API uses authentication, roles, and realms for security. Administrators assigned to the Mobile System role can purge MCS analytics data with the ADM API. You access this API through the mobile client SDK, or directly through REST calls. To avoid MCS server conflicts, you can run only one purge job at a time. MCS ignores purge requests made while a purge is in progress.

To access the Analytics Data Management API through the mobile client SDK, use a backend manager class.

- For Android apps, you use the MobileManager class as described in .

  Calling Platform APIs Using the SDK for Android
- For iOS apps, you use the OMCMobileBackendManager class as described in .

  Calling Platform APIs Using the SDK for iOS
- For Cordova and JavaScript apps, you use the mobileBackendManager class as described in Calling Platform APIs Using the SDK for Cordova and Calling Platform APIs Using the SDK for JavaScript.

  For detailed information about the platform APIs, see REST APIs for Oracle Mobile Cloud Service.

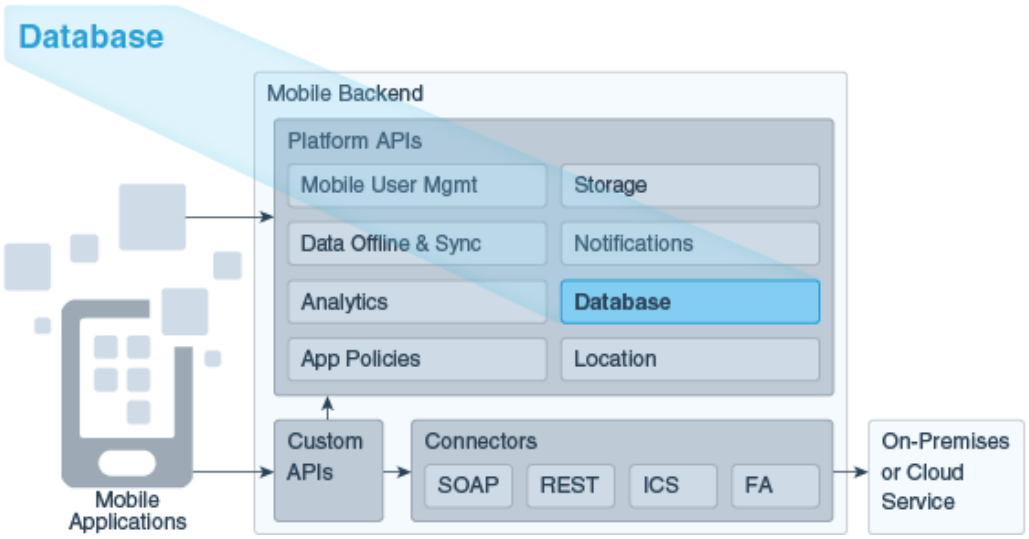
Troubleshooting Analytics Reports

MCS generates analytics reports based on artifacts created on the MCS server, events defined in the mobile app code, and by the mobile client SDK. Depending on these factors, none, or all, of the reports may be available.

Problem	Solution
No reports	When you first log into MCS , there aren't any reports because you have yet to create a mobile backend for the mobile app.
Only the API Calls Count and API Calls Response Time appear, but there are no user, session, or event reports.	MCS creates two sets of mobile analytics reports: reports for server activity and reports for sessions and events defined in the mobile app's code. In this case, only the calls from testing the endpoints are recorded, because no data has yet been sent from the mobile app. Only the endpoint reports are available if you've just created a mobile backend, but haven't yet registered a mobile app as its client app. While you can call the endpoints of Custom Code APIs directly by testing them, MCS can't chart session or events reports because these calls don't originate from a registered mobile app, whose code defines event and sessions. MCS creates user, session, and event reports when a registered mobile app that uses the mobile client SDK (or has sessions defined in its code) calls the Analytics API. Keep in mind that if there isn't a mobile app to provide events, then there will be no data for funnel reports. In this case, MCS displays a page that says, "You have no analytics data for this mobile backend."
API calls generate analytics data, eventually using up database space resulting in system issues or loss of service.	MCS automatically generates analytics data for every API invocation. This data accumulates over time and can use up all available database space, which can result in system instability or even complete loss of service. Before storage capacity reaches full, modify the `Analytics_ApiCallEventCollectionEnabled` policy by setting the policy to `false` to prevent automatic generation of analytics data with each API call.

18
Database

Database APIs help you create and manage database tables for use in mobile apps. As a service developer, you can call the Database Access API from custom API implementations to create and access database tables, and use the Database Management API to manage and view table metadata.



What Can I Do with Database APIs?

As noted above, there are two database APIs:

- The **Database Access API**, which is available only from custom code implementations using the custom code SDK, lets you to create and access database tables. For security reasons, you can't call this API from client apps. To try out calls to this API, open a custom API, go to the Custom Catalog, and then click **Database Access**.
- The **Database Management API** can be accessed through custom code implementations and HTTP REST calls to manage table metadata and deploy tables. To try out calls to this API from the UI, click ☰ to open the side menu, and click **Applications > APIs**. In the Platform APIs section at the bottom of the APIs page, click **Database Management**.

This chapter discusses how to use these Database APIs to perform common tasks. For more details on using the platform APIs, see REST APIs for Oracle Mobile Cloud Service.

Database Access API

All your mobile apps' interactions with the Database Access API are made through custom API implementations. You can't access this API directly from client apps. This section covers how to use the custom code SDK in a custom API implementation to interact with the database. To learn about designing APIs, see Custom API Design. To learn about implementing a custom API, see Implementing Custom APIs. For complete details for each custom code SDK database method, see Accessing the Database Access API from Custom Code .

Calling the Database Access API from Custom Code

Before we delve into how to implement a custom API to perform database tasks, let's go over a simplified description of how to call the Database Access API from custom code. Here we talk about some API operations that you learn about later. While they may not make sense now, these steps should give you some context for how you use the operations that you will learn about.

To call the Database Access API from custom code, you add endpoints (resources) and operations (methods) to the custom API, and then you add route definitions to your custom code implementation for the custom API. We are going to talk about how to implement the route definitions in the custom code.

To call the API from your custom code:

1. Add the route definition to the custom code.

   You implement a route definition by calling the `service` method for the API's endpoint operation. Say, for example, that your API has a `GET` operation for the `/mobile/custom/FIF_Incidents/incidents` endpoint. To implement this from your custom code, you call `service.get()`. The `service` method's arguments are the URI and a function that takes both the request object and the response object as arguments. For example:

   ```
   service.get(
   '/mobile/custom/FIF_Incidents/incidents', function (req, res) {
     // your code goes here
   });
   ```

2. From the route definition, call the appropriate `req.oracleMobile.database` method to send your request to the Database Access API, such as `get()`, `getAll()`, or `insert()`. Accessing the Database Access API from Custom Code describes the available methods and the arguments that each method takes, and provides example code.

   Here's a complete route definition. This route definition calls the `getAll()` method, which, in turn, calls the Database Access API's `GET /mobile/platform/database/objects/{table}` operation. When the `getAll()` method receives a response from the API, it calls either the `result` function or the `error` function, depending on whether an error occurred.

Notice that the first argument is the name of the table, and that the second argument is a JSON object that contains a `fields` property. This instructs the `getAll()` method to return only the `customer` and `status` fields.

```
/**
 * GET CUSTOMER AND STATUS FOR ALL INCIDENTS
 */
service.get('/mobile/custom/incidentreport/incidents',
function (req, res) {
    req.oracleMobile.database.getAll(
      'FIF_Incidents', {fields: 'customer,status'}).then(
      function (result) {
        res.status(statusCode).send (result.result);
      },
      function (error) {
        res.status(statusCode).send(error.error);
      }
    );
  });
```

The response to this call would look like this:

```
{
  "items":[
    {
      "status":"Open",
      "customer":"Lynn Smith"
    },
    {
      "status":"Completed",
      "customer":"John Doe"
    }
  ]
}
```

Creating and Restructuring Database Tables

You might think that before you can access a database table, you need to first add it to the schema. However, you can create a new table simply by adding a row to the table. This action is referred to as a *implicit* table creation.

> **Note:**
>
> Typically, you take advantage of implicit table creation when you're developing your mobile app. When you deploy your mobile app to another environment, you use explicit table creation to create the tables in that environment as described in Database Management API.

You use the following methods to insert rows into a table:

- `insert()`: Add one or more rows.

- `merge()`: Add or update one or more rows.

When you call these methods for a table that doesn't exist, a new table with the row(s) is created by deriving the table specifications from information in the `object` and `options` arguments.

To specify the table structure:

- Call either `insert()` or `merge()`, both of which require `table` and `object` arguments. In the `object` argument, which is a JSON object, include all the columns that you want in the table, and provide mock or real data for each column. The column type and size are based on the content. For example, if the value is `100` then the column will be `NUMBER(3,0)`. Don't worry about the size being too small. If you later post `3.25`, the column is resized to `NUMBER(5,2)`, which is large enough for both `100` and `3.25`. Also don't worry about adding all the columns that you need. If you later decide you want more columns, then add the new columns to a JSON object and send it in an `insert()` or `merge()` call. The table will be restructured automatically to add the new columns.

  > **Note:**
  >
  > The maximum size for a string column is 4000 characters. If you need to store a larger string, then you can use the Storage API to store the object.

  Here's an example of the JSON object:

  ```
  {
    "incidentReport": 1,
    "title": "Water heater is leaking",
    "customer": "Lynn Smith",
    "address": "200 Oracle Parkway Redwood City, CA 94065",
    "phone": "(555) 212-4567",
    "technician": "jwhite",
    "status": "Open",
    "notes": "lynnf|Initial incident report description",
    "priority": 1,
    "imageLink": "http://link.to.storage"
  }
  ```

- By default, a set of predefined columns are added and populated automatically whenever you add or update a record using `insert()` or `merge()`.

  If you don't want all these columns in your table, then use the `extraFields` property in the optional `options` argument to specify which columns to include, such as `createdOn,createdBy` (be sure to include `id` if you aren't specifying a primary key). If you later decide you want to add more predefined columns, you can just add them to the `extraFields` property the next time you add a row.
  If you don't want any of these columns, then set the `extraFields` property to `none`. However, if you don't add any predefined columns when you create the table, then you can't add any later.

  The predefined fields are:

  - `id`: The row key. This column is added only if both the `primaryKeys` and `extraFields` properties are absent. The `id` is an integer set and incremented automatically.
  - `createdBy`: Who created it.
  - `createdOn`: When it was created.
  - `modifiedBy`: When it was last modified.
  - `modifiedOn`: Who modified it last.

  The dates are in W3C date-time format, and include hours, minutes, seconds, and a decimal fraction of a second (`YYYY-MM-DDThh:mm.ss.SSSZ`).

- If you want a primary key, use the `primaryKeys` property in the `options` argument to specify which columns to use for the primary key. For example, `incidentReport,technician`. Note that the order that you list the fields is the order that you use when you retrieve or update a row. Because you can't retrieve the primary key order from the table metadata, make sure that you document the order of the primary fields.

You can see code examples for these two methods in the next section.

> **Note:**
>
> You also can use the Database Management API to create a table. However, you typically use the Database Access API for the initial creation and then use the Database Management API to copy the table structure to other environments, as described in Copying Table Structures to Another Environment.

The following table summarizes what aspects of a table can be changed implicitly:

Object	Can It Change?
Table Name	No. The name is set when the table is first created.
Primary Key	No. The primary key is defined when the table is created.
Predefined Columns	Yes. You can allow predefined columns in the table when it's created by the call. However, you can't add these predefined columns at a later point if the table was not originally intended to use them. If predefined columns are allowed, then any of them (other than `id`, that is) can be added by subsequent calls.
Columns	Yes. Although columns are created with the table, subsequent calls can add columns. These calls can also alter the column size. However, you can't change the column type after the table has been created.

> **Note:**
>
> You can also disable implicit table creation. If the `Database_CreateTablesPolicy` environment policy is neither `allow` (the default setting) nor `implicitOnly`, adding a row to a non-existent table will fail.

Preventing Passing SQL Using Implicit Table Creation

When the `Database_CreateTablesPolicy` environment policy is set to `allow` (default setting) or `implicitOnly`, the Database Access API dynamically constructs SQL statements that create and alter tables from user input.

To prevent users from using implicit table creation to pass SQL statements, set this policy to either `none` or `explicitOnly` in the staging and production environments. You should also do this in the development environment when:

- All the tables required by an application have been created.
- The mobile backend is ready to be deployed to another environment.

Adding and Updating Table Rows

You use the `insert()` and `merge()`methods to add and update rows:

- `insert()` adds one or more rows.
- `merge()` adds or updates one or more rows. Whether an add or update is performed depends on whether the table uses `id` or primary key fields to uniquely identify rows.
  - `id` field: If you include an `id` property in the `object`, then the matching row is updated if it exists. Otherwise a new row is added.
  - Primary key fields: If the table uses primary key fields, the matching row is updated if it exists. Otherwise, a new row is added.

> **Note:**
>
> If you submit a batch of rows, then all the rows must have the same set of columns.

To call either of these methods:

- Pass the table name in the first argument.
- If the table doesn't exist, and you want to limit which predefined columns to include, set the `extraFields` property in the `options` argument. For example:

```
options =
  {'extraFields' : 'createdOn,createdBy'}
```

  If you want all the predefined columns, omit this property. If you don't want any predefined columns, set it to `none`. It doesn't hurt to include it in subsequent adds,

but make sure you include it in your first add if you don't want the full set of predefined columns.

- If the table doesn't exist, and you want to specify a primary key, make sure you set the `primaryKeys` property in the `options` argument. For example:

  ```
  options =
    { 'primaryKeys' : 'incidentReport,technician' }
  ```

  The primary key list must be URL encoded.

- Put the row data in the request body in JSON format. The JSON object can contain data for one row or several rows.

  Here is an example of data for one row:

  ```
  {
    "status" : "Open",
    "code" : "3"
  }
  ```

  Here is an example of data for multiple rows:

  ```
  [
      {
          "status":"Open",
          "code":3},
      {
          "status":"Completed",
          "code":9}
  ]
  ```

Here's an example of using the `insert()` method to add two rows to the FIF_Status table. The first argument is the table name, and the second argument is the `object` argument, which contains the rows to add to the table. The third argument is the `options` argument, which specifies to not add any extra (predefined) fields, and to create a primary key based on the `code` field.

```
service.post('/mobile/custom/incidentreport/initStatus', function (req,
res) {
  req.oracleMobile.database.insert(
    'FIF_Status',
    [
      {
        "status": "Closed",
        "code": "0"},
      {
        "status": "Completed",
        "code": "9"}
    ],
    {extraFields: 'none', primaryKeys: 'code'}).then(
    function (result) {
      res.status(statusCode).send (result.result);
      },
      function (error) {
```

```
        res.status(statusCode).send(error.error);
    }
  );
});
```

Retrieving Table Rows

You can retrieve a single table row by its primary key or ID, and you can retrieve a set of table rows.

To retrieve a row by its primary key or ID, call the `get()` method. You use the `keys` argument to identify the row that you want.

- If the table uses the `id` column for the row key, then set `keys` to the row's ID.
- If the table has a primary key, then set `keys` to the primary key values in the order in which the primary keys were specified when the first row was added to the table (which resulted in the creation of the table). Use an array for a composite key. For example, if the `options.primaryKeys` property was set to `incidentReport,technician` when the table was created, then the values must be listed in that order, such as: `['5690','jwhite']`.

Here's an example of using the `get()` method to retrieve a row from the FIF_Incidents table. The first argument is the table name, and the second argument is the `keys` argument:

```
/**
 * GET INCIDENT BY ID
 */
service.get('/mobile/custom/incidentreport/incidents/:id',
  function (req, res) {
    req.oracleMobile.database.get(
      'FIF_Incidents', req.params.id).then(
      function (result) {
        res.status(statusCode).send (result.result);
      },
      function (error) {
        res.status(statusCode).send(error.error);
      }
    );
  });
```

The response body looks like this:

```
{
  "items":[
    {
      "id":168,
      "title":"Oven not working",
      "technician":"jwhite",
      "status":"Open",
      "customer":"John Doe",
      "incidentReport":"5690",
      "createdBy":"jdoe",
      "createdOn":"2015-11-16T23:42:18.281823+00:00"
```

```
    }
  ]
}
```

To get a set of rows from a table, call the `getAll()` method.

- To filter the rows, add the columns to search on and the values to match to the `qs` property in the optional `httpOptions` argument. For example, this requests all the incident reports for the technician J. White:

  ```
  httpOptions.qs = {technician : 'jwhite'};
  ```

- To specify which columns to return, use the `fields` property in the `options` argument.

  For example, to get a quick phone list:

  ```
  options={'fields' : 'customer,phone'}
  ```

Here's an example of using `getAll()` to retrieve the `customer` and `status` fields for all rows in the FIF_Incidents table that match the query string that's specified in `httpOptions.qs`.

```
/**
 * GET ALL INCIDENTS
 */
service.get('/mobile/custom/incidentreport/incidents',
function (req, res) {
    httpOptions={};
    httpOptions.qs = {technician : 'jwhite'};
    req.oracleMobile.database.getAll(
      'FIF_Incidents', {fields: 'customer,status'}, httpOptions).then(
      function (result) {
        rres.status(statusCode).send (result.result);
      },
      function (error) {
        res.status(statusCode).send(error.error);
      }
    );
  });
```

The response body looks like this:

```
{"items":[
  {"title":"Water heater is leaking",
  "technician":"jwhite",
  ,"customer":"Lynn Smith"
  ...
  "incidentReport":25
  "createdOn":"2015-03-05T12:10:15.171284-07:00"},
  {"title":"Dryer doesn't dry",
  "technician":"jwhite",
  ,"customer":"Lynn Smith"
  ...
```

```
  "incidentReport":67
  "createdOn":"2015-08-07T14:22:37.171284-07:00"}
]}
```

Deleting Table Rows

To delete a row, you call the `delete()` method.

You use the `keys` argument to identify the row that you want to delete.

- If the table uses the `id` column for the row key, then set `keys` to the row's ID.
- If the table has a primary key, then set `keys` to the primary key values in the order in which the primary keys were specified when the first row was added to the table (which resulted in the creation of the table). Use an array for a composite key. For example, if the `options.primaryKeys` property was set to `incidentReport,technician` when the table was created, then the values must be listed in that order, such as: `['5690','jwhite']`.

Here's an example of deleting a row from the FIF_Incidents table. The first argument to the `delete()` method is the table name, and the second argument is the `keys` argument.

```
/**
 * DELETE INCIDENT BY ID
 */
service.delete('/mobile/custom/incidentreport/incidents/:id',
  function (req, res) {
    req.oracleMobile.database.delete(
      'FIF_Incidents', req.params.id).then(
      function (result) {
        res.send(result.statusCode, result.result);
      },
      function (error) {
        res.send(error.statusCode, error.error);
      }
    );
  });
```

If the table has a primary key, then the response body looks like this:

```
{ "rowCount" : 1 }
```

If the `id` is the key value for the table, then the response body looks like this:

```
{"items":[{"id":42}]}
```

Database Management API

In addition to the Database Access API, there's also a Database Management API, which lets you manage the tables that you created through the Database Access API. This API lets you view table metadata, create, drop, re-create tables, and create indexes for them.

You can use the Database Management API only if you have been granted the database management role (`Mobile_DbMgmt`). If you don't have this role, then you can't create a table or use the `GET` operation to see which tables have been created. You don't need this role to use the Database Access API. For more information about roles, see Team Members.

You can access the Database Management API through custom API implementations and HTTP REST calls. To try out calls to the API, click  to open the side menu. Next, click **Applications** then **APIs**. In the Platform APIs section located at the bottom of the page, click **Database Management** . For further details about each API operation, see Here, we give a brief overview of the Storage API endpoints. For detailed information, see REST APIs for Oracle Mobile Cloud Service..

Creating a Table Explicitly

You can create a table from a JSON object using the `POST` method for the `/mobile/system/databaseManagement/tables` endpoint. To restructure a table, use the `PUT` method for the same endpoint. The `PUT` method drops the existing table and re-creates it.

To create a table explicitly:

1. If you want to include predefined columns in the table, set the `Oracle-Mobile-Extra-Fields` header to a comma-separated list of the columns to include from amongst `id`, `createdBy`, `createdOn`, `modifiedBy`, and `modifiedOn`. If you don't want any of these columns, specify `none`. The `id` column, which is a row key, is added to the table only if no primary key is specified.
2. Create the JSON object for the request body. The JSON attributes are:
   - `name`: The table name.
   - `columns`: An array of the table columns. For each column, specify:
     - `name`: The column name.
     - `type`: The data type. The binary data type is not supported.
     - `size`: (Optional) The size or precision of the column.
     - `subSize`: (Optional) For decimal columns, the scale of the column, meaning the number of places after the decimal point.
   - `primaryKeys`: An array of column names.
   - `requiredColumns`: An array of column names.
3. Call the `POST` method for the `/mobile/system/databaseManagement/tables` endpoint.

Here's an example of a JSON object for creating a table. When used in a `POST` request, a table called `Movies` is created with the specified columns and primary key.

```
{ "name" : "Movies",
  "columns": [
    {"name": "title", "type": "string", "size": 50},
    {"name": "synopsis", "type": "string"},
    {"name": "inTheaters", "type": "boolean"},
    {"name": "releaseDate", "type": "dateTime"},
    {"name": "runningTime", "type": "integer", "size": 3},
```

```
    {"name": "totalGross", "type": "decimal", "size": 10, "subSize": 2}],
  "primaryKeys" : [ "title" ],
  "requiredColumns": ["title", "releaseDate" ]
}
```

The Database Management API creates and executes the following SQL statement based on this request. In this case, the `Oracle-Mobile-Extra-Fields` request header was set to `none`, so the table does not have any predefined fields.

```
CREATE TABLE "Movies" (
    "title" VARCHAR2(50) NOT NULL,
    "synopsis" VARCHAR2(4000),
    "inTheaters" CHAR(1),
    "releaseDate" TIMESTAMP NOT NULL,
    "runningTime" NUMBER(3,0),
    "totalGross" NUMBER(10,2),
    CONSTRAINT "Movies_PK" PRIMARY KEY ("title"))
```

This example also illustrates some of the data types allowed by the Database Management API and the Database Access API:

Type	Description	Size / Subsize	Database Type
`string`	A JSON string	Maximum of 4000 bytes	`VARCHAR2`
`dateTime`	An ISO- or date-formatted JSON string		`TIMESTAMP`
`boolean`	A JSON boolean		`CHAR(1)` “1” true, “0” false
`decimal`	A JSON number	Precision (the total number of digits). Optional. / Scale (number of decimal digits). Optional.	• `NUMBER` • `NUMBER(size)` • `NUMBER(*,subsize)`
`integer`	A JSON number with no decimal digits		`NUMBER(size,0)` and `NUMBER(*0)`

The `size` and `subSize` attributes are optional. Don’t provide them for columns of type `dateTime` and `boolean`. As a best practice, unless you have a valid business constraint, don't provide `size` or `subSize` for integers and decimals because doing so limits what values are acceptable and makes it harder to resize the column. When possible, allow the database to size and store the value as efficiently as possible. However, you should provide the `size` attribute for string columns. The maximum size for a string column is 4000 characters. If you need to store a larger string, then you can use the Storage platform to store the object.

Copying Table Structures to Another Environment

When you promote a mobile backend to a staging or production environment, you can use the Database Management API’s operations to copy the table structures. These are the table structures that you created either implicitly through calls to the `insert()` and `merge()` methods or explicitly through the Database Management API.

> **Note:**
>
> As noted above, only a team member with the database management role (`MobileEnvironment_DbMgmt`) in the target environment can use the Database Management API.

Typically, the flow to copy table structures to another environment is as follows:

1. Use the Database Management API's `GET` operations to get the table metadata from the source environment.

   > **Note:**
   >
   > The metadata lists the primary key fields in alphabetical order, and not in the order that you specified when the table was created. When you use this metadata to recreate a table, you must reorder the fields correctly.

2. From the target environment in the MCS UI, export the environment policy file.
3. Change the `Database_CreateTablesPolicy` policy for the target environment to `explicitOnly`. For information about updating environment policies, see Environment Policies.

   > **Note:**
   >
   > In a development environment, schema creation occurs implicitly because `Database_CreateTablesPolicy` is set to `allow` by default (`*.*.Database_CreateTablesPolicy=allow`). By the time the mobile database is deployed to a staging environment, this policy should be disabled to prevent tables from changing.

4. Import the environment policy file back into the target environment.
5. Create a cURL script that sets up the tables in the target environment with the table metadata you retrieved in step 1, using `POST` commands to the `/mobile/system/databaseManagement/tables/{table}` endpoint. Run the script to create the tables.

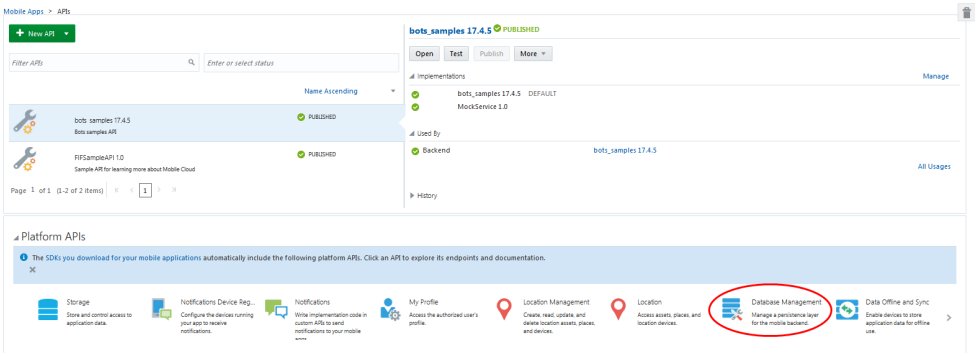
   Remember to reorder the primary fields in the correct order.
6. Repeat Step 3, but set the `Database_CreateTablesPolicy` policy to `none`.

Creating or Deleting an Index on a Table

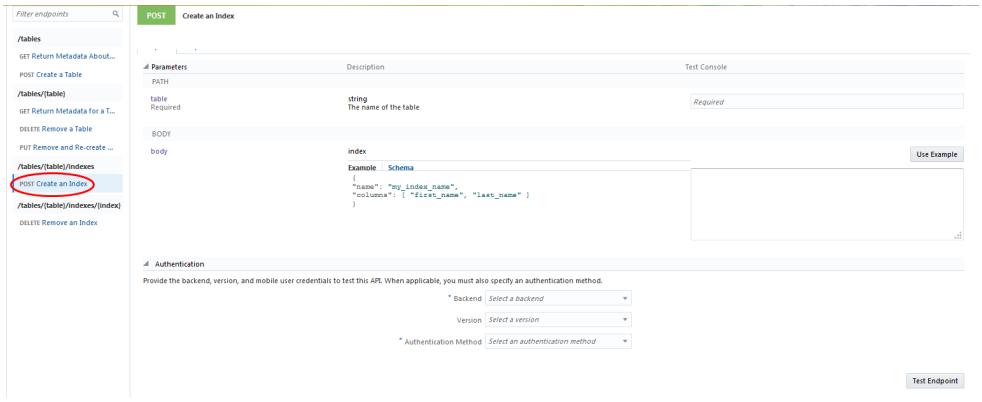
To improve the speed of data retrieval, you can use the Database Management API to create an index for a table. This API is also used to delete an index for a table.

To create a database index for an existing table:

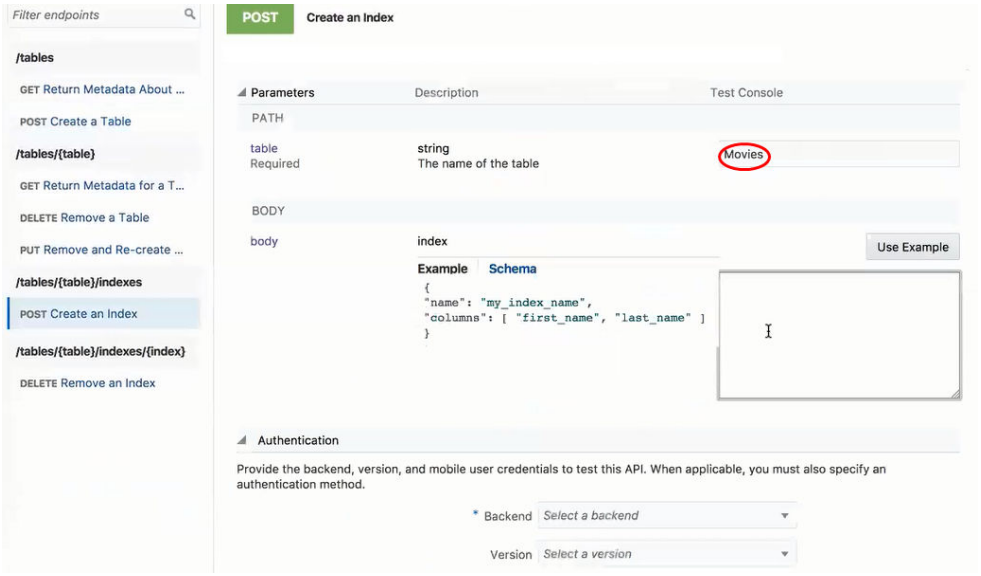
1. In the Platform APIs section located at the bottom of the APIs page, click **Database Management** .



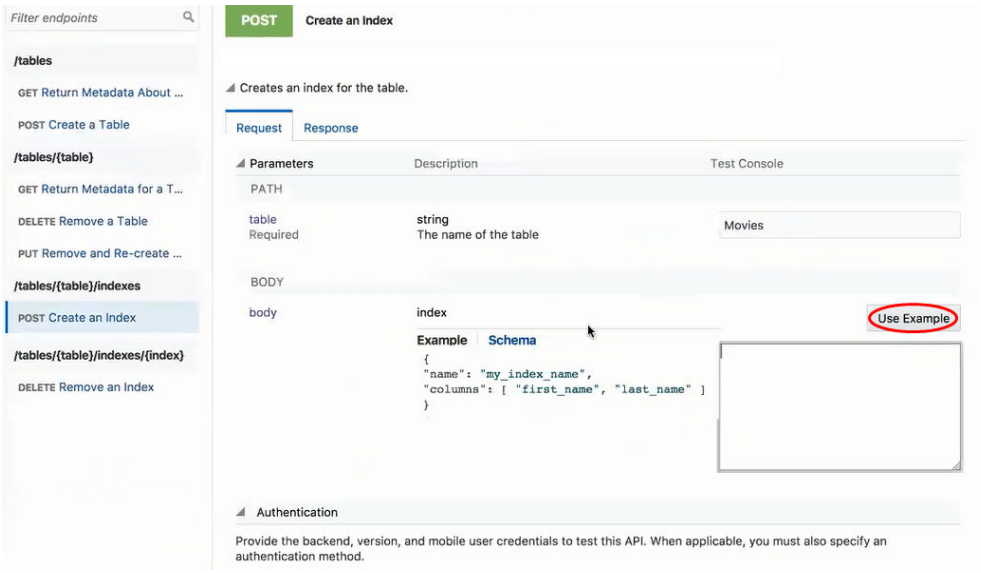
2. On the left panel, click **Create an Index**.



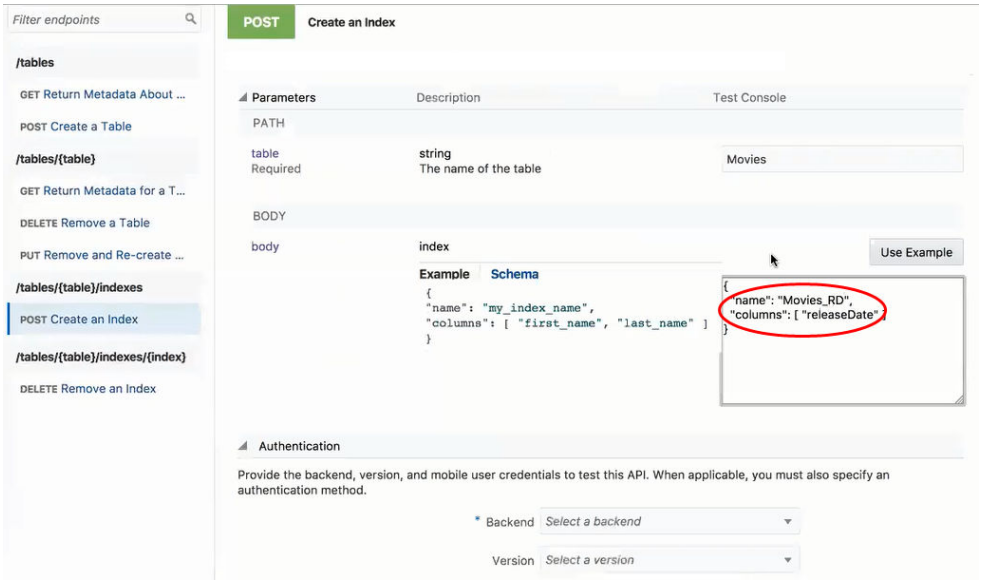
3. Fill in the name of the database table you want to index.



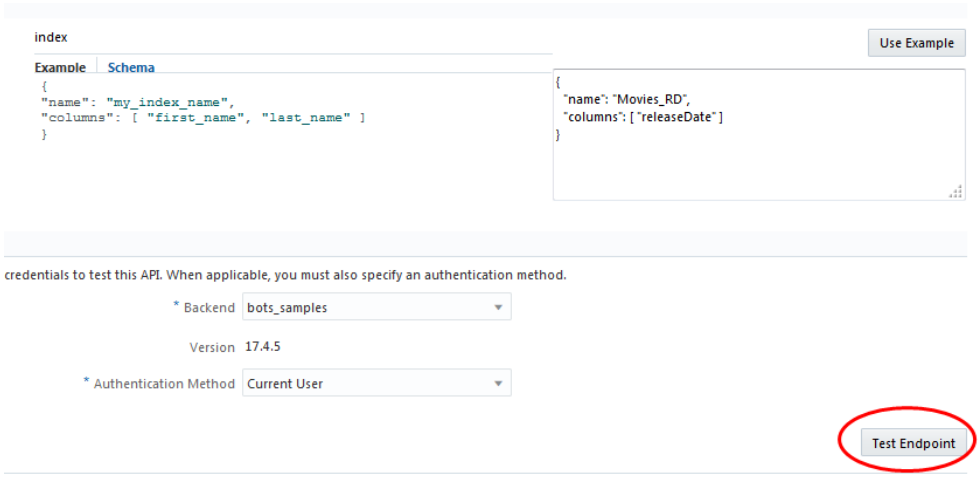
4. Click **Use Example** to use the example code for the index.



5. In the example code, replace the index and column names with whatever names you want to use.



6. In the **Backend** menu, select the backend, then version, you want to use to test the API.
7. Select **Current User** for the authentication method.
8. Click **Test Endpoint**.



You should receive a 201 response indicating your index has been created.

> **Note:**
>
> If you create an index on a table, then call PUT/mobile/system/databaseManagement/tables, any user-defined indexes will be dropped. However default indexes, like the one created on a primary key, will be recreated.

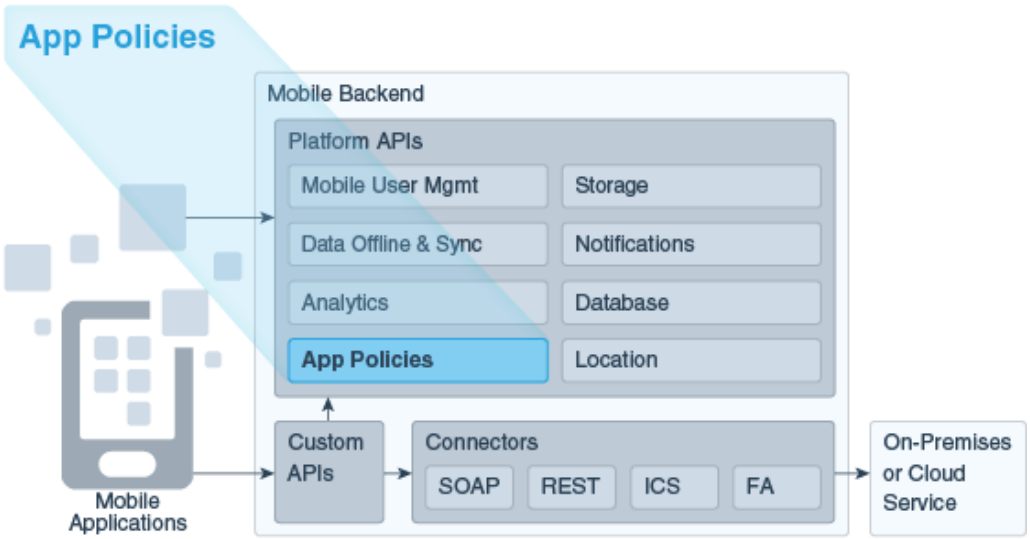
Deleting an Index

The process for deleting an index is very much like creating one. Just choose **Remove an Index** from the left panel. Then enter the names of the index and the table, as well as the backend and authentication method. Finally, click **Test Endpoint** to see that the index has been removed.

19

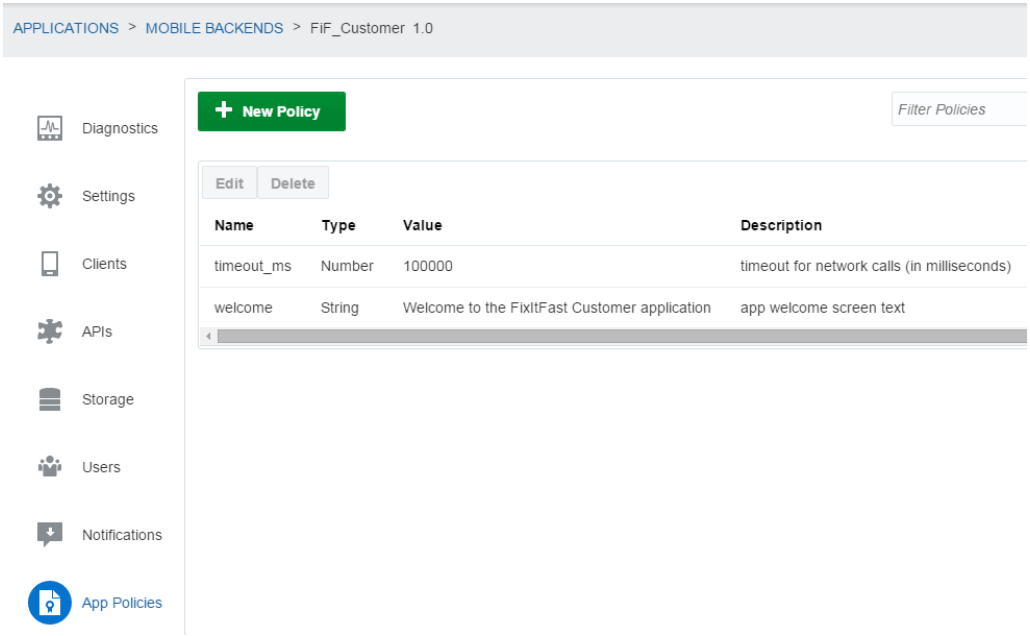
App Policies

As a mobile app developer, you can use the App Policies API to create read-only custom properties in a mobile backend and access them in your application with REST calls.



What Are App Policies and What Can I Do With Them?

App policies are custom properties that you can define and adjust in a mobile backend and then reference from your apps through a simple REST call. Once you have defined an app policy, you can update its value anytime, even after you have published the mobile backend. This lets you make changes to the appearance and behavior of a deployed app without having to update the app itself.



Here are some of the things that you might use app policies for:

- Determining when a given feature is enabled in the app. For example, an app for a retailer might have a feature to display a section for holiday sales that should only be displayed when there is a current sale.
- Fonts, colors, names of images to use, and other things that are typically stored as part of an app's configuration.
- Timeout values for network calls. Having an app policy for this can allow your mobile cloud administrator to tune app responsiveness based on prevailing network performance.

Setting an App Policy

1. Make sure you're in the environment where you want to set the app policy.
2. Click  to open the side menu and select **Applications > Mobile Backends**.
3. Open the backend. (Select it and click **Open**.)
4. Click the **App Policies** tab.
5. Click **New Policy**, fill in the property name, type, value, and description, and then click **Create**.

The new app policy appears in a table on the page.
You can later use the **Edit** and **Delete** buttons in the table to edit the policy or remove it entirely. After the mobile backend has been published, you can still change a policy's value, but you can not add, delete, or rename policies or change the policy type.

> **Note:**
>
> You can only set app policies and change their values from within the MCS user interface. You can't do this programmatically from app code.

Retrieving App Policies in App Code

You can retrieve information on the app policies associated with a mobile backend using the REST API or any of the client SDKs. The REST API enables you to retrieve an array of all of the policies for the mobile backend. The SDKs also enable you to retrieve information on specific policies.

REST

Using the following call, you can retrieve all of the app policies associated with a mobile backend.

```
GET {BaseURL}/mobile/platform/appconfig/client
```

The response body is a JSON object containing all of the app policies configured for that mobile backend. For example, if the mobile backend contains `fifTechReqTimeout`, `fifTechWelcomeMsg`, and `fifTechBgImage` policies, the response might look something like this:

```
{
    "fifTechReqTimeout":100000,
    "fifTechWelcomeMsg":"Hello",
    "fifTechBgImage":"/mobile/platform/storage/collections/appObjects/
objects/bgImage42"
}
```

From there, you can process them in your app code.

Android SDK

To fetch app policies for the first time, you use the `MobileBackend` object's `getAppConfig()` method to return all app policies as a `JSONObject`:

```
JSONObject appPolicies = oracle.cloud.mobile.mobilebackend
                         .MobileBackendManager.getMobileBackend().getAppCon
fig();
```

Once you have fetched the app policies, you can query the app config for the values of individual properties.

To return the value of a specific app policy of type `String`, where `myPolicyName` is the name of the policy and "`No policy configured`" is the string returned if `myPolicyName` doesn't exist:

```
String myPolicyValue =
oracle.cloud.mobile.mobilebackend.MobileBackendManager
```

```
                        .getMobileBackend().getAppConfig().getString(myPolic
yName, "No policy configured");
```

To return the value of a specific app policy of type `Boolean`, where `myPolicyName` is the name of the policy and `false` is the value returned if `myPolicyName` doesn't exist:

```
Boolean myPolicyValue =
oracle.cloud.mobile.mobilebackend.MobileBackendManager
                        .getMobileBackend().getAppConfig().getBoolean(myPol
icyName, false);
```

To return the value of a specific app policy of type `Integer`, where `myPolicyName` is the name of the policy and `0` is the value returned if `myPolicyName` doesn't exist:

```
Integer myPolicyValue =
oracle.cloud.mobile.mobilebackend.MobileBackendManager
                        .getMobileBackend().getAppConfig().getInt(myPolicyN
ame, 0);
```

iOS SDK

To fetch app policies for the first time, you use an ansynchronous callback. Here's some code that will fetch the app config from the mobile backend and loop until the network call returns with either the app config or an error:

```
OMCMobileBackend* mbe = [[OMCMobileBackendManager sharedManager]
defaultMobileBackend];

__block OMCAppConfig* appConfig = nil;
__block NSError* error = nil;
__block BOOL executing = YES;
[_mbe appConfigWithCompletionHandler:^(OMCAppConfig* appConfig_, NSError*
error_) {
    appConfig = appConfig_;
    error = error_;
    executing = NO;
}];

while (executing) {
    [[NSRunLoop currentRunLoop] runUntilDate:[NSDate dateWithTimeInterval:
0.5 sinceDate:[NSDate date]]];
}

if (error != nil) {
    return;
}
```

Once you have fetched the app policies, you can query the app config for the values of individual properties. You can also insert an optional parameter to return a value if the policy is not found.

```
NSString* welcome = [appConfig stringForProperty:@"welcome"
default:@"bogus"];
```

```
int timeout = [appConfig integerForProperty:@"TIMEOUT" default:42];
boolean enabled = [appConfig booleanForProperty:@"enableLocation"
default:NO];
```

Cordova SDK and JavaScript SDK

To fetch app policies, call `loadAppConfig()` on your mobile backend object, e.g.

```
backend = mcs.MobileBackendManager.getMobileBackend("JSBackend");
...
backend.loadAppConfig(success, error);
```

Updating an App Policy Value in a Published Mobile Backend

Even after a mobile backend has been published, you can still change the value of an app policy. However, you can not change its name or type.

1. Make sure you're in the environment where you want to update the app policy.
2. Click  to open the side menu and select **Applications > Mobile Backends**.
3. Open the mobile backend. (Select it and click **Open**.)
4. Click the **App Policies** tab.
5. In the table of app policies, select the policy and click **Edit**.
6. Edit the value and click **Save**.

Part IV
Custom APIs

This part contains the following chapters:

20

Creating APIs Fast with the Express API Designer

What is the Express API Designer?

The Express API Designer enables you to create an API using sample data. This data-first approach lets you build an API quickly and with a minimum of effort. This designer is an alternative to the API Designer, where less is generated but you have more control of the API definition. See Which API Designer Should I Use? for a more detailed comparison.

How Do You Get Started?

Using the Express API Designer, you get a set of generated endpoints when you paste in a set of sample data that's formatted as a JSON instance. Within the context of the API Designer, this collection of endpoints is known as a resource. Resources are the building blocks of the API.

How Do You Use the API?

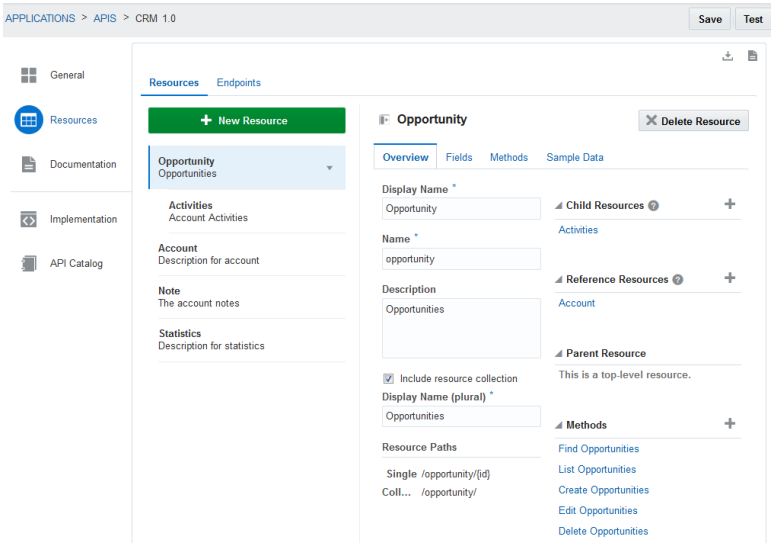
With your methods sketched in, you can then start using the API as part of your development effort by testing its endpoints and taking a look at mock data that it returns. Your service developers can implement a service for this API using JavaScript and Node. For more design and customization options, use the API Designer instead. See Custom API Design.

What are Resources?

A resource represents a real world object and the operations that can be performed upon it. In other words, the GET, POST, and PUT operations on the `/incidents` endpoint would simply be known as an “incident”.

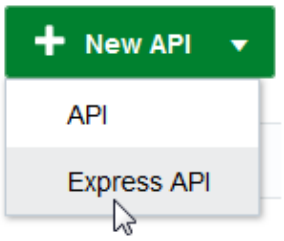
How Do I Get Started with Resources?

When you add a resource to your API, MCS creates a set of CRUD methods on these endpoints and constructs the JSON request and response schemas for you as well. To find out more about creating these schemas on your own, see Creating Resources with JSON Schemas, but if you want to see the ones that MCS creates for you, click **Export RAML** () to download a a RAML file, or toggle between the designer and the RAML document by selecting **Enter RAML Source Display Mode** ().



Creating An API

1. Click the side menu (☰), choose and then **APIs**.
2. Click **New API** and then choose **Express API**.



3. Complete the New Express API dialog by adding the API's name, its display name, and the description for the Service Catalog in the MAX Designer. When you're done, click **Create**.

   By completing this dialog, you open the Express API Designer. The Express API Designer defaults to its General page, where you can the change the API name or description. Now you're ready to add a resource.

4. Click **Resources** in the left navbar, then click **New Resource** to open the Create Resource wizard.

   > **Note:**
   >
   > When you click **New Resource** you create a top-level resource. This resource can't be selected as a child resource.

5. Describe your resource by adding a name, a display name, and a brief description. Enter a display name in plural form for the collection.

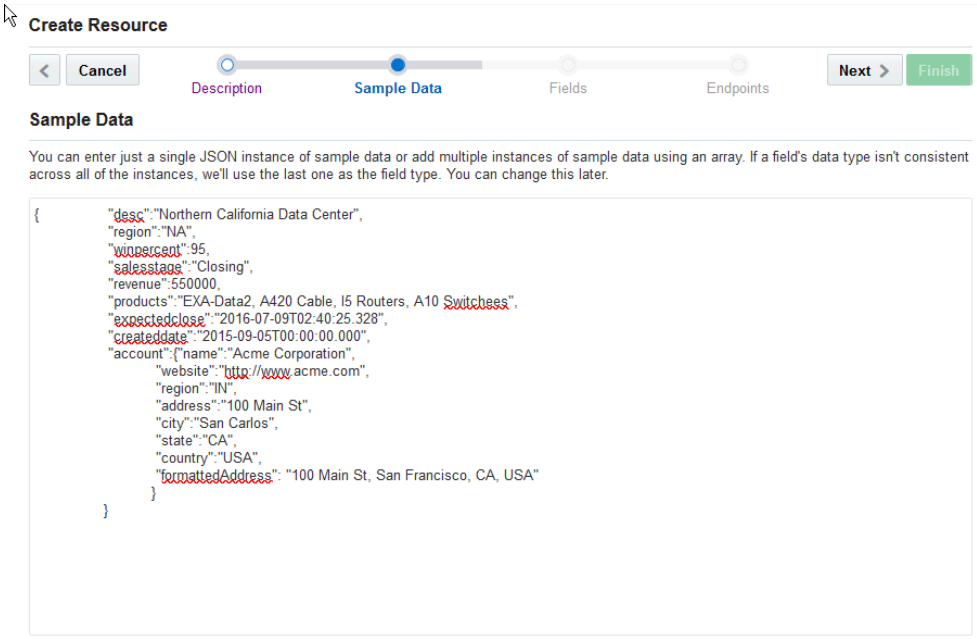
> **Tip:**
>
> The name and description that you enter here display in the Data Palette in MAX.

When you add a resource to your API, MCS creates a set of CRUD methods on these endpoints and constructs the JSON request and response schemas for you as well. To find out more about creating these schemas on your own, see Creating Resources with JSON Schemas, but if you want to see the ones that MCS creates for you, click **Export RAML** ( ) to download a a RAML file, or toggle between the designer and the RAML document by selecting **Enter RAML Source Display Mode** ( ).

Resources typically have two GET methods: one that returns a single item of an object, and one that returns multiple items (a collection). If you select **Also expose a collection of these resources**, MCS creates both GET methods and labels them Find and List, respectively. If your API supports create actions (POSTs), you need to add a collection.

Not all resources require both GET methods (or other methods that MCS creates for you, like POST, PATCH, and DELETE). You can remove any methods you don't want from the Express API Designer after you've finished creating the current resource.

6. Click **Next** and then add JSON arrays or instances of sample data in the Sample Data page. This is the mock data that helps you test the API. Within MAX, the mock data helps users visualize their app.



7. If you don't want to add sample data now, click **Finish** to exit the Create Resource wizard and go back to the Express API Designer. You can add fields and sample data from here later on. Otherwise, click **Next** to review the fields created from the sample data.

Click the **Sample Data** tab to review the sample date you previously entered. Don't worry if field names or labels aren't exactly what you want. You can edit all these fields from the Express API Designer after you're done creating the resource.

8. Click **Endpoints** and review all the methods created for you. When you return to the API Designer, you can select the methods that you want your resource to use.

9. Click **Finish** when you're done.

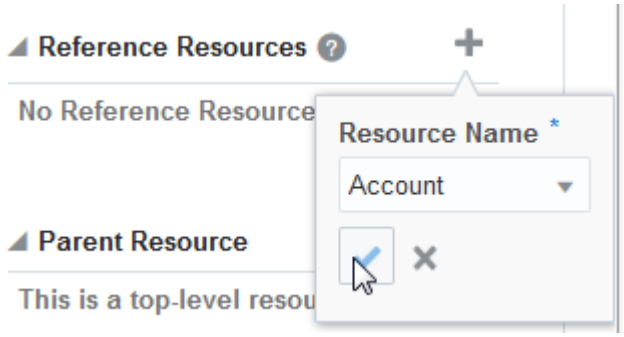
After you've created your resource, the Express API Designer opens so you can select the fields and methods you want to use to complete your resource. You can also shape request and response payloads for your methods. See Completing Your Resources.

To configure security for your API, export the RAML and then import it into the API Designer.

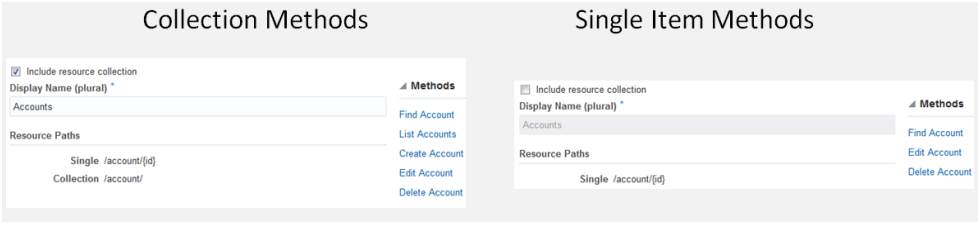
Completing Your Resources

When you click **Resources** from the Express API Designer navbar (or when you click **Finish** from the Create Resource wizard), you end up on the Overview tab in the Express API Designer, where you refine your resources by doing the following:

- Changing the resource's display name(s) and description.
- Creating reference or child relationships. You can learn more about peer and child relationships in Referenced Resources.



- Toggle the **Include Resource Collection** option to allow (or prevent) the return of multiple items from a collection. When you select this option, the General tab displays the methods available to a collection: List (GET /items) and Create (a POST call on a collection).



These methods display as hyperlinks that open pages for editing the method's requests and responses. Shaping Payloads tells you more about editing methods.

Adding Additional Fields

1. Click the **Fields** tab.

   For each resource, MCS creates a field called *id*. You can't delete this field, whose role is described in Fields.

2. If your resource needs more fields, click **New Field** and then complete the dialog by defining the field name along with the display name and description. If you use this API in MAX, the field names and descriptions that you enter here display in the Service Catalog.

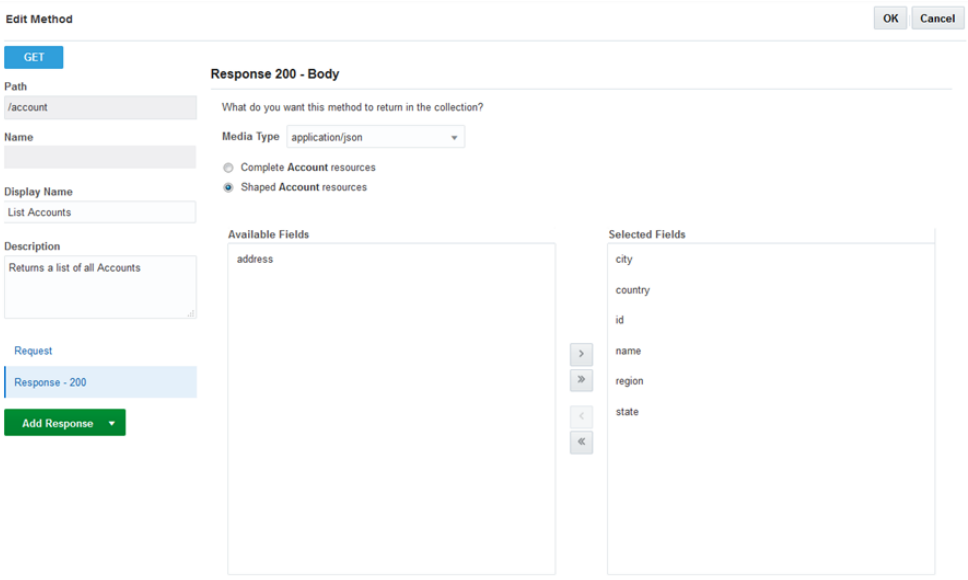
In addition to these display-related values, you also use this dialog to specify the format (string, integer, geolocation coordinates, and so on) expected by this field. By choosing the Reference field type, you can allow the field to reference the fields defined for a peer or child resource that's selected from the Reference Resource list. You can find out more in Fields.

Shaping the Payload for Your Resource

Once you've defined the fields for your resource, you're ready to select which fields are sent to, and returned from, the service. This is known as shaping the request and response payloads, which you can do as part of editing the methods.

1. Click a link in the **Methods** tab to open the **Edit Method** page.
2. Choose the request or a response type along with media type.
3. Click the **Shaped** option and move the fields you don't want to include in the payload from the Selected Fields window to the Available Fields window.

   By default, all of the fields are included in the payload. See Methods to learn about custom methods and payloads.



4. Click **OK** to save your changes.

See Shaping Payloads to find out about shaping data for different types of methods.

Adding More Sample Data

Use the Sample Data tab to add the mock data that helps you test your API. Mock data also guides MAX users as they map field data to their UI components. While MCS includes a row of sample data in the RAML document when you create fields manually for your resource, it may not reflect the data returned by your service. You can take a look at this sample data by toggling the RAML display mode option ( ). An array of MCS-generated sample data might look like this:

```
[
                {
                    "id": "id0",
                    "amount": "amount0",
                    "name": "name0",
                    "date": "date0"
                },
                {
                    "id": "id1",
                    "amount": "amount1",
                    "name": "name1",
                    "date": "date1"
                },
                {
                    "id": "id2",
                    "amount": "amount2",
                    "name": "name2",
                    "date": "date2"
                }
            ]
```

To get started populating your resource with sample data:

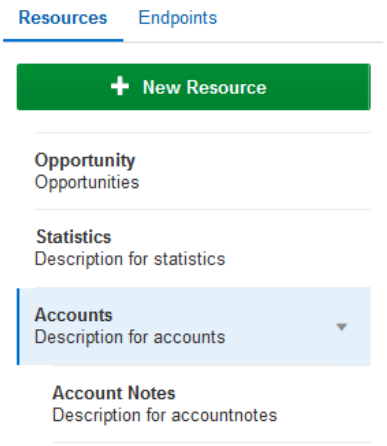
1. Click **New Row**.
2. Complete the Create Sample Data dialog.

   Because this template lets you enter sample values for all of the fields that you've defined for the resource, your sample data stays in step with the field schema definition.

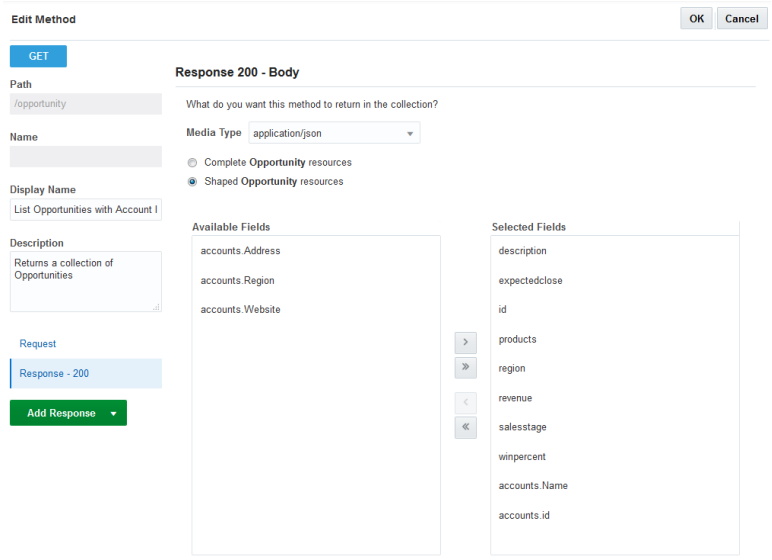
Referenced Resources

Your resources can reference each other as peers; that is, they occupy the same level. Suppose your API includes two resources that complement each other but are distinct. For example, an API that returns CRM (Customer Relationship Management) data might have two such resources: Accounts and Opportunities. The Accounts resource includes a set of fields that describe different facets of an account, like the company name and location. The information returned for these fields may relate to, but doesn't overlap, the information returned by Opportunities resource, whose fields return data that allow status meters to measure the opportunity's win percent. Your API might include resources that reference each other in a different way, as a parent-child relationship. The Accounts resource might have a subsidiary resource called

Account Notes, which is wholly dependent on the Accounts resource. If you deleted the Accounts resource, you'd delete the Account Notes resource along with it.



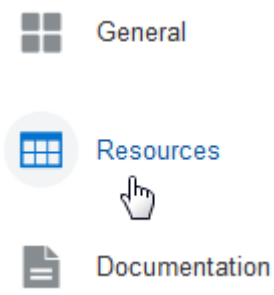
You can include the fields from a referenced resource in the payloads. When the Opportunities resource references the Accounts resource, for example, its payload for the Find Opportunities' 200 response includes *account.id* and other fields defined for the Accounts resource.



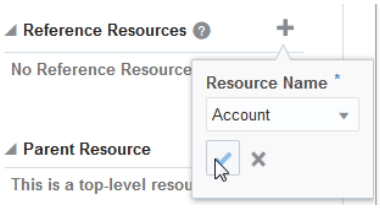
Referencing Resources

To reference a resource:

1. Click **Resources**.



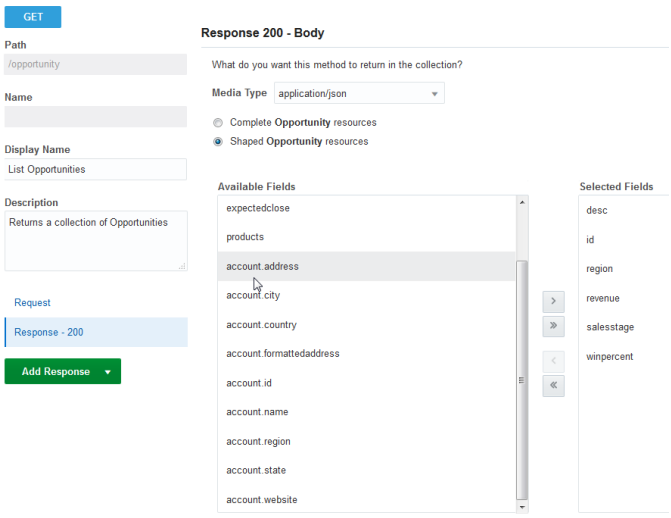
2. Click a resource.
3. Click **Add** (+) and then choose a child or a parent resource.



   To reference a child resource, first click **Add** and then complete the Create Resource dialog. MCS will create a set of method definitions for the child resource. Next, choose the child resource from the Resource Name list.

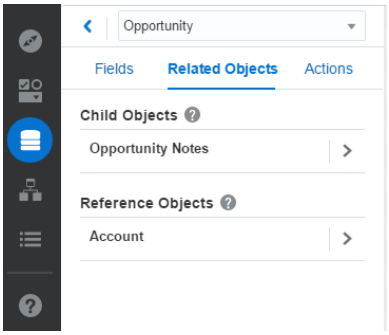
4. Click the Fields tab. MCS lists the resource with the fields. You can choose this resource (or other peer or child resources that you've reference in the API) for reference fields.
5. Click the Methods tab and then click one of the links to open the Edit Method page. By clicking **Response–200** in the Edit Method page, you can take a look at the referenced fields. Shaping Payloads describes these referenced fields, which are noted as *resource.field name* (like *accounts.region*, for example).

   The payloads for the POST and PATCH requests include the reference object itself, not its individual fields. There are no fields (referenced or otherwise) for either GET request because they don't include payloads.



6. Click **Save**.

After you've made your API available to MAX by publishing it, take a look at the MAX Designer's Service Catalog to see the various relationships between your resources.



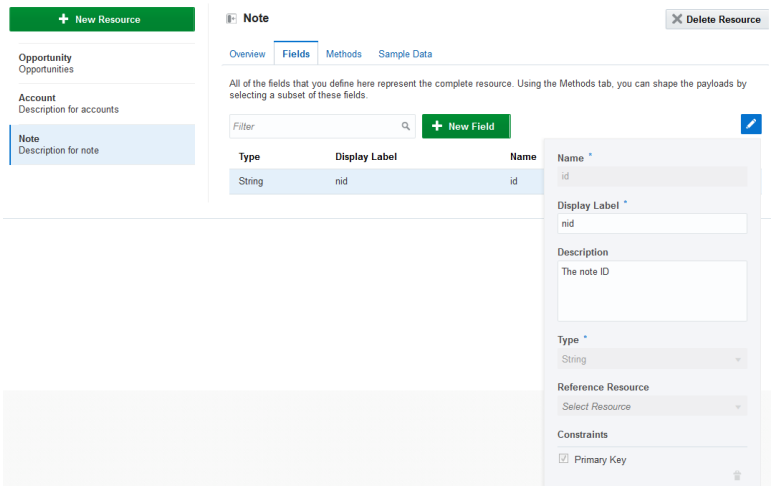
Fields

Fields describe the different aspects of a resource. They are like properties: they describe the data they hold by type (like a string, number, or reference) and format (date-time, URI, and so on). Fields can behave differently depending on context (or more specifically, on the payload definition).

> **Note:**
>
> The fields that populate list views in MAX are read-only, while the ones used in form-based create and update screens can accept user input.

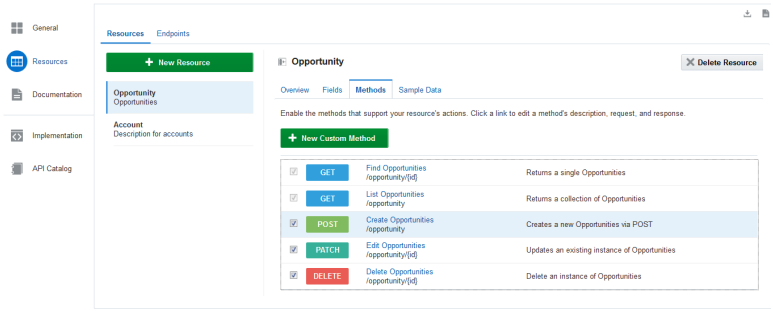
The Fields tab lets you take inventory of the fields for a selected object. It's where you can create a complete (or canonical) resource by defining all of the possible fields. After you've completed the resource, you can decide which methods can accept and return a subset of these fields by shaping the payloads in the Methods tab.

MCS adds the *id* field for you when you create a resource. Because of its role as a UUID (universally unique identifier), this field acts as the primary key. You can't delete this field, change its field type from a string, or change it from being a primary key, but by clicking **Edit** (), you can use the field editor to change its display name and description to reflect the resource.



Methods

MCS creates a set of CRUD (Create, Read, Update, and Delete) methods for you when you create a resource. Using the Methods tab, you can select from among these methods, add new ones, and shape the request and response payloads.

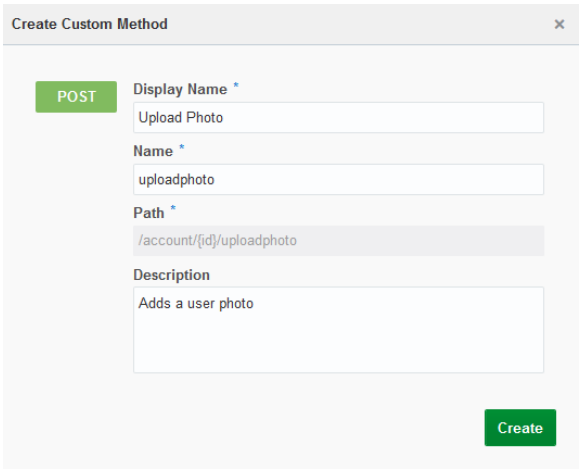


Selecting Methods

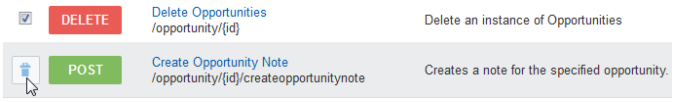
While all of the methods are selected by default, they may not all apply to your resource. You can select the CREATE, POST, or PATCH methods as needed, but because each resource needs at least one GET endpoint (or two if it's exposed as a collection), you can't remove the GET methods.

Custom Methods

Custom methods (which are always POST methods) allow your resource to perform a task or server-side action that falls outside of the functions enabled by the default set of CRUD methods. For example, you can define a custom method that enables an upload action on an Image component. Using the Fix-It-Fast app as an example, you could define an action to close an incident that's triggered by a swipe tile. Clicking **New Custom Method** opens the Create Custom Method dialog that lets you define a custom method on a nested resource (which MCS adds for you). After you've created the method, you can use the Edit Method page to shape the payload of its request body and add its responses for the 200 status code and the 500 status code. See Shaping Payloads.

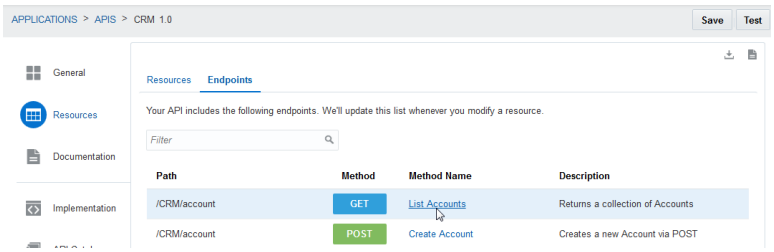


You can delete a custom method, but you can't delete any of the default set of methods that MCS creates for you.



Shaping Payloads

The Edit Methods page not only lets you change the method's display name and description, but also allows you to shape its request and response bodies by including, or excluding, the fields that filter the returned data and populate the create, update, list and detail screens. You can open this page by clicking the method links in the Overview or Methods tabs for a selected resource, or from the read-only list of all the methods defined for the APIs that display in the Endpoints tab.

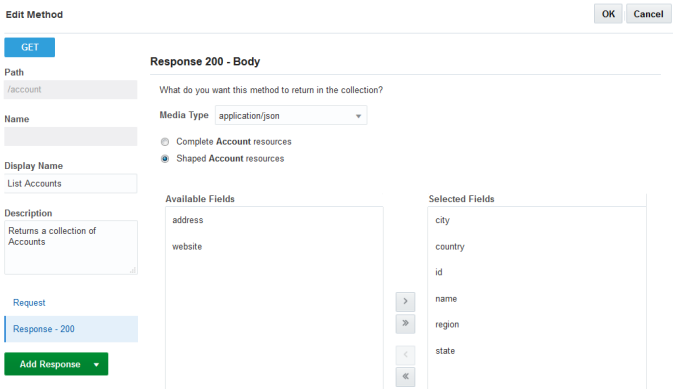


GET Payloads

There are no request bodies for GET methods; there are only response bodies. The Edit Methods page lets you select filtering criteria for the data returned for a list or a detail. In MAX, these surface as query parameters.

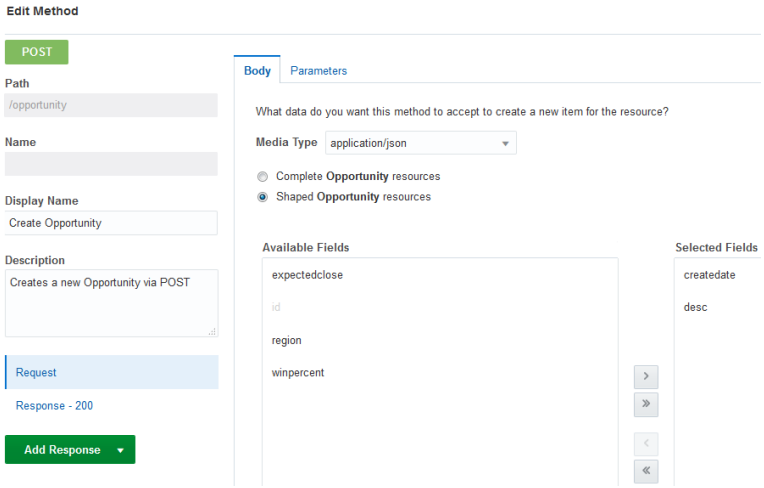
For each 200 response, MCS adds all of the fields that you created for the resource per the default option, **Complete**. While you can choose this option for detail screens, you might want to pare down the payload for a list screen by clicking the **Shaped** option. You can then shuttle the fields that you don't want from the Selected window to

the Available window. When the subset of fields in the Selected window suits your needs, click **OK**.



POST and PATCH Payloads

For POST and PATCH requests, you shape the payload with the fields that are sent to these methods to create or update an item.



Media Types for Request and Response Bodies

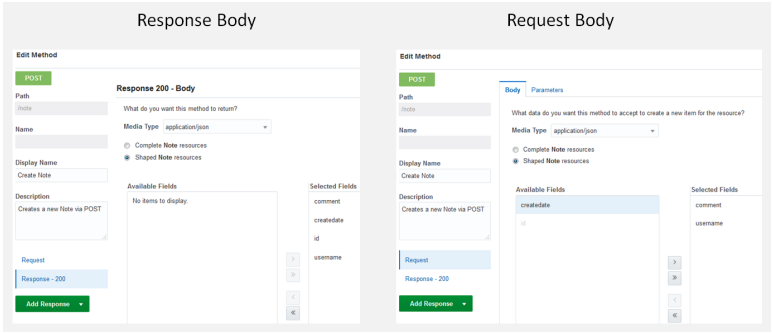
As part of the payload configuration, you can set the content type as **application/json**, **application/octet-stream**, or **image/***. For binary streams, choose **application/octet-stream**. See Enabling Uploadable Images .

Read-Only Fields

For POST and PATCH fields, you can create read-only fields by shaping the request and response bodies. By including a field in both the request and response payloads, you allow it to accept user input. By including it in the response body only, you confine the field to read-only display.

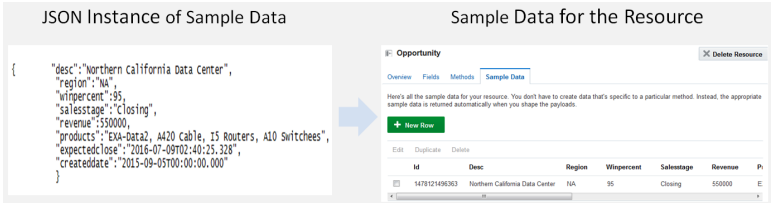
By default, MCS adds the ID field to the response body because this field typically holds a server-generated value that users shouldn't edit. Other than the ID field, there

may be other cases where your request and response bodies don't align. For example, to ensure that users can't inadvertently compromise the integrity of your data by updating the date field in an edit screen, you'd first add the field to the response payload's Selected window and then update the request payload by shuttling the date field from the Selected window to the Available window.



Sample Data

The Sample Data tab displays all of the data used by a resource for any purpose. In other words, the data is not specific to any method. As noted in Creating An API, you can add this data manually, or derive it from the instances and arrays of sample data that MCS uses to generate the both the resource's fields and the resource itself.



By adding a single JSON instance similar to the following, you can complete the resource by defining key-value pairs.

```
{               "desc":"Northern California Data Center",
                "region":"NA",
                "winpercent":95,
                "salesstage":"Closing",
                "revenue":550000,
                "products":"EXA-Data2, A420 Cable, I5 Routers, A10
Switchees",
                "expectedclose":"2016-07-09T02:40:25.328",
                "createddate":"2015-09-05T00:00:00.000"
                }
```

> **Tip:**
>
> Because MCS creates the `id` field for each resource, you don't need to include it your JSON.

MCS does more than just create fields from the JSON: it infers their data types as well. From the `"revenue": 550000`, key-value pair in the above sample, for example, MCS can interpret the field type as an integer rather than as a string.

You can create your top-level resources using this data-first approach. By nesting instances, you can create multiple top-level resources and establish reference relationships for them. The following example shows how nesting an instance creates a peer resource called *Account*:

```
{               "desc":"Northern California Data Center",
                "region":"NA",
                "winpercent":95,
                "salesstage":"Closing",
                "revenue":550000,
                "products":"EXA-Data2, A420 Cable, I5 Routers, A10
Switchees",
                "expectedclose":"2016-07-09T02:40:25.328",
                "createddate":"2015-09-05T00:00:00.000",
                "account":{"name":"Acme Corporation",
                           "website":"http://www.acme.com",
                           "region":"IN",
                           "address":"100 Main St",
                           "city":"San Carlos",
                           "state":"CA",
                           "country":"USA",
                           "formattedAddress": "100 Main St, San
Francisco, CA, USA"
                          }
               }
```

Using arrays, you can create top-level resources along with multiple rows of sample data:

```
[
                 {
                     "desc": "Anvils",
                     "region": "NA",
                     "winpercent": 30,
                     "salesstage": "appointment",
                     "revenue": "35000",
                     "expectedclose": "2016-07-09T02:40:25.328",
                     "account": {
                         "name": "Acme"
                     }
                 },
                 {
                     "desc": "Horns",
                     "region": "SA",
```

```
            "winpercent": 90,
            "salesstage": "closing",
            "revenue": 25000,
            "expectedclose": "2016-07-09T02:40:25.328",
            "account": {
                "name": "Road Runner"
            }
        },
        {
            "desc": "Bank Vaults",
            "region": "EU",
            "winpercent": 25,
            "salesstage": "prospect",
            "revenue": 15000,
            "expectedclose": "2016-07-09T02:40:25.328",
            "account": {
                "name": "Coyote"
            }
        }
]
```


> **Note:**
>
> You can only create top-level resources with sample data, so you can't add a child resource by nesting an array. Referenced Resources tells you how to add child resources.

As noted in Completing Your Resources, you can add or remove fields, or change the field display name and data type using the field editor. Because you need to define a value for each key, your resource's GET methods will always return a full set of data. In cases where this may not reflect real-world scenarios, you can edit your data using the Sample Data tab. To find out more, see Adding More Sample Data.

Using the Express API Designer with MAX

While the Express API Designer can help you jump-start your API development, it's also the quickest way for you to develop APIs for use with Mobile Application Accelerator (MAX).

MAX is a web-based development environment for mobile apps that caters to business users. Resources developed in the Express API designer can be treated as business objects that can be easily incorporated into MAX apps.

> **Tip:**
>
> You can learn more about the MAX App along with information on building, testing, and distributing apps in Designing Your App. If you want hands-on experience with using business objects to build a mobile app, follow the Create a Mobile App in Record Time with MAX! tutorial.

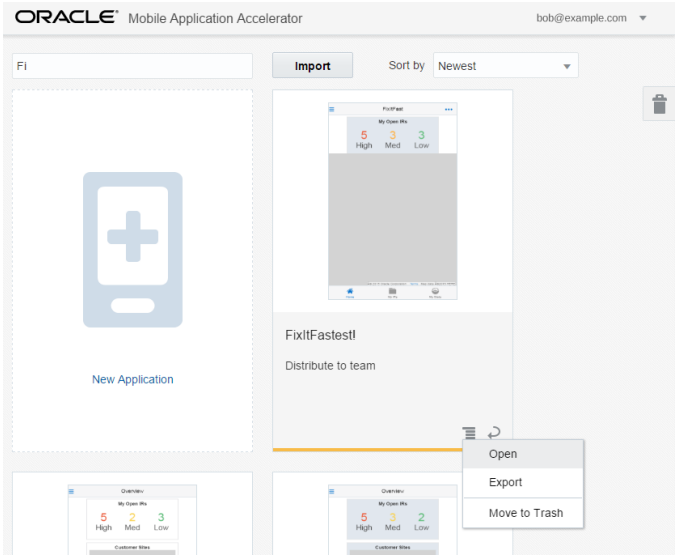
Who Uses MAX?

There are two types of MAX users:

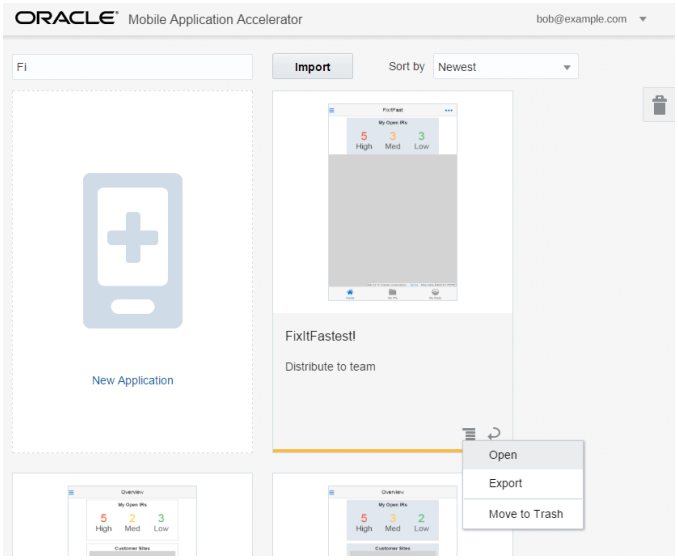
- Mobile Cloud developers (mobile app developers and service developers), who use MAX as part of their testing
- Business users, who create line-of-business (LoB) apps.

To create these apps, MAX users don't need to know platform-specific languages, nor do they even need to know anything about MCS in particular: a business user may be completely unaware that a mobile backend manages the app that he's building, or that a custom code API enables his app to use enterprise data.

These users access MAX in different ways: developers access MAX from within by clicking **MAX Apps** in the left navbar. Because they focus on building apps (rather than the backend services that these apps consume), business users access MAX directly after they log into . Unlike Mobile Cloud developers, business users are MAX-only users: they're granted the `BusinessUser` role, so they never see (and can't log into it).



These users access MAX in different ways: developers access MAX from within MCS by clicking **MAX Apps** in the left navbar. Because they focus on building apps (rather than the backend services that these apps consume), business users access MAX directly after they log into MCS. Unlike Mobile Cloud developers, business users are MAX-only users: they're granted the `BusinessUser` role, so they never see MCS (and can't log into it).

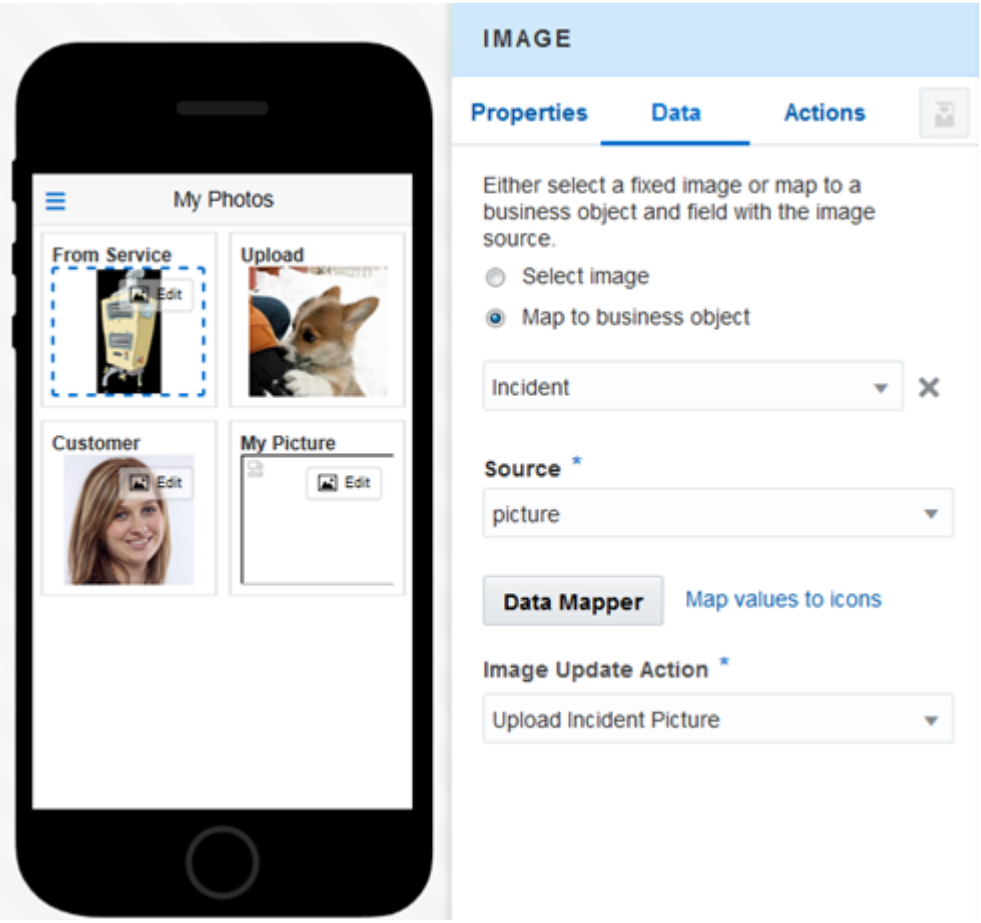


Enabling Uploadable Images

Users of MAX apps can upload images when the Image component is mapped to a business object that includes an upload action. You can add this action by creating a custom function for your business component, which is a POST method on a nested resource. To create this action:

1. Click **Add New Custom Method**. The path for this custom method points to a backend action. For example, the path for the POST might be something like/ `opportunity/{id}/uploadpicture`.
2. Because you're sending binary streams through this API, you need to select **application/octet-stream** as the media type for this method's request in the Edit Method page. This media type signals MAX that this action supports binary streams.
3. In MAX's Data Mapper, populate the Image component's Source field with the appropriate business object field.
4. To enable the action on the mapped field, clear the **Read Only** option in the Image component's Properties page. When you clear this property, MAX superimposes an edit overlay (

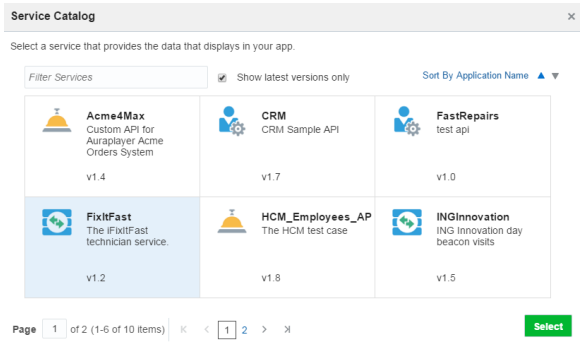
) on the image component in the Preview. It allows MAX to populate the Data tab's **Image Update Action** menu with actions that support binary streams.



Tips for User-Friendly Business Objects in MAX

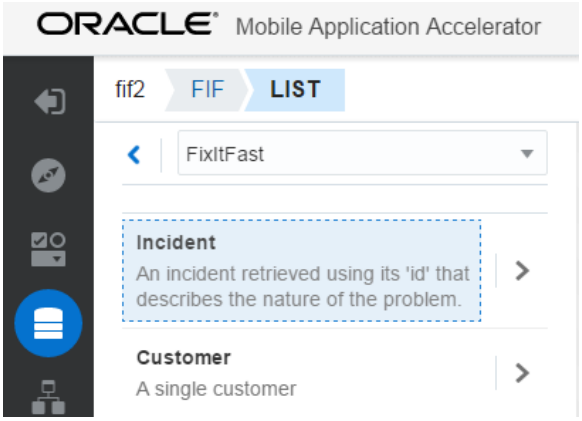
You can help business users pick services and map data by adding metadata in MCS.

...Is surfaced here in MAX
The service name and description in the Service Catalog:

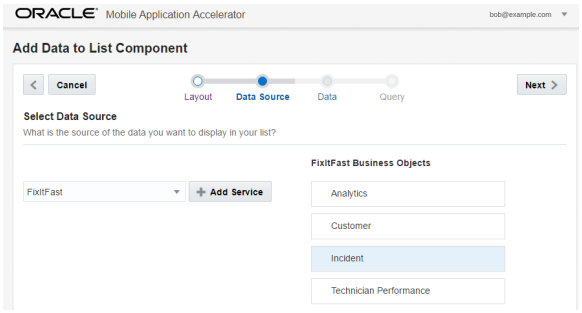


...Is surfaced here in MAX

- Business object name and description in the Data Palette:

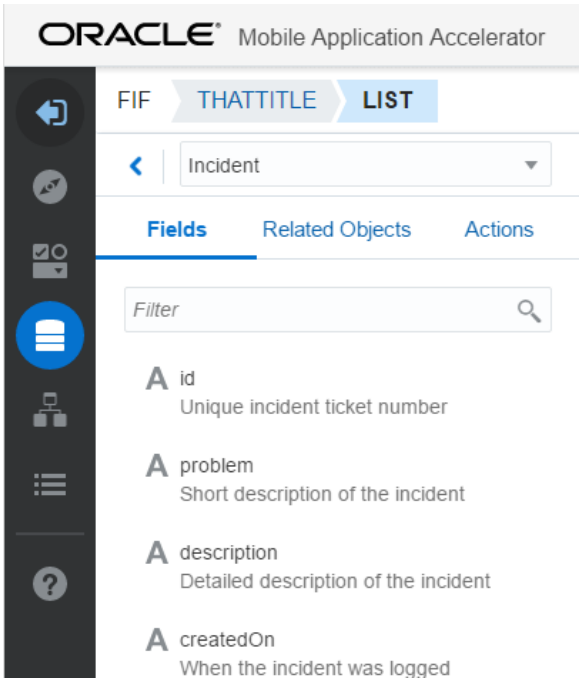


- The Data Source page of the Add Data QuickStart and the Data Mapper:

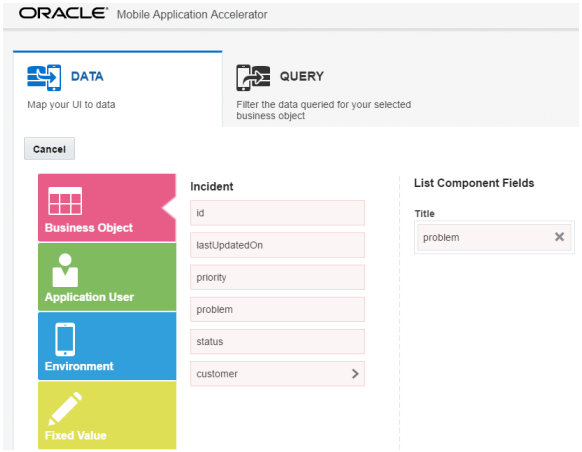


...Is surfaced here in MAX

- The field names and descriptions in the Fields tab of the Data Palette

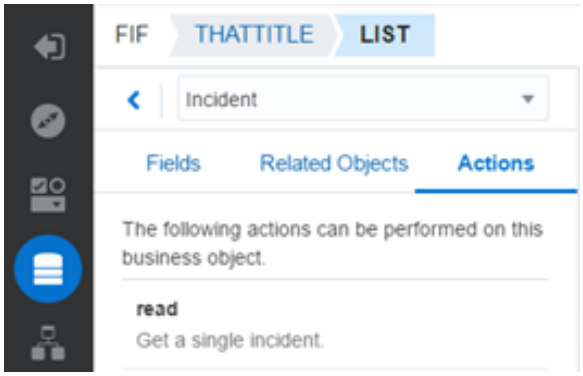


- The Data page of the Add Data QuickStart and the Data Mapper:

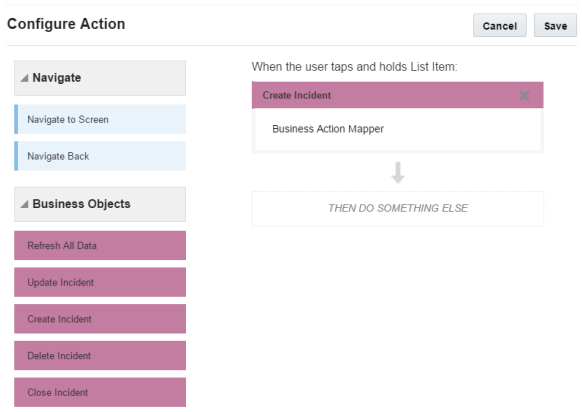


...Is surfaced here in MAX

The Actions tab of the Data Palette:

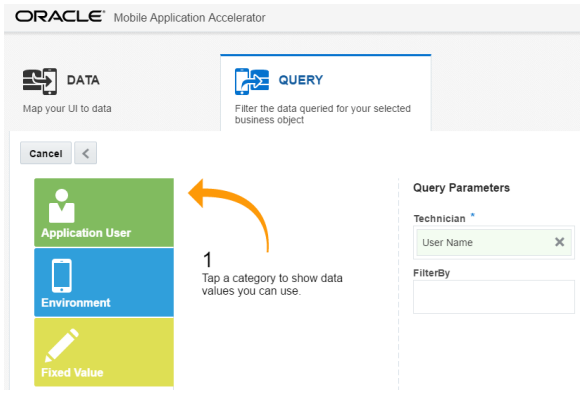


The Configure Action page of the Properties Inspector:

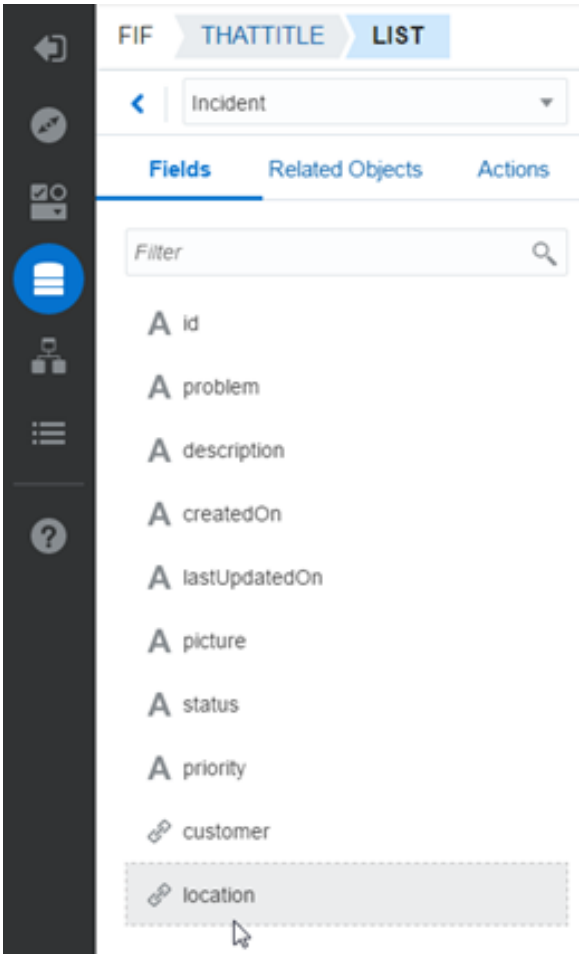


...Is surfaced here in MAX

The Query page of the Add Data QuickStart and the Data Mapper:

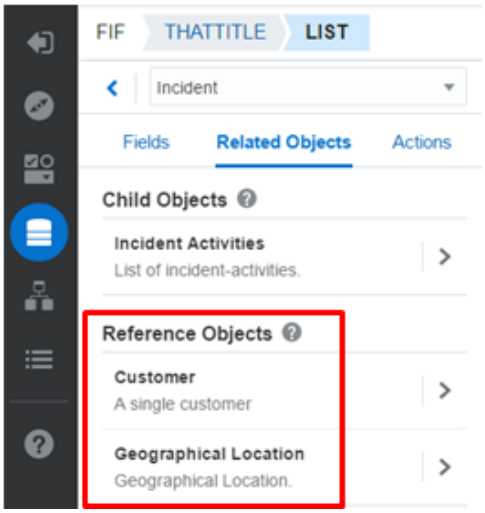


- The Fields tab of the Data Palette. Reference objects are identified with a chain link ( ).

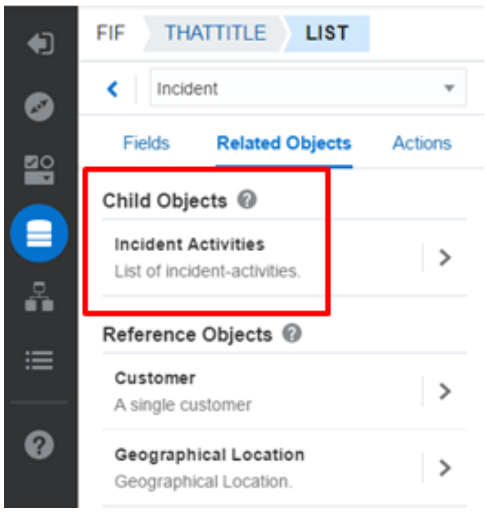


- The Data page of the Add Data QuickStart and the Data Mapper

...Is surfaced here in MAX
The Related Objects tab of the Data Palette (under Reference Objects):



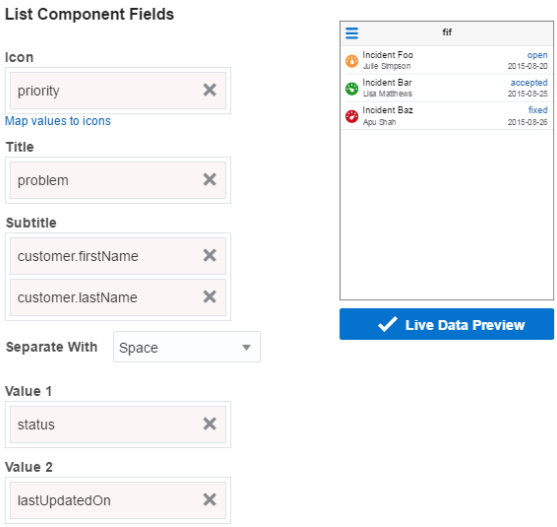
The Related Objects tab of the Data Palette (under Child Objects):



The Data Source page Data pages of the Data Mapper and the Add Data QuickStart for a detail screen.

...Is surfaced here in MAX

- The Live Data view for both the Data Mapper and the Add Data QuickStart:



- The Preview:

...Is surfaced here in MAX

Video: An Introduction to Mobile Application Accelerator (MAX)

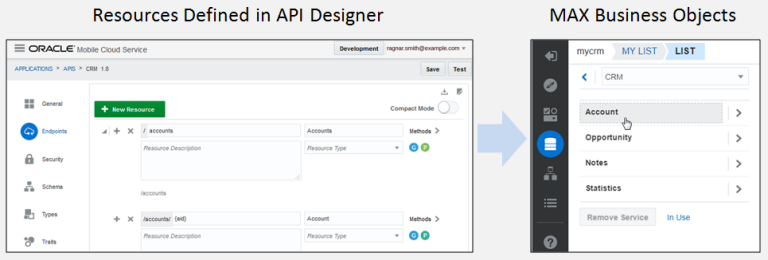
To see how you can build, test, and publish mobile apps using MAX, take a look at this video:

Video

Creating Resources with JSON Schemas

As an alternative to the Express API Designer, you can build an API with resources using the API Designer.

If you use the API Designer instead of the Express API Designer, you need to enable your API to surface in the MAX Designer by creating JSON schema definitions on its endpoints. These schema define the resources, their fields, and their methods. You can build these schemas from scratch, or you can import a RAML file (even the one generated by the Express API Designer). To get a comprehensive view of creating an API for MAX including adding JSON schemas, go through the tutorial, Shaping MCS APIs for MAX .



> **Tip:**
>
> Before you read on, take a look at the JSON schema specification.

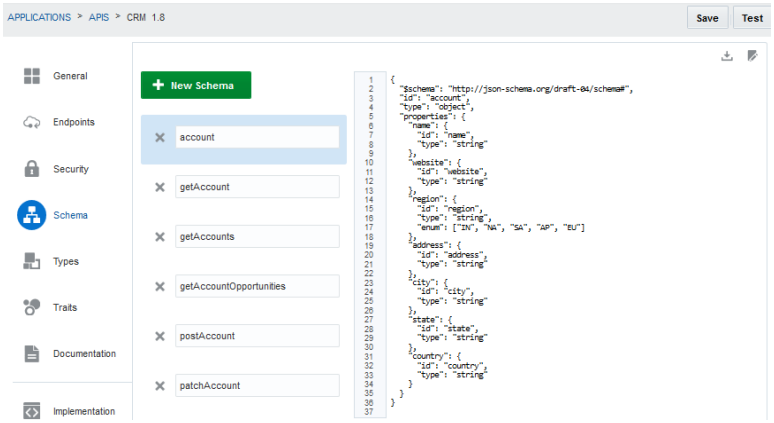
Defining Fields in a Schema

To create fields, you need to define JSON schemas for the endpoint requests and responses.

These schemas define the fields as property members, like `name` and `website` in the following example:

```
{
"$schema": "http://json-schema.org/draft-04/schema#",
  "id": "account",
  "type": "object",
  "properties": {
    "name": {
      "id": "name",
      "type": "string"
    },
    "website": {
      "id": "website",
      "type": "string"
    },
…
}
```

They also designate the kind of data that the fields can hold and the kind of user input and actions that they allow.



Defining Field Types, Formats, and Enums

Define the kind of data that your field holds by using combinations of the JSON schema `type`, `format`, and `enum` keywords.

Some things to keep in mind:

- Define enumerated values (enums) in the schema so that business users won't have to enter them as fixed values in the MAX Designer. For example:

```
"region": {
      "id": "region",
      "type": "string",
      "enum": ["IN", "NA", "SA", "AP", "EU"]
```

- When defining the field format for a date, we recommend UTC (Coordinated Universal Time):

```
"properties": {
          "lastUpdatedOn": {
            "type": "string",
            "format": "date-time",
            "description": "When the incident was last updated"
          },
```

Field Formats

You can add constraints on the values that users enter by adding validators like `required`, `minlengnth`, `maxLength`, `minimum`, and `maximum` to the property:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "id": "postOpportunity",
  "type": "object",
   "allOf": [
    {"$ref": "opportunity"}
  ],
  "required": [
    "desc",
```

```
    "region"
  ]
}
```

For fields that require input in a special format like a phone number, use the `pattern` keyword and then define a regular expression:

```
"pattern": "^(\\([0-9]{3}\\))?[0-9]{3}-[0-9]{4}$"
```

Example 20-1 Taking a Look at Properties in the JSON Schema

In the following example, a schema called *account* that defines of the base fields for a business object. Notice the `type` keyword defines the kind of data allowed in each field (`string`).

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "id": "account",
  "type": "object",
  "properties": {
    "name": {
      "id": "name",
      "type": "string"
    },
    "website": {
      "id": "website",
      "type": "string"
    },
    "region": {
      "id": "region",
      "type": "string",
      "enum": ["IN", "NA", "SA", "AP", "EU"]
    },
    "address": {
      "id": "address",
      "type": "string"
    },
    "city": {
      "id": "city",
      "type": "string"
    },
    "state": {
      "id": "state",
      "type": "string"
    },
    "country": {
      "id": "country",
      "type": "string"
    }
  }
}
```

For a base object, the properties don't include an ID (defined as `aid` in the following example). IDs aren't present when POST calls create records. Instead, the ID is assigned by the server. The following schema defines a field for the account ID called `aid`, which allows data to be returned by a GET call. In addition to the account ID, this schema allows all of fields defined for the `account` schema as well, because it includes the `allOf` keyword and assigns `account` as the pointer to the `ref` keyword.

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "id": "getAccount",
  "type": "object",
  "allOf": [
    {"$ref": "account"}
  ],
  "properties": {
    "aid": {
      "id": "aid",
      "type": "string"
    }
  }
}
```

Defining Child Objects

By defining a schema for a nested resource, you can create a child object. Unlike a reference (or peer) resource, a child object can't exist on its own. It only has meaning within the context of its parent resource.

The following schema defines a child object for the nested resource, `/accounts/{aid}/opportunities`. In this example, the canonical (or base) link returns the child object's resource (`opportunities`). The `links` keyword gives the location for the child resource, opportunities.

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "id": "getAccountOpportunities",
  "type": "array",
  "items": {
    "$ref": "getOpportunities"
  },
  "links": [
  {
    "rel": "canonical",
    "href": "/opportunities?aid={aid}"
  }]
}
```

> **Tip:**
>
> You can have different links defined in an array.

This example shows a schema on another nested resource, `/opportunities/{oid}/notes` to return the notes for a specific opportunity. In this case, the nested resources defines a grandchild object using the ID (`oid`) as part of the canonical link:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "id": "getOpportunityNotes",
  "type": "array",
  "items": {
    "$ref": "getNotes"
  },
  "links": [
  {
    "rel": "canonical",
    "href": "/mobile/custom/CRM/notes?oid={oid}"
  }]
}
```

Defining Fields for List, Details, Create, and Update Screens

Field behaviors can be described as *summary*, *creatable*, and *updatable*, that is, whether fields can accept user input, like those in a create or update screen, or appear as a read-only field in a list component.

These behaviors – and their related collection, create, read, update, and delete actions – are based on endpoints. By defining schemas for an endpoint's request and response, you tell MAX how it can use these fields to populate the different types of screens created by the QuickStarts.

Every business object needs at least one endpoint. Some might require more than one. For example, you can define GET and POST methods on a top-level resource (like `/employees`). Its GET method allows users to return all of the fields defined in the schema for the response. The schema defined for the POST method's request defines the fields that can be used to create an item. To return a specific item, define a GET method on a nested resource (`/items/{id}`).

> **Note:**
>
> In MAX, POST methods are always used for fields used for create actions. Read actions are always GET methods.

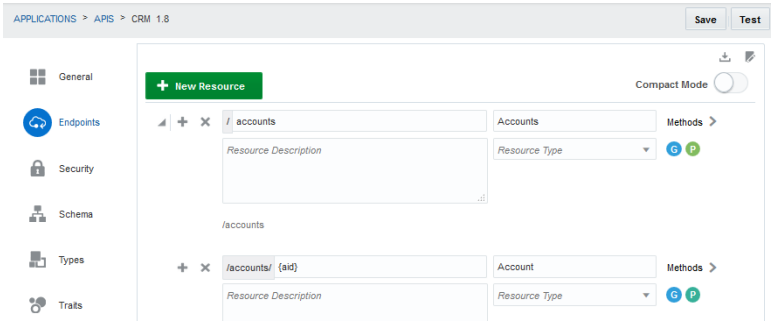
Field Behavior	Description	Used in These MAX Components	Method	Tips
Collection	Returns multiple items (or records) of the object. Calls GET on the collection resource (`/items`) to return all fields. See Collection Actions.	• List Components • DVT Components	GET	Specify the fields that you want to include in the schema for a collection endpoint. Add mock data field values for the request and the response.
Read	Gets a single item of the object. Calls GET on the item resource (`/items/{id}`) to return the properties for an item. An object can be a singleton, in which case this calls GET on the item resource (`/item`). See Collection Actions.	Detail Screen (read-only fields in a Form component)	GET	
Create	Creates a single item of the object. Calls POST on the collection resource (`/items`) with a request body that contains all of the creatable fields (which can be either required or optional), along with the user-provided values. This returns the new object with its new unique ID (which can be used subsequently in a read action). See Create Actions.	Create Screen (form fields)	POST	Specify the fields that should be included in Create screens in the schema. Add mock data field values in the request and response.

Field Behavior	Description	Used in These MAX Components	Method	Tips
Update	Updates a single item of the object. Calls PATCH on the item resource (`/items/{id}`) with one or more updatable properties. See Update Actions.	Edit Screen (form fields)	PATCH (and sometimes, PUT)	Specify the fields that users can update in the schema. Provide mock data for the field values for the request and response. You should consider using the PATCH method because it updates the server with only the fields that have been modified. See Using the PUT Method for Update Actions.
Delete	Deletes a single item of the object. Calls DELETE on the item resource (`/items/{id}`). See Delete Actions.		DELETE	

Collection Actions

Typically, collection actions are based on two different GET methods.

One endpoint returns a list of multiple items of the object using the top-level resource. The other returns a particular item and uses a nested resource. Together, these two endpoint definitions represent a single resource that supports both the collection and read actions.



This example shows a schema for the response for collection action. In this case it's a GET method on the top-level resource, `/accounts`.

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "id": "getAccounts",
  "type": "array",
  "items": {
        "properties": {
            "aid": {
              "id": "aid",
              "type": "string"
            },
            "name": {
                "id": "name",
                "type": "string"
            },
            "region": {
              "id": "region",
              "type": "string",
              "enum": ["IN", "NA", "SA", "AP", "EU"]
            },
            "city": {
              "id": "city",
              "type": "string"
            },
            "state": {
              "id": "state",
              "type": "string"
            },
            "country": {
              "id": "country",
              "type": "string"
            }
        }
  }
}
```

This example shows the schema for the response of a read action, defined for a GET action on a nested resource (`/accounts/{aid}`):

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "id": "getAccount",
  "type": "object",
  "allOf": [
    {"$ref": "account"}
  ],
  "properties": {
    "aid": {
      "id": "aid",
      "type": "string"
    }
```

```
  }
}
```

Defining a Collection Using a Single Resource

You can create a resource that returns a list of items using a GET endpoint on a single resource. In this case (which is more the exception than the rule), there isn't an additional endpoint for retrieving an individual item. In the following example, the Analytics resource has a collection action that returns a list of metrics (GET `/stats`). However, it does not use an endpoint that points to a specific resource (like GET `/stats/{sequence}`) to return an individual metric. The JSON response can be an array or an object. Objects include information about the data set, such as the number of items in the set, a token for the next set of items, and so on.

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "id": "stats",
  "title": "Analytics",
  "type": "object",
  "properties": {
    "metrics": {
      "type": "array",
      "description": "Metrics are individual measurements related to
incident activity, techncian performance",
      "items": {
        "type": "object",
        "properties": {
          "month": {
            "type": "string",
            "description": "Date Dimension for which a data point is
provided"
          },
          "technician": {
            "type": "string",
            "description": "Technician for whom the data is provided."
          },
          "radius": {
            "type": "number",
            "description": "radius in miles from the technician location,
where incidents were reported."
          },
          "incidentsAssigned": {
            "type": "number",
            "description": "Incidents Assigned to Technician"
          },
          "incidentsClosed": {
            "type": "number",
            "description": "Incidents Closed by Technician"
          }
        }
      }
    }
  }
}
```

> **Note:**
>
> MAX can only detect objects that have one top-level array. MAX can't detect the primary collection when an object has more than one top-level array like `metrics2` in the following snippet. In cases like this, the MAX can't make this collection available for data mapping.
>
> ```
> {
>   "count": 2,
>   "metrics1": [
>       {...}
>   ],
>   "metrics2": [
>       {...}
>   ]
> }
> ```

Create Actions

You can add a create action by defining a POST method.

You can define the creatable fields in the JSON schemas for both the POST request and response.

The following example shows a schema for the POST request called `postAccount` that defines creatable fields from the referenced `account` schema. Some of the fields returned from the account schema are optional, but in this schema, the `name` and `region` are designated as required fields; app users can't create a new item without defining them.

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "id": "postAccount",
  "type": "object",
   "allOf": [
    {"$ref": "account"}
  ],
  "required": [
    "name",
    "region"
  ]
}
```

In addition to these required fields, the `allOf` keyword allows app users to add values into any of the fields defined in the `account` schema (shown below) to create new items. While the `name` and `region` fields (which are also defined in the `account` schema) are required, the other fields are optional.

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "id": "account",
```

```
  "type": "object",
  "properties": {
    "name": {
      "id": "name",
      "type": "string"
    },
    "website": {
      "id": "website",
      "type": "string"
    },
    "region": {
      "id": "region",
      "type": "string",
      "enum": ["IN", "NA", "SA", "AP", "EU"]
    },
    "address": {
      "id": "address",
      "type": "string"
    },
    "city": {
      "id": "city",
      "type": "string"
    },
    "state": {
      "id": "state",
      "type": "string"
    },
    "country": {
      "id": "country",
      "type": "string"
    }
  }
}
```

> **Note:**
>
> In MAX, the POST method is the only way to enable create actions. Having a POST method enables MAX to populate create screens with fields that allow user input (creatable fields). If a business object doesn't have a POST method, then app users won't be able to create items.

Read Only Fields

To create read-only fields in a form, define fields in the JSON schema for the POST response that have no counterparts in the POST request schema. In the following table, the `getAccount` schema, which is defined for the POST response, includes the `aid` field, which holds the server-generated ID for an account. Because this is a read-only value, one which app users shouldn't update, it's not included in the field definitions of the POST request schema, `postAccount`, or the `account` schema that it references.

Response Schema	Request Schema

<pre>{
  "$schema": "http://json-
schema.org/draft-04/schema#",
  "id": "getAccount",
  "type": "object",
  "allOf": [
  {"$ref": "account"}
  ],
  "properties": {
  "aid": {
  "id": "aid",
  "type": "string"
  }
  }
}</pre>
	<pre>{
	  "$schema": "http://json-
	schema.org/draft-04/schema#",
	  "id": "postAccount",
	  "type": "object",
	  "allOf": [
	  {"$ref": "account"}
	  ],
	  "required": [
	  "name",
	  "region"
	  ]
	}</pre>

Content Types for Creatable Fields

At runtime, mobile apps return the content types specified in the POST endpoint, which can be `application/json` or `application/x-www-form-urlencoded`. You can specify `application/x-www-form-urlencoded` as the content type for a creatable field in the POST request, but also specify `application/json` as the content type for the read only fields returned by the response.

Update Actions

You can allow users to update a field's value by defining a JSON schema on a PATCH endpoint.

Schemas for PATCH endpoints enable MAX to populate edit screens (and other forms) with updatable fields. When forms are modified using PATCH, only the fields that users have updated are sent to the server, not the entire object.

> **Note:**
>
> When you define your PATCH endpoint, always specify the content in the request body as type as `application/json` instead of the JSON patch format (`application/json-patch+json`).

Using the PUT Method for Update Actions

In addition to the PATCH method, you can make fields editable by defining a JSON schemas for the requests and responses of a PUT method.

Although you can use both PUT and PATCH for update actions, keep in mind that the PUT method replaces all of the fields defined for a schema object (even if none of them have been modified). That means that the request payload must include the entire object. The request payload for the PATCH method, on the other hand, includes

only the fields that have changed. Because of this, we recommend using PATCH (if the service supports it, that is).

Delete Actions

The delete action is defined for an object. It enables users to remove an entire record, not just a field.

You can define a DELETE method on a nested resource like `/accounts/{aid}`, for example.

Custom Actions

In addition to the CRUD actions, resources can also have custom actions that require custom code, transactional semantics, or unique processing on the objects.

In general, custom actions don't return a payload. Instead, they perform server-side tasks and return success and failure responses.

Keep the following in mind when you create a custom action:

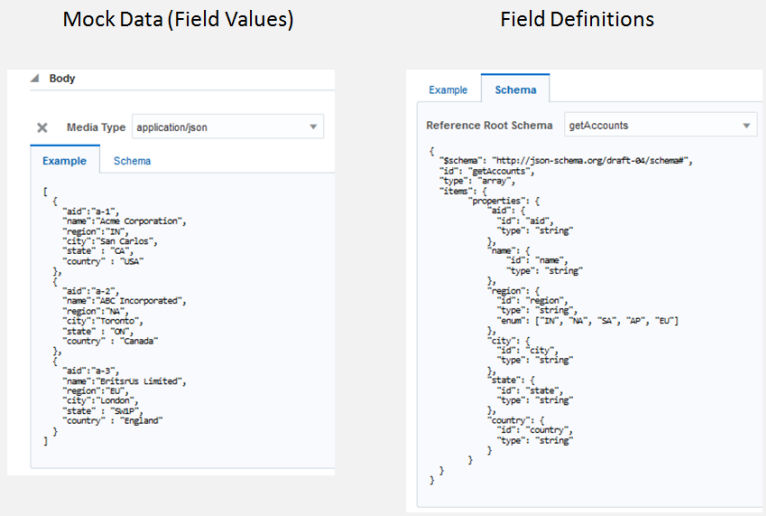
- Use POST methods for custom actions.
- Create the POST method for a nested resource like `/incidents/{id}/closeIncident`.
- If needed, define a request body for the POST method.
- Use a JSON hyper-schema `links` property to define the sub-resource. For example:

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "id": "incident",
    "title": "Incident Detail",
    "type": "object",
    "properties": {...},
    "links": [
      {
        "rel": "self",
        "title": "Incident",
        "href": "/incidents/{id}",
        "method": "GET",
        "targetSchema": {"$ref": "incident"}
      },
```

Creating Mock Data

Creating mock data for the fields defined in your JSON schemas helps you test the API. When you define these values, be sure that they align with the fields that you've defined in your schema.

> **Note:**
>
> Take care when you define your mock data, because MCS doesn't verify mock data against a schema.



Which API Designer Should I Use?

When creating your APIs, you can use either the API Designer or the Express API Designer. Which you choose boils down to a few important factors:

- If you want full control of the development process, choose the API Designer.
- If you'd rather get going fast with no coding, or you need to develop APIs to use with the Mobile Application Accelerator (MAX), the Express API Designer is your best bet.

This table highlights some of the key differences:

Category	API Designer	Express API Designer
Overview	Enables you to define custom APIs in a visual editor that gives you control over endpoint definition and security. You can also define a schema, resource types, and traits. You implement the API by writing a Node.js module.	Enables you to use sample data to quickly create APIs without writing any code. Based on the sample data you provide, the designer generates resources with GET, POST, PATCH, CREATE, and DELETE methods.

Category	API Designer	Express API Designer
Who's it best for?	• Developers who want to craft, or explicitly design, a custom API. • Developers who prefer working with the details, such as defining the method requests and responses, configuring a schema, and setting security The focus is on flexibility and control of the development process.	• Developers needing an API with only the basic CRUD operations (create, read, update or delete), who want to get up and running quickly. • Developers who want to jump-start their API designs before switching to the API Designer for fine-tuning. • Developers creating APIs for use with Mobile Application Accelerator (MAX). The focus is on speed, creating a spec to export to the API Designer for further development, and creating APIs to use with MAX.
Can use to set secure access?	Yes. You can add user authentication and role-based access to resources.	No. However, you can export the RAML to the API Designer and add role-based security settings with the tools there.
MAX Friendly?	Yes. But you must shape the API to surface in the MAX Designer by defining the JSON schema (one built from scratch, or a RAML file generated by the Express API Designer).	Yes. You create an API with an object-centric focus. This kind of API can be used out-of-the-box to build mobile apps with MAX.
Coding needed?	Yes. After you define the custom API's REST endpoints with the API Designer, you then need to implement internal logic through Node.js.	No, though you can modify the generated implementation.

21

Custom API Design

In Oracle Mobile Cloud Service (MCS), you can create custom REST APIs that can be used by your mobile apps. If you're a mobile app developer, use the API Designer to sketch out and test the endpoints that you define and then have a service developer fill out the details of the API (add resource types or traits, provide a schema, and set the access to the API and its endpoints), and implement it in JavaScript. If you're a service developer, use the API Designer to explicitly configure a complete API that you can test with mock data. Alternatively, you can generate custom APIs from a REST or Fusion Applications connectors without writing any code.

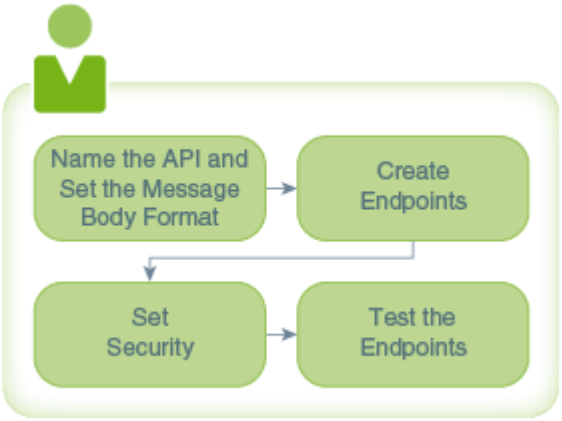
Unlike the MCS platform APIs, which provide a core set of known services, custom APIs let you use Node.js to code any service your mobile app needs, published through a REST interface. You can relay data by using an MCS connector to a backend service, which transforms complex data into mobile-friendly payloads. By using custom APIs to build a catalog of reusable services, you can save lots of time that might otherwise be spent periodically re-creating and maintaining implementation details in your mobile apps.

If you want to create an API quickly by providing sample data and letting MCS define a set of endpoints for you, use the Express API Designer.

API Design Process

The API Designer guides you through the process of creating a custom API.

You can quickly create a draft version of the API in just a few steps:



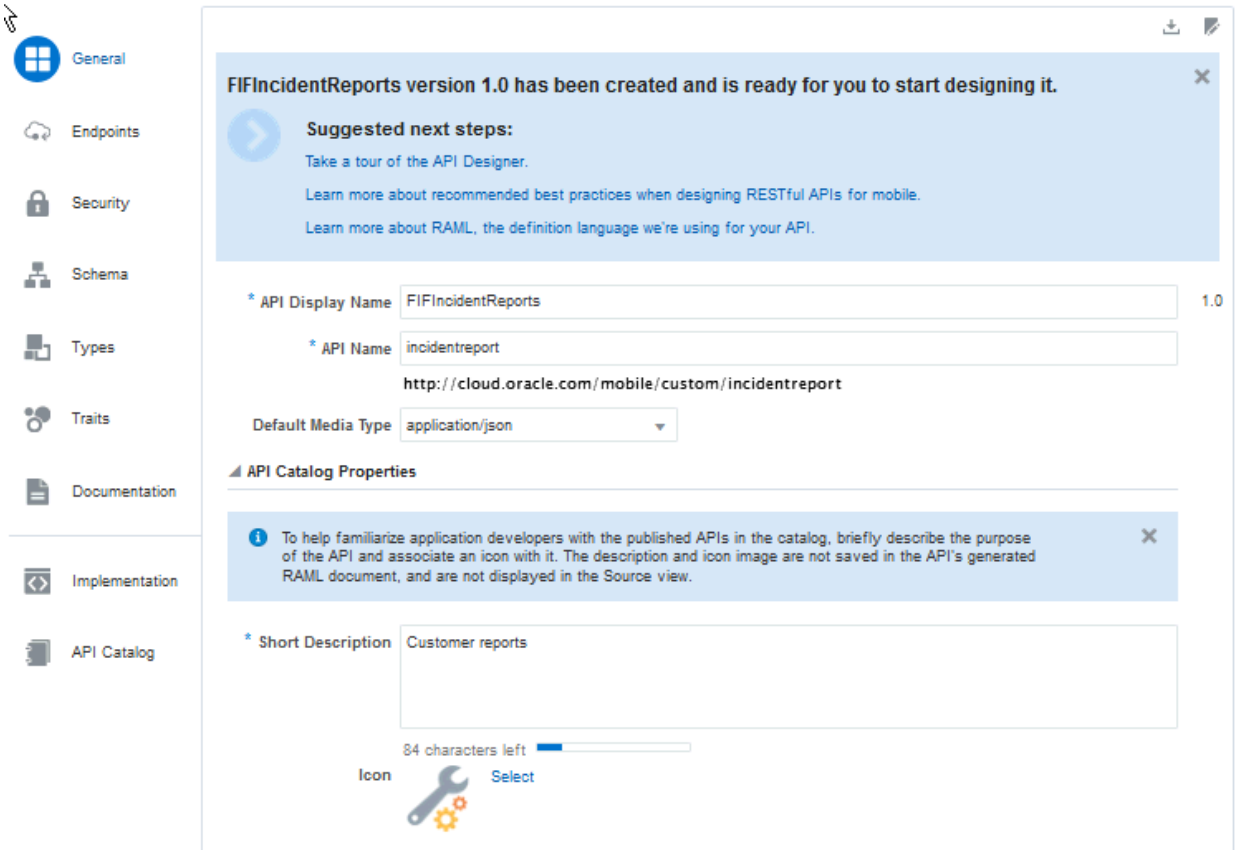
1. Add the basics (name of the API, the message media type, and a brief description).
2. Define an endpoint by setting a resource and at least one method for it.
3. Set access security.
4. Test your endpoint after you've defined at least one resource.

You can create mock data to quickly test and validate an endpoint even when you haven't completely finished configuring your API. When you define your message body, you can provide placeholder values to verify that the correct data is being sent or returned. See Testing API Endpoints Using Mock Data.

Completing Your Custom API

To fully complete your API, use the API Designer to help you add the essential components for a robust API:

- Provide the API metadata (that is, the basic attributes of the API, which are the API display name, API name, and short description) or, if you already have a RAML document that contains the configuration of your API, then you can upload it to the API Designer. All the information (metadata, resources, methods, and the schema for the message body) is extracted from the RAML document and loaded into the API Designer, letting you quickly proceed to testing your endpoints or editing your API configuration. To provide a valid RAML file, see RAML.
- Add one or more root and nested resources.
- Add methods to act on the resources.
- Create a schema to describe the body of data.
- Test your endpoints during design time with sample data and make any changes as needed.
- Allow anonymous access to your API or specify which roles can access it.
- Add documentation for your custom API



Later on, as you create more APIs, you might find that you are repeatedly defining the same methods, using the same parameters, etc. You can reduce the redundancy by creating resource types and traits. If your API is still in the draft state, then you can go back into your configuration and add the resource types and traits that you've defined.

The API Designer

The API Designer helps you configure a custom API with task-specific tabs that you use to name your API, define its endpoints, set security, add API documentation, add a schema, define resource types and traits, and test the API.

When you double-click an existing API, it automatically opens in the API Designer. Only APIs in draft state can be edited. If you open a published API, then it's displayed as read-only information. To make changes to a published API, you need to create a new version of it (see Creating a New Version of an API).

While you're configuring the API, you can switch between the Design view and the Source view. In the Design view (the default view), you enter values in fields. In the Source view, you manually define the API's properties in a source code editor. Click **Enter RAML Source Editor Mode** to toggle between the Design and Source views.

If you already have a RAML document, then you can import it and edit it in the API Designer. Click **Upload a RAML Document** or drag and drop your RAML document in the New API dialog to download your API definition.

> **Note:**
>
> If you came to the API Designer by clicking the **APIs** navigation link from a mobile backend, the feature to upload a RAML document is not available.

MCS APIs are based on the RESTful API Modeling Language (RAML) standard. Once you've begun to configure your API, MCS generates a RAML document of the configuration. See RAML to learn more about it.

If you want to work on the RAML document outside of MCS, you can export it by clicking **Export RAML document** at the top of the page.

Generating Custom APIs for Connectors

Oracle Mobile Cloud Service (MCS) can generate custom code from connectors to connect to external services. As a service developer, you can select a Fusion Applications connector or a REST connector that has been created with a valid descriptor, generate the custom API, and use the generated API to make it easier to call these services from the implementations of your custom APIs, or directly from a mobile app.

A connector is a means of enabling a mobile backend to communicate with an external service such as enterprise system or third-party APIs, which in turn, allows a mobile app to interact with the functions of that service. A connector API is a configuration for communicating with a specific external service to send and receive data.

As a service developer, you can generate a custom API that exposes the methods of a connector API and provides a default implementation, without writing code.

The custom API is generated with an endpoint for each resource in the connector API, and it is opened in the API Designer for you to continue to specify details of the API, such as roles. The default implementation, passes through all the requests coming from the generated custom API to the target connector API, is also generated and assigned to the generated API. As soon as you have assigned roles to the API if they are required for security on the connector you can use the implementation to test the API. You can download and modify the implementation and then upload it.

Creating a Generated Custom API for a Connector

Being able to create a custom API for a connector means that it is much easier to create a prototype which you use to test a connector. As you find things you want to change, you can quickly make a change to the connector, and generate a new custom API and implementation. Once you are satisfied you can generate a final version of the custom API and implementation.

- First, you develop a REST connector or Fusion Applications connector that is defined using a descriptor.
- Generate the custom API from the connector. It opens in the API Designer, where you can define one or more roles or specify the authentication required by the API.
- You can immediately call the generated API from the mobile device. The default implementation passes through all the requests coming from the generated API to the target connector API.

- You will probably want to download the implementation and modify it to shape the data returned.
- You may want to revisit the connector and make changes to the connector resources or descriptor. If you do you must generate a new custom API and implementation. If you make changes to the generated custom API, these changes are not reflected in the connector. You should make the appropriate changes in the connector and then generate the custom API and implementation again.

Limitations of Generated Custom APIs for Connectors

You can only generate a custom API for a REST or Fusion Applications connector which is defined using a descriptor. You cannot generate a custom API for another type of connector, or where the REST or Fusion Applications connector does not have a descriptor.

If you want to send multipart form data or use the `http` options object, you might need to replace the `callConnector` method in the implementation with your own code. See Calling Connector APIs from Custom Code.

How Do I Generate a Custom API from a Connector

Before you can generate your custom API, you must have created the connector that the API will be configured for. If the connector isn't valid you'll see a popup explaining that you can only generate custom connector API code for:

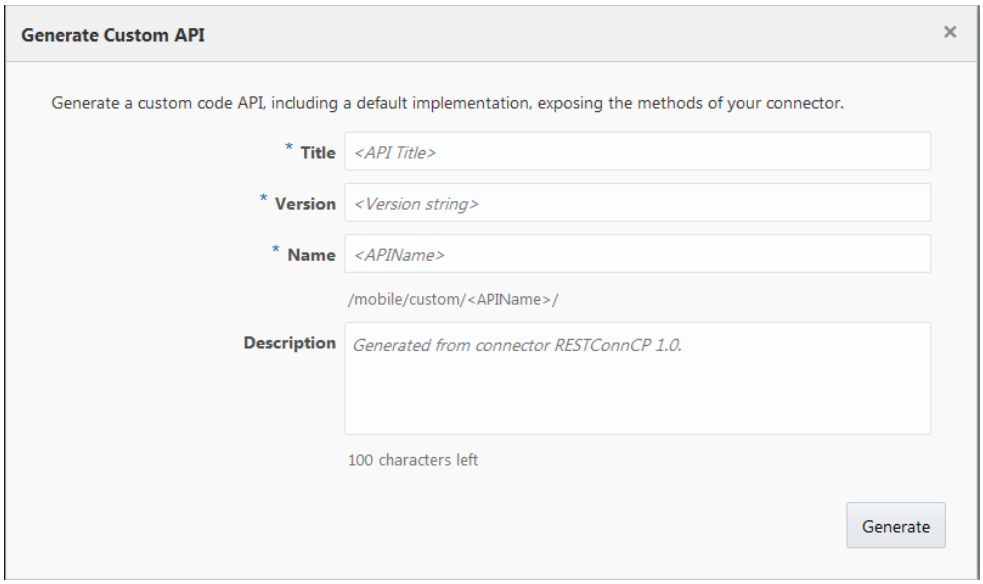
- REST connectors that use a descriptor URL
- Fusion Applications connectors

> **Note:**
>
> Make sure that you have the descriptor defined for the connector, and that you have selected the resources and methods you want to generate code for. The connector should be as complete as possible

1. Make sure that you're in the environment for which you want to generate the custom API.
2. Click  and select **Applications > APIs** from the side menu.

   The Connectors page appears. Select the connector API you want to generate custom code for. You can filter the list to see only the connector APIs that you're interested in or click **Sort** to reorder the list.
3. Click **More** and from the drop-down list, select **Generate Custom API**.

   The Generate Custom API dialog appears.



4. Provide the following information for the generated custom API:

   a. **Title:** Enter a descriptive name (an API with an easy-to-read name that clearly identifies the API makes it much easier to locate in the list of custom APIs).

      For example, `myCustomAPI`.

      > **Note:**
      >
      > The names you give to a custom API (the value you enter in the API name field) must be unique among custom APIs.

   b. **Version:** Enter a version number.

      If you enter a version number that already exists, you'll get a message letting you know that number is already in use.

   c. **Name:** The title you entered is automatically entered here as the name. You can change it if you want. This name is used a unique name for your custom API.

      By default, this name is appended to the relative base URI as the resource name for the custom API. You can see the base URI below the Name field.

      > **Note:**
      >
      > The custom API name must consist only of alphanumeric characters. It can't include special characters, wildcards, slashes /, or braces {}.

      If you edit the name for the API here, the base URI is automatically updated.

      Other than a new version of this custom connector API, no other custom connector API can have the same resource name.

   d. **Description:** You can accept the default description, or provide a brief description, including the purpose of this API.

After you've filled in all the required fields, click **Generate**.

The draft API is generated and displayed in the General page of the API Designer (see The API Designer) where you can continue to edit it.

You can find the new custom connector API listed under **Applications > APIs**.

Completing the Custom API

The generated API opens in the API Designer.

- An endpoint exists for all the resources selected in the connector, along with an implementation that you can use to test the API.
- By default, security is set that login is required and security is enterprise level so you need to add the roles that can access the API. See Security in Custom APIs

As soon as you assign appropriate roles, you can test the custom API.

Working with the Implementation

The default generated implementation passed through all requests. You can edit the implementation to shape the data returned, which is useful if there is a lot of data.

1. Make sure that you're in the environment where you can download the implementation.
2. Click  and select **Applications > APIs** from the side menu.

   The APIs page appears. Select the custom API that you have generated. You can filter the list to see only the custom APIs that you're interested in or click **Sort** to reorder the list.
3. Click the **Implementations** navigation link, select the implementation which will have the same name as the custom API, and click **Download**.
4. The download is a zip file with the default name `<custom-api><version>.zip`. Expand it to a suitable location. The implementation files are:
   - `callConnector.js`, passes the client's request to the connector, and sends back the connector's response.
   - `<custom_api>.js` , provides the main body of the scaffolding of the custom API implementation. You can uncomment lines in this to shape the data returned from the connector.
   - `<custom_api>.raml`, the RAML definition of the custom API.
   - `package.json`, the package descriptor file.
   - `ReadMe.md`, has a description of the implementation files.
   - `samples.txt`, code samples.
   - `swagger.json`, the Swagger definition of the custom API.
   - `toolsConfig.json`, used by the command-line development tools.
5. In an appropriate editor, open `<custom_api>.js`, which is the only file in the generated implementation which you should edit.

To shape the response from the connector, uncomment the relevant lines and if necessary change the `type` and `limit`. See the `service.use` examples in the sample of `<custom_api>.js` below.

```
service.use(bodyParser.raw({type: 'application/octet-stream', limit:
'100mb'}));
```

and

```
service.use(bodyParser.text({type: 'text/*', limit: '1mb'}));
```

This is the first few lines of the `<custom_api>.js` generated implementation file.

```
// no need to add body-parser as a dependency in package.json - it's
provided by custom code container
var bodyParser = require('body-parser');

// passes client's request to the connector, sends back connector's
response
var callConnector = require('./callConnector.js');

/**
 * Mobile Cloud custom code service entry point.
 * @param {external:ExpressApplicationObject}
 * service
 * @see {@link http://expressjs.com/en/4x/api.html}
 */
module.exports = function(service) {

// uncomment if using customizer to customize binary request with
content-type 'application/octet-stream' - it will be parsed into a
Buffer and assigned to req.body. Otherwise these requests streamed
through (recommended approach if no customization is required).
//service.use(bodyParser.raw({type: 'application/octet-stream', limit:
'100mb'}));
// uncomment if using customizer to customize text request with text
content-type - it will be parsed into a string and assigned to
req.body. Otherwise these requests streamed through (recommended
approach if no customization is required).
//service.use(bodyParser.text({type: 'text/*', limit: '1mb'}));

// In the product UI, in Diagnostics -> Logs tab, ServerSetting button
allows to set backend log level: set your mbe log level to FINE (FINER,
FINEST) to see the generated custom code sdk calls.


    service.post('/mobile/custom/sample_api/emps', function(req,res) {
        // uncomment customizer to customize request and/or response
        callConnector(req, res/*,customizer*/);
    });

    service.get('/mobile/custom/sample_api/emps', function(req,res) {
        // uncomment customizer to customize request and/or response
```

```
            callConnector(req, res/*,customizer*/);
        });

    ...
```

There is a sample customizer in the same generated implementation file. You can edit it and pass it as a last parameter to `callConnector` to override the request sent to the connector and/or the connectors response. See the comments in the code for examples of what you can do.

```
// Edit this sample customizer and pass it as a last parameter to
callConnector to override request sent to connector and/or connector's
response.
// Without customizer callConnector streams request to connector, then
connector's response is streamed back to client - recommended approach in
case no customization is required.
var customizer = {
    // allows to customize request sent to connector. If omitted then the
request streamed to the connector - recommended approach in case no
request customization is required.
    request: {
        // used - with post and put only - to customize request body
        // If not specified then request body is streamed directly to the
connector - no need to define this function unless you need to override
the payload.
        body: function(req) {
            console.log('customizer.request.body: req.body = ', req.body);
            var body = req.body;
            // OVERRIDE request body here - substitute this sample code:
            if (typeof body == 'string'){
                // to enable string parsing uncomment
service.use(bodyParser.text... - otherwise req.body would never be a string
                body += ' customized request';
            } else if (typeof body == 'object'){
                if (Buffer.isBuffer(body)){
                    // to enable binary parsing uncomment
service.use(bodyParser.raw... - otherwise req.body would never be a Buffer
                    body = Buffer.concat([Buffer.alloc(8, '00000000'),
body]);
                } else {
                    // json parsing is enabled by default
                    body['customized-request'] = true;
                }
            }
            console.log('customizer.request.body ->', body);
            return body;
        }/*,
        // advanced: uncomment to add options to connector request, see
https://github.com/request/request#requestoptions-callback
        options: function(req) {
            var options = {headers: {myHeader: 'myHeaderValue'}};
            console.log('customizer.request.options ->', options);
            return options;
```

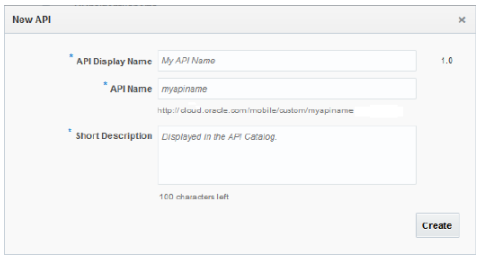
```
    }*/
},
```

Spec Out a Custom API

As a mobile developer, you might want to quickly spec out an API for your backend then configure it later, or hand it to someone like the service developer to complete. You can construct a functioning API with just a few steps: name your API, define an endpoint, and test the endpoint. These next steps use a simplified FixItFast example. It doesn't show you how to add method parameters, or schemas, or resource types and traits.

1. Make sure that you're in the environment containing the mobile backend for which you want to create a custom API.
2. Click  and select **Applications > Mobile Backends** from the side menu.
3. Select the mobile backend that you want to associate the API with from the list of backends and click **Open**.
4. Click the **APIs** navigation link.
5. Select **New API > API**.

   The New API dialog opens. Here's where you enter the basic information for your API:



   a. Enter a name in the **API Display Name** field that is easy to read and describes your API. For example, `FixItFast Incident Reports`. This name appears in the API Catalog, which other developers can see.

      The name you give to a custom API (the values you enter in the API Display Name and the API Name fields) must be unique. No two custom APIs can have the same name.

   b. Enter a name in the **API Name** field for the internal name of the API. It's part of the metadata of the API, that is, it appears in the custom API URI. It won't appear in the API Catalog, so you can use a more concise form of the display name if you choose. For example, `incidentreports`.

   c. Add a brief description that tells others what the API does.

6. Click **Create**.

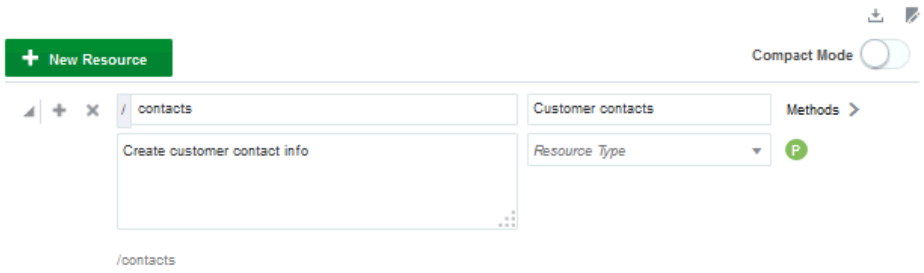
   The General page of the API Designer is displayed. If you want to change the name of your API or its description, then you can do it here.

7. Select the default media type, that is, the content type of the message body. REST APIs commonly use the `application/json` or the `application/xml` media type.

That's all you need to do to set the basic information for your API. If you'd like, you can choose a different icon to associate with the API display name or just go with the default and select a different icon later.

8. Click **Endpoints** in the navigation bar to define endpoints for the API.

   a. Click **New Resource** and enter the resource name and the display name of the resource (the field next to the resource name field). For instance, you could have `contacts` as the resource name and `Customer contacts` as the display name. Resources are listed by their display names on the left side of the API Test page. Enter a brief description of the resource so others can understand what the resource does.



   Tip: This image shows a "P" under the **Methods** link. When a method is defined for an endpoint, an icon for the method appears below the **Methods** link. The icons are a shortcut you can use later to quickly see what methods are defined for the resource and you can go directly to the method definition by clicking on an icon.

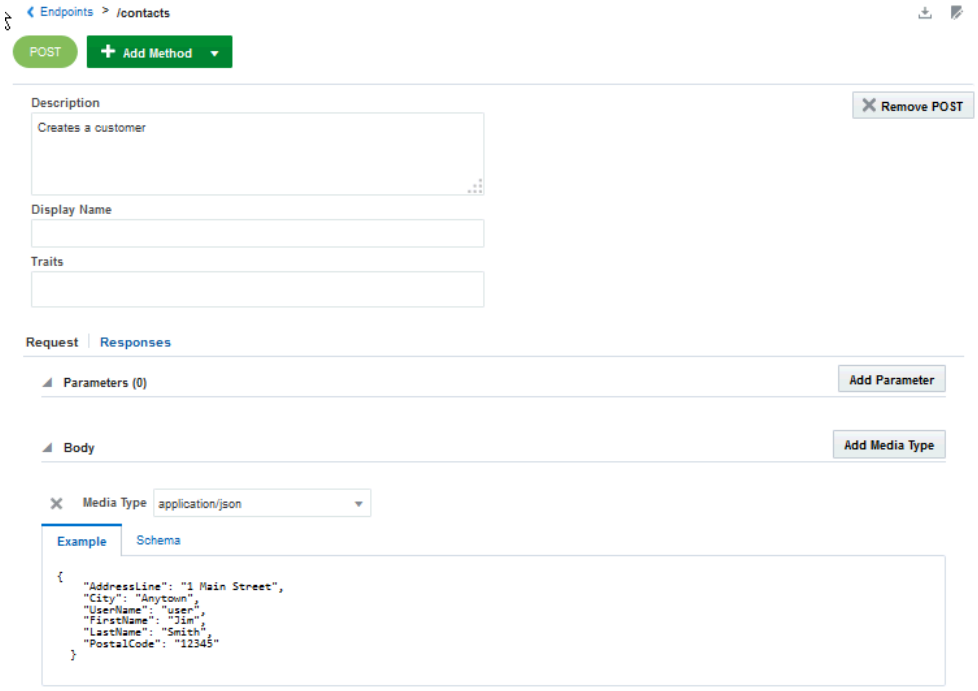
   If you want to add another top-level resource, then click **New Resource** again and enter names and descriptions.

   b. (Optional) If you want to add a nested resource (a child resource of `contacts`), click **Add** (+) next to the **Resource** name field. Enter a name, a display name, and a description of the nested resource. Click **Add** (+) again to add more nested resources if you need them.

   Endpoints are what really define an API. They are the resources and the methods that act on those resources.
   If you want to know more about resources, see API Resources.

9. Click **Methods** next to the resource display name and define a method for the resource.

   For each method, you need to define a request and a response. You can add parameters to filter the information for the request and response message bodies if you need them.



a. Click **Add Method**, select an operation and, optionally, add a description of the method in the Description field.

   For example, you could select a `POST` method to create a customer and add "Creates a customer" as the description. Notice that a POST icon appears next to **Add Method**. All methods defined for a resource have icons displayed at the top of the page. When you want to view or edit a specific method, just click the icon for it.

b. Click **Add Media Type** and select the format of the request message body, which is usually JSON or XML.

c. Add a schema (a template of the message body) or an example of the message body using mock data. Click **Example** or click **Schema** to paste the message body.

   Here's an example body you could use for the FixtItFast example:

```
{
  "AddressLine":"1 Main Street',
  "City":"Anytown",
  "UserName":"user",
  "FirstName":"Jim",
  "LastName":"Smith",
  "PostalCode":"12345"
}
```

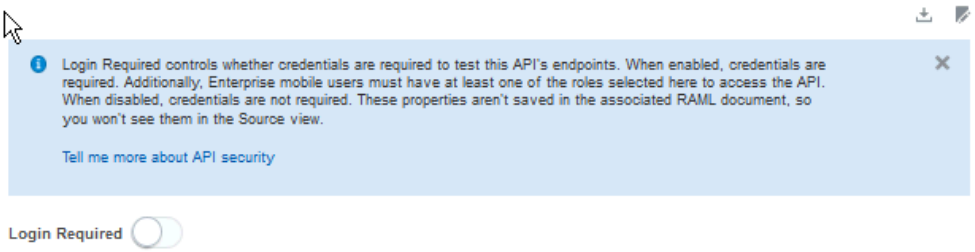
d. Add a response body by clicking **Add Response** and selecting a response code. Don't forget to add a description for the response body.

   Using the example, you would select **201 — Created** for the `POST` method and enter the following description: **Request fulfilled, new customer added.**

You can add parameters to filter information for the response body. You can also enter a response message body. If you're using the FixItFast example, then a response body isn't needed for the `POST` method.

e. Save your method definitions by going to the top of the Methods page and clicking **Save**.

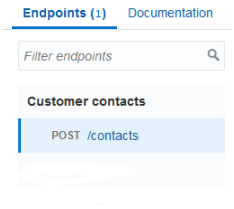
10. Set security access for your API by clicking **Endpoints** to get back to the Endpoints page. From there, click the **Security** navigation link.
11. Switch **Login Required** to `OFF` so you don't have to provide mobile user credentials or access tokens for authentication and click Save.



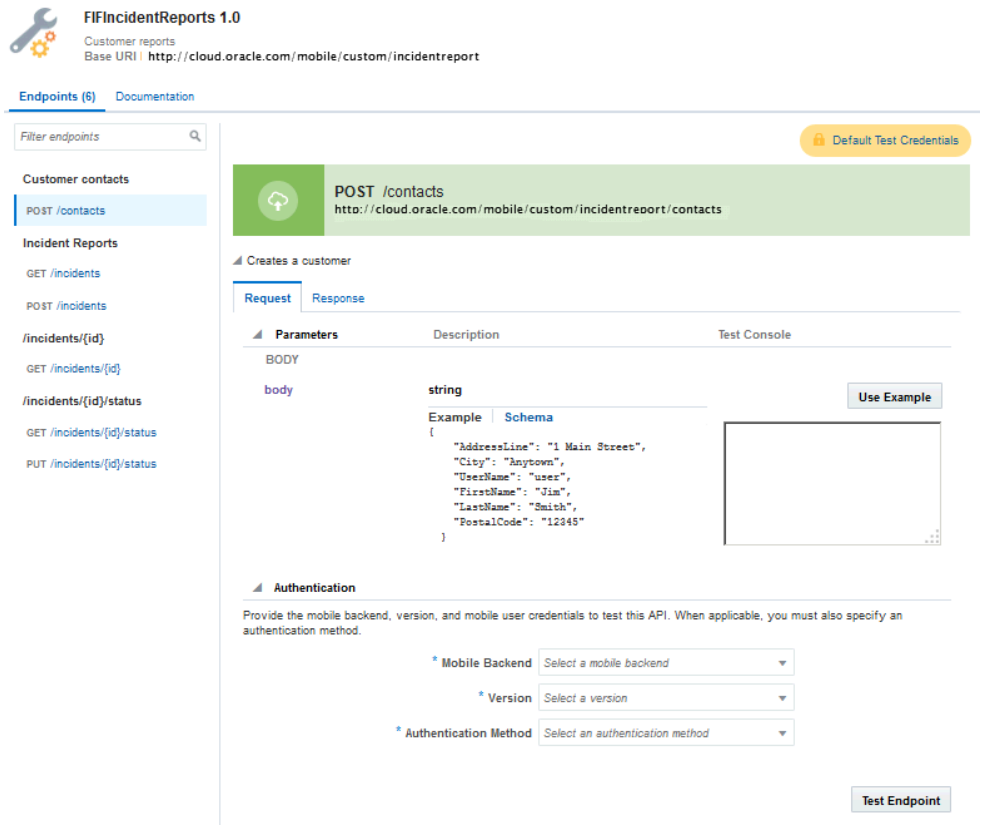
See Security in Custom APIs to learn more about securing access to the API. Now you're ready to test your endpoint.

12. Click **Test** to go to the API testing page.

The endpoints defined for the API are listed on the left side of the page. Click an endpoint to load it. You can see each method's request and response configurations for each resource.



You can check the definition of each method and if you want to modify a parameter name or an example, enter the change in the box to the right of the field. If you click **Use Example** by a message body, then the current body is copied into the text editor and you can make any changes.



13. In the Authentication section, select the mobile backend that this API is associated with and the mobile backend's version number.

    Because you set Login Required to OFF, you don't need to specify the authentication method or provide credentials.

    If you defined more than one endpoint, then set the default test credentials so you won't have to fill out the Authentication field for each method. Click **Default API Designer Test Credentials** at the top of the page and select the associated mobile backend and its version number. When you click **Save** (✔), the values are applied to the Authentication fields of each method.

14. Click **Test Endpoint**.

    You can view the request and response status and data of the test under the Response Status section. If you used the FixItFast example and your test was successful, then you should see a 201 status.

That's all you need to do to spec out your custom API. As long as the API is in a draft state, you or a teammate can edit the API configuration as needed. For steps on how to fully configure a custom API, see Creating a Complete Custom API.

Creating a Complete Custom API

Previously, you learned how to spec out an API using the API Designer. You gave a name to the API, added at least one resource and method and tested your endpoint. At this point you have a draft version of the API but it isn't quite complete. In this section, you'll fill in more details (such as defining the method requests and response, adding a schema, and setting secure access) to make a more robust API. Just in case

you're starting from scratch though or want more details about setting the basics, the complete set of steps to creating a custom API are presented.

Click  and select **Applications > APIs** from the side menu. If an API has already been created (whether in a Draft or a Published state), you'll see a list of APIs. If no custom APIs exist, then you'll see a page with the **New API** button. Click the API you spec'd out already or click **New API** to get started.

Setting Up Your API

Let's use the FixItFast example to create a custom API. In this example, you work for the FixItFast appliance repair company. You need to find a way to track the repair calls and responses. It would also be helpful to know which technicians are assigned to the repair jobs. You want to create an API that lists the customer service calls based on the customer who called to report the problem, the customer location, and the technician assigned to the job. You'll create the following API with the following properties:

- An API called `FIFIncidentReports`
- A base URI: `https://fif.mcs.cloud.oracle.com/mobile/custom/fif-incidentreport/`
- An `application/json` media type
- An icon to associate with the API display name (a PNG file that we selected)

When you click **Create**, a Draft state of the API is created and added to the list of custom APIs.

First, set the basic characteristics for your API by going to the General page.

1. Make sure that you're in the environment containing the mobile backend for which you want to create a custom API.
2. Click  and select **Applications > APIs** from the side menu.
3. Select **New API > API**.

   You select **API** to craft custom APIs with the API Designer. **Express API** enables you to create API quickly without having to write any code as long as you have sample data to provide. See Creating An API to learn about the Express API Designer. If you're developing mobile apps with the Mobile Application Accelerator (MAX), the Express API designer is the quickest way to develop APIs for use with MAX. See Creating APIs Fast with the Express API Designer for information about MAX.
4. Enter a name for the API in the **API Display Name** field that will appear in the list of APIs (required).

   The display name can contain alphanumeric characters and special characters (! ? & @ ( ) _ - . ' "). The name can't begin with a space and can't exceed 100 characters.
   The name you give to a custom API (the values you enter in the API Display Name and the API Name fields) must be unique among custom APIs. For example, if a custom API exists with the API name `My API`, then you can't create another custom API with the same name.
5. Enter a name for the API in the **API Name** field that will appear in the API configuration (required).

This name is appended to the relative base URI as the resource name for the API. The API name must begin with a letter (A - Z) and can contain numbers (0 - 9) and underscores (_). The name can't exceed 100 characters. A validation error message is displayed if you enter a name that's already in use.

If you edit the name of the API here, then the change will be made automatically to the resource name in the local URI.

6. Add a brief description of your API and click **Create**.

   You're taken to the API Designer page where you can complete the basic information for your API:

   - Default media type for the payload (`application/json` is selected by default, click the drop-down list to select another type).
   - API Catalog Properties to make it easier for you and other developers to locate the API. Provide a brief description of your API and select an icon to associate with your API.

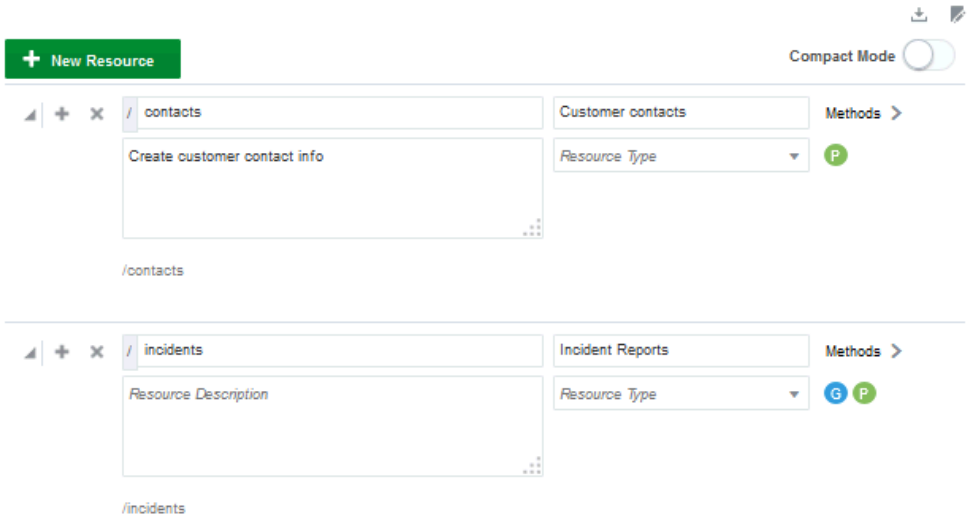
     If you want to use your own icon, then you can upload an icon (it must be in a PNG format) or if you're creative, then you can download Photoshop QuickStart to use an icon template to create an icon. You should be familiar with using Photoshop to create an icon. Follow the icon guidelines for sizing and color information. For sizing information, see the Full Palette Icon section of the ALTA ICON STYLE chapter in the Oracle Alta Web Design Guide. You'll need a 48x48 icon image within a 70x70 canvas. For color guidelines, see the Icon Palette section of the ALTA COLORS chapter of the same guide.

Now that you've provided the basic information, it's time to define endpoints for your API.

Defining Endpoints

You create resources to define the endpoints of your API. A resource is the crux of an API. It has a type, some data associated with it, a relationship to other resources, and contains one or more methods that act on it. A resource can be nearly anything: an image, a text file, a collection of other resources, a logical transaction, a procedure, etc. See API Resources.

1. Click the **Endpoints** navigation link to begin.
2. Click **New Resource** and add some basic information.



Each time you click **New Resource**, you create a top-level (root) resource. If you want to add a child (nested) resource, then click **Add** (**+**) next to the top-level resource. Click **X** to delete a resource.

> **Note:**
>
> See the icons under the **Methods** links? Each time you define a method for a resource, an icon for it appears under the Methods link. Use them as a shortcut to see what methods have already been define for a resource. Click on an icon to go directly to its definition on the Methods page.

3. Provide the resource path, which is the URI (relative to the base URI). For example, if the base URI is `/mobile/custom/fif-incidentreport`, then you could add the resource, `incidents`, that is `/mobile/custom/fif-incidentreport/incidents`.
4. Provide the display name, which is a name for the resource that makes it easy to identify in the API documentation.

   Resources are listed by their display names on the left side of the API Test page.
5. Provide a brief description of the resource.

   After you enter a description, the URI is displayed below the description field.
6. (Optional) Provide a RAML resource type, which is the resource type (`resourcesType:`). You don't need to specify a resource type. If you want to use a resource type but you don't have one defined, then click the **Types** link and define one. See Creating Resource Types.

When you create a method for a resource, a symbol for that method appears below the Methods link. You can immediately see what methods have defined for a resource if you need to examine a resource definition. Click on an icon to go directly to that method definition.

You can clear the clutter to locate a resource more quickly by switching to **Compact Mode** (it's to the right of **New Resource**). The compact display hides the resource description, resource type, and path.

Adding Methods to Your Resources

Methods are actions that can be performed on a resource. The Methods page shows you one method at a time. After at least two methods are defined, you can click on the icon for a method at the top of the page to see its details.

1. Add some methods to the resource by clicking **Methods**.

   If the resource you're defining methods for has path parameters, then they are displayed above **Add Method**.

   a. (Optional) Click **Required** if you want the path parameters to be passed with each method.

      The parameter name is displayed.

   b. Provide a display name for the parameter and example code.

   c. From the drop-down list, select the valid value type for the parameter.

2. Click **Add Method** and select the method that you want:

Method	Description
`GET`	Retrieve or read a resource
`POST`	Create a new resource
`PUT`	Update a resource
`DELETE`	Remove a resource
`HEAD`	Read the HTTPS metadata
`PATCH`	Perform a partial update of a resource
`OPTIONS`	Request information, such as the options or requirements of the resource

   After you've selected a method, it's no longer listed in the method list because you use a method only once per resource (e.g., you can't define two `DELETE` methods for a single resource). An icon for each method that you define is displayed at the top of the page. Click on a method icon to go directly to its definition.

3. (Optional) You can enter a brief description of the method in the Description field.

4. (Optional) You can enter a display name for the method.

5. (Optional) Provide any traits to apply to the method.

   If you don't have any resource traits defined, click **<Endpoints** to go back to the main Resources page and click the **Traits** link to define one. Traits let you define a collection of similar operations. See Creating Resource Traits.

After you've defined methods for the resource, you can define the requests and responses for those methods. See Defining a Request for the Method and Defining a Response for the Method.

Defining a Request for the Method

Now that you've selected a method, define the request you're making of the service that you want to connect to. For instance, if you selected a `POST` method, then now you can define what to create. You do this by adding parameters and a request body, which contains the description of the data to send to the service.

1. Click **Request** to define a request.
2. Click **Add Parameter** and select a parameter type: Query or Header. Select **Required** if the parameter is required for the method.
   a. Give the parameter a name and a display name.
   b. Select a valid value type: String, Number, Integer, Date, or Boolean.
   c. (Optional) Provide a description of the parameter and an example you can use when you test the validity of the endpoint. For example, you could have a resource, `incidents`, and add a query parameter, `contact` that takes a number value, and another parameter, `gps` that takes a string value:

```
/incidents:
  get:
    description: |
      Retrieves all incident reports for the filters below.
    queryParameters:
      contact:
        displayName: Contact ID
        description: |
          filter reports by contact
        type: string
        example: |
          lynn@gmail.com

        required: false
      technician:
        displayName: Technician ID
        description: |
          filter reports by technician
        example: "joethetechnician"
      gps:
        displayName: gps
        description: |
          location of contact or technician
        example: "39.355589 -120.652492"
```

   In this example, a `GET` method is defined with the query parameters, `contact`, `technician`, and `location`.
   d. (Optional) Click **More Properties** to add nested properties to the parameter. Click **Repeat** to add multiples of the current parameter.
   e. Click **Add Parameter** to add another top-level parameter for the method.
3. Depending on the method you selected, click **Add Media Type** and define the method body. The body contains the data that you're sending to the server. For instance if you're defining a `POST` method, you'll need to define the item you're

creating, such as a new customer listing or service request. If you're defining a `GET` method, you don't need to send a method body so you don't need to specify a media type.

a. Select the media type for your method body, that is the format of the message that you're sending, such as text, images, or web forms.

   Depending on the type (for instance, you wouldn't enter a schema for an image type), you have the option of adding a schema or an example, or both. When defining a schema, add only the data necessary for the purpose of the resource. That is, don't add unnecessary data that will only slow down the transmission and potentially increase the potential for errors.

b. (Optional) Click **Schema** and enter a schema (in JSON format) in the editor pane. A schema is like a template for the body. It's what you use to define the contents of the message.

   For an example of a schema, see Providing a Schema.

c. (Optional) Click **Example** and enter an example (in JSON format) in the editor pane, which is used by the mock implementation as a mock response for the method. Using mock data can help you verify the behavior of your methods. See Testing API Endpoints Using Mock Data. The example shows mock values for the data being sent in the message body as defined in the POST method of the `incidents` resource:

```
body:
  application/json:
    example: |
      {
        "Title": "Leaking Water Heater",
        "Username": "joh1017",
        "imageLink": "storage/collections/2e029813-d1a9-4957-a69a-
fbd0d7431d77/objects/6cdaa3a8-097e-49f7-9bd2-88966c45668f?
user=lynn1014",
        "Notes": "my water heater is broken"
      }
```

4. Click **Add Media Type** to add additional media types. If you decide that you don't want the method, then click **X** in the banner to delete it.

Defining a Response for the Method

Depending on the request, you may or may not need a response. A response describes the process for returning results from the service. You might want to define a response that verifies that the data you requested was returned or you might want a response that just acknowledges whether or not the request was received. Defining a response is similar to defining a request. The main difference is that you'll need to select a status code to let you know the result of the connection.

1. Click **Response** to define one or more responses.
2. Click **Add Response** and select the status code that you want returned.

   A status code of 200 is provided by default but if that isn't the code you want, then select one from the drop-down list.

   - 2*xx* indicates a successful connection

- 3*xx* indicates a redirection occurred
- 4*xx* indicates a user error occurred
- 5*xx* indicates a server error occurred

To help whoever uses the API to understand the reason for a potential error in the API you're configuring, use an HTTP status code to return code that best matches the error situation.

3. Provide a description of what the code designates.
4. Click **Add Header**, select a response **Header** or **Query**, provide the name of the header or query and a display name for the header, and the valid value type for the header.
5. Click **Add Media Type** and select the format of the response. Depending on the media type you select, you can add parameters, schemas, or examples just as you did for the Request body.
   a. For text-based media type (e.g., `application/json` or `text/xml`), click **Schema** to enter a schema (in JSON format) for the body.

      As with the request body, add only pertinent data to the response body. Don't include more data than you actually need for the operation.

   b. Click **Example** to add mock data (in JSON format) for your response body. Use mock data to verify the behavior of your methods before testing with real data. See Testing API Endpoints Using Mock Data.

   c. For form-based media type (e.g., `multipart/form-data`), click **Add Parameter** and select **Required** if the parameter is mandatory. Then provide a name and select a value type. Optionally, you can give your parameter a name.

   d. For image-based media type (e.g., `image/png`), you don't have to do anything because there are no schemas or attributes to provide.

The following example shows that a response for the `POST` method of the `incidents` resource was created with a status code of 201 indicating a new resource was successfully created. The example also shows a return response format of `application/json`, a `Location` header that was added, and the message body containing mock data:

```
responses:
  201:
    description: |
      The request has been fulfilled and resulted in a new resource
      being created. The newly created resource can be referenced
      by the URI(s)returned in the entity of the response, with the
      most specific URI for the resource given by a Location header
      field.

    headers:
      Location:
        displayName: Location
        description: |
          Identifies the location of the newly created resource.

        type: string
        example: |
```

```
          /20934

        required: true

    body:
      application/json:
        example: |
          {
            "id": 20934,
            "title": "Lynn's Leaking Water Heater",
            "contact": {
               "name": "Lynn Adams",
               "street": "45 O'Connor Street",
               "city": "Ottawa",
               "postalcode": "a1a1a1",
               "username": "johnbeta"
              },
            "status": "New",
            "driveTime": 30,
            "priority": "high",
            "notes": "My notes",
            "createdon": "2014-01-20 23:15:03 EDT",
            "imageLink": "storage/collections/2e029813-d1a9-4957-a69a-
fbd0d74331d77/objects/6cdaa3a8-097e-49f7--9bd2-88966c45668f?user=lynn1014"
          }
```

When you've defined your response, you can decide to test your endpoints (see Testing API Endpoints Using Mock Data) or click **<Endpoints** in the navigation bar to return to the main Resources page. From there, you can proceed to another page in the API Designer to create a root, resource types or traits, or add API documentation.

If you decide you don't want the method, then click **X** in the banner to delete it.

Testing API Endpoints Using Mock Data

You can provide mock data in your request and response message bodies during the design phase of your API configuration. This lets you examine the context of each call without having to use real time data or interact with a real time service. For example, to test whether your code correctly handles an invalid ID, you can add an example in your request body with mock data containing an invalid ID. When you finish the test, you can replace the example with other code to test some other aspect of the method.

In the FixItFast example, the mock data in the response body lets you verify if the correct customer information is being returned. Here's an example of mock data that the service developer could create for the response body of the POST operation of the `contact` resource in the FixItFast example:

```
{
 "id": 20934,
 "title": "Lynn's Leaking Water Heater",
       "contact": {
       "name": "Lynn Adams",
       "street": "45 O'Connor Street",
       "city": "Ottawa",
       "postalcode": "ala1a1"
```

```
       "username":"johneta"
       }
 "status": "new",
 "driveTime": 30,
 "priority": "high",
 "createdon": "2015-04-23 18:12:03 EDT"
}
```

When you create a custom API, a mock implementation is created automatically. The mock implementation lets you invoke the API from your mobile application before you've implemented the custom code. This lets you develop and test the mobile applications and the custom code simultaneously. If you're satisfied with the configuration, you can add a real implementation.

Until you create your first implementation, the default implementation is the mock implementation. After you create a real implementation, it becomes the default implementation for the API.

Click the **Implementations** navigation link to upload an implementation or to see any existing implementations. You can change the default implementation on the Implementations page. After you upload an implementation, you see a list of existing implementations, which includes the mock implementation.

See Testing with Mock Data to learn more about testing an API with a mock implementation. See Implementing Custom APIs to create a real API implementation.

For details on testing fully-implemented custom APIs, see Testing Your Custom API.

Providing a Schema

You have the option of adding a JSON schema, which describes the structure of your data and is written in JSON. If you want to add a schema, go to the **Schema** page and click **New Schema**. After you've defined at least one schema, you can select one from the list.

To define a schema, provide:

- The schema name
- The schema definition (in JSON format) in the editor pane, which you can manually enter or copy and paste into the editor

For example, a schema called `schema#` is defined as follows:

```
schemas:
- reports: |
      {
        "$schema": "http://json-schema.org/draft-04/schema#",
        "type": "array",
        "description": "Incident Reports array",
        "items": {
          "type": "object",
          "properties": {
            "id":  { "description": "Unique id for the incident report",
                     "type": "integer" },
            "title": { "description": "Title for the incident report",
                       "type": "string" },
```

```
            "createdon": { "description": "Date and time of creation",
                           "type": "string" },
            "contact": { "decription": "Contact information of customer
filing the report",
                         "type": "object",
                         "properties": {
                           "id" : { "description": "Unique id for the
customer",
                                    "type" : "string" },
                           "name" : { "description": "First and last
name of contact",
                                     "type" : "string" },
                           "street": { "description": "Street address of
contact",
                                     "type" :  "string"},
                           "city" : { "description": "City of contact",
                                     "type" : "string"},
                           "postalcode" : { "description" : "Postalcdoe
of contact",
                                            "type": "string" }
                           }
                 },
             "status" : { "description": "The current status of the
incident",
                          "type" : "string" },
             "priority" : { "description": "The current priority of the
incident",
                          "type" : "string" },
             "driveTime" : {"description" : "Calculated field based on
location",
                          "type" :  "integer"},
             "imageLink" : { "description" : "Link to image from Storage",
                           "type": "string" }
           },
         }
      }
```

Add more schemas to define by clicking **New Schema**. Click **X** to delete a schema. See Schemas for details about the structure of a JSON schema.

> **Note:**
>
> You can define multiple schemas for use with the given API. Schemas are specific to the API and aren't shared across other APIs.

Security in Custom APIs

In MCS, an API is protected through its association with a mobile backend to allow only authorized users and devices to access the API and its endpoints.

For enterprise applications, you can use HTTP Basic Authentication, OAuth, or SSO OAuth Token credentials to control user authentication and authorization of access to resources:

- With OAuth, when you create a mobile backend or register with an existing mobile backend, a set of OAuth consumer keys (that is, client credentials) consisting of a client ID and client secret are generated for you. The values of these keys are unique to the mobile backend (for information about authenticating with OAuth, see Authenticating with OAuth in Direct REST Calls). You authenticate yourself to the OAuth server by providing your client credentials and receive an access token that is passed in each API call via a header. Only a user with a valid token can access the API.

  Alternatively, you can provide a Single Sign-On OAuth token provided by your Remote Identity Provider if the **Enable SSO** option is selected for the mobile backend. For information on how to enable single sign-on for a mobile backend, see Authentication in MCS.

- With HTTP Basic Authentication, when a mobile backend is created, a mobile backend ID and an anonymous access key are generated for it. You authenticate yourself to MCS by providing these items, which are passed in each API call via a header. You must provide this information to access the API. You can obtain the mobile backend ID and anonymous access key from the mobile backend landing page. Select the mobile backend associated with the API and expand the Keys section. To learn more about authenticating with HTTP Basic, see Authenticating with HTTP Basic in Direct REST Calls.

- With Social Identity, when you register an app with a social identity provider (for example, Facebook), an access token is generated by the provider. You authenticate yourself to MCS by specifying the social identity provider and providing the access token.

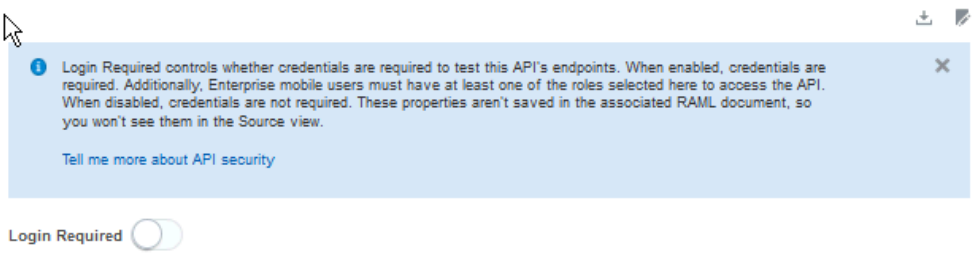
  To find out how to get an access token, see Getting a Facebook User Access Token Manually.

To learn about authentication in MCS, see Enterprise Single Sign-On in MCS.

Setting Access to the API

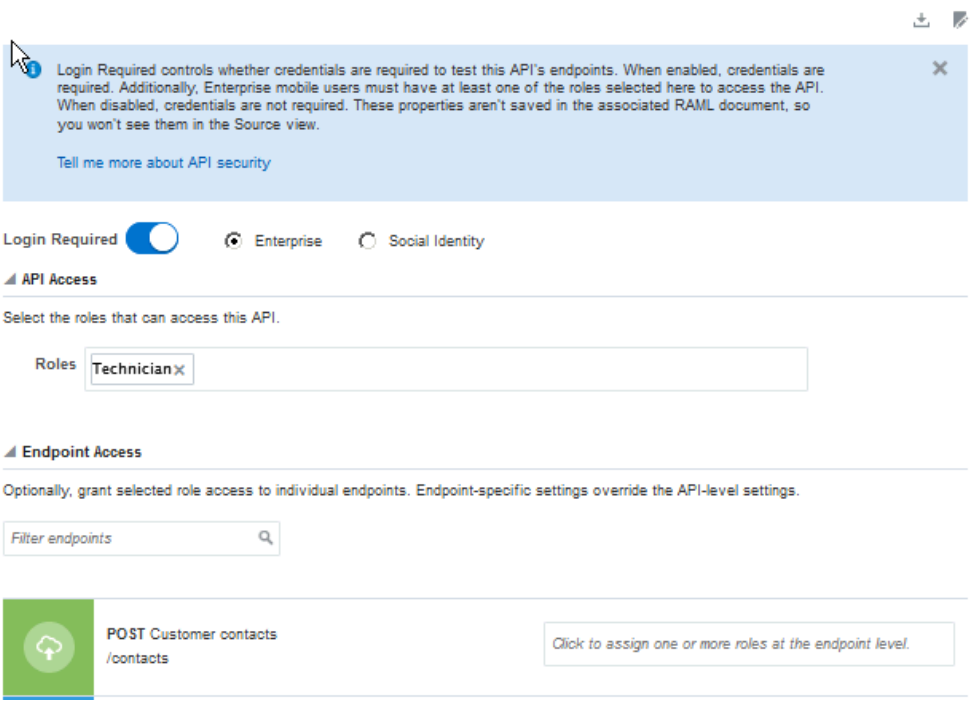
You have the option of requiring developers to login and provide authentication credentials to access the API.

- Set **Login Required** to `OFF` to allow access to the API from a mobile app as an anonymous user. Also, you won't need to use authentication credentials on the API's Test page.



This setting is particularly useful when you're in the early phases of configuring your API and you just want to validate some endpoints or when the data being requested or received is from a service that doesn't require security.

- Set **Login Required** to `ON` to require authenticated access to the API:



  - Select **Enterprise** to set access for mobile users who login with their MCS username and password or who have configured Single Sign-On authentication providers.

    When you set **Login Required** to `ON` and select **Enterprise**, the API Access and Endpoint Access fields are exposed and you **must** select at least one role to access the API. This ensures that only those mobile users that have the selected role or roles can access the API endpoints. Click in the **Roles** field to select one or more roles.

    Optionally, you can further refine access to the API by selecting roles for specific endpoints. Only mobile users having the role selected for a specific endpoint can access it. For example, you can allow only users with a Mobile Develop role to access the `DELETE` method. Click in the field for each endpoint and select one or more roles.

  - Select **Social Identity** to set access for mobile users who want to use their social media accounts for authentication.

    If you choose this setting, you can save your API configuration and move on to the Test page. In addition to specifying the mobile backend and its version, you'll be asked to select the social authentication provider and provide the access token generated for you by the selected provider.

> **Note:**
>
> You can obtain information about the current mobile and social users by including the `ums.getUserExtended()` method in the custom code for the API. See Accessing the Mobile Users API from Custom Code.

Testing Your Custom API

To validate your API endpoints, the Test page lets you test with sample response data. You'll see a list of all the resources that you've defined on the left side of the page. Use the Filter endpoints field to display only the resources that you want to test. You test only one endpoint at a time.

> **Note:**
>
> A few things before you start testing your API:
>
> - If **Login Required** is turned `ON` and Enterprise is selected, you must have a role assigned that allows access to the API.
> - If **Login Required** is turned `ON` and Enterprise or Social Identity is selected, you must provide values for all fields in the Authentication section of each method to test it.
> - If **Login Required** is turned `OFF`, providing authentication credentials is optional.
> - Save your configuration *before* you test. If you haven't, then you can check the **Always save before testing** option in the Save Before Testing confirmation dialog that appears when you click **Test**. That way, any changes that you make to the API configuration are automatically saved.

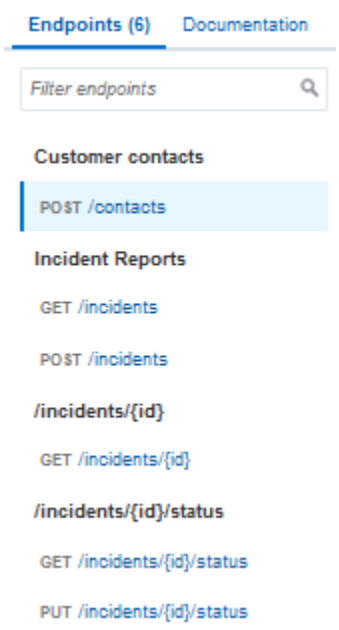
1. If you are in the design phase and just want to see if your endpoints are valid, or if you want to test multiple endpoints during the session, then set the default API test credentials.
   a. Click **Default API Designer Test Credentials** at the top of the page.
   b. Select a mobile backend to associate the API with and the version of the mobile backend.
   c. Select one the authentication method to use for testing:. **HTTP Basic**, **OAuth Consumer**, **Social**, or **Single-Sign On**.
   d. If **Enterprise** is selected on the Security page, mobile users must enter their mobile user credentials (username and password).

      Credentials for social identity or for single-sign on are not required.
   e. Click **Save** (✔).

      The mobile user credentials that you enter will be used as the default credentials for all test calls made within Mobile Cloud Service.

If you need to test only a few methods, skip Step 1 and fill out the fields in the Authentication section for each method (see Step 5).

2. Select the method that you want to test from the list of endpoints on the left side of the test page.



When you select an endpoint, the method banner for it is displayed with the base URI is displayed below the operation name. If you provided an alternate name for the operation, then that name appears, otherwise the default operation name is shown. Only one method per endpoint is displayed at a time for testing.

3. Click **Request**.
4. Expand **Parameters** to view the query or header parameters that you provided.
   a. (Optional) Click **Example** to view the example body, if you provided one. Enter an alternate example to test with by clicking **Use Example**. The provided example body is copied into the text box. You can edit the example as needed.
   b. (Optional) Click **Schema** to view the request body schema if you provided one.
5. Click **Response**.
6. Expand the status code area and click **Example** or **Schema** to review the example or schema for the response body, if you provided one.
7. Click **Request** again to enter Authentication information.
8. If **Login Required** is `OFF`, click **Test Endpoint**. Otherwise, skip this step and go to the next step.
9. Expand **Authentication**and, if **Login Required** is `ON`, select the mobile backend and its version that are associated with this API and enter your authentication credentials:

- If Enterprise is selected, select the authentication method you want to use for testing and provide your mobile user credentials.
- If **Enterprise** is selected and Single Sign-On is enabled for the associated mobile backend, select **Single Sign-On** as the authentication method and enter either the MCS-issued SSO OAuth token (hover over the ? icon and follow the instructions) or the third-party issued SSO token that you obtained from your trusted remote identity provider.

  For information on configuring a Single Sign-On provider, see Configuring Identity Management (SSO and OAuth).
- If **Social Identity** is selected, select a social authentication provider and enter the access token that you got from your provider.

> **Note:**
>
> MCS automatically URI encodes the username and password that you enter. An error can result if the username and password entries contain special characters (that is, you've entered pre-URI encoded values). If you enter values for these fields that are already encoded, another layer of encoding is added. During authentication, these values are decoded once, and the original encoded values are revealed, which will fail authentication so don't enter URI-encoded values for username and password.

10. Click **Test Endpoint**.

    Click **Request** to see the metadata for the transaction, such as header information and the body of the request. Click **Response** to see the details of the response returned. The response code tells you whether or not the connection was successful.

Test each of your operations and modify them as needed to validate your endpoints. When your custom API is completed, you can go to the APIs page and check out the Implementations, Deployments, Used By, and History fields to find out how often the API is being called, what mobile backends are using it, and more. See Managing an API.
To learn how to get a Single Sign-On OAuth token, see Enterprise Single Sign-On in MCS.

To find out how to get an access token from a social authentication provider, see Getting a Facebook User Access Token Manually.

Creating Resource Types

A **resource type** is a partial resource definition that specifies a description and methods and their properties. Resources that use a resource type inherit its properties, such as its methods. You don't have to use a resource type, but if you find that you're defining resources with the same methods, you can increase efficiency by defining resource types to reduce the redundancy.
Using the incident report example, you might want to get reports from several departments (billing, service technicians, and clerks). For each department, you want to get a list of employees involved with a particular incident and you want the name, ID, and extension number for each employee. You can define a resource type, `employee_contact` that defines a `GET` method that retrieves all the personnel

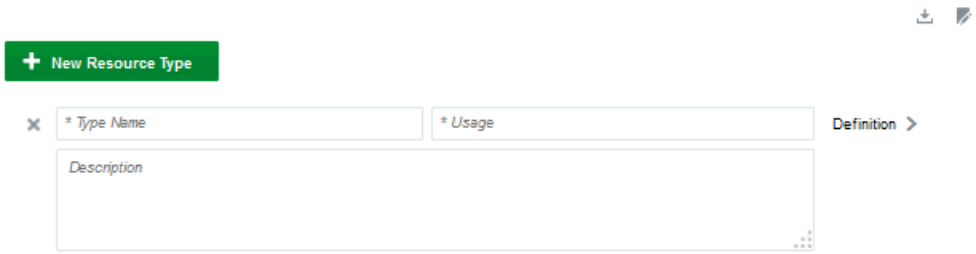
information that you need. Instead of defining an `employee_contact` for each branch of the company, you can apply the `employee_contact` resource type to each incident report resource.

> **Note:**
>
> Resource types can't be used with nested resources.
>
> You can define multiple resource types for use with the given API. Resource types are specific to the API and aren't shared across other APIs.

Adding a resource type through the API Designer is simple:

1. Click **Types** and then click **New Resource Type**.

   The Types page is displayed:



2. Enter a name for the resource type.

   For example, a resource type called `orderinfo` could be used each time appliance parts are ordered.
   Valid resource type names are character strings and can include underscore (_) and hyphens (-). Camel case is allowed (for example, `employeeContact`). Don't include special characters, such as slashes, asterisks (*), and exclamation points (!).

3. (Optional) Add a description of the type.

4. Enter a brief sentence that describes the purpose of the type in the **Usage** field, then enter a description of the type in the **Description** field.

   For example, a resource type called `orderinfo` , the usage might be: `Defines a standard parts order`. The description might be: `Always get model's serial number and part number.`

5. Click **Definition** to define the resource type in the source editor.

6. Click **Save** when you're done defining the type.

7. (Optional) Click **Test** to test your resource type.

   Edit your definition as needed. When you're finished, return to the Types page to add another type or navigate to another page in the wizard.

The resource type is added to the list of available resource types for use with the given API. To learn more about resource types, see Resource Types and Traits in the RAML specification.

Creating Resource Traits

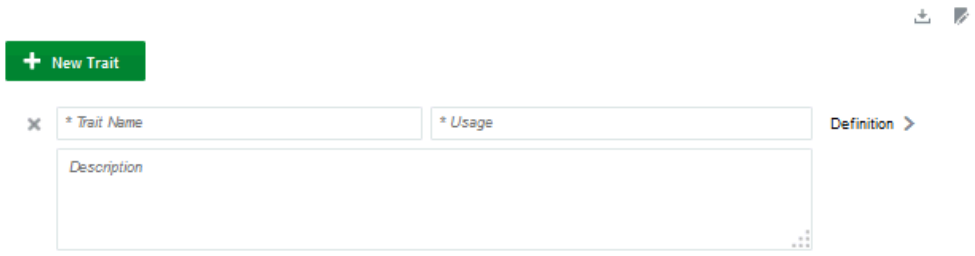
A **trait** is a partial method definition that provides method-level properties such as a description, headers, query string parameters, and responses. Define traits for obtaining descriptive information like version numbers or vendor information. Methods that use one or more traits inherit those traits' properties. As with resource types, if you're defining methods with the same attributes multiple times, then define a trait to prepopulate a method with certain attributes. You don't have to use resource traits, but they're useful if you have several methods with the same operational structure.

> **Note:**
>
> You can define multiple resource traits for use with the given API. Resource traits are specific to the API and aren't shared across other APIs.

Here's how to define a resource trait:

1. In the API Designer, click the **Traits** navigation link and click **New Trait**.

   The Traits page is displayed:



2. Enter a name for the trait.

   For example, a resource trait called `parts-inventory` could define a standard method of looking up the availability and location of specific parts.
   Valid resource trait names are character strings and can include underscores (_) and hyphens (-). Camel case is allowed (for example, `applianceModel`). Do not include special characters, such as slashes, asterisks (*), and exclamation points (!).

3. Enter a brief sentence that describes the purpose of the trait in the **Usage** field, then enter a description of the trait in the **Description** field.

   For example, if you have a trait called `parts-inventory`, the usage might be: `Apply to GET methods for all part requests.` The description might be: `Always determine if parts are in stock and list warehouse locations.`

4. Click **Definition** to define the resource trait in the source editor.

5. Click **Save** so you don't lose your work.

The resource trait is added to the list of available resource traits for use with the given API. To learn more about resource traits, see Resource Types and Traits in the RAML specification.

Providing API Documentation

A good, even great API is useless without documentation describing it so others can use the API too. While the API Designer can't write that documentation for you, you can upload it through the API Designer so that the next time you or someone else selects this API from the API Catalog, a full description of the API is available (its purpose, its resources and schemas, the security policies that it uses, and helpful code comments).

1. In the API Designer, click the **Documentation** navigation link and click **Documentation**.
2. Enter a title for your API document.
3. You can either manually write your API documentation using Markdown syntax in the source editor or copy and paste your documentation into the editor.

   Click **Markdown Reference** to see how to use Markdown. It lets you write an easy-to-read plain text that can easily be converted to structurally valid XHTML for viewing in a browser. See How Do I Write in Markdown?

   Here's an example of part of the API documentation for the FIFIncidentReports API:

```
#%RAML 0.8
title: FIFIncidentReports
version: 1.0
baseUri: /mobile/custom/incidentreport
protocols: [HTTP,HTTPS]
mediaType: application/json
/contacts:
  displayName: Customers
  post:
    description: |
      Creates a customer

    protocols: [HTTP,HTTPS]
    body:
      application/json:
        example: |
          {
              "AddressLine": "1 Main Street",
              "City": "Anytown",
              "UserName": "user",
              "FirstName": "Jim",
              "LastName": "Smith",
              "PostalCode": "12345"
            }

    responses:
      201:
        description: |
          The request has been fulfilled and resulted in a new
resource being created

/incidents:
  displayName: Incident Reports
  get:
    description: |
      Retrieves all incident reports for the filters below. One
better filter less they damage the mobile device.

    protocols: [HTTP,HTTPS]
    queryParameters:
      contact:
        displayName: Contact ID
        description: |
          filter reports by contact

        type: string
        example: |
          lynn@gmail.com
```

4. Click **Save** so you don't lose your work.

   You can add more documentation by clicking **New Title** and adding content in the editor field for that document. You can replace the default title provided by entering text in the title field. Each time you click **New Title**, the title field and editor for the most recent document is appended below the previous document. When you click **Save**, only the current document is displayed. Click a title tab to view that particular document.
   To see the API documentation for a specific API, select the API from the API Catalog, click **Test**, and then on the Test page, click the **Overview** tab.

How Do I Write in Markdown?

Markdown is a simple set of syntax that you can use to produce basic formatting structures such as section heads, paragraphs, ordered and itemized lists, block quotes, and links.

Construct	Markdown	Output
Header: Use hash marks (#) to denote headers	#First-Level Heading ## Second-Level Heading ### Third-Level Heading	First-Level Heading Second-Level Heading Third-Level Heading
Paragraph: Separate paragraphs with one or more blank lines.	This is a paragraph. This is a second paragraph.	This is a paragraph This is a second paragraph.
Simple List: Use +, -, or * followed by a space to denote list items. List markers are interchangeable.	- list item 1 + list item 2 * list item 3	- list item 1 - list item 2 - list item 3
Nested List: Use +, -, or * followed by a space to denote list items and indent nested list item by exactly four spaces.	-list item 1 + list item 1a + list item 1b -list item 2	- list item1 - list item 1a - list item 1b - list item 2
Ordered List: Precede each item with a number in a consecutive sequence followed by a space.	1. list item 1 2. list item 2 * list item 2a * list item 2b 3. list item 3	1. list item 1 2. list item 2 2a. list item 2a 2b. list item 2b 3. list item 3
Emphasis Italics: Wrap text with an asterisk (*) or single underscore.	*text* _more text_	*text* *more text*
Emphasis Bold: Wrap text with two asterisks (*) or double underscores.	**text** __more text__	**text** **more text**
Inline code: Use back quotes (`) around the text.	This is an `inline code` example.	This is an `inline code` example.
Code Block: Indent each line by four spaces	Format a block of preformatted code: This is a code line.	Format a block of preformatted code: `This is a code line.`
Links: Put the link text in brackets, followed immediately by the URL in parentheses.	This is an [example link](http://example.com).	This is an example link.

If you want to find out more about Markdown, see *What is Markdown?*

Getting Diagnostic Information

You can view the response code and returned data to determine if your endpoints are valid. A response status other than 2*xx* doesn't necessarily mean that the test failed. If the operation was supposed to return a null response, then the response should show a 4*xx* code.

For every message you send, MCS tags it with a correlation ID. A correlation ID associates your request with other logging data. The correlation ID includes an Execution Context ID (ECID) that's unique for each request. With the ECID and the Relationship ID (RID), you can use the log files to correlate messages across Oracle Fusion Middleware components. By examining multiple messages, you can more easily determine where issues occur. For example, you can retrieve records from Oracle Fusion Middleware Logging using the call's ECID. From the Administration page, you can click **Logs** to view logging data.

Depending on your MCS access permissions, you or your mobile cloud administrator can view the client and server HTTP error codes for your API's endpoints on the Request History page, allowing you to see the context of the message status when you're trying to trace the cause of an error. Every message sent has a set of attributes such as the time the event occurred, the message ID, the Relationship ID (RID), and the Execution Context ID (ECID).

To learn more about getting and understanding diagnostics, see Monitoring Performance and Troubleshooting .

After you've configured your custom API, you can provide an API implementation, that is, create your own custom code and add it to your mobile backend to access the API. See Implementing Custom APIs.

API Design Considerations

When you configure your custom API, there are some things you can do to ensure you have a well-formed API, including making sure that URLs and resources are well-formed, that reasonable read and connect timeouts have been set, and, if you're providing a RAML file, that it's correctly configured.

Here are some things to consider when you configure your API and some detailed descriptions of more advanced constructs that you can use to refine your API.

Valid URLs

In creating your RESTful API, it's important that you define a valid URL. You can see the URL for your API as you define it from the API name that you provide and the resources and methods that you add. To ensure that you have a valid URL, it must adhere to the following best practice guidelines:

- Provide a relevant and easily identifiable resource name. Using identifiers in your URLs make for a more understandable resource than using a query string. Which makes more sense to you, the resource name `/customers/2223` or `/customers/api?type=customerid=2223`?
- Resources can be grouped into a collection, so make the collection resource name consistent with the attribute names used to refer to the collection.

For example, if an attribute is a collection of favorite bookmarks, be obvious and name the collection `favoriteBookmarks` instead of `favoriteLinks`.

- Always make the resource names plural nouns and alternate between plural nouns and singular resource identifiers (rid): `/services/1.0/items/{rid}/subitems/{rid}/`

  For example: `/customers/2223/orders/555`

  To ensure that the API is sync-compatible, always put the identifier immediately after its related resource name as shown in the previous example, where `2223` is the designation of a specific customer and `555` is the designation of a specific order. A poorly formed URL to indicate a specific customer could look like this: `/customers/orders/2223/555` or `/customers/orders/locations/2223`.

- Use lowercase for resource names and use camel case for attribute names.

  For example: `/services/1.0/items?limit=10&totalResults=true`

- Keep resource identifiers down to 32 characters or fewer due to the limitations of some browsers.
- Keep URLs as short as possible. A long rambling URL is difficult to read and all the more difficult to debug.
- When defining the URL, you can be as concrete or abstract as desired, but you should use the curly brace {} notation to indicate URI parameters. This makes the corresponding RAML more detailed and easier to test.
- Ensure that all date formats are in the form: `YYYY-MM_DD[THH:mm:ss.sss]Z`.

  For example: `2014-10-07T18:35:50.123Z`

- For pagination, use the `limit` and `offset` query parameters so that the Synchronization library uses paged downloads correctly. If you don't need to support pagination, you don't need to specify these parameters.
- To ensure sync compatibility, use the `orderBy` query parameter to specify sorting. For example: "`orderBy=propA,propB:desc,propC:asc`". In this example, the default sort order is by ascending value.

  For details on designing sync-compatible custom APIs, see Making Custom APIs Synchronizable.

- Provide values for query parameters as a URL-encoded JSON string. For example:

```
[
  {
    "property":"propertyName",
    //Supports Equals, NotEquals, LessThan, GreaterThan,
LessThanOrEqual,GreaterThanOrEqual
    "comparison":"Equals",
    "value":"Must be a string",
  },
  {
    "property":"Another clause, only support ANDS not ORs",
    ...
  }
]
```

API Timeouts

Sometimes when an API fails, it's due to a stream or connection timeout. Stream timeouts happen when, after a successful connection to the server, data is being transmitted and the network time outs before all the data can be sent or received. Connection timeouts happen when the network connection is never made.

To ensure that connectors have sufficient time to make a connection and that data can be transmitted, the HTTP read and connection timeouts should have smaller values than the API timeout.

The `Network_HttpRequestTimeout` value determines the amount of time spent transmitting an HTTP request before the operation times out. The default value is 40,000 ms. The value of this policy can affect your API timeout values, which should be less than the value of the policy. Note that policy values are specific to a particular environment. The value for this policy in a development environment can be different from its value in a runtime environment. Your mobile cloud administrator can increase or decrease the timeout value from the Administration tab.

If you have mobile cloud administrator privileges, then you can select an environment in the Administration view and export the `policies.properties` file to see a list of the current environment policies and their values. For information about API environment policies and policy settings, see Oracle Mobile Cloud Service Environment Policies. For information about environment policies in general, see Environment Policies.

API Resources

A key element of an API is the *resource*. A **resource** is the conceptual mapping to an entity or to a set of entities and is identified by its relative base URI. In other words, a resource is a *thing* (noun) that's located at an address to which you want to transmit information or receive information. It has at least one method (verb) that operates on it. A method is what you use to retrieve, create, update, or delete a representation of a resource. For example, `GET incidents`.

A top-level resource is a resource defined at the root level (also referred to as the root resource). A resource that's defined as a child of another resource is a nested resource. Nested resources let you specify aspects of the parent resource. A nested resource is identified by its URI relative to the parent resource URI. For example, let's say you have a root resource defined as `.../incidents`, and you have a nested resource, `{id}`. The API definition in RAML looks like:

```
title: FIFIncidentReports
version: 1.0
baseURI: /mobile/custom/fif-incidentreport
protocols: [HTTPS]
mediatType: "application/json"
/incidents:
  displayName: Incident Reports
  get:
    description: |
      Retrieves all incident reports.
.
.
.
```

```
/{id}:
  uriParameters:
    id:
      displayName: id
      description: |
          The unique id of the incident report.
```

A resource is always preceded with a slash (/), whether it's a root or nested resource. For information about constructing a valid RAML document, see RAML.

If you think of a resource as a collection of objects and a nested resource as an item in that collection, then your resource path shows the parent resource in plural form and a nested resource in singular form. For example:

```
.../mobile/custom/fif-incidentreport/incidents/{id}
```

The root resource is `incidents` and the instance of an incident is `{id}`. You can give the resource an easy-to-read display name on the Endpoints page. If you don't provide a display name, then the resource URI is used as the name.

A common practice when designing a resource is to have `PUT` and `POST` methods return the same objects that are sent in the request.

URI Parameters

If you want to allow API calls that change or restrict the value of the relative base URI, then you can override it by setting a base URI parameter. The URI of a resource can contain parameters, which are variable elements, for example `{id}`.

Like resources, parameters have a name. The RAML generated for our `fif-incdentreport` shows the resource parameter named `id`, a display name (`id`, although the display name doesn't have to be the same as the parameter name), and a value type (in this example, the value type is `integer`):

```
/{id}:
  uriParameters:
    id: displayName: id
    description: |
      the unique id of the incident report

    type: integer
    required: true
 get:
   description: |
     Retrieves the incident report with the specified id.
```

You place the path parameter after the resource name. Use a semicolon to separate multiple parameters. For parameters that can have multiple values, separate the values with commas.

In the example, the URI parameter */{id}* is a variable that identifies a specific incident report by its ID number. The parameter contains the properties `displayName` and `type`. The URI would look like this:

```
.../fif-incidentreport/incidents/{id}
```

If the parameter, `id`, has a value of `1234`, then the resulting URI would look like this:

```
.../fif-incidentreport/incidents/1234
```

Parameters can be added as part of the URI path as a child (nested) resource or added as a query. There are no hard and fast rules to adding parameters to the URI path versus adding parameters as a query. One possible consideration is whether the parameter is essential to the request. For example, to get data for a specific report, you would use an identifier (`id`) of the resource in the URI path as shown in the previous `fif-incidentreport` URI example.

However, if you're using the parameter as a filter to narrow down the data, then add it in the query. For example, you would use `technician` as a query parameter `.../fif-incidentreport/incidents?technician=joe` to filter reports only by a particular technician.

Endpoint Requirements for Sync Compatibility

To ensure optimal synchronization of data when a custom API is used by the Synchronization library on a client, the custom API must include a specific set of server-side endpoints.

For example, let's say a custom API endpoint is defined that returns a collection of Department records and is consumed by a client that uses the Synchronization library. Records are retrieved from the collection endpoint, `/Departments`, and stored in the client's local cache by the library. Later on, the library identifies two records in the cache that require updating because they've expired (`/Departments/Finance` and `/Departments/HR`).

In this case, to get the most up-to-date data, the Synchronization library retrieves only the records that need to be updated, and not the entire collection.

On the server side, via the associated Synchronization library, these endpoints are called individually on behalf of the client. The data is returned to the client in a single payload and response, saving multiple round trips for each required object.

To support this, the Synchronization library requires that the custom API includes `GET` methods for both the collection resource (`GET /{collection}`) and the object resource (`GET /{collection}/{objectId}`). That is, in our Department example, the following endpoints are needed:

- `GET /Departments`
- `GET /Departments/{DeptId}`

To go a step further, if the offline API collection objects that were retrieved can be modified, say by the addition, update, or deletion of an object, the Synchronization library calls the appropriate custom code APIs to enact the change on the objects on the server side. To support creating, updating, or deleting the object requires that the following types of endpoints are implemented on the server-side custom API:

- `GET /{collection}`
- `GET /{collection}/{objectId}`
- `PUT /{collection}/{objectId}`
- `POST /{collection}`
- `DELETE /{collection}/{objectId}`

The inclusion of the `PUT`, `POST`, and `DELETE` operations are optional. If, for example, your application never deletes an object in a collection, you don't need to implement the `DELETE` operation.

> **Note:**
>
> The Synchronization library doesn't support the `PATCH` operation.

See Making Custom APIs Synchronizable to learn more about configuring a sync-compatible custom API.

Schemas

A JSON **schema** defines the structure of your API in a JSON-based data format. The JSON schema can be used to validate JSON data. You can define a schema from the **Schema** page. Let's look at the schema from the `IncidentReports` example:

```
{
        "$schema": "http://json-schema.org/draft-04/schema#",
        "type": "array",
        "description": "Incident Reports array",
        "items": {
          "type": "object",
          "properties": {
            "id":  { "description": "Unique id for the incident report",
                     "type": "integer" },
            "title": { "description": "Title for the incident report",
                       "type": "string" },
            "createdon": { "description": "Date and time of creation",
                           "type": "string" },
            "contact": { "decription": "Contact information for customer
filing the report",
                         "type": "object",
                         "properties": {
                           "id" : { "description": "Unique id for the
customer",
                                    "type" : "string" },
                           "name" : { "description": "First and last
name of contact",
                                     "type" : "string" },
                           "street": { "description": "Street address of
contact",
                                     "type" :  "string"},
                           "city" : { "description": "City of contact",
                                     "type" : "string"},
```

```
                                "postalcode" : { "description" : "Postalcdoe
of contact",
                                                 "type": "string" }
                                }
                        },
                "status" : { "description": "The current status of the
incident",
                             "type" : "string" },
                "priority" : { "description": "The current priority of the
incident",
                             "type" : "string" },
                "driveTime" : {"description" : "Calculated field based on
location",
                             "type" :  "integer"},
                "imageLink" : { "description" : "Link to image from Storage",
                              "type": "string" }
              },
            }
      }
```

This schema contains the following keywords:

- `$schema`: denotes that this schema is based on the draft v4 specification. It must be located at the root of the JSON schema. You should always include this keyword in your JSON schema.
- `type`: defines a JSON constraint, so the data must be an array.
- `description`: describes the contents of the schema.
- `items`: define the items in the array. In an incident report, we want to assign attributes to each report. In this example, all items are of type `object` and each object has a set of properties, such as report ID, title, contact info, status, priority level, etc.

For a complete list of keywords to use in your JSON schema, see `http://json-schema.org/`.

To add a schema for your API, see Providing a Schema.

RAML

When you create an API using the MCS interface, the API definition is stored as a RAML document. RAML is a simple efficient way to describe RESTful APIs. REST stands for Representational State Transfer (REST) and is a way to perform basic operations (create, read, update or delete) information on a server using simple HTTP calls.

You can also upload a RAML document that you create from scratch into the API Designer. The API Designer takes the input that you provide and creates a RAML file that documents the contents of the custom API. Note that the RAML defines only the API itself, not the implementation of the API. You must create custom code using JavaScript to implement the API. For information on how to implement an API, see Implementing Custom APIs.

> **Note:**
>
> The feature to upload a RAML document isn't available if you came to the API page by clicking **APIs** from the navigation list of a mobile backend.

If you upload a RAML file, then the values for the required Name fields are extracted from the RAML file. You still have to add the short description. At a minimum, your RAML file must include the API name, a base URI (`/mobile/custom/apiname`), and a version number.

For your RAML file to be valid, it must specify a media type, base URI, the HTTPS protocol, and a version number:

```
#%RAML 0.8
---
title: api_title
version: 1.0
protocols: [HTTPS]
baseURI: /mobile/custom/api_name
mediaType: application/json
```

> **Note:**
>
> MCS requires the HTTPS protocol for custom APIs. If you upload a RAML document that configures the API using the HTTP protocol, then it's automatically edited to use HTTPS.

For new a API, a default version of 1.0 is automatically applied when you save the configuration (unless the mobile cloud administrator has changed the value of the `Asset_DefaultInitialVersion` environment policy). However, if you upload an API configuration, then the version value displayed is taken from the file.

> **Note:**
>
> The version value uses a specific format. Versions are specified with an integer. For example, in your RAML file specifying `version: 2.0` is valid while `version: v2.0` isn't.

RAML lets you define resource types and traits for describing resources and methods, which results in a more succinct RESTful API by reducing repetition in the design. The principle components of a RAML (`.raml`) document are:

- Basic API information consisting of:
  - API Display Name: the easy—to—read name of the API, which appears in the API list (for example, `FIFIncident Reports`)
  - Base URI: The address of the resource (`/mobile/custom` for custom APIs)

  - API Name: name of the API (`fif-incidentreport`) in the configuration
  - Short description: Brief description of your API
- Resource types and traits, which allow you to characterize resources to avoid unnecessary repetition in the API definition
- Resources (the conceptual mappings to one or more entities), resource methods, and schema

To ensure that your RAML document is correctly configured, follow these tips:

- Although RAML allows both HTTP and HTTPS protocols, MCS requires the HTTPS protocol for custom APIs. If you upload a RAML document that configures the API using the HTTP protocol, then it's automatically edited to use HTTPS.
- If you define a top-level resource with an empty relative URI (that is, /:), then you can't add a subresource to it.

  An error message will alert you that the structure is invalid. For example, the following resource definitions will fail:

  ```
  /:
    /reports:
  ```

  You need to make reports a top-level resource:

  ```
  /:
  /reports:
  ```

- Top-level resources shouldn't contain empty relative URI subresources, for example:

  ```
  /books:
    /:
  ```

- Avoid creating duplicate paths, for example:

  ```
  /reports/{id}:
  /reports:
    /{id}:
  ```

  Multiple subresources in the resource name are valid. For example:

  ```
  /reports:
    /county/branchid/reportissue:
  ```

- Add comments only in a property's `description:` field. Adding a comment using a comment line (for example, `#report issue by technician`) is not supported by the RAML source editor. Comments added in a comment line are stripped out by the parser.

For a thorough discussion about RAML, see http://raml.org/.

Editing a Custom API

You can always edit an API as long as it's in the Draft state. A published API can't be changed.

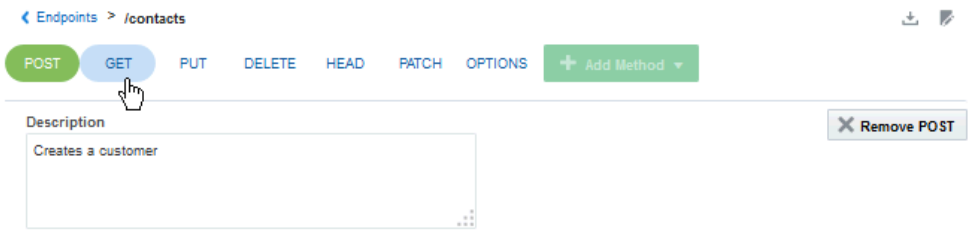
To edit a custom API:

1. Make sure that you're in the environment containing the API you want to edit.
2. Click ☰ and select **Applications > APIs** from the side menu.

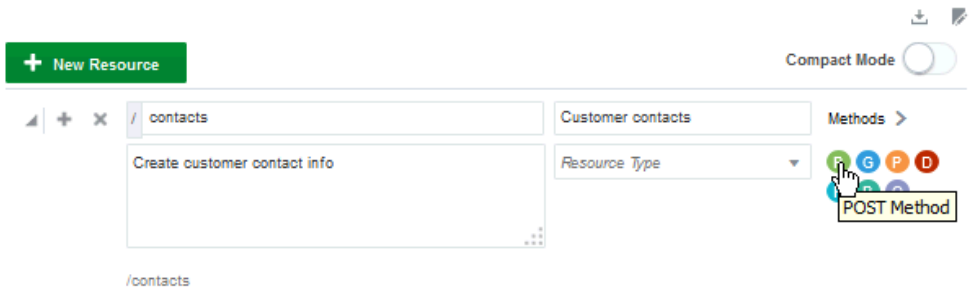
   Now that at least one custom API exists, the APIs page is displayed.
3. Select the draft API that you want to edit and click **Open**.

   You can filter the list by version number or status. You can also sort the list alphabetically by name or by last modified date.
4. Edit the fields for general information, resource, schemas, traits, types, and security policies as needed.

   Each time you create a method for a resource, an icon for the method appears at the top of the Methods page. Click on one of these icons to go directly to the method definition:



On the Resources page, icons for the methods defined for the resource are displayed below the Methods navigation link. You can quickly see what types of methods have been defined for a resource. Click on an icon to go the method definition:



Remember you can always click **Save and Close** to save your current changes and finish the rest of your changes later.

5. Save your changes if you didn't select the option to always save the configuration before testing when you created the API.
6. Test your changes.

Your edited version is still in a Draft state and you can continue to edit your custom API until you're satisfied with the configuration. At that point, you're ready to publish your custom API. See Publishing a Custom API. If you need to make a change to a published API, you'll have to create a new version of it. See Creating a New Version of an API.

After you've published it, you can then deploy your API to other environments. See API Lifecycle for information on specific parts of an API lifecycle. For general information about lifecycle in MCS, see Lifecycle.

Video: End-to-End Custom API Demo

To see the process of designing and developing a custom API, including how it fits in with a mobile backend and a connector, take a look at this video:

Video

Troubleshooting Custom APIs

When an incorrect value is entered in a field, a message window displays the error and, depending on the field, the correct syntax or value type to use. In some cases (such as when a malformed schema or RAML is uploaded), the error message includes a **Show Details** link that displays a description of the error. See Viewing Log Messages.

To learn more about common errors that can occur when you configure custom code, see Common Custom Code Errors.

When troubleshooting an unexpected result, consider that the cause might be due to a rerouting of the call to the mobile backend as described in Making Changes After a Backend is Published (Rerouting). If the mobile backend was rerouted, check to see if the following conditions were met:

- If the API was accessed using social identity, then the access token of the provider that was entered in the Authentication header must be the access token of the provider of the target mobile backend (that is, the mobile backend to which the original mobile backend was redirected).
- If the API was accessed by a mobile user, then the user must be a member of the realm that is associated with the target mobile backend (the mobile backend to which the original mobile backend is being redirected).

22

Implementing Custom APIs

As a service developer, you use the custom code service to implement the custom APIs that your team creates for its mobile apps.

What Can I Do with Custom Code?

Using JavaScript, Node, and the custom code SDK, you can implement the APIs that have been designed in the API Designer (or by means of a RAML document). Say, for example, that your mobile app developer has designed the following API, which has one resource (`/incidents`), and two endpoints (`GET /incidents`, and `POST /incidents`).

```
#%RAML 0.8
title: IncidentReport
version: 2.0
baseUri: /mobile/custom/incidentreport
...
/incidents:
  displayName: Incident Reports
  get:
    description: |
      Retrieves all incident reports.
...
  post:
    description: |
      Creates a new incident report.
```

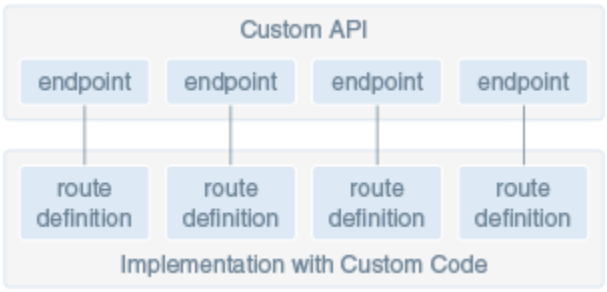
You, as the service developer, implement all the endpoints in the API. That is, you write code to return incident reports for the first endpoint and to store incident reports for the second endpoint.

Your custom API implementation can call platform APIs (such as Storage and Notifications), other custom APIs, and external REST and SOAP web services. And it can access the external web services either directly or through connectors.

> **Note:**
>
> To use your implementation (custom code) in a mobile backend, you must first define the custom API as described in Custom API Design. The implementation is then accessed by your apps through calls to the API's endpoints.

How Does Custom Code Work?

Using the custom code service, you write JavaScript code to implement a custom API. The coding model is based on Node, which is a JavaScript framework that enables you to write server-side code and that provides a runtime environment for it. For each API endpoint, which is the resource (URI) plus the HTTP method such as `GET` or `POST`, you need a route definition that specifies how to respond to a client request to that endpoint. In other words, for each URI and HTTP method combination in your API, you need to add a JavaScript method to your custom code that handles the request. Route definitions follow the coding style promoted by Express, which is a module that runs in Node. We'll show you how to write these methods.



After you've written your custom code, you package it as a Node module, and then upload it.

For more information about route definitions, see Key JavaScript Constructs in Custom Code. For information about the Express coding style, see `http://expressjs.com/starter/basic-routing.html`. For information about Node, see `www.nodejs.org`. If you're interested in how the custom code service handles custom API requests and responses, then see What Happens When a Custom API Is Called?

> **Note:**
>
> Note: The purpose of the examples in this chapter is to illustrate how to interface with the custom code service. The examples are not intended to teach best practices for writing Node.js REST API implementations.

What's the Foundation for the Custom Code Service?

The custom code SDK is available to custom API implementations and is what you use to call platform APIs, connectors, and other custom APIs, as described in Calling APIs from Custom Code. In addition, the custom code service is backed by the following JavaScript libraries, which you can use when you implement your custom API.

JavaScript Library	Description
Node	Node provides the backbone for the custom code service. When you implement a custom API, you create a Node.js module. Behind the scenes, a router module takes care of creating an HTTP server for a Node instance and routing the HTTP calls that come from the service to the custom API's implementation that runs inside the instance. You don't need to write code for this.
Request	Request is framework for Node that simplifies the making of HTTP calls. The service wraps Request calls with additional code that's necessary for the custom code service.
Express	Express is a lightweight web application framework for Node. The custom code service uses it to expose API endpoints. To implement your custom API, you write route definitions similar to how you would use Express to write routes for a web app.
Bluebird	The custom code service uses the Bluebird promises library to implement the promises that the custom code SDK methods return.
Body-parser	The custom code service uses this library to parse incoming request bodies.
Http-proxy-agent	This module provides an http.Agent implementation that connects to a specified HTTP proxy server.
Https-proxy-agent	This module provides an http.Agent implementation that connects to a specified HTTPS proxy server.
Express-method-override.	The custom code uses this library to override the method of a request based on an X-HTTP-Method-Override header, a custom query parameter, or a post parameter.
Agentkeepalive	This library is an implementation of http.Agent that keeps connections alive for some time to reduce the number of times that TCP connections are closed, which thus saves resources.

As shown in the next table, the default library versions depend on whether your environment was provisioned from the current release or upgraded from an earlier release. This table lists the default versions of the libraries for this release and the prior release.

JavaScript Library	Environment Provisioned from Current Release	Environment Upgraded from Prior Release
Node	8.9.4	6.10.0
Request	2.83.0	2.74.0
Express	4.16.2	4.14.0
Bluebird	3.5.1	3.4.6
Agentkeepalive	3.3.0	3.1.0
Body-parser	1.18.2	1.15.2
HTTP-proxy-agent	2.0.0	1.0.0
HTTPS-proxy-agent	2.1.0	1.0.0
Method-override	2.3.10	2.3.6

If your custom API implementation isn't compatible with the default library versions for your environment, do one of the following to change the versions for that implementation:

- Add a `node` property to the configuration section in the custom API implementation's `package.json` file as described in Declaring the Node Version. You can set it to `0.10`, `6.10` or `8.9`. The Node version in the `package.json` file overrides the `CCC_DefaultNodeConfiguration` environment policy for that custom API implementation.
- Ask your mobile cloud administrator to change the node version that is specified by the appropriate `CCC_DefaultNodeConfiguration` environment policy. The choices are `0.10`, `6.10`, and `8.9`. You can set this policy at different scopes, such as environment scope, mobile backend scope, and API scope. Whenever you change a `CCC_DefaultNodeConfiguration` environment policy, any custom API implementation that uses that default configuration will change to the new version no later than its second REST request after the version change.

> **Note:**
>
> The default maximum body size for all configurations is 1MB. To learn how to increase the maximum body size, see Custom Code Problem parsing JSON: Error: request entity too large.

Video: Node.js Technology Primer

If you don't have experience with Node.js or you'd simply like to better understand how it works with the custom code service, take a look at this video:

Video

Setting Up Tooling for Custom Code

The custom code service is based on Node. You don't need to install Node on your system to create custom API implementations, but you'll need the tooling that it provides, such as the Node package manager (npm). Having Node on your system also makes it easier for you to write the code.

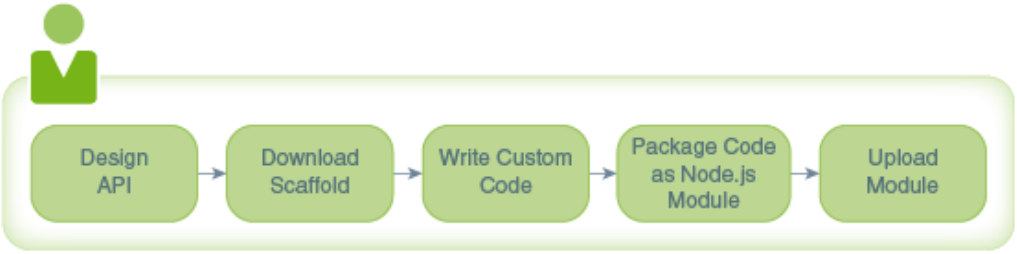
The `nodejs.org` website provides installers that contain the library and some command-line tools, such as npm. You may wish to also install an integrated development environment (IDE) with Node support for features such as syntax highlighting and code completion. One free option is to install Eclipse (`eclipse.org`) and then add the Nodeclipse plug-in (`http://www.nodeclipse.org/`).

Steps to Implement a Custom API

The main steps for defining and implementing a custom API are the following:

1. Define a custom API as described in Custom APIs.
2. Download a JavaScript scaffold for the API. This scaffold contains stub implementations for your endpoints.
3. Within the scaffold, fill in the appropriate JavaScript code for each function that corresponds with a given REST endpoint.
4. Package the finished JavaScript module.

5. Upload the module to the API Designer.



Downloading a JavaScript Scaffold for a Custom API

After you create your custom API, you can download a scaffold that is based on your API's RAML document, and then use the scaffold as a quick start for implementing your custom API.

> **Note:**
>
> Instead of downloading the scaffold, you can have it pushed directly to a Git repository. See Managing Custom Code in Git.

The scaffold comes in the form of a Node module, the key components of which are the main JavaScript file that contains stub methods for each endpoint (resource plus HTTP method), and a `package.json` file, which serves as the manifest for the module.

To download the scaffold:

1. Click to open the side menu, click **Applications** , and then click **APIs**.
2. Open the API that you want to download.
3. In the left navigation area of the API Designer, click **Implementations**.
4. Click **JavaScript Scaffold** to download the zip file.

   If Git integration is enabled, the **JavaScript Scaffold** button is replaced by a drop-down list. In this case, click **JavaScript Scaffold** and then select **Download** (if you want to download the scaffold) or **Push to Git Repository** (if you want to immediately start working with the scaffold in Git).
5. On your system, unzip the downloaded file.

> **Note:**
>
> If you later change the API, then you can download a new scaffold based on the updated endpoints. However, any coding that you may have done and uploaded previously won't be reflected in the new scaffold.

Writing Custom Code

The following sections show the constructs that are available to you and how to use them in your code.

Key JavaScript Constructs in Custom Code

The scaffold zip that you download from the API Designer includes a main JavaScript file, which contains the key constructs that you need to implement the custom API. Here's an example of a main JavaScript file for a custom API, which has these resources (URIs):

- `/incidents`, which supports the `GET` and `POST` HTTP methods
- `/incidents/:id`, which supports the `GET` HTTP method
- `/incidents/:id/uniquecode`, which supports the `GET` HTTP method

```
// A
module.exports = function(service) {


  //B
    service.post('/mobile/custom/incidentreport/incidents',
function(req,res) {
        var result = {};
        var statusCode = 201;
        res.status(statusCode).send(result);
    });

    service.get('/mobile/custom/incidentreport/incidents',
function(req,res) {
        var result = {};
        var statusCode = 200;
        res.status(statusCode).send(result);
    });

    service.get('/mobile/custom/incidentreport/incidents/:id',
function(req,res) {
        var result = {};
        var statusCode = 200;
        res.status(statusCode).send(result);
    });
```

This example illustrates these main constructs:

- (A) `module.exports = function (service) {`*`implementation`*`}`

  The `module.exports` statement is required at the beginning of all custom API implementations. It's used to export an anonymous function with a parameter (`service`) through which the custom code service passes the object that's used to expose your endpoints. The `service` parameter is an instance of an Express application object, and all the object's functionality is available. Note that in

Express example code, this parameter is often called `app`. The anonymous function contains all the API's route definitions.

- (B) Route definitions

  A route definition is an Express route method that associates an anonymous callback function with an endpoint (resource plus HTTP method). Its signature takes the following form:

  ```
  service.HttpMethod('URI', function (req, res)
  ```

  – *`service`* is the variable for the custom code service instance (or, in Express terminology, the application instance), which was defined in the `module.exports = function (service)` statement.

  – *`HttpMethod`* is one of the following methods corresponding to standard REST methods:

    * `get`
    * `delete`
    * `head`
    * `options`
    * `patch`
    * `post`
    * `put`

  – *`URI`* refers to resource defined in the API. Notice that while braces identify a parameter in the API design for the resource, you use a colon to identify a parameter in the *`uri`*. For example, if the resource is `/incidentreport/incidents/{id}`, then you use `'/mobile/custom/incidentreport/incidents/:id'` for the *`URI`*.

  – `function (req, res)` is a callback through which HTTP request and HTTP response objects are passed. It defines how the API responds to client requests to that endpoint. The `req` variable provides access to the data in the request and you can use the `res` variable to build the result. Node and Express provide properties and functions for those two variables, which enable you to retrieve information about their values and work with them. We talk about some of these next.

    For more information about the `req` and `res` objects, see `http://expressjs.com/4x/api.html#request` and `http://expressjs.com/4x/api.html#response`.

  The following example is a route definition for the endpoint `GET /incidentreport/incidents/{id}/uniquecode`, which generates a unique code.

  ```
  service.get(
    '/mobile/custom/incidentreport/incidents/:id/uniquecode',
    function (req, res) {
      console.info('get /incidentreport/incidents/' +
        req.params.id + '/uniquecode');
      res.type('application/json');
      // status defaults to 200
  ```

```
  res.send({'code': req.params.id + '-' + new Date().valueOf()});
});
```

Notice that the code example uses `req.params.id` to get the `:id` value from the URI. Here are some of the request properties that you typically use in your code:

Property	Description
`req.body`	If the request's content type is `application/JSON` or `application/x-www-form-urlencoded` then this property contains the data that was submitted in the request body in the form of a JavaScript object. For information about accessing other types of request bodies, see Accessing the Body of the Request.
`req.headers`	A map of header names and values. The names are lower case. Often used to transport extra information in the request, such as an external identifier.
`req.params`	An object that contains properties that map to parameters in the endpoint's URI. For example, if the endpoint is `attachments/:collection/objects/:objectid`, then you use `req.params.collection` and `req.params.objectid` to get the parameter values. When you use the `req.params` object to retrieve a parameter value, you must use the same case as the parameter in the endpoint. For example, if the endpoint parameter is `{id}`, then you must use `req.params.id` to get the value, and not `req.params.Id`.
`req.query`	The query string parameters that are passed in the URI. For example, if the request is `GET /incidents?q=joe&order=desc` then you use `req.query.q` and `req.query.order` to get the query parameters.

Here are some methods that you typically use to inquire about the request:

Method	Description
`req.get(field)` and `req.header(field)`	Both these methods return the value for the header named by `field`. For example, `req.header('content-type')`. The match is case-insensitive. Note that `req.header` is an alias for `req.get`.

Method	Description
`req.is(mimeType)`	Boolean method that you can use to find out if the request's `Content-Type` header matches the `mimeType`. For example, `req.is('json')`.

> **Note:**
>
> The custom code service essentially creates Express application objects and then configures them with service-specific functionality (such as identity propagation and consolidated logging) before it passes them to the custom API implementation logic for further configuration. You get preconfigured Express application objects to which you add route-specific business logic.

Here we discussed only the basic usage of Express features necessary to implement the API by using routing methods to set up callbacks. However, the entirety of the Express features are available for use in custom code. Consult the Express documentation at `http://expressjs.com/` to learn about the details, such as how to implement URI parameter parsing, set up multiple callback handlers, and use third-party middleware.

Accessing the Body of the Request

When requests that are received by the custom code have a content type of `application/x-www-form-urlencoded` or `application/json`, the payload is converted to a JavaScript object, which is then stored in `req.body`. For all other types, such as `image/jpeg` or `text/html`, `req.body` is undefined. Examples of when this occurs is when the body is a text file or an image. In those cases, when you need to access the body from the incoming request's handler, use the data event listener and end event listener to save the body to a buffer.

The following example shows how to access the body for different content types:

```
if (req.is('json') || req.is('application/x-www-form-urlencoded'))
    {
       console.info('Request Body: ' + JSON.stringify(req.body));
    } else {
        var data = [];
        // Process a chunk of data. This may be called multiple times.
        req.on('data', function(chunk) {
          // Append to buffer
          data.push(chunk);
        }).on('end', function() {
          // process full message here
          var buffer = Buffer.concat(data);
          // Convert to base64, if required
          // var base64 = buffer.toString('base64');
      });
   }
```

To learn more about Node.js events and listeners, see https://nodejs.org/api/events.html#events_events.

Inserting Logging Into Custom Code

You can use the Node `console` object to add logging messages to custom code, as shown in this example:

```
console.info(i + ' Request to get ' + url);
```

These messages appear in the diagnostic logs.

The custom code service wraps the `console` object to enable finer-grained logging. The following methods are available for logging messages at different levels:

- `console.severe`
- `console.warning`
- `console.info`
- `console.config`
- `console.fine`
- `console.finer`
- `console.finest`

By carefully applying log levels to the messages in your code, you can simplify how you debug and administer the app. This allows you to add good debug messages, and then log them only as necessary, such as during development or when diagnosing a problem. For example, you might want to add the following log messages at the suggested log levels:

Log Message	Log Level
Function entry and exit	Finest
Input, such as parameters that are sent with the request	Fine
Caught exceptions	Severe
Uncaught exceptions	Fine

To set the level at which logging is enabled for a backend, from either the mobile backend's diagnostics page or the **Administration** page, click **Logs**, and then click  (**Log Level**).

To learn how to view the logs, see Accessing Logging Messages for Custom Code.

> **Note:**
>
> Node.js has a less granular set of native methods for logging, which are also possible to use. The logging level of the native Node.js methods `console.log` and `console.dir` is equivalent to `console.info`. The Node.js method `console.warn` is equivalent to the custom-code method `console.warning`. The Node.js method `console.error` is equivalent to the custom-code method `console.severe`.

When you use console messages to locate problem code, know that the service's console calls are nonblocking. That is, there's no guarantee that logging completes before the next statement is executed. In the case of a problem that's caused by an infinite loop, you will most likely see only the first console message that's in the block of code before the infinite loop. Consider the following code, for example:

```
console.info("Log 1");
var myVar="any string";
console.info("Log 2");
myVar="a different string";
console.info("Log 3");
functionWithInfiniteLoop();
```

When this code is executed, it's possible that only `Log 1` appears in the diagnostic logs. Therefore, to locate an infinite loop, you must have just one console message, and you must put that message where you think it will flag the problem. If it doesn't flag the problem, then move the message and run another test until you identify the problem code.

When you suspect an infinite loop, follow these steps:

1. Remove or comment out all console messages.
2. Add a logging statement as the last line before the return.
3. Ensure that the log level for your backend is set to the same level as your logging statement, such as `INFO` for a `console.info()` message.
4. Test the endpoint.
5. Look in the diagnostic logs for your logging statement.
6. If you don't see the message, move the logging statement up one line and test the endpoint again.
7. Repeat the previous step until the message appears in the log.

   At this point, you know that the problem statement is just below the logging statement.

> **Tip:**
>
> If you have several lines of code, then you can reduce the number of tests by putting the logging statement in the middle of the code block and then testing the endpoint. If you don't get the log message, then put the logging statement in the middle of the top half. Otherwise, put the logging message in the middle of the bottom half. Test the endpoint. Repeat the test by dissecting the code blocks until you have narrowed the test to just two lines of code.

Storing Data Locally

Don't use the file system that's associated with the virtual machine running the Node.js instance to store data, even temporarily. The virtual machines that run Node.js instances might fluctuate in number, meaning that data written to one instance's file system might be lost when individual instances are started and stopped.

To store data from custom code, you can use the Database Access API, which is described in Accessing the Database Access API from Custom Code, or the Storage API, which is described in Accessing the Storage API from Custom Code.

Video: Working with Node - Common Code

For a demonstration of writing Node code to implement custom APIs, take a look at the Oracle Mobile Platform video series on custom code, starting with this video:

Video

Implementing Synchronization-Compatible APIs

If your mobile app uses the Synchronization library to enable offline use, as described in Data Offline and Sync, then here's some information about how to make your implementation compatible with the library.

> **Note:**
>
> To learn how to design your API so that it is compatible with the Synchronization library, see Endpoint Requirements for Sync Compatibility and API Design Considerations.

Video: Working with Custom APIs via Data Offline & Sync

If you want an overview of how to build your custom API to have synchronization-compliant REST endpoints and data, take a look at this video:

Video

Requirements for a Synchronization-Compatible Custom API

To ensure that the Synchronization library can synchronize with your custom API's data, as described in Building Apps that Work Offline Using the Synchronization Library, follow these rules:

Method	Response Body	Response Headers	Response HTTP Status Codes
GET	• To return a single item, use `setItem()` to put the item in the response, as described in Returning Cacheable Data. Note that this method adds the `Oracle-Mobile-Sync-Resource-Type` header to the response and sets it to `item`. • To return a collection, use `addItem()` to add the items to the collection, as described in Returning Cacheable Data. Note that this method associates each item with its required URI and ETag and sets the `Oracle-Mobile-Sync-Resource-Type` header to `collection`. If there're no items in the collection, then you must return a body with empty `items`, `uris`, and `etags` arrays. For example: `{ items:[], uris:[], etags:[] }`	• `Oracle-Mobile-Sync-Resource-Type`: Must be set to `item` for a single item, or `collection` for an array of items. The `setItem()` and `addItem()` methods set this header automatically for items and collections. If the response body is a file, you optionally can set this header to `file`. • `ETag`: If the `Oracle-Mobile-Sync-Resource-Type` header is set to `item` or `file`, then this header must be set to the item's ETag (in quotes). • `Oracle-Mobile-Sync-Evict`, `Oracle-Mobile-Sync-Expires`, and `Oracle-Mobile-Sync-No-Store`: Optional. See Specifying Synchronization and Cache Policies.	No special requirements

Method	Response Body	Response Headers	Response HTTP Status Codes
PUT	If the item stored on the server is different from the item in the request body, such as having a different ID in the case of an add or containing automatically calculated fields like `modifiedOn`, then return the stored item in the response body. Otherwise, returning the item in the response body is optional.	• `Location`: If the item was added, then you must include this header, which contains the item's URI. Otherwise, this header is optional. • `ETag`: Must contain the item's ETag in quotes. • `Oracle-Mobile-Sync-Resource-Type`: Must be set to `item` for a single object. The `addItem()` method sets this header automatically. If the response body is a file, you optionally can set this header to `file`. • `Oracle-Mobile-Sync-Evict`, `Oracle-Mobile-Sync-Expires`, and `Oracle-Mobile-Sync-No-Store`: Optional. See Specifying Synchronization and Cache Policies.	Note that the value in the `If-Match` header value dictates the actions to take and the response code to send. The Synchronization library sends `*` in the `If-Match` header when the conflict resolution policy is `CLIENT_WINS`. For all other conflict resolution policy configurations (that is, `SERVER_WINS` and `PRESERVE_CONFLICT`), it sends the item's ETag. If the header isn't present or is null, then assume `*`. • If there's an `If-Match` header and its value isn't `*`, then, if the item's ETag doesn't match the header's value, return `412 Precondition Failed`. • If the item to be updated no longer exists, then do one of the following: – If the `If-Match` header is `*`, then add the item and return `201 CREATED` – If there's an `If-Match` header and its value isn't `*`, then return `404 NOT FOUND`. • If the item was successfully updated, then

Method	Response Body	Response Headers	Response HTTP Status Codes
			return one of the standard PUT codes, such as `200 OK` or `204 No Content`.
POST	If the item stored in the server is different from the item in the request body, then include the stored item in the response body. Otherwise, returning the item in the response body is optional. For example, if the server adds calculated fields such as `createdOn`, then return the stored item in the response body.	• `Location`: Must contain the item's URI. • `ETag`: Must contain the item's ETag in quotes. • `Oracle-Mobile-Sync-Resource-Type`: Must be set to `item` for a single object. The `addItem()` method sets this header automatically. If the response body is a file, you optionally can set this header to `file`. • `Oracle-Mobile-Sync-Evict`, `Oracle-Mobile-Sync-Expires`, and `Oracle-Mobile-Sync-No-Store`: Optional. See Specifying Synchronization and Cache Policies.	No special requirements

Method	Response Body	Response Headers	Response HTTP Status Codes
DELETE	No special requirements	No special requirements	• If there's an `If-Match` request header and its value isn't `*`, then if the ETag of the item to be deleted doesn't match the header's value, return `412 Precondition Failed`. Note that the Synchronization library sends `*` in the `If-Match` header when the conflict resolution policy is `CLIENT_WINS`. For all other conflict resolution policy configurations, it sends the item's ETag. • If the item doesn't exist, then you can return either a `404 Not Found` or a `204 No Content`. The Synchronization library process is the same for both codes. • If the item was successfully deleted, then return one of the standard DELETE codes, such as `200 OK`, `202 Accepted`, or `204 No Content`.

If you want to learn more about how the Synchronization library uses the `412 Precondition Failed` HTTP response status code and the `If-Match` header to implement conflict resolution policies, see Synchronization Library Process Flow. Basically, if the conflict resolution policy is `CLIENT_WINS`, then the `If-Match` header is set to `*` to indicate that the server must update or delete the resource without conflict. Otherwise, the `If-Match` header is set to the item's ETag, and the custom code is expected to return `412 Precondition Failed` if the ETags don't match.

> **Tip:**
>
> Most methods require an `ETag` header in the response, and many methods require that you compare the server version's ETag with the value in the request's `If-Match` header. There are several node libraries that you can use to create ETags. For example, the NPM `etag` library that is available from `https://www.npmjs.com/package/etag`.

Returning Cacheable Data

The custom code SDK provides the following methods to format your data for use by the Synchronization library. Using these methods enables the library to optimize synchronization.

oracleMobile.sync Method	Description
`setItem(response, item)`	Set the response body to the item.
`addItem(response, item, uri, etag)`	Add the item to a collection, which will be returned in the response body in a cacheable format.
`clear(response)`	Undoes all calls to `setItem` and `addItem` .

For a response with a single JSON object, you use `setItem` to format the data, as shown in this example, and you return the ETag value in the `ETag` header:

```
var etag = require('etag');
...
service.get('/mobile/custom/incidentreport/incidents/:id/syncUniquecode',
  function (req, res) {
    var item = {'code': req.params.id + '-' + new Date().valueOf()};
    res.setHeader('Etag', etag(JSON.stringify(item)));
    req.oracleMobile.sync.setItem(res,item);
    res.end();
});
```

For a JSON object that contains an array of items, you use `addItem` to add each item to the response, as shown in the next example. Note that `addItem` attaches a URI and an ETag value to each item in the response body. The URI must uniquely identify each item.

```
var etag = require('etag');
...
service.get(
  '/mobile/custom/incidentreport/statusCodes',
  function (req, res) {
    var payload = {'inroute': 'Technician is on the way'};
    req.oracleMobile.sync.addItem(
      res,
      payload,
      '/mobile/custom/incidentreport/statusCodes/inroute',
      etag(JSON.stringify(payload))
```

```
      );
    payload = {'arrived': 'Technician is on premises'};
    req.oracleMobile.sync.addItem(
      res,
      payload,
      '/mobile/custom/incidentreport/statusCodes/arrived',
      etag(JSON.stringify(payload))
      );
    payload = {'completed': 'Technician has left premises'};
    req.oracleMobile.sync.addItem(
      res,
      payload,
      '/mobile/custom/incidentreport/statusCodes/completed',
      etag(JSON.stringify(payload))
      );
    res.end();
  });
```

The response body for the `addItem` example looks like this:

```
{
  "items": [
    {
      "inroute": "Technician is on the way"
    },
    {
      "arrived": "Technician is on premises"
    },
    {
      "completed": "Technician has left premises"
    }
  ],
  "uris": [
    "/mobile/custom/incidentreport/statusCodes/inroute",
    "/mobile/custom/incidentreport/statusCodes/arrived",
    "/mobile/custom/incidentreport/statusCodes/completed"
  ],
  "etags": [
    "\"26-5vTpRVIO9SakJoLYEQrQ0Q\"",
    "\"27-+lktOY9aA46ySRE0O/y5Aw\"",
    "\"2c-PSRg8Cxr2rYp/9BftCmDag\""
  ]
}
```

When you use `setItem` and `addItem`, the response also includes this header:

Header	Description	Type
`Oracle-Mobile-Sync-Resource-Type`	If the response body is JSON, then the value is `item` if the JSON object includes a single item. The value is `collection` if the JSON object contains an array of items. Note that when the response is a file, you optionally can set the value to `file`. When this header isn't included in the response, the Synchronization library assumes that the type is `file`. That is, when this header is not set, then the `MobileResource` that the Synchronization library `fetchObjectBuilder` and `fetchCollectionBuilder` methods return is of type `MobileFile`.	String

Specifying Synchronization and Cache Policies

For the mobile apps that use the Synchronization library, you might want to override their settings for whether to cache the data that you return and when to expire and delete the data. For example, if the data contains private information, you might want to prevent a mobile app from caching that data. This table shows the `Oracle-Mobile-Sync` HTTP headers to override these settings.

Header	Description	Type
`Oracle-Mobile-Sync-Evict`	Specifies the date and time after which the expired resource should be deleted from the app's local cache. Uses RFC 1123 format, for example `EEE, dd MMM yyyy HH:mm:ss z` for `SimpleDateFormat`. The following synchronization policies are set for the resource object that is created from the response: • Eviction policy: `EVICT_ON_EXPIRY_AT_STARTUP` • Expiration policy: `EXPIRE_AFTER` with the `expireAfter` property set to date and time provided in the header value .	Number
`Oracle-Mobile-Sync-Expires`	Specifies when to mark the returned resource as expired. Uses RFC 1123 format, for example `EEE, dd MMM yyyy HH:mm:ss z` for `SimpleDateFormat`.	Number
`Oracle-Mobile-Sync-No-Store`	When set to `true`, instructs the client to not cache the resource.	Boolean

Calling Web Services and APIs from Custom Code

Your custom code will most likely need to access one or more of the following types of APIs and services:

- Platform APIs: Your custom code can connect with platform services, such as Storage, Notifications, and Location, through their APIs.
- Custom APIs: Your custom code can interact with all the other custom APIs that are in your environment.
- Connector APIs: Your custom code can serve as wrappers for connector APIs.
- External web services: Typically, you create connector APIs with which to interact with external services, but you also can connect with remote web services directly from custom code.

Calling APIs from Custom Code discusses how to access platform, custom, and connector APIs from custom code.

If you need to make a third-party web service call that doesn't require you to shape the data, and you don't need integrated diagnostics, tracking, or analytics for that call, then

you might choose to call the service directly instead of setting up a connector. You can call a web service directly from your custom code using Node APIs such as the HTTP API. For information about the Node HTTP API, see `nodejs.org/api/http.html`.

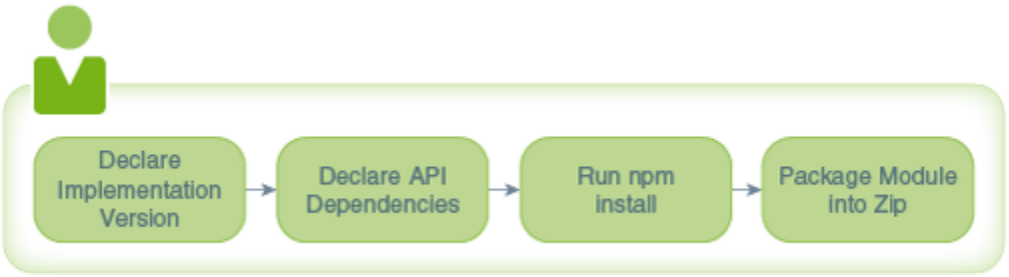
Note that HTTP and HTTPS are the only supported protocols for making calls to the Internet from custom code.

> **Note:**
>
> If the third-party web service changes its API, then a connector requires just one change, whereas with direct calls, you must make sure you find and change all the direct calls. Also, consider that if you're testing against a test web service, you'll have to modify the URLs for the direct calls when you switch to the production service.

Packaging Custom Code into a Module

After you've written custom code to implement an API, and before you upload and deploy it, follow these steps to package the implementation:

1. Declare the implementation version in the `package.json` manifest file.
2. Optionally declare the Node version in the `package.json` file.
3. Declare in the `package.json` file the API dependencies on other modules.
4. Run the Node.js package manager (npm) to download the dependencies.
5. Put all the implementation files in a zip file.



Required Artifacts for an API Implementation

An API implementation is packaged as a zip archive containing, at minimum, the following artifacts:

- A root directory that has the name of the custom code module.
- The `package.json` file. Within this file, you specify in JSON format the name of the module and any dependencies that your custom code has, such as any connector APIs. See package.json Contents for information on the contents and syntax on the `package.json` file.

> **Note:**
>
> By Node convention, this file *must* be within the root directory.

- At least one JavaScript file that contains the implementation code.
- If there are any additional modules that you are using (in addition to Express and the base Node features), then a `node_modules` directory containing those modules. See Packaging Additional Libraries with Your Implementation.

package.json Contents

Like all npm packages, custom API implementations require that you identify the project and its dependencies in a package manifest named `package.json`. Here's an example of the syntax and the properties of a `package.json` file for a custom API implementation:

```
{
  "name" : "incidentreports",
  "version" : "1.0.0",
  "description" : "FixItFast Incident Reports API",
  "main" : "incidentreports.js",
  "dependencies": {
    "async": "0.9.0"
  },
  "oracleMobile" : {
    "dependencies" : {
      "apis" : {"/mobile/custom/employees" : "3.5.1"},
      "connectors" : {"/mobile/connector/RightNow": "1.0"}
    }
  }
}
```

The key attributes are the following:

name
A descriptive name for the implementation. The name can contain only characters that can be used in a URI. It may not start with a period (`.`) or underscore (`_`). The value of this attribute in combination with the value of the `version` attribute must be unique among all API implementations.

version
The version of the implementation. If you provide a new version of an implementation, then this attribute should be incremented and the `name` value should stay the same.

description
An optional description of the implementation.

main
The name of the main JavaScript file that implements the API. If this file isn't in the same folder as the `package.json` file, then use a path name that's relative to the `package.json` folder.

dependencies
The specification of dependencies to other Node modules required for the implementation. When you have such dependencies, use `npm` to install those modules in this directory. See Packaging Additional Libraries with Your Implementation.

oracleMobile / dependencies / api
The specification of the version for a custom API or a connector API that you reference in your custom code.

Declaring the API Implementation Version

Use the `version` attribute in the `package.json` file for the custom code module to specify the implementation version, as shown in the following example:

```
{
  "name" : "incidentreport",
  "version" : "1.0.0",
  "description" : "Incident Report Custom API",
  "main" : "incidentreport.js",
  "oracleMobile" : {
    "dependencies" : {
      "apis" : { },
      "connectors" : {"/mobile/connector/RightNow": "1.0"}
    }
  }
}
```

If you have previously uploaded an implementation and that implementation is still in Draft state, then you can continue to upload modified implementations without incrementing the version number. After you publish a version, that version is final. If you want to make changes to a published implementation, then you must increment the version number.

You can publish implementations independently of APIs, and you can increment their version numbers separately as well. This lets you make changes to a published implementation, such as minor modifications or bug fixes, without requiring the API itself to be updated.

To create another version of an API implementation, change the `version` attribute, such as `"version": "1.0.1"`, and then upload a zip file of the modified implementation. When you upload a new version of an implementation, it becomes the default version (active implementation) for that API. You can change the default version in the API's Implementations page.

If the new version is backward-compatible, then use a minor incremental increase. For example, if the previous version is 1.3, then the updated version number could be 1.4 or 1.7. If the new version isn't backward-compatible, then use a major incremental increase. For example, if the previous version is 1.3, then the updated version number could be 2.0 or 2.1.

For more information about publishing and deploying APIs, see Lifecycle Scenarios.

Declaring the Node Version

To use a version of the Node library other than the instance's default version, add a node property to the `configuration` section as shown in the following example:

```
{
  "name" : "incidentreport",
  "version" : "1.0.0",
  "description" : "Incident Report Custom API",
  "main" : "incidentreport.js",
  "oracleMobile" : {
    "configuration" : {
      "node" : "6.10"    }
  }
}
```

To learn about the default Node version and the available node versions, see `CCC_DefaultNodeConfiguration` in Environment Policies and Their Values.

Packaging Additional Libraries with Your Implementation

If your API implementation depends on other JavaScript modules, such as Async, then you must add them to your custom code zip file. The additional modules aren't shared across APIs. For example, you must include the Async module in every implementation package that uses it. Your implementation can't use any modules that depend on installing a binary (executable) on the server.

1. In the `package.json` file for the implementation module, declare the modules that the implementation module depends on. Specify both the module name and the version number in the following format:

   ```
   "dependencies": {
        "Module1Name":"VersionNumber",
        "Module2Name":"VersionNumber",
   },
   ```

2. In the directory containing the `package.json` file for the custom code module, run:

   ```
   npm install
   ```

   This command downloads the stated dependencies from the public npm repository and places them in the `node_modules` subdirectory.

   > **Note:**
   >
   > If the module on which you're creating the dependency is in a folder on your file system instead of in the public npm repository, add the path to the folder as an argument to the command:
   >
   > `npm install folder-name`

For more information on using the npm package manager, see `https://docs.npmjs.com/cli/install`.

3. Package the whole folder containing the `package.json` file in a zip archive.

Uploading the Custom Code Module

1. On your system, prepare the required artifacts for the implementation, as described in Required Artifacts for an API Implementation.
2. From the API Catalog, open the custom API that the custom code implements.
3. In the left navigation bar, click **Implementations**.
4. At the bottom of the API Implementation page, click **Upload an implementation archive**, and then go to the implementation zip file on your system.

> **Note:**
>
> You also can upload an implementation from the command line. See Offline Debugging with the MCS Custom Code Test Tools.

Managing Custom Code in Git

When you first generate a JavaScript scaffold for your implementation, you can have it pushed directly to a Git repository.

Setting Up a Git Repository for Custom Code

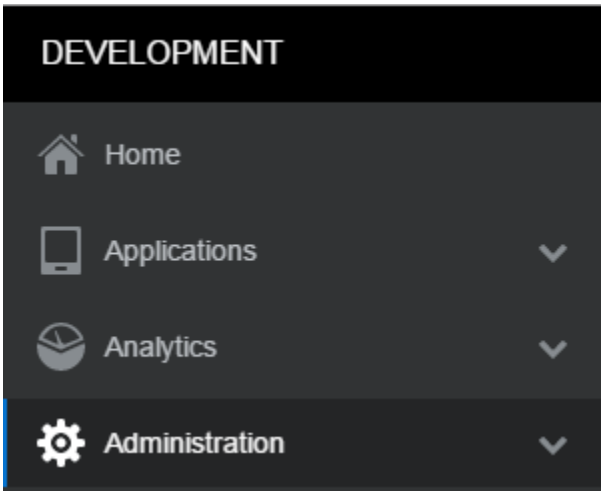
As a mobile cloud administrator, you can specify a Git repository in which to store your team's custom API implementations. This enables your team's developers to put their API implementations under version control starting with the creation of the JavaScript scaffold for the API.

You can use either an existing Git repository or create a new one in Oracle Developer Cloud Service to use. To do the latter, see Using Git in Oracle Developer Cloud Service in *Using Developer Cloud Service*.

Designating a Git Repository for Custom Code

If you already have a Git repository set up that you want to use for you team's custom code, you can specify that repository as the place where new JavaScript scaffolds are pushed. To designate the Git repository, you need to have access permission for the **Administration** page.

You can see if you have access to that page by clicking  and checking to see if there is an item for **Administration** as shown in the screenshot below.



1. Click to open the side menu and select **Administration**.
2. Click the link for **Developer Cloud Service Git Integration**.
3. In the dialog, enter the URL for the Git repository in the **Source Code Git Repository** field and click **Save**.

Setting Up a Git Repository in Oracle Developer Cloud Service

If you don't already have a Git repository for your custom code, you can set one up in Oracle Developer Cloud Service.

1. Make sure that you have a user account in Oracle Developer Cloud Service.

   If you have a user account, you will have received an email notification with your account details.

   If you don't have a user account, contact your team's identity domain administrator to set up an account for you (and for anybody else who will be using the repository). You need the Developer Service User role (`DEVELOPER_USER`) in Oracle Developer Cloud Service.
2. Sign in to Oracle Developer Cloud Service.
3. Set up a project to hold the Git repository by following the steps at Creating a Project in the Oracle Developer Cloud Service docs.

   As part of creating the project, a Git repository will be created.

See Managing Git Repositories in Oracle Developer Cloud Service.

Generating a Scaffold in a Git Repository

1. Click to open the side menu, and then click **APIs**.
2. Open the API that you want to implement.
3. In the left navigation area of the API Designer, click **Implementations**.
4. Click **New Implementation > Push to Git Repository**.

You can now clone the repository (or pull from the repository if you have already cloned it) to fill in the scaffold with your custom code.

Testing and Debugging Custom Code

You can test and debug your custom code directly within the UI. It's also possible to test your custom code outside of the UI.

Testing with Mock Data

When you create a custom API, you get a mock implementation, which application developers can use to test their mobile applications while you are implementing the custom code. When you call an endpoint for a mock implementation, it returns the request example, if one has been provided.

The mock implementation is the default implementation until you upload an implementation. Whenever you upload an implementation, it is automatically deployed as the default implementation. You can always change this, including reverting to the mock implementation, for testing purposes. To change the default implementation, select it on the Implementations page and click **Set as Default**.

You can create example (mock) data to provide default request and response bodies for the test UI. You can use either the API Designer or the RAML to add example (mock) data. To provide an example for an endpoint from the API Designer, from the Endpoints page, go to the desired method, click either the **Requests** tab or the **Responses** tab, select the appropriate media type, and then enter the mock data in the **Example** tab.

Here is an example of providing mock data in the RAML.

```
/status:
      get:
        description: |
          Gets status of specified report.
        responses:
          200:
            description: |
              OK.
            body:
              application/json:
                example: |
                  { "code": "New",
                     "notes": "My hot water tank's model is AB234"
                  }
```

Testing Custom Code from the UI

As soon as you upload your custom API implementation, you can test it by clicking **Test** in the API Designer. You can also test from the API Catalog by selecting the API, and then clicking **Test**.

The test page displays all the operations. Click an operation, fill out the necessary fields, and then click **Test Endpoint**.

If the API isn't configured for anonymous access, then you must provide a user name and the password. The user must have been assigned one of the roles that can access the endpoint. If the endpoint doesn't have any roles configured for it, then the user must belong to a role that's associated with the API. You can learn more about roles and anonymous access at Security in Custom APIs.

> **Note:**
>
> The API must either allow anonymous access or be associated with at least one role. If neither of these is true, then you'll get an unauthenticated error.

For detailed steps on how to test the API, see Testing a Platform or Custom API from the UI.

Offline Debugging with the MCS Custom Code Test Tools

Within the zip file of the client SDK for each platform is the `mcs-tools.zip` file, which contains the MCS Custom Code Test Tools that you can use to iteratively debug your custom code.

The core of the tools is an npm module that enables you to run an offline custom code container, run tests on the code, and package and deploy an implementation back to MCS.

Detailed instructions on using the tools are located in the `README.MD` file that is packaged within the `mcs-tools.zip`.

You can get the client SDKs and the accompanying test tools from the Oracle Technology Network's MCS download page.

Other Tools for Testing Custom Code Outside of the UI

Besides the Custom Code Test Tools, you can use tools that were designed for testing web services, such as cURL. To learn how to test your custom API from these tools, see Testing Platform and Custom APIs Remotely.

> **Note:**
>
> The API must either allow anonymous access or be associated with at least one role. If neither of these is true, then you will get an unauthenticated error.

Accessing Logging Messages for Custom Code

When your API implementation doesn't return the expected results, use the diagnostic logs to troubleshoot the problem.

To pinpoint where the error occurred, click  to open the side menu. Next, click **Administration**, and then click **Request History**. Next, find the request, click **View related log entries**  in the **Related** column, and then select **Log Messages Related by API Request**. To see a message's details, click the time stamp. From the

Message Details dialog, you can click the up and down arrows to see all the related log messages.

You can get to the **Request History** page from either the **Administration** page or a mobile backend's **Diagnostics** page. Note that if there isn't sufficient information in a request to enable the service to determine the associated backend, then the related log messages appear only in the **Logs** page that is available from the **Administration** page.

Every message is tagged with a request correlation ID that associates all messages for a request. When you view a message's details, you can click the request correlation ID to see the other messages for the same request.

If you don't see any messages that help identify the source of the problem, then you can change to a finer level for logging messages. Click **Log Level** in the **Logs** page, change the log level for the mobile backend, and then rerun the test.

To learn about the different types of log messages and how to filter and correlate messages, see Viewing Log Messages. For use cases for diagnosing custom code and connector issues, see Diagnosing Custom Code.

Let's use the following endpoint to see how to custom code logging works. In this code, the `ums.updateUser` method makes a `PUT` request to `/platform/users/~`.

```
service.put(
  '/mobile/custom/incidentreport/key',
  function (req, res) {
    req.oracleMobile.ums.updateUser({key: req.body.key}).then(
      function (result) {
        res.send(result.statusCode, result.result);
      },
      function (error) {
        res.send(error.statusCode, error.error);
      }
    );
  });
```

The service always logs a message whenever a call ends, regardless of the log level setting. In the following figure, the bottom (earliest) message was logged when the `PUT` request to `/platform/users/~` ended. The top (later) message was logged when the `service.put` call to `/mobile/custom/incidentreport/key` ended.

Timestamp	Messag...	Call	Message
4/29/15 01:21:15.022 ...	INFO	FiF 1.0 > incidentreport 2.0 > ...	The API invocation ended.
4/29/15 01:21:15.015 ...	INFO	FiF 1.0 > users 1.0 > PUT /{us...	The API invocation ended.

Logging Request and Response Messages

If you would like to see the bodies of the requests and responses, then ask your mobile cloud administrator to change the `CCC_LogBody` environment policy to `true`. When you do this, the service logs a CCC message whenever a body is passed in a

request or a response. The following figure shows the log entries for a request to the example endpoint. In addition to the standard call messages, this log also has a message for each of the following events (reading from the bottom (earliest) up):

- When the `PUT /mobile/custom/incidentreport/key` request was received.
- When the `ums.updateUser` method made the `PUT /platform/users/~` request .
- When the response was returned by the `PUT /platform/users/~` operation.
- When the response was returned by the `PUT /mobile/custom/incidentreport/key` operation.

When you set the log level to `Info`, the service logs the request bodies with a message type of `INFO`. Response bodies are logged with a message type that corresponds to the response status. For example, if the response status is `401`, then the log message that contains the response body has a message type of `WARNING`.

Message Type	Call	Message
INFO	FiF 1.0 > incidentreport 2.0 > P...	The API invocation ended.
INFO	FiF 1.0 > incidentreport 2.0	CCC response body: {"id":"4f88e008-9ff4-41fd
INFO	FiF 1.0 > incidentreport 2.0	OracleMobile.rest response rid:0:4:1 body: {"id
INFO	FiF 1.0 > users 1.0 > PUT /{use...	The API invocation ended.
INFO	FiF 1.0 > incidentreport 2.0	OracleMobile.rest request rid:0:4:1 body: {"key
INFO	FiF 1.0 > incidentreport 2.0	CCC request body: {"key":"abCdeFG123"}

Note that setting the `CCC_LogBody` environment policy to `true` might have a negative effect on performance.

> **Note:**
>
> By default, the body is truncated after 512 characters. Use the `CCC_LogBodyMaxLength` environment policy to change the maximum body length. To always include the full message, no matter how long it is, set `CCC_LogBodyMaxLength` to `-1`.

Getting More Details

To get the maximum amount of log messages, set the log level to `FINEST`. With this level, the service logs the following messages:

- A `FINEST` message, which contains the HTTP verb and URI, whenever a request is received by any of the custom API's endpoints
- A `FINEST` message, which contains the HTTP verb, URI, and status code, whenever a response is sent by any of the custom API's endpoints

- A `FINEST` message, which contains the HTTP verb and URI, whenever a request is sent to another platform or custom API.
- A `FINEST` message, which contains the HTTP verb, URI, and status code, whenever a response is received from a call to another platform or custom API.

If the `CCC_LogBody` environment policy is set to `true` and the log level is `FINEST`, then the following occurs:

- If a request body exists, then the `FINEST` message that contains the request's HTTP verb and URI also shows the body.
- If a response body exists and the response status code is less than `400`, then the `FINEST` message that contains the HTTP verb, URI, and status code for the response also shows the body.
- If a response body exists and the response status code is `400` or higher, then the response body is logged in a separate message. Immediately after, it logs the `FINEST` message for the response. The message type is either `WARNING` or `SEVERE`, depending on the status code.

Message...	Call	Message
INFO	FiF 1.0 > incidentreport 2.0 > PUT	The API invocation ended.
FINEST	FiF 1.0 > incidentreport 2.0	CCC response put /mobile/custom/incidentreport/key statusCode: 200 body...
FINEST	FiF 1.0 > incidentreport 2.0	OracleMobile.rest response put /internal-rt/mobile/platform/users/~ rid:0:4:1 ...
INFO	FiF 1.0 > users 1.0 > PUT /{usern:	The API invocation ended.
FINEST	FiF 1.0 > incidentreport 2.0	OracleMobile.rest request put /internal-rt/mobile/platform/users/~ rid:0:4:1 b...
FINEST	FiF 1.0 > incidentreport 2.0	CCC request put /mobile/custom/incidentreport/key body: {"key":"abCdeFG1...

Note that setting the log level to `FINEST` might have a negative effect on performance.

Minimizing the Performance Cost of Logging Bodies

If you are concerned about the performance cost of logging bodies, but you want to see the request and response bodies for exceptional cases, set the `CCC_LogBody` environment policy to `true`, and set the logging level to `WARNING` or `SEVERE`. With these settings, whenever there is a status code of `400` or higher, a message is logged for both the request and the response. Both messages are logged at the time that the response is received. The message type is `WARNING` or `SEVERE`, depending on the status code. The message shows the body, if there is one.

Message Type	Call	Message
WARNING	FiF 1.0 > incidentreport 2.0 > ...	The API invocation ended.
WARNING	FiF 1.0 > incidentreport 2.0	CCC response body: {"type":"http://www.w3.o
WARNING	FiF 1.0 > incidentreport 2.0	CCC request body: {"keys":"abCdeFG123"}
WARNING	FiF 1.0 > incidentreport 2.0	OracleMobile.rest response rid:0:4:1 body: {"
WARNING	FiF 1.0 > incidentreport 2.0	OracleMobile.rest request rid:0:4:1 body: {"ke
WARNING	FiF 1.0 > users 1.0 > PUT /{us...	The API invocation ended. We can't update t

Creating Custom Log Messages

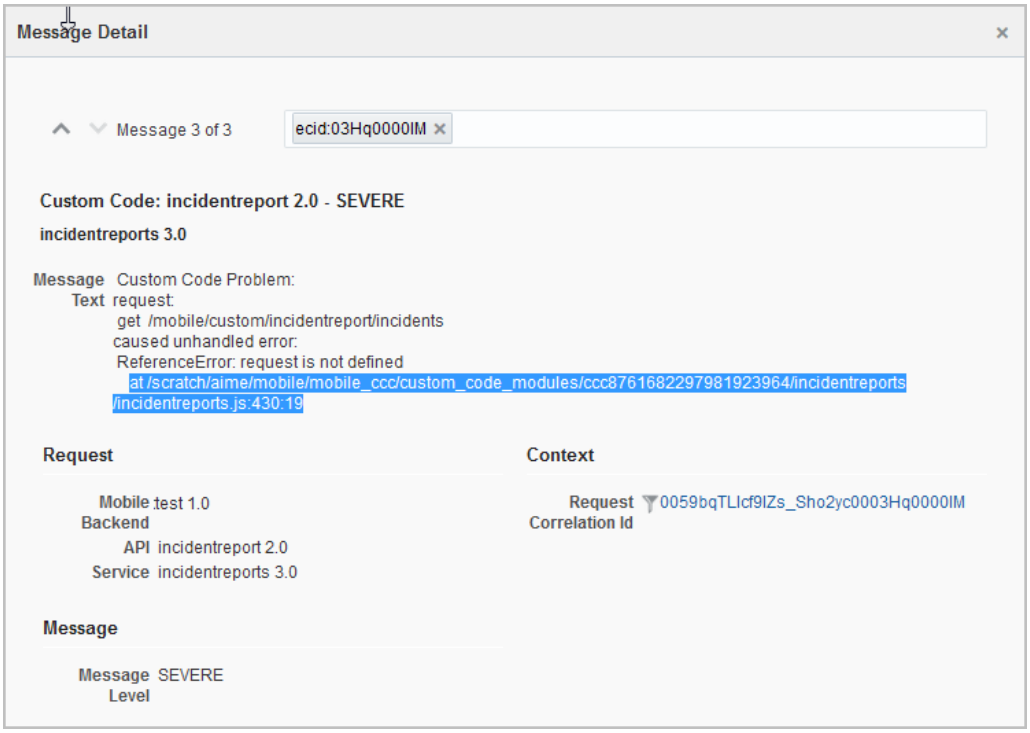
To help with debugging, you can use the `console` object in your code to generate your own messages, as described in Inserting Logging Into Custom Code, and then view them from the logs.

Troubleshooting Custom API Implementations

The following topics provide information about diagnosing and resolving common problems in custom code.

Diagnosing Syntax Errors

If a request failure is caused by a syntax error, then the Message Detail dialog box for the associated log message displays the module and line number where the error occurs, as shown here:



To learn about accessing log messages, see Viewing Log Messages.

If you'd like to see the stack traces for custom code syntax errors in request responses, then ask your mobile cloud administrator to change the `CCC_SendStackTraceWithError` environment policy to `true`. When you do this, you'll see a request response like the following example whenever a request results in a syntax error in the custom code. The stack trace shows the line number where the error occurred.

```
{"message": "Custom Code Problem: ReferenceError: nonExist is not defined
\n at /scratch/aime/mobile/mobile_ccc/custom_code_modules/
ccc2455344468806884059/incidentreports/incidentreports.js:354:17\n at
callbacks (/scratch/aime/mobile/mobile_ccc/mcs-node-router/node_modules/
express/lib/router/index.js:164:37)\n  ..."
}
```

Common Custom Code Errors

The following topics discuss common errors, possible causes, and solutions.

Custom Code Problem parsing JSON: Error: request entity too large

This error is typically caused by a request body that's larger than the JSON body parser's default maximum input, which is 1MB.

To change the JSON body parser limit for Node 6.10 and later, add this code to the implementation's main JavaScript file, and set the desired limit:

```
var bodyParser = require('body-parser');
module.exports = function(service) {
```

```
      service._router.stack[3].handle = bodyParser.json({limit: '2mb'})
};
```

To change the JSON body parser limit for Node 0.10, add this code to the implementation's main JavaScript file, and set the desired limit:

```
var bodyParser = require('body-parser');
module.exports = function(service) {
    service.stack[3] = { route: "", handle: express.json({limit: '2mb'})
};
```

Custom Code Problem in oracleMobile.rest callback: Argument error, options.body

The common cause for this error is assigning a JavaScript object to `optionsList.body`, where `optionsList` is the first parameter in a call to `req.oracleMobile.rest.post(optionsList, handler)`.

The solution is to do one of the following:

- Store the object in `options.json`, instead of `optionsList.body`. This solution automatically converts the object to a JSON string and sets relevant parts of the request, such as the content type and length. For example:

  ```
  optionsList.json = {first: 'John', last: 'Doe'};
  ```

- Use `JSON.stringify` to convert the object to a JSON string before setting the `optionsList.body` value. For example:

  ```
  optionsList.body = JSON.stringify(first: 'John', last: 'Doe'};
  optionsList.headers = {'Content-Type': 'application/json');
  ```

Your custom code container is in the process of recovering from an unhandled error in a earlier request

This issue occurs when a previous request results in an uncaught exception. When you receive this response, rerun the current request. It should succeed as soon as the system has recovered from the uncaught exception for the previous request.

You should examine the logs for the previous requests to see if you can find the cause of the uncaught exception.

Connection fails due to untrusted URL

To protect client apps, the service passes all external URLs through McAfee Web Gateway v7.x/6.9.x (Cloud), which requires that all external URLs are trusted. This requirement applies to external service URLs for connector APIs as well as those that you access directly from custom code.

Attempting to connect with an untrusted connector endpoint results in a 403 error, which might be wrapped in a 500 error.

To resolve the issue, add the untrusted URL to the list of trusted URLS for McAfee Web Gateway v7.x/6.9.x (Cloud) at `http://trustedsource.org/`. Note that the process can take from three to five business days.

database.getAll(table, options, httpOptions) doesn't return all the rows in a table

This issue occurs when there are more rows in the table than the `Database_MaxRows` environment policy allows the service to return. The default value is `1000`.

Ask your mobile cloud administrator to increase the `Database_MaxRows` value.

This mobile user doesn't have the necessary permissions to call this endpoint

In the UI, open the API and click **Security**. If **Login Required** is turned on and **Enterprise** is selected, then look at the roles that have been configured. If no roles are configured, then no one has permission to log in to the mobile backend. If one or more roles are configured, ensure that the user has a necessary role.

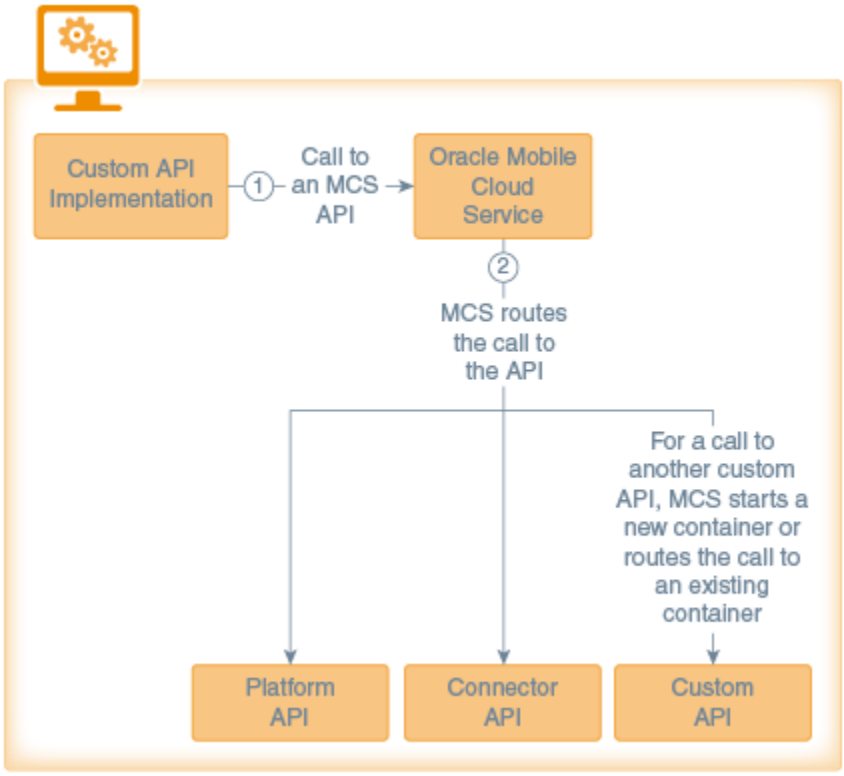
What Happens When a Custom API Is Called?

You might be curious about how the service handles calls to a custom API. Here is a high-level summary. When the service receives a custom API request, it sends the request to the custom code service. The custom code service then directs the request to one of the following:

- Custom code container for the API implementation: A container is a Node instance. This container wraps the custom API implementation with JavaScript that handles tasks such as server startup, authentication, authorization, and logging. There is one container for each deployed version of an implementation for each associated mobile backend version.
- Custom code agent: The agent controls the creation and destruction of custom code containers, controls server startup, and exposes the REST endpoints for creating and destroying a container.

Basically, a custom API implementation is launched on demand in a container that is instantiated by the custom code agent. This container, which runs in Node, handles the requests and returns the responses.

When the custom code calls a platform API or a connector API, it makes the call back through the service, and then the service routes the call to that API. If the call is to a different custom API, then the service routes the call to that API's container if it exists, or it creates the container and then routes the call to it.


Custom API Implementation
1
Call to an MCS API
Oracle Mobile Cloud Service
2
MCS routes the call to the API
For a call to another custom API, MCS starts a new container or routes the call to an existing container
Platform API
Connector API
Custom API

23
Calling APIs from Custom Code

As a service developer, you might want to access platform APIs, connector APIs, and other custom APIs from your custom code. The custom code SDK provides methods that simplify making requests to these APIs.

How to Send Requests to MCS APIs

You use custom code SDK methods to send requests to MCS APIs. When you call one of these methods, that method makes a RESTful HTTP call to the MCS API. This SDK makes the HTTP calls mostly transparent to you, but you'll see that a method's arguments and its return value are similar to what you would see with a RESTful HTTP request and response.

These methods and their arguments conform to a common pattern. This section describes this pattern, and the following sections provide the details that are specific to the API's methods:

- Calling Platform APIs from Custom Code
- Calling Custom APIs from Custom Code
- Calling Connector APIs from Custom Code

> **Note:**
>
> Note: The purpose of the examples in this chapter is to illustrate how to interface with the custom code service. The examples are not intended to teach best practices for writing Node.js REST API implementations.

API Request Pattern

The custom code SDK methods that make requests to custom, platform, and connector APIs follow this pattern:

```
req.oracleMobile.<service>.<method>(required arguments, options,
httpOptions)
```

The `<service>` identifies the API that you want to call.

- For platform APIs, this is the name of the platform, such as `storage`, `ums`, or `notification`.
- For connector APIs, this can be either `connectors` or `connectors.<api>`. Later, we discuss how to choose which one to use.
- For custom APIs, this can be either `custom` or `custom.<api>`. Later, we discuss how to choose which one to use.

You use `options` to specify optional API-specific properties. The next section discusses the `options` properties that are shared by many of these methods. Each method description in the subsequent sections discusses additional `options` properties that apply to that method, if any.

The `httpOptions` argument is like the Node.js `http.request(options)` argument. You use this argument to pass properties not covered by required arguments and `options`. For example, if you need to pass the `timeout` property to specify the number of milliseconds to wait for a request to respond before terminating the request, then you would pass it in `httpOptions`. Another example of when you use `httpOptions` is to pass query parameters to a connector. To learn more about `http.request(options)`, go to the API documentation at https://github.com/mikeal/request and scroll down to the section entitled "request(options, callback)".

> **Tip:**
>
> When you use `httpOptions.qs` to pass the query string, you can use `encodeURIComponent(<string>)` for the `qs` value to ensure that your code handles multibyte characters.

You can omit the `options` and `httpOptions` arguments. When you do so, they are treated as null values. Any value that you provide in `options` that affects a parameter in `httpOptions` overrides the `httpOptions` parameter. The methods ignore any property in the `options` and `httpOptions` arguments that they do not support.

> **Note:**
>
> You might notice that you don't need to worry about authentication when you send requests to custom, platform, and custom APIs from custom code. The service re-uses the access token that's passed into the custom code and takes care of authentication for you. With connectors, if you need to use different credentials for the external service, you can use `options.externalAuthorization` to pass the value to be used in the `Authorization` header for the external service.

To learn how to send direct requests to third-party web services without going through a connector, see Calling Web Services and APIs from Custom Code.

Common options Argument Properties

Several custom code SDK methods that access APIs accept an optional `options` argument, which is a JSON object. Here are the `options` properties:

Property	Description	Type	Default Value
`accept`	The value for the `Accept` header. Use this property to list the media types that you prefer for the response body. Note that for most methods, the media type for a response body is `application/json`.	String	Empty, which indicates no preference for response type.
`contentType`	The value for the `Content-Type` header. This property specifies the content type of the request body. For most methods, this is `application/json`.	String	Empty. Note that if the `inType` is `json`, then the service sets the `Content-Type` header to `application/json`.
`inType`	For Storage, connector API, and custom API SDK functions that take a request body, use this option to specify whether the request body is `json` or `stream`. If `json`, then the method sets the `Content-Type` header to `application/json` automatically. You typically set this property when the custom code builds the request body that you are sending to the API.	String	Undefined. If this property isn't set, then the method passes the request body as is. The request is serviced by the Node Request module, which accepts a string or a buffer.

Property	Description	Type	Default Value
`outType`	The response body type. The value can be one of the following: • `json`: Convert the response body to a JSON object. Note that if there are JSON parse errors, then the response body remains a string. • `stream`: Return the response body in a readable stream that can be piped. • `binary`: Do not convert the response body to a string. • `encoding`: Convert the response body to a string using the specified encoding. This property is supported only by the Storage API and the connector and custom APIs. All other APIs use the default response behavior.	String	Undefined. The response body is converted to a string using the UTF8 encoding.

Property	Description	Type	Default Value
`encodeURI`	When true, encodes the URI and the following arguments and properties: • Encodes `table`, `keys`, `fields`, `extraFields`, `primaryKeys`, and `sql` arguments and properties for `database` methods • Encodes `collectionId`, `mobileName`, `objectId`, `orderBy`, and `user` arguments and properties for `storage` methods. This option is useful for multibyte characters.	Boolean	`false`

API Response Patterns

The return value for a custom code SDK call to an API depends on the value of the `options.outType` property.

- If the `outType` is `stream`, then, if there's no error, the return value is a stream that you can pipe, as shown in Handling a Stream.
- If the `outType` is undefined or any value other than `stream`, then the return value is a promise object. To learn more about the promise object, see Handling a Promise.

Handling a Stream

When the response is a `stream`, then, if there's no error, the return value is a stream that you can pipe. Otherwise, you can process the error as shown in this example:

```
req.oracleMobile.storage.store('attachments', req, {
  mobileName: 'Technician Notes',
  contentType: req.header('content-type'),
  inType: 'stream',
  outType: 'stream'
})
.on('error', function (error) {
  res.status(error.status).send(error.message);
})
```

```
.on('response', function (response) {
  console.info('HEADERS received from response:', response.headers);
})
.pipe(res);
```

For more information about streaming, see https://github.com/request/request.

Handling a Promise

A **promise** provides access to the result of an asynchronous request. At the time a promise is returned, the request may or may not have completed. Most custom code SDK methods return promises. In the following examples, `<promiseFunction>` represents a custom code SDK method that returns a promise, such as `req.oracleMobile.storage.getCollections`.

When you call a promise function, you typically use the `then` function to handle the success or failure as shown here:

```
<promiseFunction>.then(successFunction, errorFunction)
```

- `<promiseFunction>` is the call that returns a promise, such as `req.oracleMobile.storage.getCollections` in the next code example.
- `successFunction` is a user-defined function that is called if the prior promise function resolves successfully. This occurs when the request completes with a response status code less than `400`. The `successFunction` takes a single argument, which is what the prior `<promiseFunction>` returned on success. With custom code SDK methods, this is a JSON object with the following properties:
  - `result`: The body of the result.
  - `statusCode`: The HTTP status code.
  - `headers`: A JSON object that contains all the HTTP response headers, such as `{accept-charset:'UTF-8',content-type:'application/json'}`.
  - `contentType`: The value of the `Content-Type` header if that header was included in the response.
  - `contentLength`: The value of the `Content-Length` header if that header was included in the response.
- `errorFunction` is a user-defined function that is called if and when promise function doesn't resolve successfully. This is when the response status is equal to or greater than `400`, or if there is a severe error. The `errorFunction` takes a single argument, which is what the `<promiseFunction>` returned on error. With custom code SDK methods, this is a JSON object with the following properties:
  - `statusCode`: The HTTP status code.
  - `error`: The body of the error or the error message.
  - `headers`: All the response HTTP headers.

> **Note:**
>
> The `then` function takes an optional `progressFunction` argument. However, the custom code SDK doesn't use this argument, and you can omit it from the call.

Here's an example of how to call a custom code SDK method to access a custom, platform, or connector API and use `then` to handle the promise that it returns. In this example:

- In this example, the `<promiseFunction>` is `req.oracleMobile.storage.getCollections`. This is a function from the storage component of the custom code SDK, which either resolves with a successful promise or rejects with an error promise.
- If `getCollections` completes successfully, then it passes the successful promise to the first argument for `then`, which is `function(result)`.
- If `getCollections` results in an error, then it passes the error promise to the second argument, which is `function(error)`.

```
// Get metadata about the backend's collections.
service.get('/mobile/custom/incidentreport/collections',
  function (req, res) {
    req.oracleMobile.storage.getCollections({sync: true}).then(
      function (result) {
        res.status(result.statusCode).send(result.result);
      },
      function (error) {
        res.status(error.statusCode).send(error.error);
      }
    );
});
```

A promise and its result can be assigned to a variable. This means that the result can live longer than the function call alone, allowing you to chain multiple success and failure functions calls against the result. For example, you can write code like this:

```
var collections = req.oracleMobile.storage.getCollections({sync: true});

collections.then(successFunction1, errorFunction1);
...
collections.then(successFunction2, errorFunction2);
```

> **Note:**
>
> Because the custom code SDK uses the Bluebird promises library, we recommend that you use this library to process these promises. If you only use the `then()` function from the promises library, then you don't need to include Bluebird in your package.
> There are several promises libraries that you can choose from for your custom code implementation, but the extent to which they will work with the custom code SDK promises is not known. To learn more about Bluebird promises, go to https://github.com/petkaantonov/bluebird.

The next sections show some common examples of ways in which you can handle promises.

Chaining Calls

When you need to invoke a series of calls in a synchronous manner, waiting for one operation to complete before starting the next one, then you can take advantage of the fact that most custom code SDK methods return a promise. A promise handles some of the complexity of making synchronous calls in an asynchronous environment like Node, and provides a simple way to handle both success and failure cases through callback methods.

As we discussed in API Response Patterns, the simplest way to extract the result of a promise is to use the `then` function. In your custom code, you can provide two arguments to the `then` function.

- A function to invoke on success, which takes a single argument – the success promise.
- A function to invoke on error, which takes a single argument – the error promise.

Here's an example of using the `then` function to handle the result of a promise function. As you can see, it has two arguments:

- `function(result)`, which sends the `getById` result.
- `function(error)`, which sends the error message.

```
service.get('/mobile/custom/incidentreport/attachments/:id',
  function (req, res) {
    req.oracleMobile.storage.getById('attachments', req.params.id, {sync:
true}).then(
      function (result) {
        res.status(result.statusCode).send(result.result);
      },
      function (error) {
        res.status(error.statusCode).send(error.error);
      }
    );
  });
```

When you need to call more than one API operation from a route definition, you can use `then` to chain the calls, so that one call completes successfully before the next one is called. In this example, the route definition:

1. Posts an incident to the database and returns the result.
2. If the post completes successfully, gets the user info.
3. If the user info is retrieved successfully, posts an analytics event.

Notice that none of the `then` functions take a second argument (the error function). If an error (rejected) promise is passed to a `then` function that doesn't have a second argument, then the code skips to the first `then` function with a second argument. In this example, because there aren't any, all errors trickle to the `catch` function.

```
service.post('/mobile/custom/incidentreport/incidents',
function (req, res) {

  /* Post the incident and send the response.
   * Then, if the post was successful,
   * get the username,
   * then use the username to post an event.
   *
   **/
  postIncident()
  .then(getUser)
  .then(postEvent)
  .catch(function (errorResult) {
        console.warn(errorResult);
  });

  function postIncident() {
    return req.oracleMobile.database.insert('FIF_Incidents', req.body)
    .then(
      function (successResult) {
        res.status(successResult.statusCode).send(successResult.result);
        // By default, Bluebird wraps this with a
        // resolved promise
        return {status: "resolved"};
      },
      function (errorResult) {
        res.status(errorResult.statusCode).send(errorResult.error);
        throw errorResult;
      }
    );
  };

  function getUser() {
    return req.oracleMobile.ums.getUser({fields: 'username'});
  };

  function postEvent(successResult) {
    var userName = successResult.result.username;
    /*
     * Record the NewIncident event
     */
    var timestamp = (new Date()).toISOString();
```

```
    // Events are posted as an array
    var events = [];
    // Put events in context
    events.push(
      {name: 'context',
        type: 'system',
        timestamp: timestamp,
        properties: {userName: userName}
      });
    // Start the session
    events.push(
      {name: 'sessionStart',
        type: 'system',
        timestamp: timestamp
      });
    // Add the custom event:
    events.push(
      {name: 'NewIncident',
        type: 'custom',
        component: 'Incidents',
        timestamp: timestamp,
        properties: {customer: req.body.customer}
      });
    // End the session:
    events.push(
      {name: 'sessionEnd',
        type: 'system',
        timestamp: timestamp
      });
    // Post the batch of events. Apply the passed-in session ID to all.
    // The postEvent result is returned by this function
    return req.oracleMobile.analytics.postEvent(
      events,
      {sessionId: req.header('oracle-mobile-analytics-session-id')});
  };
});
```

Joining Calls

`Promise.join` lets you make several asynchronous calls and then use the results after all calls are complete. The promise that the join returns is an array of the results.

For example, the following code makes three calls to the `incidentreport` custom API to get information for the result body. After all calls complete successfully, the `then` function's success handler extracts the necessary information to compile the result, and then sends it.

Note that the `join` functions aren't necessarily called in the order in which they occur in the code. The only guarantee is that all the `join` functions successfully complete before a success promise is returned.

```
/* Promise.join example
 *
 * Promise.join takes multiple promises as arguments.
 * If all promises succeed, then it returns a promise
```

```
 * that holds an array of the results of the promises.
 */
var Promise = require("bluebird");
module.exports = function(service) {
  ...
  service.get('/mobile/custom/incidentreport/
join/:custId/:incidentId/:techId', function (req, res) {
    // Three functions that return promises.
    var customer = req.oracleMobile.custom.incidentreport.get(
      "customers/" + req.params.custId, {outType: 'json'});
    var incident = req.oracleMobile.custom.incidentreport.get(
      "incidents/" + req.params.incidentId, {outType: 'json'});
    var technician = req.oracleMobile.custom.incidentreport.get(
      "technicians/" + req.params.techId, {outType: 'json'});

    Promise.join(customer, incident, technician).then(
      function (joinResult) {
        // Anonymous handler that's called if all 3 promises succeeded.
        // Harvest a piece of data from each promise result.
        var report = {
          customerContact: joinResult[0].result.email,
          description: joinResult[1].result.title,
          technicianContact: joinResult[2].result.email};
        res.type('application/json');
        res.status(200).send(report);
      },
      function (error) {
        // Anonymous handler to handle errors
        console.info(error);
        res.status(error.statusCode).send(error.error);
      }
    );
  })
  ...
}
```

Waiting for a Dynamic Set of Calls to Complete

Use `Promise.all` when you have a dynamic set of calls and you must wait until all calls complete before you take some action. If any of the promises in the array don't succeed, then the returned promise is rejected with the reason for rejection.

```
/* Promise.all example
 *
 * Promise.all takes an array of promises as an argument (promiseArray).
 * If all promises succeed, then it returns a promise that holds
 * an array of the results from the promiseArray's promises.
 */

var Promise = require("bluebird");
module.exports = function(service) {
...
  service.get('/mobile/custom/incidentreport/
all/:custId/:incidentId/:techId', function (req, res) {
```

```
    // Put the functions that return promises in the array
    promiseArray = [];
    promiseArray.push(req.oracleMobile.custom.incidentreport.get(
      "customers/" + req.params.custId, {outType: 'json'}));
    promiseArray.push(req.oracleMobile.custom.incidentreport.get(
      "incidents/" + req.params.incidentId, {outType: 'json'}));
    promiseArray.push(req.oracleMobile.custom.incidentreport.get(
      "technicians/" + req.params.techId, {outType: 'json'}));
    // Call Promise.all with the array
    Promise.all(promiseArray).then(
      function (allResult) {
        var report = {
          customerContact: allResult[0].result.email,
          description: allResult[1].result.title,
          technicianContact: allResult[2].result.email};
        res.type('application/json');
        res.status(200).send(report);
      },
      function (error) {
        console.dir(error);
        res.status(error.statusCode).send(error.error);
      }
    );
  })
  ...
}
```

Creating a Function that Returns a Promise

Here are some examples of creating and using functions that return a promise. The first example shows how to return a resolved promise and a rejected promise.

```
// Simple function that returns a resolved promise.
// Note the object passed to Promise.resolve is the
// object the promise is resolved with.
function resolve() {
  return Promise.resolve({status: "resolved"});
}

// Simple function that returns a rejected promise.
// The object passed to Promise.reject describes the error.
function reject() {
  return Promise.reject({error: "rejected"});
}
```

In this example, the `compareEtags` function takes a successful (resolved) promise as its argument. It rejects the promise if the request had an ETag header and the ETag for the result doesn't match the ETag passed in the header.

```
var Promise = require("bluebird");
var etag = require('etag');
module.exports = function(service) {
...
```

```
  service.get('/mobile/custom/incidentreport/incidents/:id/ifmatch',
function (req, res) {
    function compareEtags(result) {
      thisEtag = result.headers.etag;
      if (req.header('if-match') &&
        thisEtag != req.header('if-match')) {
        return Promise.reject({
          statusCode: 412,
          error: "Precondition Failed" +". If-Match ETag: " +
req.header('if-match') + ", this Etag: " + thisEtag
        })
      } else {
        // result is already a resolved promise
        return result;
      }
    }
    // The custom code SDK get method returns a promise,
    // which is then passed to the custom function compareEtags.
    // On success, compareEtags passes the result from the get.
    // If there's an ETag header, then the function rejects the
    // promise if the result's ETag doesn't match.
    //
    // All rejections are caught by the last then.
    req.oracleMobile.custom.incidentreport.get(
      "incidents/" + req.params.id, {outType: 'json'})
      .then(compareEtags)
      .then(
        function (result) {
          // res.setHeader('Etag', etag(JSON.stringify(result.result)));
          res.status(result.statusCode).send(result.result);
        },
        function (error) {
          res.status(error.statusCode).send(error.error);
        }
      );
  });
...
}
```

Accessing Mobile Backend Information from Custom Code

The MBE API lets you inquire about the mobile backend that the request is coming from.

This API has one method.

mbe.getMBE()

This method retrieves information about the backend that made the request. Note that this method is synchronous and doesn't return a promise.

Arguments

This method doesn't have any required arguments and doesn't take the `options` and `httpOptions` arguments.

Response

The response body is a JSON object that contains the `name`, `version`, and `id` properties.

Examples

Here's an example of calling this method to get the backend's name and version number to pass to the Notifications API:

```
service.get('/mobile/custom/incidentreport/notifications',
  function (req, res) {
    req.oracleMobile.notification.getAll({
      mbe: req.oracleMobile.mbe.getMBE().name,
      version: req.oracleMobile.mbe.getMBE().version})
    .then(
      function (result) {
        res.status(result.statusCode).send(result.result);
      },
      function (error) {
        res.status(error.statusCode).send(error.error);
      }
    );
  });
```

Here's an example of the JSON object that the method returns:

```
{
  name: 'myMBE',
  version: '1.0',
  id: 'ab72abb7-b337-4673-8584-ca5163df5d24'
}
```

Calling Platform APIs from Custom Code

You can use the `req.oracleMobile.<service>` methods to call a platform API, where `<service>` identifies the platform that you want to call. These subsections provide the details for each platform:

- Accessing the Analytics API from Custom Code
- Accessing the App Policies API from Custom Code
- Accessing the Database Access API from Custom Code

- Accessing the Devices API from Custom Code
- Accessing the Location API from Custom Code
- Accessing the Location Management API from Custom Code
- Accessing the Notifications API from Custom Code
- Accessing the Storage API from Custom Code
- Accessing the Mobile Users API from Custom Code

Further details, such as the HTTP response status codes and the schema for the request and response bodies, can be found in REST APIs for Oracle Mobile Cloud Service.

Accessing the Analytics API from Custom Code

The Analytics API lets you log runtime events, such as a user submitting an inquiry or placing an item into a shopping cart, so that you can observe performance and usage patterns.

For information about what you can do with the posted events and how you can report on them, see Analytics.

This API has one method.

analytics.postEvent(events, options, httpOptions)

This method accepts a batch of events and validates them. If they are valid, they are sent to the Analytics database. If one or more events in a batch are not valid, then no events are sent to the Analytics database.

When adding events to the batch, keep the following in mind:

- There are two types of events — `custom` and `system`. Use the custom events to record the events that you want to analyze. Use the system events to group your custom events. Note that if you don't specify the event type, then the event defaults to `custom`. To learn more about each type, see:
  - Tracking Sessions and Logging Events for Mobile Apps
  - Defining Sessions
- Events are JSON objects. All events must have a `name` and a `timestamp`, and `component` and `properties` are optional.
- With custom events, you can add your own custom properties to `properties`. For example:

```
{
  name: 'NewIncident',
  type: 'custom',
  component:  'Incidents',
  timestamp: timestamp,
  properties: {customer: 'Lynn White'}
}
```

Custom properties must be strings and the property names can’t be reserved names. For the list of reserved names, see Adding Custom Properties to Events.

- You can group events by session. For example, a session can mark the beginning and ending of a function within the application or when an application starts and stops. You start a session by adding a system event with the name `sessionStart`. You use a `sessionEnd` event to end the session.

  You use a user-defined session ID to associate events with a session. You have two ways to specify a session ID for an event. You can add a `sessionId` property to an event, and you can set the `options.sessionId` property. Here's examples of starting and stopping a session. In these examples, the session ID is set explicitly, but you can also set it using `options.sessionId`.

  ```
  {
    name: 'sessionStart',
    type: 'system',
    sessionId: '2d64d3ff-25c7-4b92-8e49-21884b3495ce',
    timestamp: timestamp
  }
  {
    name: 'sessionEnd',
    type: 'system',
    sessionId: '2d64d3ff-25c7-4b92-8e49-21884b3495ce',
    timestamp: timestamp
  }
  ```

- If you want to provide context to a session, then precede the `sessionStart` event with a system event named `context`. You can also intersperse context events with custom events to indicate changes in context, such as a location change. Here's an example of a context event:

  ```
  {
    name: 'context',
    type: 'system',
    timestamp: timestamp,
    properties: {
      userName: 'joe',
      model: 'iPhone5,1',
      longitude: '-122.11663',
      latitude: '37.35687',
      manufacturer: 'Apple',
      osName: 'iPhone OS',
      osVersion: '7.1',
      osBuild: '13E28',
      carrier: 'ATT'
  }
  ```

- To associate the batch of events with an application, include the `options.applicationKey` property set to the application's key.

For information about what you can do with the posted events and how you can report on them, see Analytics.

Arguments

`events`: Required. This is an array of `event` objects. To learn about the `event` properties, see the `POST /mobile/platform/analytics/events` operation in REST APIs for Oracle Mobile Cloud Service.

`options`: Optional. JSON object. This object can have these properties in addition to those listed in Common options Argument Properties:

Property	Description	Type	Default
`applicationKey`	Identifies the application key that MCS assigns to your application when you register it with the mobile backend. You can find this key in the **Clients** page for the mobile backend. For example, `9a5b4150-c756-4758-87c3-ec2814289799`.	String	None
`deviceId`	Identifies the device. This is the ID that is returned when you register the device with MCS using the Devices API.	String	None
`sessionId`	Specifies a default session ID. Use a session ID to group all events by a user-defined session. When present, the `sessionID` value in the `event` object overrides this value.	String	None

Response

The response body is a JSON object with a `message` attribute. For example, `{"message":"1 events accepted for processing."}`

Example

Here's an example that records events when incidents are created. After it successfully saves an incident in the database, it gets the user name for the context event, and then it records the event. This example uses the promises `then()` function to insure that each API call completes successfully before invoking the next, as described in Chaining Calls.

In this example, the request body looks like this:

```
{
  title:'Water heater is leaking',
  technician:'jwhite',
  customer:'Lynn Smith'
}
```

This code expects the request to include the session ID in the `Oracle-Mobile-Analytics-Session-ID` header. It sets the `options.sessionId` property to this value.

```
service.post('/mobile/custom/incidentreport/incidents',
function (req, res) {

  /* Post the incident and send the response.
   * Then, if the post was successful,
   * get the username,
   * then use the username to post an event.
   *
   **/
  postIncident()
  .then(getUser)
  .then(postEvent)
  .catch(function (errorResult) {
        console.warn(errorResult);
  });

  function postIncident() {
    return req.oracleMobile.database.insert('FIF_Incidents', req.body)
    .then(
      function (successResult) {
        res.send(successResult.statusCode, successResult.result);
        // By default, Bluebird wraps this with a
        // resolved promise
        return {status: "resolved"};
      },
      function (errorResult) {
        res.send(errorResult.statusCode, errorResult.error);
        throw errorResult;
      }
    );
  };

  function getUser() {
    return req.oracleMobile.ums.getUser({fields: 'username'});
  };

  function postEvent(successResult) {
    var userName = successResult.result.username;
    /*
     * Record the NewIncident event
     */
    var timestamp = (new Date()).toISOString();
    // Events are posted as an array
    var events = [];
    // Put events in context
    events.push(
      {name: 'context',
        type: 'system',
        timestamp: timestamp,
        properties: {userName: userName}
      });
    // Start the session
```

```
    events.push(
      {name: 'sessionStart',
        type: 'system',
        timestamp: timestamp
      });
    // Add the custom event:
    events.push(
      {name: 'NewIncident',
        type: 'custom',
        component: 'Incidents',
        timestamp: timestamp,
        properties: {customer: req.body.customer}
      });
    // End the session:
    events.push(
      {name: 'sessionEnd',
        type: 'system',
        timestamp: timestamp
      });
    // Post the batch of events. Apply the passed-in session ID to all.
    // The postEvent result is returned by this function
    return req.oracleMobile.analytics.postEvent(
      events,
      {sessionId: req.header('oracle-mobile-analytics-session-id')});
  };
});
```

Accessing the App Policies API from Custom Code

The App Policies API lets you retrieve the app policies that have been set for the current mobile backend. For example, a mobile backend might have app policies for the string that appears in an app's welcome message, the background color, and a timeout value.

This API has one method.

appConfig.getProperties(httpOptions)

This method retrieves the app policies that have been set for a mobile backend. These are the policies that you create from the mobile backend's App Policies page.

See App Policies.

Arguments

This method doesn't have any required arguments and doesn't take the `options` argument.

Response

The response body is a JSON object where the name/value pairs represent the app policies.

Examples

Here's an example of calling this method:

```
service.get(
  '/mobile/custom/incidentreport/appPolicies',
  function (req, res) {
    req.oracleMobile.appConfig.getProperties().then(
      function (result) {
        res.status(result.statusCode).send(result.result);
      },
      function (error) {
        res.status(error.statusCode).send(error.error);
      }
    );
  });
```

Here's an example of the response body:

```
{
  "fifBgColor": "blue",
  "fifWelcomeMessage": "Hello",
  "fifShowArg": true
}
```

Accessing the Database Access API from Custom Code

You can use the Database Access API to retrieve, add, update, and delete rows in a database table. When you add a row, the API implicitly creates the table if it doesn't exist.

This API has the following methods:

- database.delete(table, keys, options, httpOptions): Deletes a row.
- database.get(table, keys, options, httpOptions): Retrieves a row from a table.
- database.getAll(table, options, httpOptions): Retrieves specified fields from all rows in a table.
- database.insert(table, object, options, httpOptions): Adds rows to a table.
- database.merge(table, object, options, httpOptions): Adds or updates rows in a table.

For detailed information about how to use these methods to create and access database tables, see Database.

database.delete(table, keys, options, httpOptions)

This method lets you delete a row from the table.

Arguments

`table`: Required. String. The name of the database table to delete the row from.

`keys`: Required. String. If the table's row key is `id`, then provide the `id` value. Otherwise, provide the primary key values in the order in which the primary keys were specified when the first row was added to the table (which resulted in the creation of the table). Use an array for a composite key. For example, if the `options.primaryKeys` property was set to `incidentReport,technician` when the table was created, then the values must be listed in that order, such as `['5690','jwhite']`.

`options`: Optional. JSON object. This object can have the following property in addition to those listed in Common options Argument Properties:

Property	Description	Type	Default
`encodeURI`	Set to `true` to URI-encode the `table` and `keys` values. This option can be useful for multibyte values.	Boolean	`false`

Response

The response body is a JSON object. If the table's row key is `id`, then the response is an array that contains the deleted row's `id` value. Otherwise, the response is the `rowCount` indicating if 0 or 1 row was deleted.

Examples

Here's an example of calling the method to delete a record with the `id` specified in the request URI:

```
service.delete('/mobile/custom/incidentreport/incidents/:id',
  function (req, res) {
    req.oracleMobile.database.delete(
      'FIF_Incidents', req.params.id).then(
      function (result) {
        res.status(result.statusCode).send(result.result);
      },
      function (error) {
        res.status(error.statusCode).send(error.error);
      }
    );
  });
```

Here's an example of the response for this request.

```
{"items":[{"id":42}]}
```

Note that if you have defined primary keys for the table (instead of using the system-defined `id` column for the row key), then the response shows the `rowCount` of the deleted rows. For example:

```
{
  "rowCount": 1
}
```

database.get(table, keys, options, httpOptions)

This method lets you retrieve a row from a table.

Arguments

`table`: Required. String. The name of the database table to retrieve the row from.

`keys`: Required. String. If the table's row key is `id`, then provide the `id` value. Otherwise, provide the primary key values in the order in which the primary keys were specified when the first row was added to the table (which resulted in the creation of the table). Use an array for a composite key. For example, if the `options.primaryKeys` property was set to `incidentReport,technician` when the table was created, then the values must be listed in that order, such as `['5690','jwhite']`.

`options`: Optional. JSON object. This object can have the following property in addition to those listed in Common options Argument Properties:

Property	Description	Type	Default
`encodeURI`	Set to `true` to URI-encode the `table` and `keys` values. This option can be useful for multibyte values.	Boolean	`false`
`expectSingleResult`	Set to `true` to return an object instead of an array and to return `404` (not found) if the row for the specified `keys` doesn't exist.	Boolean	`false`

Response

By default, the response body is a JSON object containing an `items` array with just one item, which contains the column names and corresponding values. To return a single object, include `options.expectSingleResult` in the request and set it to `true`.

Examples

Here's an example of calling the method to retrieve the row with the `id` specified in the request URI. Because the `expectSingleResult` option is omitted, the response body will contain an array, and the response status will always be `200`.

```
service.get('/mobile/custom/incidentreport/incidents/:id',
  function (req, res) {
    req.oracleMobile.database.get(
      'FIF_Incidents', req.params.id).then(
      function (result) {
        res.status(result.statusCode).send(result.result);
      },
      function (error) {
        res.status(error.statusCode).send(error.error);
      }
```

```
    );
  });
```

Here's an example of the response for this request.

```
{
  "items":[
    {
      "id":2,
      "createdBy":"jdoe",
      "createdOn":"2018-01-31T20:14:24.4948+00:00",
      "modifiedBy":"jdoe",
      "modifiedOn":"2018-01-31T20:14:24.4948+00:00",
      "title":"Water heater is leaking",
      "technician":"jwhite",
      "status":"Open",
      "customer":"Lynn Smith",
      "incidentReport":"7890"
    }
  ]
}
```

Here's an example of including the `expectSingleResult` option with a value of `true`. The response body will contain an object, and the response status will be `404` if the row doesn't exist.

```
service.get('/mobile/custom/incidentreport/incidents/:id',
  function (req, res) {
    req.oracleMobile.database.get(
      'FIF_Incidents', req.params.id, {expectSingleResult:
true}).then(
      function (result) {
        res.status(result.statusCode).send(result.result);
      },
      function (error) {
        res.status(error.statusCode).send(error.error);
      }
    );
  });
```

Here's an example of the response for this request.

```
{
    "id": 2,
    "createdBy": "jdoe",
    "createdOn": "2018-01-31T20:14:24.4948+00:00",
    "modifiedBy": "jdoe",
    "modifiedOn": "2018-01-31T20:14:24.4948+00:00",
    "title": "Water heater is leaking",
    "technician": "jwhite",
    "status": "Open",
    "customer": "Lynn Smith",
```

```
    "incidentReport": "7890"
}
```

database.getAll(table, options, httpOptions)

This method lets you retrieve the specified fields from all the rows in a table.

> **Note:**
>
> The `Database_MaxRows` environment policy restricts the number of rows that the service returns for this call. The default value is `1000`. Ensure that this value is sufficient for your needs. If your request doesn't return all the rows that you expected, ask your mobile cloud administrator to increase the `Database_MaxRows` value.

Arguments

`table`: Required. String. The name of the tables to retrieve the rows from.

`options`: Optional. JSON object. This object can have the following properties in addition to those listed in Common options Argument Properties:

Property	Description	Type	Default
`encodeURI`	Set to `true` to URI-encode the `table` and `fields` values. This option can be useful for multibyte values.	Boolean	`false`
`fields`	A comma separated list of the fields to return. For example, `customer, status`.	String	If you omit this argument, then the method returns all fields.

Response

The response body is a JSON object containing an `items` array, where each item represents a row, and contains the column names and corresponding values.

Examples

Here's an example of calling the method to retrieve the `customer` and `status` fields from the FIF_Incidents table:

```
service.get('/mobile/custom/incidentreport/incidents',
function (req, res) {
    req.oracleMobile.database.getAll(
      'FIF_Incidents', {fields: 'customer,status'}).then(
      function (result) {
        res.status(result.statusCode).send(result.result);
      },
      function (error) {
        res.status(error.statusCode).send(error.error);
```

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

Here's an example of the response for this request.

```
{
  "items":[
    {
      "status":"Open",
      "customer":"Lynn Smith"
    },
    {
      "status":"Completed",
      "customer":"John Doe"
    }
  ]
}
```

The `/database/objects/{table}` resource supports a query parameter to filter by column values which rows to retrieve. This example uses the `httpOptions` argument to pass a request query string that filters the results for a matching technician.

```
service.get('/mobile/custom/incidentreport/incidents',
function (req, res) {
    httpOptions={};
    httpOptions.qs = {technician : 'jwhite'};
    req.oracleMobile.database.getAll(
      'FIF_Incidents', {}, httpOptions).then(
      function (result) {
        res.status(result.statusCode).send(result.result);
      },
      function (error) {
        res.status(error.statusCode).send(error.error);
      }
    );
  });
```

database.insert(table, object, options, httpOptions)

This method lets you add one or more rows to a table.

When the `Database_CreateTablesPolicy` environment policy is `allow`, then the following actions can occur:

- If the table doesn't exist, then it is created.
- If a column doesn't exist, then the table is altered to include it.
- If the value is larger than the column size, then the column is resized.

Ask your mobile cloud administrator about the `Database_CreateTablesPolicy` environment policy setting.

Arguments

`table`: Required. String. The name of the database table to add the row to.

`object`: Required. JSON object containing the table data. If you're adding one row, then you can use this format:

```
{
  status : 'Open',
  code : '3'
}
```

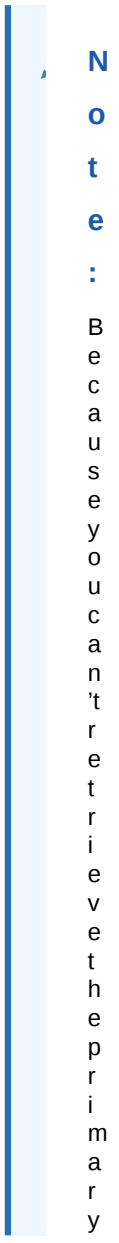
If you're adding multiple rows, then use this format:

```
[
    {
        status:'Open',
        code:3},
    {
        status:'Completed',
        code:9}
]
```

`options`: Optional. JSON object. This object can have the following properties in addition to those listed in Common options Argument Properties:

Property	Description	Type	Default
`encodeURI`	Set to `true` to URI-encode the `table`, `extraFields`, and `primaryKeys` values. This option can be useful for multibyte values.	Boolean	`false`
`extraFields`	For an implicit table creation, optionally provide a comma-separated list that specifies which predefined columns to include in the table from amongst `id`, `createdBy`, `createdOn`, `modifiedBy`, and `modifiedOn`. For example, `createdOn,createdBy`. To not include any predefined columns, specify `none`.	String	To include all the predefined columns, do not include this property. Note that the `id` column is added to the table automatically if both the `primaryKeys` and `extraFields` properties are absent.

Property	Description	Type	Default
`primaryKeys`	For an implicit table creation, provide a URL-encoded, comma-separated list specifying which attributes of the JSON object in the request body constitute the table's primary key. For example, `lastName,firstName`.	String	If you do not specify a primary key, then the service adds an `id` column to the table, and generates the column's values automatically, as long as you don't also include `extraFields` without `id` in the list.



Note:

Because you can't retrieve the primary

Property	Description	Type	Default
		keyorderfromthetablemetadata, makesurethatyoudocumentthe	

Property	Description	Type	Default
		order of the primary fields.	

Response

The response body is a JSON object. If the table is indexed on `id`, then the response is an array of the new rows' `id` values. Otherwise, the response is the `rowCount` of the records added.

Examples

Here's an example of calling the method to add two rows. If the table doesn't exist, then the service creates it. This table doesn't have extra fields, and its primary key is `code`:

```
service.post('/mobile/custom/incidentreport/initStatus', function (req,
res) {
  req.oracleMobile.database.insert(
    'FIF_Status',
    [
      {
        status: 'Closed',
        code: '0'},
      {
        status: 'Completed',
        code: '9'}
    ],
    {extraFields: 'none', primaryKeys: 'code'}).then(
    function (result) {
      res.status(result.statusCode).send(result.result);
```

```
    },
    function (error) {
      res.status(error.statusCode).send(error.error);
    }
  );
});
```

Here's an example of the response for this request.

```
{
  "rowCount": 2
}
```

Note that if a table's row key is the system-defined `id` column (instead of user-defined primary keys), then the response shows the `id` values for the new rows. For example:

```
{"items":[{"id":42},{"id":43}]}
```

database.merge(table, object, options, httpOptions)

This method lets you add or update rows in a table. Whether the operation performs an add or update depends on whether the table uses `id` or primary key fields to uniquely identify rows.

- `id` field: If you include an `id` property in the table data in the `object` argument, then the operation performs an update. Otherwise it adds the row.
- Primary key fields: If the table uses primary key fields, then the operation performs an update if a row exists with matching primary key values. Otherwise, it adds the row.

Note that if you submit a batch of rows, all the rows must have the same set of columns.

When the `Database_CreateTablesPolicy` environment policy is `allow`, then the following actions can occur:

- If the table doesn't exist, then it is created.
- If a column doesn't exist, then the table is altered to include it.
- If the value is larger than the column size, then the column is resized.

Ask your mobile cloud administrator about the `Database_CreateTablesPolicy` environment policy setting.

Arguments

`table`: Required. String. The name of the database table to add the row to.

`object`: Required. JSON object containing the table data. If you're adding one row, then you can use this format:

```
{
  status : 'Open',
```

```
  code : '3'
}
```

If you're adding multiple rows, then use this format:

```
[
    {
        status:'Open',
        code:'3'},
    {
        status:'Completed',
        code:'9'}
]
```

`options`: Optional. JSON object. This object can have the following properties in addition to those listed in Common options Argument Properties:

Property	Description	Type	Default
`encodeURI`	Set to `true` to URI-encode the `table`, `extraFields`, and `primaryKeys` values. This option can be useful for multibyte values.	Boolean	`false`
`extraFields`	For an implicit table creation, optionally provide a comma-separated list that specifies which predefined columns to include in the table from amongst `id`, `createdBy`, `createdOn`, `modifiedBy`, and `modifiedOn`. For example, `createdOn,createdBy`. To not include any predefined columns, specify `none`.	String	To include all the predefined columns, do not include this property. Note that the `id` column is added to the table automatically if both the `primaryKeys` and `extraFields` properties are absent.

Property	Description	Type	Default
`primaryKeys`	For an implicit table creation, provide a URL-encoded, comma-separated list specifying which attributes of the JSON object in the request body constitute the table's primary key. For example, `lastName,firstName`.	String	If you do not specify a primary key, then the operation adds an `id` column to the table, and generates the column's values automatically, as long as you don't also include `extraFields` without `id` in the list.

Note:

Because you can't retrieve the primary

Property	Description	Type	Default
		keyorderfromthetablemetadata, makesurethatyoudocumentthe	

Property	Description	Type	Default
		order of the primary fields.	

Response

The response body is a JSON object. If the table is indexed on `id`, then the response is an array of the new rows' `id` values. Otherwise, the response is the `rowCount`.

Examples

Here's an example of calling the method to add or update two rows. If the table doesn't exist, then the operation creates it. This table doesn't have extra fields, and its primary key is `code`:

```
service.post('/mobile/custom/incidentreport/initStatus', function (req,
res) {
  req.oracleMobile.database.merge(
    'FIF_Status',
    [
      {
        status: 'Closed',
        code: '0'},
      {
        status: 'Completed',
        code: '9'}
    ],
    {extraFields: 'none', primaryKeys: 'code'}).then(
    function (result) {
      res.status(result.statusCode).send(result.result);
    },
```

```
    function (error) {
      res.status(error.statusCode).send(error.error);
    }
  );
});
```

Here's an example of the response for this request.

```
{
  "rowCount": 2
}
```

Note that if a table's row key is the system-defined `id` column (instead of user-defined primary keys), then the response shows the `id` values for the new rows. For example:

```
{"items":[{"id":42},{"id":43}]}
```

Accessing the Devices API from Custom Code

Use this API to configure which devices that are running a mobile app can receive notifications.

This API has the following methods:

- devices.deregister(device, httpOptions): Deregister a mobile client instance that no longer needs to receive notifications..
- devices.register(device, httpOptions): Register a mobile client instance that receives notifications.

devices.deregister(device, httpOptions)

Call this method to deregister a a mobile client instance that no longer needs to receive notifications.

Arguments

`device`: Required. JSON object that follows the root (mobile client instance) request schema that's shown for the `POST /mobile/platform/devices/deregister` operation in REST APIs for Oracle Mobile Cloud Service.

If the `notificationProvider` property isn't provided, then the service assumes `APNS` for iOS, `GCM` for Android, and `WNS` for Windows.

Examples

Here's an example of calling this method to deregister a device.

```
service.post(
  '/mobile/custom/incidentreport/devices/deregister',
  function (req, res) {
    req.oracleMobile.devices.deregister(
      {
        "notificationToken": "b14d6dfbd9d56e09f098",
```

```
        "notificationProvider: "APNS",
        "mobileClient": {
          "id": "my.app.id",
          "platform": "IOS"
        }
      }
    ).then(
      function (result) {
        res.status(result.statusCode).send(result.result);
      },
      function (error) {
        res.status(error.statusCode).send(error.error);
      }
    );
  });
```

devices.register(device, httpOptions)

Call this method to register a new device.

Arguments

`device`: Required. JSON object that follows the root (mobile client instance) request schema that's shown for the `POST /mobile/platform/devices/register` operation in REST APIs for Oracle Mobile Cloud Service.

Response

The response body is a JSON object that follows the root (mobile client instance) response schema that's shown for the `POST /mobile/platform/devices/register` operation in REST APIs for Oracle Mobile Cloud Service.

Examples

Here's an example of calling this method to register a device.

```
service.post(
  '/mobile/custom/incidentreport/devices/register',
  function (req, res) {
    req.oracleMobile.devices.register(
      {
        "notificationToken": "b14d6dfbd9d56e09f098",
        "notificationProvider: "APNS",
        "mobileClient": {
          "id": "my.app.id",
          "version": "1.0",
          "platform": "IOS"
        }
      }
    ).then(
      function (result) {
        res.status(result.statusCode).send(result.result);
      },
      function (error) {
        res.status(error.statusCode).send(error.error);
```

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

Here's an example of the response body:

```
{
  "id": "27fee547-bdd0-4688-9497-475ec5ed0dfd",
  "notificationToken": "b14d6dfbd9d56e09f098",
  "notificationProvider: "APNS",
  "mobileClient": {
    "id": "my.app.id",
    "user": "joe",
    "version": "1.0",
    "platform": "IOS"
  },
  "modifiedOn": "2015-06-17T18:37:59.424Z"
}
```

Accessing the Location API from Custom Code

The Location API lets you query about location devices, their assets, and the places where they're located.

This API has the following methods:

- location.assets.getAsset(id, httpOptions): Retrieves the asset that matches the ID or name.
- location.assets.query(queryObject, httpOptions): Retrieves the assets that match the query parameters that you specify in the request body.
- location.devices.getDevice(id, httpOptions): Retrieves the device that matches the ID or name.
- location.devices.query(queryObject, httpOptions): Retrieves the devices that match the query parameters that you specify in the request body.
- location.places.getPlace(id, httpOptions): Retrieves the place that matches the ID or name.
- location.places.query(queryObject, httpOptions): Retrieves the places that match the query parameters that you specify in the request body.

You can learn about location devices, assets, and places in Location.

See Accessing the Location Management API from Custom Code for the methods to add, delete, and update assets, devices, and places.

location.assets.getAsset(id, httpOptions)

Call this method to retrieve the asset that matches the specified ID or name.

Arguments

`id`: Required. Must be one of the following:

- String that contains the ID of the asset to retrieve.

- JSON object that contains either the `id` property or the `name` property, where the property value indicates the search value. If the object contains both properties, then the SDK retrieves the asset with the matching name.

Response

The response body is a JSON object that follows the `Asset` schema that is shown for the `GET /mobile/platform/location/assets` and `GET /mobile/platform/location/assets/{id}` operations in REST APIs for Oracle Mobile Cloud Service

Examples

Here's an example of calling this method to retrieve an asset by ID.

```
service.get(
  '/mobile/custom/incidentreport/assets/:id',
  function (req, res) {
    req.oracleMobile.location.assets.getAsset(req.params.id).then(
      function (result) {
        res.status(result.statusCode).send(result.result);
      },
      function (error) {
        res.status(error.statusCode).send(error.error);
      }
    );
  });
```

Here's an example of calling this method to retrieve an asset by name.

```
service.get(
  '/mobile/custom/incidentreport/assets/:name',
  function (req, res) {
    req.oracleMobile.location.assets.getAsset({name:req.params.name}).then(
      function (result) {
        res.status(result.statusCode).send(result.result);
      },
      function (error) {
        res.status(error.statusCode).send(error.error);
      }
    );
  });
```

Here's an example of the response body:

```
{
   "id":111,
   "createdOn":"2015-08-06T18:37:59.424Z",
   "createdBy":"jdoe",
   "modifiedOn":"2015-08-06T18:37:59.424Z",
   "modifiedBy":"jdoe",
   "name":"RC_WH_01_F01_B023",
   "label":"forklift",
   "description":"Forklift in the FixItFast Warehouse in Redwood City",
   "lastKnownLocation":{
```

```
      "gpsPoint":{
         "latitude":37.5548,
         "longitude":-121.1566
      }
   },
   "devices":[
      {
         "id":345,
         "createdOn":"2015-08-06T18:37:59.424Z",
         "createdBy":"jdoe",
         "modifiedOn":"2015-08-08T07:22:44.654Z",
         "modifiedBy":"tsmith",
         "name":"RC_WH_01_F01_B001",
         "description":"Beacon in FixitFast Warehouse in Redwood City",
         "beacon":{
            "iBeacon":{
               "uuid":"B9407F30-F5F8-466E-AFF9-25556B57FE6D",
               "major":"1.0",
               "minor":"1.1"
            }
         },
         "attributes":{
            "manufacturer":"Abc Company",
            "manufacturerId":"10D39AE7-020E-4467-9CB2-DD36366F899D",
            "status":"Active",
            "visibility":"Public"
         },
         "links":[
            {
               "rel":"canonical",
               "href":"/mobile/platform/location/devices/345"
            },
            {
               "rel":"self",
               "href":"/mobile/platform/location/devices/345"
            }
         ]
      }
   ],
   "attributes":{
      "EquipmentManufacturer":"Abc Company",
      "beaconID":"AE2924505-66045"
   },
   "links":[
      {
         "rel":"canonical",
         "href":"/mobile/platform/location/assets/111"
      },
      {
         "rel":"self",
         "href":"/mobile/platform/location/assets/111"
      }
   ]
}
```

location.assets.query(queryObject, httpOptions)

Call this method to retrieve the assets that match the query parameters that you specify in `queryObject`.

Arguments

`queryObject`: Required. String. The parameters that describe the desired results. For details, see the body parameter for the `POST /mobile/platform/location/assets/query` operation in REST APIs for Oracle Mobile Cloud Service. If you don't have any query parameters, then use an empty body ({}).

Response

The response body is a JSON object that contains an array of items that follow the `Asset` schema that is shown for the `POST /mobile/platform/location/assets/query` operation in REST APIs for Oracle Mobile Cloud Service. The result also contains paging information. For example:

```
"totalResults":2,
"offset":0,
"limit":40,
"count":2,
"hasMore":false
```

Examples

Here's an example of calling this method. It returns all assets that have the string `1225` in the name or description (case-insensitive).

```
service.get(
  '/mobile/custom/incidentreport/assets,
  function (req, res) {
    req.oracleMobile.location.assets.query({"search":"1225"}).then(
      function (result) {
        res.status(result.statusCode).send(result.result);
      },
      function (error) {
        res.status(error.statusCode).send(error.error);
      }
    );
  });
```

Here's an example of the response body:

```
{
    "items":[
        {
            "devices":[
                {
                    "id":3401,
                    "createdBy":"jdoe",
                    "name":"RC_WH_01_F01_B001",
```

```
                    "createdOn":"2015-08-06T18:37:59.424Z",
                    "modifiedOn":"2015-08-08T07:22:44.654Z",
                    "beacon":{
                        "iBeacon":{
                            "uuid":"B9407F30-F5F8-466E-AFF9-25556B57FE6D",
                            "major":"1.0",
                            "minor":"1.1"}},
                    "modifiedBy":"tsmith",
                    "links":[
                        {
                            "rel":"canonical",
                            "href":"/mobile/platform/location/devices/
3401"},
                        {
                            "rel":"self",
                            "href":"/mobile/platform/location/devices/
3401"}
                    ],
                    "attributes":{
                        "manufacturer":"Example Company",
                        "manufacturerId":"10D39AE7-020E-4467-9CB2-
DD36366F899D",
                        "status":"Active",
                        "visibility":"Public"},
                    "description":"Beacon on 1st Floor in FixitFast
Warehouse in Redwood City"}
            ],
            "label":"hospital bed",
            "lastKnownLocation":{
                "placeId":244},
            "id":333,
            "createdBy":"jdoe",
            "name":"hospital bed #233",
            "createdOn":"2015-08-06T18:37:59.424Z",
            "modifiedOn":"2015-08-06T18:37:59.424Z",
            "modifiedBy":"jdoe",
            "links":[
                {
                    "rel":"canonical",
                    "href":"/mobile/platform/location/assets/333"},
                {
                    "rel":"self",
                    "href":"/mobile/platform/location/assets/333"}
            ],
            "attributes":{
                "EquipmentManufacturer":"Example Company",
                "SJId":"6754843090"},
            "description":"model 1225 hospital bed"},
        {
            "devices":[
                {
                    "id":648,
                    "createdBy":"jdoe",
                    "name":"RC_WH_01_F01_B001",
                    "createdOn":"2015-08-06T18:37:59.424Z",
```

```
                    "modifiedOn":"2015-08-08T07:22:44.654Z",
                    "beacon":{
                        "iBeacon":{
                            "uuid":"B9407F30-F5F8-466E-AFF9-25556B57FE6D",
                            "major":"1.0",
                            "minor":"1.1"}},
                    "modifiedBy":"tsmith",
                    "links":[
                        {
                            "rel":"canonical",
                            "href":"/mobile/platform/location/devices/
648"},
                        {
                            "rel":"self",
                            "href":"/mobile/platform/location/devices/648"}
                    ],
                    "attributes":{
                        "manufacturer":"Example Company",
                        "manufacturerId":"10D39AE7-020E-4467-9CB2-
DD36366F899D",
                        "status":"Active",
                        "visibility":"Public"},
                    "description":"Beacon on 1st Floor in FixitFast
Warehouse in Redwood City"}
            ],
            "label":"hospital bed",
            "lastKnownLocation":{
                "placeId":360},
            "id":888,
            "createdBy":"jdoe",
            "name":"hospital bed #233",
            "createdOn":"2015-10-16T09:24:41.354Z",
            "modifiedOn":"2015-10-16T09:24:41.354Z",
            "modifiedBy":"jdoe",
            "links":[
                {
                    "rel":"canonical",
                    "href":"/mobile/platform/location/assets/888"},
                {
                    "rel":"self",
                    "href":"/mobile/platform/location/assets/888"}
            ],
            "attributes":{
                "EquipmentManufacturer":"Example Company",
                "SJId":"6754843090"},
            "description":"model 1225 hospital bed"}
    ],
    "totalResults":2,
    "offset":0,
    "count":2,
    "hasMore":false
}
```

location.devices.getDevice(id, httpOptions)

Call this method to retrieve the device that matches the specified ID or name.

Arguments

`id`: Required. Must be one of the following:

- String that contains the ID of the device to retrieve.
- JSON object that contains either the `id` property or the `name` property, where the property value indicates the search value. If the object contains both properties, then the SDK retrieves the device with the matching name.

Response

The response body is a JSON object that follows the `Location device` schema that is shown for the `GET /mobile/platform/location/devices` and `GET /mobile/platform/location/devices/{id}` operations in REST APIs for Oracle Mobile Cloud Service.

Examples

Here's an example of calling this method to retrieve a device by ID.

```
service.get(
  '/mobile/custom/incidentreport/devices/:id',
  function (req, res) {
    req.oracleMobile.location.devices.getDevice(req.params.id).then(
      function (result) {
        res.status(result.statusCode).send(result.result);
      },
      function (error) {
        res.status(error.statusCode).send(error.error);
      }
    );
  });
```

Here's an example of calling this method to retrieve a device by name.

```
service.get(
  '/mobile/custom/incidentreport/devices/:name',
  function (req, res) {

req.oracleMobile.location.devices.getDevice({name:req.params.name}).then(
      function (result) {
        res.status(result.statusCode).send(result.result);
      },
      function (error) {
        res.status(error.statusCode).send(error.error);
      }
    );
  });
```

Here's an example of the response body:


```
{
  "id": 12345,
  "createdOn": "2015-08-06T18:37:59.424Z",
  "createdBy": "jdoe",
  "modifiedOn": "2015-08-08T07:22:44.654Z",
  "modifiedBy": "tsmith",
  "name": "RC_WH_01_F01_B001",
  "description": "Beacon on 1st Floor in FixitFast Warehouse in Redwood
City",
  "place":
  {
    "id": 111,
    "createdOn": "2015-08-06T18:37:59.424Z",
    "createdBy": "jdoe",
    "modifiedOn": "2015-08-06T18:37:59.424Z",
    "modifiedBy": "jdoe",
    "name": "FixitFast Redwood City Warehouse",
    "label": "FixitFast Warehouse",
    "parentPlace": 42,
    "description": "FixitFast Warehouse in Redwood City",
    "address" : {
      "gpsPoint" : {
        "latitude": 37.5548,
        "longitude": -121.1566
      }
    },
    "attributes" : {
      "equipmentManufacturer": "Abc Corp"
    },
    "links": [
      {
        "rel": "canonical",
        "href": "/mobile/platform/location/places/111"
      },
      {
        "rel": "self",
        "href": "/mobile/platform/location/places/111"
      }
    ]
  },
  "beacon": {
    "iBeacon" : {
      "uuid": "B9407F30-F5F8-466E-AFF9-25556B57FE6D",
      "major": "1.0",
      "minor": "1.1"
    }
  },
  "attributes" : {
    "manufacturer": "Abc Company",
    "manufacturerId": "10D39AE7-020E-4467-9CB2-DD36366F899D"
    "status": "Active",
    "visibility": "Public"
  },
```

```
  "links": [
    {
      "rel": "canonical",
      "href": "/mobile/platform/location/devices/12345"
    },
    {
      "rel": "self",
      "href": "/mobile/platform/location/devices/12345"
    }
  ]
}
```

location.devices.query(queryObject, httpOptions)

Call this method to retrieve the devices that match the query parameters that you specify in `queryObject`.

Arguments

`queryObject`: Required. String. The parameters that describe the desired results. For details, see the body parameter for the `POST /mobile/platform/location/devices/query` operation in REST APIs for Oracle Mobile Cloud Service. If you don't have any query parameters, then use an empty body ({}).

Response

The response body is a JSON object that contains an array of items that follow the `Location device` schema that is shown for the `POST /mobile/platform/location/devices/query` operation in REST APIs for Oracle Mobile Cloud Service The result also contains paging information. For example:

```
"totalResults":2,
"offset":0,
"limit":40,
"count":2,
"hasMore":false
```

Examples

Here's an example of calling this method. It returns the devices that have the string `warehouse` in either the name or description (case-insensitive).

```
service.get(
  '/mobile/custom/incidentreport/devices,
  function (req, res) {
    req.oracleMobile.location.devices.query({{ "search":
"Warehouse"}}).then(
      function (result) {
        res.status(result.statusCode).send(result.result);
      },
      function (error) {
        res.status(error.statusCode).send(error.error);
      }
```

```
    );
  });
```

Here's an example of the response body:

```
{
    "items":[
        {
            "id":33,
            "name":"RC_WH_01_B09_C004",
            "description":"Beacon on 2nd Floor NW in FixItFast Warehouse
in Redwood City",
            "protocol":"altBeacon"},
        {
            "id":12,
            "name":"RC_WH_01_F01_B001",
            "description":"Beacon on 1st Floor SE in FixItFast Warehouse
in Redwood City",
            "protocol":"altBeacon"},
        {
            "id":61,
            "name":"RC_WH_01_F01_B008",
            "description":"Beacon on 2nd Floor SW in FixItFast Warehouse
in Redwood City",
            "protocol":"altBeacon"},
        {
            "id":58,
            "name":"RC_WH_02_F01_B011",
            "description":"Beacon on 1st Floor NW in FixitFast Warehouse
in Redwood City",
            "protocol":"altBeacon"},
        {
            "id":114,
            "name":"RC_WH_01_K22_A999",
            "description":"Beacon on 3rd Floor NW in FixitFast Warehouse
in Redwood City",
            "protocol":"altBeacon"}
    ],
    "totalResults":5,
    "offset":0,
    "count":5,
    "hasMore":false
}
```

location.places.getPlace(id, httpOptions)

Call this method to retrieve the place that matches the specified ID or name.

Arguments

`id`: Required. Must be one of the following:

- String that contains the ID of the place to retrieve.

- JSON object that contains either the `id` property or the `name` property, where the property value indicates the search value. If the object contains both properties, then the SDK retrieves the place with the matching name.

Response

The response body is a JSON object that follows the `Place` schema that is shown for the `GET /mobile/platform/location/places` and `GET /mobile/platform/location/places/{id}` operations in REST APIs for Oracle Mobile Cloud Service.

Examples

Here's an example of calling this method to retrieve a place by ID.

```
service.get(
  '/mobile/custom/incidentreport/places/:id',
  function (req, res) {
    req.oracleMobile.location.places.getPlace(req.params.id).then(
      function (result) {
        res.status(result.statusCode).send(result.result);
      },
      function (error) {
        res.status(error.statusCode).send(error.error);
      }
    );
  });
```

Here's an example of calling this method to retrieve a place by name.

```
service.get(
  '/mobile/custom/incidentreport/places/:name',
  function (req, res) {
    req.oracleMobile.location.places.getPlace({name:req.params.name}).then(
      function (result) {
        res.status(result.statusCode).send(result.result);
      },
      function (error) {
        res.status(error.statusCode).send(error.error);
      }
    );
  });
```

Here's an example of the response body:

```
{
  "id": 111,
  "createdOn": "2015-08-06T18:37:59.424Z",
  "createdBy": "jdoe",
  "modifiedOn": "2015-08-06T18:37:59.424Z",
  "modifiedBy": "jdoe",
  "name": "FixitFast Redwood City Warehouse",
  "label": "FixitFast Warehouse",
  "parentPlace": 42,
  "description": "FixitFast Warehouse in Redwood City",
```

```
  "address" : {
    "gpsPoint" : {
      "latitude": 37.5548,
      "longitude": -121.1566
    }
  },
  "attributes" : {
    "equipmentManufacturer": "Abc Corp"
  },
  "links": [
    {
      "rel": "canonical",
      "href": "/mobile/platform/location/places/111"
    },
    {
      "rel": "self",
      "href": "/mobile/platform/location/places/111"
    }
  ]
}
```

location.places.query(queryObject, httpOptions)

Call this method to retrieve the places and, optionally, the associated devices that match the query properties that you specify in the `queryObject`.

Arguments

`queryObject`: Required. String. The parameters that describe the desired results. For details, see the body parameter for the `POST /mobile/platform/location/places/query` operation in REST APIs for Oracle Mobile Cloud Service. If you don't have any query parameters, then use an empty body ({}).

Response

The response body is a JSON object that contains an array of items that follow the `Place` schema that is shown for the `POST /mobile/platform/location/places/query` operation in REST APIs for Oracle Mobile Cloud Service. The result also contains paging information. For example:

```
"totalResults":2,
"offset":0,
"limit":40,
"count":2,
"hasMore":false
```

Examples

Here's an example of calling this method. It returns all places that have the string `warehouse` in the name or description (case-insensitive). By default, the response includes the `children` array, which contains information about descendent places. In

this request, the `includeDescendantsInResult` property is set to `none`. Therefore the request doesn't include that array.

```
service.get(
  '/mobile/custom/incidentreport/places',
  function (req, res) {

req.oracleMobile.location.places.query({"search":"warehouse","includeDescen
dantsInResult":"none" }).then(
      function (result) {
        res.status(result.statusCode).send(result.result);
      },
      function (error) {
        res.status(error.statusCode).send(error.error);
      }
    );
  });
```

Here's an example of the response body:

```
{
    "items":[
        {
            "devices":[
                {
                    "id":12345,
                    "createdBy":"jdoe",
                    "name":"RC_WH_01_F01_B001",
                    "createdOn":"2015-08-06T18:37:59.424Z",
                    "modifiedOn":"2015-08-08T07:22:44.654Z",
                    "beacon":{
                        "iBeacon":{
                            "uuid":"B9407F30-F5F8-466E-AFF9-25556B57FE6D",
                            "major":"1.0",
                            "minor":"1.1"}},
                    "modifiedBy":"tsmith",
                    "links":[
                        {
                            "rel":"canonical",
                            "href":"/mobile/platform/location/devices/
12345"},
                        {
                            "rel":"self",
                            "href":"/mobile/platform/location/devices/
12345"}
                    ],
                    "attributes":{
                        "manufacturer":"Abc Company",
                        "manufacturerId":"10D39AE7-020E-4467-9CB2-
DD36366F899D",
                        "status":"Active",
                        "visibility":"Public"},
                    "description":"Beacon on 1st Floor in FixitFast
Warehouse in Redwood City"}
```

```
                ],
                "label":"FixItFast Warehouse",
                "id":112,
                "createdBy":"jdoe",
                "name":"FixItFast Redwood City Warehouse",
                "createdOn":"2015-08-06T18:37:59.424Z",
                "modifiedOn":"2015-08-06T18:37:59.424Z",
                "address":{
                    "gpsPoint":{
                        "latitude":122,
                        "longitude":37}},
                "modifiedBy":"jdoe",
                "links":[
                    {
                        "rel":"canonical",
                        "href":"/mobile/platform/location/places/112"},
                    {
                        "rel":"self",
                        "href":"/mobile/platform/location/places/112"}
                ],
                "attributes":{
                    "hours":"9am-6pm"},
                "hasChildren":false,
                "parentPlace":42,
                "description":"FixItFast Warehouse in Redwood City"},
            {
                "devices":[
                    {
                        "id":111,
                        "createdBy":"jdoe",
                        "name":"RC_WH_01_F01_B001",
                        "createdOn":"2015-08-06T18:37:59.424Z",
                        "modifiedOn":"2015-08-08T07:22:44.654Z",
                        "beacon":{
                            "iBeacon":{
                                "uuid":"B9407F30-F5F8-466E-AFF9-25556B57FE6D",
                                "major":"1.0",
                                "minor":"1.1"}},
                        "modifiedBy":"tsmith",
                        "links":[
                            {
                                "rel":"canonical",
                                "href":"/mobile/platform/location/devices/
111"},
                            {
                                "rel":"self",
                                "href":"/mobile/platform/location/devices/111"}
                        ],
                        "attributes":{
                            "manufacturer":"Abc Company",
                            "manufacturerId":"10D39AE7-020E-4467-9CB2-
DD36366F899D",
                            "status":"Active",
                            "visibility":"Public"},
                        "description":"Beacon on 1st Floor in FixitFast
```

```
Warehouse in Redwood City"},
                {
                    "id":222,
                    "createdBy":"jdoe",
                    "name":"RC_WH_01_F01_B996",
                    "createdOn":"2015-08-08T18:37:59.424Z",
                    "modifiedOn":"2015-08-12T07:22:44.654Z",
                    "beacon":{
                        "iBeacon":{
                            "uuid":"B9407F30-F5F8-466E-
AFF9-25552345908234DD0",
                            "major":"1.0",
                            "minor":"1.1"}},
                    "modifiedBy":"tsmith",
                    "links":[
                        {
                            "rel":"canonical",
                            "href":"/mobile/platform/location/devices/
222"},
                        {
                            "rel":"self",
                            "href":"/mobile/platform/location/devices/222"}
                    ],
                    "attributes":{
                        "manufacturer":"Abc Company",
                        "manufacturerId":"10D39AE7-020E-4467-9CB2-
DD36366F899D",
                        "status":"Active",
                        "visibility":"Public"},
                    "description":"Beacon on 2nd Floor in FixitFast
Warehouse in Redwood City"}
            ],
            "label":"FixItFast Warehouse",
            "id":325,
            "createdBy":"jdoe",
            "name":"FixItFast Palo Alto Warehouse",
            "createdOn":"2015-08-06T19:27:59.424Z",
            "modifiedOn":"2015-08-06T19:27:59.424Z",
            "address":{
                "gpsCircle":{
                    "latitude":123,
                    "longitude":37,
                    "radius":300}},
            "modifiedBy":"jdoe",
            "links":[
                {
                    "rel":"canonical",
                    "href":"/mobile/platform/location/places/325"},
                {
                    "rel":"self",
                    "href":"/mobile/platform/location/places/325"}
            ],
            "attributes":{
                "hours":"9am-6pm"},
            "hasChildren":false,
```

```
            "parentPlace":42,
            "description":"FixItFast Warehouse in Palo Alto"}
    ],
    "totalResults":2,
    "offset":0,
    "count":2,
    "hasMore":false
}
```

Accessing the Location Management API from Custom Code

The Location Management API lets you create, update, and delete location devices, places, and assets.

You can learn about location devices, assets, and places in Location.

The `Authorization` request header for these methods must use OAUTH. Otherwise the methods return a `404` HTTP status code.

This API has the following methods:

- location.assets.register(assets, context, httpOptions): Creates one or more assets.
- location.assets.remove(id, context, httpOptions): Deletes assets.
- location.assets.update(id, asset, context, httpOptions): Updates a single asset.
- location.devices.register(devices, context, httpOptions): Creates one or more location devices.
- location.devices.remove(id, context, httpOptions): Deletes location devices.
- location.devices.update(id, device, context, httpOptions): Updates a single location device.
- location.places.register(places, context, httpOptions): Creates one or more places.
- location.places.remove(id, context, httpOptions): Deletes places.
- location.places.removeCascade(id, context, httpOptions): Deletes the place that matches the ID as well as all its child places.
- location.places.update(id, place, context, httpOptions): Updates a single place.

For methods to query and retrieve information about assets, devices, and places, see Accessing the Location API from Custom Code.

Location Management Context Argument

All the Location Management API methods require a `context` argument, which is a JSON object with the following properties. This information is required to get authorization to manage location information. In addition, the mobile app must use OAuth authorization.

Note that the custom code can call mbe.getMBE() to get the mobile backend information.

Property	Desc	Type
`mbe`	The name of the mobile backend.	String
`username`	The name of a user who is an MCS team member and has the `MobileEnvironment_System` role. Team members and their roles are managed from Oracle Cloud Infrastructure Classic Console. SeeAssign MCS Team Member Roles.	String
`version`	The version of the mobile backend.	String

> **Note:**
>
> If the `Authorization` request header doesn't use OAuth, then the methods return `404`. If the `username` is not an MCS team member who has the `MobileEnvironment_System` role, then the methods return `403`.

location.assets.register(assets, context, httpOptions)

This method lets you create one or more assets.

Arguments

`assets`: Required. JSON object that follows the request root schema (Assets Array) that is shown for the `POST /mobile/system/locationManagement/assets` operation in REST APIs for Oracle Mobile Cloud Service. Here's an example:

```
{
  "items":[
      {
          "name":"hospital bed #233",
          "label":"hospital bed",
          "description":"model 1225 hospital bed",
          "lastKnownLocation":{
              "placeId":244
          },
          "devices":[
              1111
          ],
          "attributes":{
              "EquipmentManufacturer":"Example Company",
              "SJId":"6754843090"
          }
      }
  ]
}
```

`context`: Required. JSON object as described in Location Management Context Argument.

Response

The response body, which shows the stored assets, is a JSON object that follows the response root schema (Assets Array) that is shown for the `POST /mobile/system/locationManagement/assets` operation in REST APIs for Oracle Mobile Cloud Service.

Examples

In this example, the request body would look like this:

```
{
  "userName":"anAdministrator",
  "assets": {
    "items":[
      {
        "name":"hospital bed #233",
        "label":"hospital bed",
        "description":"model 1225 hospital bed",
        "attributes":{
          "EquipmentManufacturer":"Example Company",
          "SJId":"6754843090"
        }
      }
    ]
  }
}
```

This example puts the `username` in the `context` object and passes `assets` as the request body.

```
service.post('/mobile/custom/incidentreport/assets', function (req, res) {
  req.oracleMobile.location.assets.register(
    req.body.assets,
    {
      username: req.body.userName,
      mbe: req.oracleMobile.mbe.getMBE().name,
      version: req.oracleMobile.mbe.getMBE().version
    }).then(
    function (result) {
      res.type('application/json');
      res.status(result.statusCode).send(result.result);
    },
    function (error) {
      console.dir(error);
      res.status(error.sta).send(sCode, error.error);
    }
  );
});
```

Here's an example of the response body.

```
{
  "items": [
    {
      "id": 12,
      "createdOn": "2016-11-05T02:33:36.154Z",
      "createdBy": "anAdministrator",
      "modifiedOn": "2016-11-05T02:33:36.154Z",
      "modifiedBy": "anAdministrator",
      "name": "hospital bed #233",
      "label": "hospital bed",
      "description": "model 1225 hospital bed",
      "lastKnownLocation": null,
      "attributes": {
        "EquipmentManufacturer": "Example Company",
        "SJId": "6754843090"
      },
      "links": [
        {
          "rel": "canonical",
          "href": "/mobile/platform/location/assets/12"
        },
        {
          "rel": "self",
          "href": "/mobile/platform/location/assets/12"
        }
      ]
    }
  ]
}
```

location.assets.remove(id, context, httpOptions)

Use this method to delete assets.

Arguments

`id`: Required. IDs of the assets to remove. This argument can be either a single value or an array of values.

`context`: Required. JSON object as described in Location Management Context Argument.

Response

If you provide a single value, then the service doesn't return a response body. The status code is 204 if the asset was deleted and 404 if it doesn't exist.

If you provide an array of IDs, then the status code is 200 for a successful request. The response contains a `batch` object with an array of responses for the individual delete requests. For schema details, see the Delete Multiple Assets operation in REST APIs for Oracle Mobile Cloud Service.

Here's an example:

```
{
    "batch":[
        {
            "body":{
                "id":353,
                "message":"asset was deleted successfully."},
            "code":200},
        {
            "body":{
                "id":354,
                "message":"asset was deleted successfully."},
            "code":200},
        {
            "body":{
                "id":355,
                "message":"asset not found."},
            "code":404}
    ]
}
```

Examples

In this example, if the `id` query parameter contains multiple IDs, then it converts the query string into an array.

Note that the user name of the user who has the `MobileEnvironment_System` role is passed in the `user` query parameter.

```
service.delete('/mobile/custom/location/assets', function(req,res) {
  var contextObject = {
    username: req.query.user,
    mbe: req.oracleMobile.mbe.getMBE().name,
    version: req.oracleMobile.mbe.getMBE().version
  };
  var id = req.query.id.split(',');
  if (id.length == 0){
      id = req.query.id;
  }
  req.oracleMobile.location.assets.remove(
      id,
      contextObject
    ).then(
    function (result) {
      res.type('application/json');
      res.status(result.statusCode).send(result.result);
    },
    function (error) {
      console.dir(error);
      res.status(error.statusCode).send(error.error);
    }
  );
});
```

location.assets.update(id, asset, context, httpOptions)

This method lets you update an asset.

Arguments

`id`: Required. The ID of the asset. This ID must be an existing asset ID.

`asset`: Required. JSON object that follows the request root schema (Asset) that is shown for the `PUT /mobile/system/locationManagement/assets/{id}` operation in REST APIs for Oracle Mobile Cloud Service. Here's an example:

```
{
    "lastKnownLocation":{
        "gpsPoint":{
            "latitude":37.5548,
            "longitude":-121.1566
        }
    },
    "devices":[
        11
    ]
}
```

`context`: Required. JSON object as described in Location Management Context Argument.

Response

The response body, which shows the updated asset, is a JSON object that follows the response root schema (Asset) that is shown for the `PUT /mobile/system/locationManagement/assets/{id}` operation in REST APIs for Oracle Mobile Cloud Service.

Examples

In this example, the request body would look like this:

```
{
   "userName":"anAdministrator",
   "asset":{
      "lastKnownLocation":{
         "gpsPoint":{
            "latitude":37.5548,
            "longitude":-121.1566
         }
      },
      "devices":[
         11
      ]
   }
}
```

This example puts the `username` in the `context` object and passes `asset` as the request body.

```
service.put('/mobile/custom/incidentreport/assets/:id', function (req,
res) {
  req.oracleMobile.location.assets.update(
    req.params.id,
    req.body.asset,
    {
      username: req.body.userName,
      mbe: req.oracleMobile.mbe.getMBE().name,
      version: req.oracleMobile.mbe.getMBE().version
    }).then(
    function (result) {
      res.type('application/json');
      res.status(result.statusCode).send(result.result);
    },
    function (error) {
      console.dir(error);
      res.status(error.statusCode).send(error.error);
    }
  );
});
```

Here's an example of the response body.

```
{
  "id": 11,
  "createdOn": "2016-11-08T21:26:38.318Z",
  "createdBy": "anAdministrator",
  "modifiedOn": "2016-11-08T22:18:24.157Z",
  "modifiedBy": "anAdministrator",
  "name": "hospital bed #233",
  "label": "hospital bed",
  "description": "model 1225 hospital bed",
  "lastKnownLocation": {
    "gpsPoint": {
      "longitude": -121.1566,
      "latitude": 37.5548
    }
  },
  "devices": [
    {
      "id": 11,
      "createdOn": "2016-11-08T18:01:18.531Z",
      "createdBy": "anAdministrator",
      "modifiedOn": "2016-11-08T18:01:18.531Z",
      "modifiedBy": "anAdministrator",
      "name": "RC_WH_01_F01_B016",
      "description": "Beacon on 2nd Floor in FixitFast Warehouse in
Redwood City",
      "beacon": {
        "altBeacon": {
          "id1": "B9407F30-F5F8-466E",
```

```
            "id2": "AFF9",
            "id3": "25556B57FE6D"
          }
        },
        "attributes": {
          "manufacturer": "Abc Company",
          "status": "Active",
          "manufacturerId": "10D39AE7-020E-4467-9CB2-DD36366F899D",
          "visibility": "Public"
        },
        "links": [
          {
            "rel": "canonical",
            "href": "/mobile/platform/location/devices/11"
          },
          {
            "rel": "self",
            "href": "/mobile/platform/location/devices/11"
          }
        ]
      }
    ],
    "attributes": {
      "EquipmentManufacturer": "Example Company",
      "SJId": "6754843090"
    },
    "links": [
      {
        "rel": "canonical",
        "href": "/mobile/platform/location/assets/11"
      },
      {
        "rel": "self",
        "href": "/mobile/platform/location/assets/11"
      }
    ]
}
```

location.devices.register(devices, context, httpOptions)

This method lets you create one or more devices.

Arguments

`devices`: Required. JSON object that follows the request root schema (Devices Array) that is shown for the `POST /mobile/system/locationManagement/devices` operation in REST APIs for Oracle Mobile Cloud Service. Here's an example:

```
{
    "items":[
        {
            "name":"RC_WH_01_F01_B006",
            "description":"Beacon on 2nd Floor in FixitFast Warehouse in
Redwood City",
```

```
            "asset":333,
            "beacon":{
                "altBeacon":{
                    "id1":"B9407F30-F5F8-466E",
                    "id2":"AFF9",
                    "id3":"25556B57FE6D"
                }
            },
            "attributes":{
                "manufacturer":"Abc Company",
                "manufacturerId":"10D39AE7-020E-4467-9CB2-DD36366F899D",
                "status":"Active",
                "visibility":"Public"
            }
        }
    ]
}
```

`context`: Required. JSON object as described in Location Management Context Argument.

Response

The response body, which shows the stored devices, is a JSON object that follows the response root schema (Devices Array) that is shown for the `POST /mobile/system/locationManagement/devices` operation in REST APIs for Oracle Mobile Cloud Service.

Examples

In this example, the request body would look like this:

```
{
  "userName":"anAdministrator",
  "devices": {
    "items":[
       {
         "name":"RC_WH_01_F01_B006",
         "description":"Beacon on 2nd Floor in FixitFast Warehouse in
Redwood City",
            "beacon":{
                "altBeacon":{
                    "id1":"B9407F30-F5F8-466E",
                    "id2":"AFF9",
                    "id3":"25556B57FE6D"
                }
            },
            "attributes":{
                "manufacturer":"Abc Company",
                "manufacturerId":"10D39AE7-020E-4467-9CB2-DD36366F899D",
                "status":"Active",
                "visibility":"Public"
            }
        }
    ]
```

```
  }
}
```

This example puts the `username` in the context object and passes `devices` as the request body.

```
service.post('/mobile/custom/incidentreport/devices, function (req, res) {
  req.oracleMobile.location.devices.register(
    req.body.devices,
    {
      username: req.body.userName,
      mbe: req.oracleMobile.mbe.getMBE().name,
      version: req.oracleMobile.mbe.getMBE().version
    }).then(
    function (result) {
      res.type('application/json');
      res.status(result.statusCode).send(result.result);
    },
    function (error) {
      console.dir(error);
      res.status(error.statusCode).send(error.error);
    }
  );
});
```

Here's an example of the response body.

```
{
  "items": [
    {
      "id": 10,
      "createdOn": "2016-11-08T15:54:51.603Z",
      "createdBy": "anAdministrator",
      "modifiedOn": "2016-11-08T15:54:51.603Z",
      "modifiedBy": "anAdministrator",
      "name": "RC_WH_01_F01_B006",
      "description": "Beacon on 2nd Floor in FixitFast Warehouse in
Redwood City",
      "beacon": {
        "altBeacon": {
          "id1": "B9407F30-F5F8-466E",
          "id2": "AFF9",
          "id3": "25556B57FE6D"
        }
      },
      "attributes": {
        "manufacturer": "Abc Company",
        "manufacturerId": "10D39AE7-020E-4467-9CB2-DD36366F899D",
        "status": "Active",
        "visibility": "Public"
      },
      "links": [
        {
          "rel": "canonical",
```

```
          "href": "/mobile/platform/location/devices/10"
        },
        {
          "rel": "self",
          "href": "/mobile/platform/location/devices/10"
        }
      ]
    }
  ]
}
```

location.devices.remove(id, context, httpOptions)

Use this method to delete devices.

Arguments

`id`: Required. IDs of the devices to remove. This argument can be either a single value or an array of values.

`context`: Required. JSON object as described in Location Management Context Argument.

Response

If you provide a single value, then the service doesn't return a response body. The status code is 204 if the device was deleted and 404 if it doesn't exist.

If you provide an array of IDs, then the status code is 200 for a successful request. The response contains a `batch` object with an array of responses for the individual delete requests. For schema details, see the Delete Multiple Devices operation in REST APIs for Oracle Mobile Cloud Service.

Here's an example:

```
{
    "batch":[
        {
            "code":200,
            "body":{
                "id":121,
                "message":"device was deleted successfully."
            }
        },
        {
            "code":200,
            "body":{
                "id":122,
                "message":"device was deleted successfully."
            }
        },
        {
            "code":404,
            "body":{
                "id":123,
                "message":"device not found."
```

```
            }
        }
    ]
}
```

Examples

In this example, if the `id` query parameter contains multiple IDs, then it converts the query string into an array.

Note that the user name of the user who has the `MobileEnvironment_System` role is passed in the `user` query parameter.

```
service.delete('/mobile/custom/location/devices', function(req,res) {
  var contextObject = {
    username: req.query.user,
    mbe: req.oracleMobile.mbe.getMBE().name,
    version: req.oracleMobile.mbe.getMBE().version
  };
  var id = req.query.id.split(',');
  if (id.length == 0){
      id = req.query.id;
  }
  req.oracleMobile.location.devices.remove(
      id,
      contextObject
    ).then(
    function (result) {
      res.type('application/json');
      res.status(result.statusCode).send(result.result);
    },
    function (error) {
      console.dir(error);
      res.status(error.statusCode).send(error.error);
    }
  );
});
```

location.devices.update(id, device, context, httpOptions)

This method lets you update a device.

Arguments

`id`: Required. The ID of the device. This ID must be an existing device ID.

`device`: Required. JSON object that follows the request root schema (Device) that is shown for the `PUT /mobile/system/locationManagement/device/{id}` operation in REST APIs for Oracle Mobile Cloud Service. Here's an example:

```
{
    "attributes":{
        "status":"Inactive",
        "visibility":"Private"
```

```
    }
}
```

`context`: Required. JSON object as described in Location Management Context Argument.

Response

The response body, which shows the updated device, is a JSON object that follows the response root schema (Device) that is shown for the `PUT /mobile/system/locationManagement/devices/{id}` operation in REST APIs for Oracle Mobile Cloud Service.

Examples

In this example, the request body would look like this:

```
{
   "userName":"anAdministrator",
   "device":{
      "attributes":{
         "status":"Inactive",
         "visibility":"Private"
      }
   }
}
```

This example puts the `username` in the `context` object and passes `device` as the request body.

```
service.put('/mobile/custom/incidentreport/device/:id', function (req,
res) {
  req.oracleMobile.location.device.update(
    req.params.id,
    req.body.device,
    {
      username: req.body.userName,
      mbe: req.oracleMobile.mbe.getMBE().name,
      version: req.oracleMobile.mbe.getMBE().version
    }).then(
    function (result) {
      res.type('application/json');
      res.status(result.statusCode).send(result.result);
    },
    function (error) {
      console.dir(error);
      res.status(error.statusCode).send(error.error);
    }
  );
});
```

Here's an example of the response body.


```
{
  "id": 11,
  "createdOn": "2016-11-08T18:01:18.531Z",
  "createdBy": "anAdministrator",
  "modifiedOn": "2016-11-08T22:45:47.545Z",
  "modifiedBy": "anAdministrator",
  "name": "RC_WH_01_F01_B016",
  "description": "Beacon on 2nd Floor in FixitFast Warehouse in Redwood
City",
  "asset": {
    "id": 11,
    "createdOn": "2016-11-08T21:26:38.318Z",
    "createdBy": "anAdministrator",
    "modifiedOn": "2016-11-08T22:18:24.157Z",
    "modifiedBy": "anAdministrator",
    "name": "hospital bed #233",
    "label": "hospital bed",
    "description": "model 1225 hospital bed",
    "lastKnownLocation": {
      "gpsPoint": {
        "longitude": -121.1566,
        "latitude": 37.5548
      }
    },
    "attributes": {
      "EquipmentManufacturer": "Example Company",
      "SJId": "6754843090"
    },
    "links": [
      {
        "rel": "canonical",
        "href": "/mobile/platform/location/assets/11"
      },
      {
        "rel": "self",
        "href": "/mobile/platform/location/assets/11"
      }
    ]
  },
  "beacon": {
    "altBeacon": {
      "id1": "B9407F30-F5F8-466E",
      "id2": "AFF9",
      "id3": "25556B57FE6D"
    }
  },
  "attributes": {
    "manufacturer": "Abc Company",
    "status": "Inactive",
    "manufacturerId": "10D39AE7-020E-4467-9CB2-DD36366F899D",
    "visibility": "Private"
  },
  "links": [
```

```
    {
      "rel": "canonical",
      "href": "/mobile/platform/location/devices/11"
    },
    {
      "rel": "self",
      "href": "/mobile/platform/location/devices/11"
    }
  ]
}
```

location.places.register(places, context, httpOptions)

This method lets you create one or more places.

Arguments

`places`: Required. JSON object that follows the request root schema (Places Array) that is shown for the `POST /mobile/system/locationManagement/places` operation in REST APIs for Oracle Mobile Cloud Service. Here's an example:

```
{
    "items":[
        {
            "name":"FixItFast Redwood City Warehouse",
            "label":"FixItFast Warehouse",
            "parentPlace":42,
            "description":"FixItFast Warehouse in Redwood City",
            "address":{
                "gpsPoint":{
                    "latitude":122,
                    "longitude":37
                }
            },
            "devices":[
                12345
            ],
            "attributes":{
                "hours":"9am-6pm"
            }
        }
    ]
}
```

`context`: Required. JSON object as described in Location Management Context Argument.

Response

The response body, which shows the stored places, is a JSON object that follows the response root schema (Places Array) that is shown for the `POST /mobile/system/locationManagement/places` operation in REST APIs for Oracle Mobile Cloud Service.

Examples

In this example, the request body would look like this:

```
{
  "userName":"anAdministrator",
  "places": {
    "items":[
       {
            "name":"FixItFast Redwood City Warehouse",
            "label":"FixItFast Warehouse",
            "description":"FixItFast Warehouse in Redwood City",
            "address":{
                "gpsPoint":{
                    "latitude":89,
                    "longitude":37
                }
            },
                "attributes":{
                "hours":"9am-6pm"
            }
        }
    ]
  }
}
```

This example puts the `username` in the `context` object and passes `places` as the request body.

```
service.post('/mobile/custom/incidentreport/places', function (req, res) {
  req.oracleMobile.location.places.register(
    req.body.places,
    {
      username: req.body.userName,
      mbe: req.oracleMobile.mbe.getMBE().name,
      version: req.oracleMobile.mbe.getMBE().version
    }).then(
    function (result) {
      res.type('application/json');
      res.status(result.statusCode).send(result.result);
    },
    function (error) {
      console.dir(error);
      res.status(error.statusCode).send(error.error);
    }
  );
});
```

Here's an example of the response body.

```
{
  "items": [
    {
```

```
      "id": 10,
      "createdOn": "2016-11-08T17:55:21.816Z",
      "createdBy": "john.doe",
      "modifiedOn": "2016-11-08T17:55:21.816Z",
      "modifiedBy": "john.doe",
      "name": "FixItFast Redwood City Warehouse",
      "label": "FixItFast Warehouse",
      "description": "FixItFast Warehouse in Redwood City",
      "hasChildren": false,
      "address": {
        "gpsPoint": {
          "longitude": 37,
          "latitude": 89
        }
      },
      "attributes": {
        "hours": "9am-6pm"
      },
      "links": [
        {
          "rel": "canonical",
          "href": "/mobile/platform/location/places/10"
        },
        {
          "rel": "self",
          "href": "/mobile/platform/location/places/10"
        }
      ]
    }
  ]
}
```

location.places.remove(id, context, httpOptions)

Use this method to delete places.

Arguments

`id`: Required. IDs of the places to remove. This argument can be either a single value or an array of values.

`context`: Required. JSON object as described in Location Management Context Argument.

Response

If you provide a single value, then the service doesn't return a response body. The status code is 204 if the place was deleted and 404 if it doesn't exist.

If you provide an array of IDs, then the status code is 200 for a successful request. The response contains a `batch` object with an array of responses for the individual delete requests. For schema details, see the Delete Multiple Places operation in REST APIs for Oracle Mobile Cloud Service.

Here's an example:

```
{
    "batch":[
        {
            "body":{
                "id":222,
                "message":"place was deleted successfully."},
            "code":200},
        {
            "body":{
                "id":223,
                "message":"place was deleted successfully."},
            "code":200},
        {
            "body":{
                "id":224,
                "message":"place not found."},
            "code":404}
    ]
}
```

Examples

In this example, if the `id` query parameter contains multiple IDs, then it converts the query string into an array.

Note that the user name of the user who has the `MobileEnvironment_System` role is passed in the `user` query parameter.

```
service.delete('/mobile/custom/location/places, function(req,res) {
  var contextObject = {
    username: req.query.user,
    mbe: req.oracleMobile.mbe.getMBE().name,
    version: req.oracleMobile.mbe.getMBE().version
  };
  var id = req.query.id.split(',');
  if (id.length == 0){
      id = req.query.id;
  }
  req.oracleMobile.location.places.remove(
      id,
      contextObject
    ).then(
    function (result) {
      res.type('application/json');
      res.status(result.statusCode).send(result.result);
    },
    function (error) {
      console.dir(error);
      res.status(error.statusCode).send(error.error);
    }
  );
});
```

location.places.removeCascade(id, context, httpOptions)

Use this method to delete a parent place and all its child places.

Arguments

`id`: Required. The ID of the place. This ID must be an existing place ID.

`context`: Required. JSON object as described in Location Management Context Argument.

Examples

In this example, if the `cascade` query parameter is `true`, then the method calls `removeCascade()` instead of `remove()`.

Note that the user name of the user who has the `MobileEnvironment_System` role is passed in the `user` query parameter.

```
service.delete('/mobile/custom/location/places/:id', function(req,res) {
  var contextObject = {
    username: req.query.user,
    mbe: req.oracleMobile.mbe.getMBE().name,
    version: req.oracleMobile.mbe.getMBE().version
  };
  var removeFunc = req.oracleMobile.location.places.remove;
  if (req.query.cascade == 'true') {
      removeFunc = req.oracleMobile.location.places.removeCascade;
  }
  removeFunc(
    req.params.id,
    contextObject
  ).then(
     function (result) {
        res.type('application/json');
        res.status(result.statusCode).send(result.result);
      },
      function (error) {
        console.dir(error);
        res.status(error.statusCode).send(error.error);
      }
   )
 });
```

location.places.update(id, place, context, httpOptions)

This method lets you update a place.

Arguments

`id`: Required. The ID of the place. This ID must be an existing place ID.

`place`: Required. JSON object that follows the request root schema (Place) that is shown for the `PUT /mobile/system/locationManagement/place/{id}` operation in REST APIs for Oracle Mobile Cloud Service. Here's an example:

```
{
    "address":{
        "gpsPoint":{
            "latitude":-121.1566,
            "longitude":37.5548
        }
    },
    "devices":[
        1111
    ]
}
```

`context`: Required. JSON object as described in Location Management Context Argument.

Response

The response body, which shows the updated place, is a JSON object that follows the response root schema (Place) that is shown for the `PUT /mobile/system/locationManagement/places/{id}` operation in REST APIs for Oracle Mobile Cloud Service.

Examples

In this example, the request body would look like this:

```
{
   "userName":"anAdministrator",
   "place":{
      "address":{
         "gpsPoint":{
            "latitude":-89,
            "longitude":37
         }
      },
      "devices":[
         11
      ]
   }
}
```

This example puts the `username` in the `context` object and passes `place` as the request body.

```
service.put('/mobile/custom/incidentreport/place/:id', function (req, res)
{
  req.oracleMobile.location.place.update(
    req.params.id,
    req.body.place,
    {
```

```
      username: req.body.userName,
      mbe: req.oracleMobile.mbe.getMBE().name,
      version: req.oracleMobile.mbe.getMBE().version
    }).then(
    function (result) {
      res.type('application/json');
      res.status(result.statusCode).send(result.result);
    },
    function (error) {
      console.dir(error);
      res.status(error.statusCode).send(error.error);
    }
  );
});
```

Here's an example of the response body.

```
{
  "id": 11,
  "createdOn": "2016-11-08T23:36:55.371Z",
  "createdBy": "anAdministrator",
  "modifiedOn": "2016-11-08T23:37:45.576Z",
  "modifiedBy": "anAdministrator",
  "name": "FixItFast Redwood City Warehouse",
  "label": "FixItFast Warehouse",
  "description": "FixItFast Warehouse in Redwood City",
  "hasChildren": false,
  "address": {
    "gpsPoint": {
      "longitude": 37,
      "latitude": 89
    }
  },
  "devices": [
    {
      "id": 11,
      "createdOn": "2016-11-08T18:01:18.531Z",
      "createdBy": "anAdministrator",
      "modifiedOn": "2016-11-08T22:45:47.545Z",
      "modifiedBy": "anAdministrator",
      "name": "RC_WH_01_F01_B016",
      "description": "Beacon on 2nd Floor in FixitFast Warehouse in
Redwood City",
      "beacon": {
        "altBeacon": {
          "id1": "B9407F30-F5F8-466E",
          "id2": "AFF9",
          "id3": "25556B57FE6D"
        }
      },
      "attributes": {
        "manufacturer": "Abc Company",
        "status": "Inactive",
        "manufacturerId": "10D39AE7-020E-4467-9CB2-DD36366F899D",
```

```
        "visibility": "Private"
      },
      "links": [
        {
          "rel": "canonical",
          "href": "/mobile/platform/location/devices/11"
        },
        {
          "rel": "self",
          "href": "/mobile/platform/location/devices/11"
        }
      ]
    }
  ],
  "attributes": {
    "hours": "9am-6pm"
  },
  "links": [
    {
      "rel": "canonical",
      "href": "/mobile/platform/location/places/11"
    },
    {
      "rel": "self",
      "href": "/mobile/platform/location/places/11"
    }
  ]
}
```


Accessing the Notifications API from Custom Code

You can use the Notifications API to send a message to the mobile app users, such as an alert about an upcoming event or news that the user might be interested in. You can specify a target for the message such as a device, user, or operating system, and you can schedule the message. You can also inquire about notifications, and delete scheduled notifications that haven't been sent.

For more information about the ways in which you can use notifications, see Notifications.

This API has the following methods:

- notification.getAll(context, options, httpOptions): Retrieves all notifications.
- notification.getById(id, context, options, httpOptions): Retrieves a notification for a specific notification ID.
- notification.post(notification, context, options, httpOptions): Creates a notification.
- notification.remove(id, context, options, httpOptions): Deletes a notification.

Notifications Context Argument

All the Notifications API methods require a `context` argument, which is a JSON object with the following properties. This information is required to get authorization to send and view the notifications.

Note that the custom code can call mbe.getMBE() to get this information.

Property	Desc	Type
`mbe`	The name of the mobile backend that's associated with the notification.	String
`mbeId`	(Optional) The ID of the mobile backend that's associated with the notification. When omitted, the default is the mobile backend id that the mobile application is using.	String
`version`	The version of the mobile backend.	String

notification.getAll(context, options, httpOptions)

This method lets you retrieve the notifications that match your criteria. Only the notifications that match ALL the criteria are returned.

Arguments

`context`: Required. JSON object as described in Notifications Context Argument.

`options`: Optional. JSON object. This object can have these properties in addition to those listed in Common options Argument Properties:

Property	Description	Type	Default
`createdOnOrAfter`	Criteria: Filter by `createdOn` on or after the given UTC date/time (in `YYYY-DD-MM[Thh:mm]Z` format).	String	None
`createdOnOrBefore`	Criteria: Filter by `createdOn` on or before the given UTC date/time (in `YYYY-DD-MM[Thh:mm]Z` format).	String	None
`limit`	The maximum number of items to be returned. If the requested limit is too large, then a lower limit is substituted.	Integer	None
`offset`	The zero-based index of the first item to return.	Integer	None

Property	Description	Type	Default
`orderBy`	Specifies the ordering for the query operations. The default sort order is ascending by ID. The format is: "orderBy" "=" 1#(attr [":" "asc" | "desc"]), where the `attr` parameter may be `id`, `status`, `tag`, `platform`, `sendOn`, `createdOn`, or `processedOn`.	String	None
`processedOnOrAfter`	Criteria: Filter by `processedOn` on or after the given UTC date/time (in `YYYY-DD-MM[Thh:mm]Z` format).	String	None
`processedOnOrBefore`	Criteria: Filter by `processedOn` on or before the given UTC date/time (in `YYYY-DD-MM[Thh:mm]Z` format).	String	None
`q`	Filter results based on a case-insensitive partial match of this string with the tag. For example, `q=market` returns notifications with `tag` equal to `Marketing`, `marketing`, and `markets`.	String	None
`sendOnOrAfter`	Criteria: Filter by `sendOn` on or after the given UTC date/time (in `YYYY-DD-MM[Thh:mm]Z` format).	String	None
`sendOnOrBefore`	Criteria: Filter by `sendOn` on or before the given UTC date/time (in `YYYY-DD-MM[Thh:mm]Z` format).	String	None
`status`	Criteria: Filter by status	String	None
`tag`	Criteria: Filter by tag	String	None

Response

The response body is a JSON object that follows the `notificationPaging` schema that is shown for the `GET /mobile/system/notifications/notifications` operation in REST APIs for Oracle Mobile Cloud Service.

Examples

Here's an example of calling this method:

```
service.get('/mobile/custom/incidentreport/notifications',
  function (req, res) {
    req.oracleMobile.notification.getAll({
      mbe: req.oracleMobile.mbe.getMBE().name,
      version: req.oracleMobile.mbe.getMBE().version})
    .then(
      function (result) {
        res.status(result.statusCode).send(result.result);
      },
      function (error) {
        res.status(error.statusCode).send(error.error);
      }
    );
  });
```

Here's an example of a response body.

```
{
  "items": [
    {
      "id": 2,
      "message": "Incident Updated: Broken Dryer",
      "users": [
        "J Doe"
      ],
      "roles": [],
      "notificationTokens": [],
      "status": "New",
      "createdOn": "2015-09-24T21:58:04.465Z",
      "links": [
        {
          "rel": "canonical",
          "href": "/mobile/system/notifications/notifications/2"
        },
        {
          "rel": "self",
          "href": "/mobile/system/notifications/notifications/2"
        }
      ]
    },
    {
      "id": 3,
      "message": "Incident Updated: Malfunctioning Air Conditioner",
      "users": [
```

```
        "Lynn Smith"
      ],
      "roles": [],
      "notificationTokens": [],
      "status": "New",
      "createdOn": "2015-09-24T21:58:07.413Z",
      "links": [
        {
          "rel": "canonical",
          "href": "/mobile/system/notifications/notifications/3"
        },
        {
          "rel": "self",
          "href": "/mobile/system/notifications/notifications/3"
        }
      ]
    }
  ],
  "hasMore": false,
  "limit": 2,
  "count": 2,
  "links": [
    {
      "rel": "canonical",
      "href": "/mobile/system/notifications/notifications/?
offset=0&limit=2"
    },
    {
      "rel": "self",
      "href": "/mobile/system/notifications/notifications/"
    }
  ]
}
```


notification.getById(id, context, options, httpOptions)

This method lets you retrieve a specific notification by its ID.

Arguments

`id`: Required. String or integer. The generated notification ID.

`context`: Required. JSON object as described in Notifications Context Argument.

`options`: Optional. JSON object as described in Common options Argument Properties.

Response

The response body is a JSON object that follows the `notification` schema that is shown for the `GET /mobile/system/notifications/notifications/{id}` operation in REST APIs for Oracle Mobile Cloud Service.

Examples

Here's an example of calling the method to get a notification:

```
service.get('/mobile/custom/incidentreport/notifications/:id',
  function (req, res) {
    req.oracleMobile.notification.getById(req.params.id, {
        mbe: req.oracleMobile.mbe.getMBE().name,
        version: req.oracleMobile.mbe.getMBE().version})
    .then(
      function (result) {
        res.status(result.statusCode).send(result.result);
      },
      function (error) {
        res.status(error.statusCode).send(error.error);
      }
    );
  });
```

Here's an example of a response body.

```
{
  "id": 1,
  "message": "Incident Updated: Leaky Faucet",
  "users": [
    "Lynn Smith"
  ],
  "roles": [],
  "notificationTokens": [],
  "status": "New",
  "createdOn": "2015-09-24T21:44:45.708Z",
  "links": [
    {
      "rel": "canonical",
      "href": "/mobile/system/notifications/notifications/1"
    },
    {
      "rel": "self",
      "href": "/mobile/system/notifications/notifications/1"
    }
  ]
}
```

notification.post(notification, context, options, httpOptions)

This method lets you create a notification.

Arguments

`notification`: Required. JSON object that follows the `notificationCreate` schema that is shown for the `POST /mobile/system/notifications/notifications` operation in REST APIs for Oracle Mobile Cloud Service. Here's an example:

```
{
    message:'This is the alert message.',
    tag:'Marketing',
    notificationTokens:['APNSdeviceToken']
}
```

`context`: Required. JSON object as described in Notifications Context Argument.

`options`: Optional. JSON object as described in Common options Argument Properties.

Response

The return value includes this header:

Header	Description	Type
`Location`	Canonical resource URI for the notification.	String

The response body, which shows the stored notification, is a JSON object that follows the `notification` schema that is shown for the `POST /mobile/system/notifications/notifications` operation in REST APIs for Oracle Mobile Cloud Service.

Examples

In this example of posting a notification, the request body would look like this: `{incidentName: 'Leaky Faucet', customerName: 'Lynn Smith'}`.

```
service.post('/mobile/custom/incidentreport/notifications',
  function (req, res) {
    var notification = {
      sendOn: '2016-06-25T6:00Z',
      message: 'Incident Updated: ' +
      req.body.incidentName,
      users: [req.body.customerName]
    };
    req.oracleMobile.notification.post(notification, {
        mbe: req.oracleMobile.mbe.getMBE().name,
        version: req.oracleMobile.mbe.getMBE().version})
    .then(
      function (result) {
        res.status(result.statusCode).send(result.result);
```

```
      },
      function (error) {
        res.status(error.statusCode).send(error.error);
      }
    );
  });
```

Here's an example of the response body.

```
{
  "id": 1,
  "message": "Incident Updated: Leaky Faucet",
  "users": [
    "Lynn Smith"
  ],
  "roles": [],
  "notificationTokens": [],
  "sendOn": "2016-06-25T06:00Z",
  "status": "New",
  "createdOn": "2015-06-24T21:44:45.708Z",
  "links": [
    {
      "rel": "canonical",
      "href": "/mobile/system/notifications/notifications/1"
    },
    {
      "rel": "self",
      "href": "/mobile/system/notifications/notifications/1"
    }
  ]
}
```

notification.remove(id, context, options, httpOptions)

This method lets you delete a notification. You can delete a notification only if its status is `Scheduled`.

Arguments

`id`: Required. String or integer. The generated notification ID.

`context`: Required. JSON object as described in Notifications Context Argument.

`options`: Optional. JSON object as described in Common options Argument Properties.

Example

Here's an example of calling this method:

```
service.delete('/mobile/custom/incidentreport/notifications/:id',
  function (req, res) {
    req.oracleMobile.notification.remove(req.params.id, {
        mbe: req.oracleMobile.mbe.getMBE().name,
        version: req.oracleMobile.mbe.getMBE().version})
```

```
  .then(
    function (result) {
      res.status(result.statusCode).send(result.result);
    },
    function (error) {
      res.status(error.statusCode).send(error.error);
    }
  );
});
```

Accessing the Storage API from Custom Code

The Storage API lets you store mobile application objects in the cloud. An object can be text, JSON, or a binary object such as an image. These objects are grouped by collection. To learn about collections and objects, see Storage.

This API has the following methods:

- storage.doesCollectionExist(collectionId, options, httpOptions): Indicates if a collection exists, and, optionally, whether its ETag matches.
- storage.doesExist(collectionId, objectId, options, httpOptions): Indicates if an object exists, and, optionally, whether its ETag matches.
- storage.getAll(collectionId, options, httpOptions): Returns the metadata for every object in a collection.
- storage.getById(collectionId, objectId, options, httpOptions): Retrieves an object and its metadata.
- storage.getCollection(collectionId, options, httpOptions): Retrieves metadata about a collection.
- storage.getCollections(options, httpOptions): Returns metadata about each collection that is available through the mobile backend.
- storage.remove(collectionId, objectId, options, httpOptions): Removes an object from a collection.
- storage.store(collectionId, object, options, httpOptions): Adds an object and automatically assigns an ID for it.
- storage.storeById(collectionId, objectId, object, options, httpOptions): Adds or updates an object based on an ID that you specify.

storage.doesCollectionExist(collectionId, options, httpOptions)

You can use this method to determine whether a collection exists. You can also use it to see if the collection matches (or does not match) an ETag.

Arguments

`collectionId`: Required. String. The name of the collection. When you look at the metadata for the collection, this value corresponds to the metadata's `id` value.

`options`: Optional. JSON object. This object can have these properties in addition to those listed in Common options Argument Properties:

Property	Description	Type	Default
`encodeURI`	Set to `true` to URI-encode the `collectionId` value. This option can be useful for multibyte values.	Boolean	`false`
`ifMatch`	The call returns `true` only if the ETag of the corresponding object matches one of the values specified in this property.	String	None
`ifNoneMatch`	The call returns `true` only if the ETag of the corresponding object does not match one of the values specified by this property.	String	None

Response

This method returns a Boolean value.

Example

The following example uses this method to verify that the collection exists before it stores an object in it.

```
req.oracleMobile.storage.doesCollectionExist('attachments').then(
  function(result){
    if (result) {
      req.oracleMobile.storage.store('attachments', {id: 'incident412-
pic'}, {inType: 'json'}).then(
        function (result) {
          res.status(result.statusCode).send(result.result);
        },
        function (error) {
          res.status(error.statusCode).send(error.error);
        }
      );
    } else {
      res.status(404).send('Storage has not been configured for this app.
Please contact your admin.');
    };
  },
  function(error){
    res.status(error.statusCode).send(error.error);
  }
);
```

storage.doesExist(collectionId, objectId, options, httpOptions)

You can use this method to determine whether an object exists. You can also use it to see if the object matches (or does not match) an ETag, or if it was modified after a specified date.

Arguments

`collectionId`: Required. String. The name of the collection. When you look at the metadata for the collection, this value corresponds to the metadata's `id` value.

`objectId`: Required. String. The object being accessed. If the object was stored using the `storage.storeById()` method, then this is the ID that was provided as the `id` argument, and, if the object was stored using the `storage.store()` method, then the ID was generated. When looking at the object metadata, this argument value corresponds to the metadata's `id` attribute.

`options`: Optional. JSON object. This object can have these properties in addition to those listed in Common options Argument Properties:

Property	Description	Type	Default
`contentDisposition`	This property lets you specify the value of the `Content-Disposition` response header.	String	None
`encodeURI`	Set to `true` to URI-encode the `collectionId`, `objectId`, and `user` values. This option can be useful for multibyte values.	Boolean	`false`
`ifMatch`	The call completes successfully only if the ETag of the corresponding object matches one of the values specified in this property.	String	None

Property	Description	Type	Default
`ifModifiedSince`	Date and time in HTTP-date format. For example, `Mon, 30 Jun 2014 19:43:31 GMT`. The request completes successfully only if the object was modified after the date specified in this property. You can use this property to reduce the amount of data that is transported by not re-retrieving data if it hasn't changed.	Date	None
`ifNoneMatch`	The call completes successfully only if the ETag of the corresponding object does not match one of the values specified by this property. You can use this property to reduce the amount of data that is transported by not re-retrieving data if it hasn't changed.	String	None
`ifUnmodifiedSince`	Date and time in HTTP-date format. For example, `Mon, 30 Jun 2014 19:43:31 GMT`. The request completes successfully only if the object wasn't modified after the date specified in this property.	Date	None

Property	Description	Type	Default
`user`	This is the ID (not the user name) of a user. This query parameter allows a user with READ_ALL/ READ_WRITE_ALL permission to access another user's isolated space. A user with READ/READ_WRITE permission may access only their own space.	String	If you are inquiring about a shared collection, there is no default. If you are inquiring about an isolated collection, and you have READ_ALL/ READ_WRITE_ALL permission, then the signed-in user is assumed unless you include this property. If you have READ_ALL/ READ_WRITE_ALL permission for an isolated collection, you must include this property to inquire about objects in another user's space.

Response

This method returns a Boolean value.

Example

In this example, the code calls `doesExist` to see if the stored object still has the same ETag as when it was last retrieved (`"1"`).

```
req.oracleMobile.storage.doesExist('attachments', 'incident412-pic',
{ifMatch: '\"' + 1 + '\"'}).then(
  function (result) {
    res.status(200).send('Object has not changed.');
  },
  function (error) {
    res.status(412).send('Object was modified by someone else.');
  }
)
```

storage.getAll(collectionId, options, httpOptions)

This method returns the metadata for every object in a collection.

Arguments

`collectionId`: Required. String. The name of the collection. When you look at the metadata for the collection, this value corresponds to the metadata's `id` value.

`options`: Optional. JSON object. This object can have these properties in addition to those listed in Common options Argument Properties:

Property	Description	Type	Default
`encodeURI`	Set to `true` to URI-encode the `collectionId`, `orderBy`, and `user` values. This option can be useful for multibyte values.	Boolean	`false`
`limit`	The maximum number of items to be returned. If the requested limit is greater than 100, then 100 is used instead.	Integer	None
`offset`	The zero-based index of the first item to return.	Integer	None
`orderBy`	Use this property to sort the results by `name`, `modifiedBy`, `modifiedOn`, `createdBy`, `createdOn`, or `contentLength`. You can append `:asc` or `:desc` to specify whether to sort in ascending or descending order. For example, `modifiedOn:desc`.	String	None
`q`	The items that are returned are based on a case-insensitive partial match of the `id`, `name`, `createdBy` or `modifiedBy` property of an item. For example, if you set this property to `sam`, it could return an object with an `id` of `axsam3` and an object with a `createdBy` of `SAMANTHA`.	String	None

Property	Description	Type	Default
`sync`	When this property is present and has a value of `true`, then the return value contains the information required by the Synchronization library to cache the data locally for offline use. You can get this value from the `Oracle-Mobile-Sync-Agent` request header, when present.	Boolean	`false`
`totalResults`	When this property is present with a value of `true`, then the response body contains the `totalResults` attribute with a value that represents the total number of items in the collection. By default, the response does not contain this value.	Boolean	`false`
`user`	This is the ID (not the user name) of a user. Use * (wildcard) to get all users. This query parameter allows a user with READ_ALL/ READ_WRITE_ALL permission to access another user's isolated space. A user with READ/READ_WRITE permission may access only their own space.	String	If you are inquiring about a shared collection, there is no default. If you are inquiring about an isolated collection, and you have READ_ALL/ READ_WRITE_ALL permission, then the signed-in user is assumed unless you include this property. If you have READ_ALL/ READ_WRITE_ALL permission for an isolated collection, you must include this property to inquire about objects in another user's space.

Response

The return value includes these headers:

Header	Description	Type
`Cache-Control`	Describes how the result may be cached.	String
`Oracle-Mobile-Sync-Resource-Type`	The Synchronization library uses this header.	String

The response body is a JSON object that follows the response body schema that is shown for the `GET /mobile/platform/storage/collections/{collection}/objects` operation in REST APIs for Oracle Mobile Cloud Service.

Examples

Here's an example of calling this method. The response lists the objects by modified date, in descending order. Because the `sync` property is set to `true`, the client app can cache the response.

```
// Get metadata about the objects in the attachments collection.
// List most recently modified first.
service.get('/mobile/custom/incidentreport/attachments',
  function (req, res) {
    req.oracleMobile.storage.getAll('attachments',
    {orderBy: 'modifiedOn:desc', sync: true}).then(
      function (result) {
        res.status(result.statusCode).send(result.result);
      },
      function (error) {
        res.status(error.statusCode).send(error.error);
      }
    );
  });
```

Here's an example of a response body:

```
{
    "items":[
        {
            "eTag":"\"2\"",
            "id":"incident412-pic",
            "createdBy":"jdoe",
            "name":"Incident Picture",
            "createdOn":"2014-11-20T19:57:04Z",
            "modifiedOn":"2014-11-20T19:58:09Z",
            "modifiedBy":"jdoe",
            "links":[
                {
                    "rel":"canonical",
                    "href":"/mobile/platform/storage/collections/
attachments/objects/profile-pic"
                },
                {
                    "rel":"self",
                    "href":"/mobile/platform/storage/collections/
```

```
attachments/objects/profile-pic"
                }
            ],
            "contentType":"image/png",
            "contentLength":937647
        },
        {
            "eTag":"\"1\"",
            "id":"incident131-pic",
            "createdBy":"jsmith",
            "name":"Incident Picture",
            "createdOn":"2014-11-20T18:27:02Z",
            "modifiedOn":"2014-11-20T18:27:02Z",
            "modifiedBy":"jsmith",
            "links":[
                {
                    "rel":"canonical",
                    "href":"/mobile/platform/storage/collections/
attachments/objects/0683d48b-fdc5-4397-8ca2-824e2b0cae65"
                },
                {
                    "rel":"self",
                    "href":"/mobile/platform/storage/collections/
attachments/objects/0683d48b-fdc5-4397-8ca2-824e2b0cae65"
                }
            ],
            "contentType":"image/jpeg",
            "contentLength":5266432
        }
    ],
    "hasMore":true,
    "limit":2,
    "offset":4,
    "count":2,
    "totalResults":7,
    "links":[
        {
            "rel":"canonical",
            "href":"/mobile/platform/storage/collections/attachments/
objects/"
        },
        {
            "rel":"self",
            "href":"/mobile/platform/storage/collections/attachments/
objects?offset=4&limit=2&orderBy=name:asc&totalResults=true"
        },
        {
            "rel":"prev",
            "href":"/mobile/platform/storage/collections/attachments/
objects?offset=2&limit=2&orderBy=name:asc&totalResults=true"
        },
        {
            "rel":"next",
            "href":"/mobile/platform/storage/collections/attachments/
objects?offset=6&limit=2&orderBy=name:asc&totalResults=true"
```

```
        }
    ]
}
```

storage.getById(collectionId, objectId, options, httpOptions)

This method retrieves an object and its metadata from a collection based on the object identifier.

Arguments

`collectionId`: Required. String. The name of the collection. When you look at the metadata for the collection, this value corresponds to the metadata's `id` value.

`objectId`: Required. String. The object being accessed. If the object was stored using the `storage.storeById()` method, then this is the ID that was provided as the `id` argument, and, if the object was stored using the `storage.store()` method, then the ID was generated. When looking at the object metadata, this argument value corresponds to the metadata's `id` attribute.

`options`: Optional. JSON object. This object can have these properties in addition to those listed in Common options Argument Properties:

Property	Description	Type	Default
`contentDisposition`	This property lets you specify the value of the `Content-Disposition` response header.	String	None
`encodeURI`	Set to `true` to URI-encode the `collectionId`, `objectId`, and `user` values. This option can be useful for multibyte values.	Boolean	`false`
`ifMatch`	The call completes successfully only if the ETag of the corresponding object matches one of the values specified in this property.	String	None

Property	Description	Type	Default
`ifModifiedSince`	Date and time in HTTP-date format. For example, `Mon, 30 Jun 2014 19:43:31 GMT`. The request completes successfully only if the object was modified after the date specified in this property. You can use this property to reduce the amount of data that is transported by not re-retrieving data if it hasn't changed.	Date	None
`ifNoneMatch`	The call completes successfully only if the ETag of the corresponding object does not match one of the values specified by this property. You can use this property to reduce the amount of data that is transported by not re-retrieving data if it hasn't changed.	String	None
`ifUnmodifiedSince`	Date and time in HTTP-date format. For example, `Mon, 30 Jun 2014 19:43:31 GMT`. The request completes successfully only if the object wasn't modified after the date specified in this property.	Date	None
`range`	This property lets you request a subset of bytes. For example, `bytes=0-99` gets the first 100 bytes.	String	None

Property	Description	Type	Default
`sync`	When this property is present and has a value of `true`, then the return value contains the information required by the Synchronization library to cache the data locally for offline use. You can get this value from the `Oracle-Mobile-Sync-Agent` request header, when present.	Boolean	`false`
`user`	This is the ID (not the user name) of a user. This query parameter allows a user with READ_ALL/ READ_WRITE_ALL permission to access another user's isolated space. A user with READ/READ_WRITE permission may access only their own space.	String	If you are inquiring about a shared collection, there is no default. If you are inquiring about an isolated collection, and you have READ_ALL/ READ_WRITE_ALL permission, then the signed-in user is assumed unless you include this property. If you have READ_ALL/ READ_WRITE_ALL permission for an isolated collection, you must include this property to get an object from another user's space.

Response

The return value includes these headers:

Header	Description	Type
`Accept-Ranges`	This header indicates that byte ranges may be provided when requesting an object resource.	String
`Cache-Control`	Describes how the result may be cached.	String
`Content-Disposition`	This response header is returned if the `options` argument included the `contentDisposition` property. The value for the response header is the same as the value for the property.	String

Header	Description	Type
`Content-Length`	The size of the object in bytes.	Number
`Content-Type`	The media type of the object, such as `image/jpeg`.	String
`Etag`	Each item has an ETag value. This value changes each time the item is updated. The value includes the starting and ending quotation marks (for example, `"2"`).	String
`Last-Modified`	The date and time when the resource was last modified. This date is in RFC-1123 format. For example, `Fri, 29 Aug 2014 12:34:56 GMT`.	Date
`Oracle-Mobile-Canonical-Link`	A relative URI that you can use to uniquely reference this object.	String
`Oracle-Mobile-Created-By`	The user name of the user who created the object.	String
`Oracle-Mobile-Created-On`	The date and time, in ISO 8601 format (for example, `2014-06-30T01:02:03Z`), when the object was created.	String
`Oracle-Mobile-Modified-By`	The user name of the user who last modified the object.	String
`Oracle-Mobile-Modified-On`	The date and time, in ISO 8601 format (for example, `2014-06-30T01:02:03Z`), when the object was last modified.	String
`Oracle-Mobile-Name`	The display name for the object.	String
`Oracle-Mobile-Self-Link`	A relative URI that you can use to uniquely reference this object within the specified isolation level.	String
`Oracle-Mobile-Sync-Expires`	This header is used by the Synchronization library.	String
`Oracle-Mobile-Sync-No-Store`	This header is used by the Synchronization library.	Boolean

The response body is the stored object.

Example

Here is an example of calling this method. Because the `sync` property is set to `true`, the client app can cache the response.

```
req.oracleMobile.storage.getById('attachments', 'incident412-notes',
{sync: true}).then(
  function (result) {
```

```
    res.status(result.statusCode).send(result.result);
  },
  function (error) {
    res.status(error.statusCode).send(error.error);
  }
);
```

storage.getCollection(collectionId, options, httpOptions)

This method returns metadata about a particular collection.

Arguments

`collectionId`: Required. String. The name of the collection. When you look at the metadata for the collection, this value corresponds to the metadata's `id` value.

`options`: Optional. JSON object. This object can have these properties in addition to those listed in Common options Argument Properties:

Property	Description	Type	Default
`encodeURI`	Set to `true` to URI-encode the `collectionId` value. This option can be useful for multibyte values.	Boolean	`false`
`ifMatch`	The call completes successfully only if the ETag of the corresponding object matches one of the values specified in this property.	String	None
`ifNoneMatch`	The call completes successfully only if the ETag of the corresponding object does not match one of the values specified by this property.	String	None
`sync`	When this property is present and has a value of `true`, then the return value contains the information required by the Synchronization library to cache the data locally for offline use. You can get this value from the `Oracle-Mobile-Sync-Agent` request header, when present.	Boolean	`false`

Response

The return value includes these headers:

Header	Description	Type
`Cache-Control`	Describes how the result may be cached.	String
`Etag`	Each item has an ETag value. This value changes each time the item is updated. The value includes the starting and ending quotation marks (for example, `"2"`).	String

The response body is a JSON object that follows the `Collection` schema that is shown for the `GET /mobile/platform/storage/collections/{collection}` operation in REST APIs for Oracle Mobile Cloud Service.

Examples

Here's an example of calling this method. Because the `sync` property is set to `true`, the client app can cache the response.

```
req.oracleMobile.storage.getCollection('attachments', {sync: true}).then(
  function (result) {
    res.status(result.statusCode).send(result.result);
  },
  function (error) {
    res.status(error.statusCode).send(error.error);
  }
);
```

Here's an example of a response body:

```
{
    "id":"attachments",
    "description":"Attachments for technician notes.",
    "contentLength":6205619,
    "eTag":"\"1.0\"",
    "links":[
        {
            "rel":"canonical",
            "href":"/mobile/platform/storage/collections/attachments"},
        {
            "rel":"self",
            "href":"/mobile/platform/storage/collections/attachments"}
    ]}
```

storage.getCollections(options, httpOptions)

This method returns metadata about each collection that is available through the mobile backend.

Arguments

`options`: Optional. JSON Object. This object can have these properties in addition to those listed in Common options Argument Properties:

Property	Description	Type	Default
`limit`	The maximum number of items to be returned. If the requested limit is too large, then a lower limit is substituted.	Integer	None
`offset`	The zero-based index of the first item to return.	Integer	`0` (zero)
`sync`	When this property is present and has a value of `true`, then the return value contains the information required by the Synchronization library to cache the data locally for offline use. You can get this value from the `Oracle-Mobile-Sync-Agent` request header, when present.	Boolean	`false`
`totalResults`	When this property is present with a value of `true`, then the then the response body contains the `totalResults` property with a value that represents the total number of items in the collection. By default, this property is not returned.	Boolean	`false`

Response

The return value includes these headers:

Header	Description	Type
`Cache-Control`	Describes how the result may be cached.	String
`Oracle-Mobile-Sync-Resource-Type`	The Synchronization library uses this header.	String

The response body is an array of `items` in JSON format that follows the `Collection Array` schema that is shown for the `GET /mobile/platform/storage/collections` operation in REST APIs for Oracle Mobile Cloud Service.

Example

Here is an example of calling this method. Because the `sync` property is set to `true`, the client app can cache the response.

```
req.oracleMobile.storage.getCollections({sync: true}).then(
  function (result) {
    res.status(result.statusCode).send(result.result);
  },
  function (error) {
    res.status(error.statusCode).send(error.error);
  }
);
```

Here's an example of a response body:

```
{
    "items":[
        {
            "id":"logs",
            "description":"Application logs.",
            "contentLength":0,
            "eTag":"\"1.0\"",
            "links":[
                {
                    "rel":"canonical",
                    "href":"/mobile/platform/storage/collections/logs"},
                {
                    "rel":"self",
                    "href":"/mobile/platform/storage/collections/logs"}
            ]},
        {
            "id":"attachments",
            "description":"Attachments for technician notes.",
            "contentLength":6205619,
            "eTag":"\"1.0\"",
            "links":[
                {
                    "rel":"canonical",
                    "href":"/mobile/platform/storage/collections/
attachments"},
                {
                    "rel":"self",
                    "href":"/mobile/platform/storage/collections/
attachments"}
            ]}
    ],
    "hasMore":false,
    "limit":100,
    "offset":0,
```

```
    "count":2,
    "links":[
        {
            "rel":"canonical",
            "href":"/mobile/platform/storage/collections/"},
        {
            "rel":"self",
            "href":"/mobile/platform/storage/collections?
offset=0&limit=100"}
    ]}
```

storage.remove(collectionId, objectId, options, httpOptions)

This method removes an object from a collection based on the object identifier.

Arguments

`collectionId`: Required. String. The name of the collection. When you look at the metadata for the collection, this value corresponds to the metadata's `id` value.

`objectId`: Required. String. The ID of the object to remove.

`options`: Optional. JSON object. This object can have these properties in addition to those listed in Common options Argument Properties:

Property	Description	Type	Default
`encodeURI`	Set to `true` to URI-encode the `collectionId`, `objectId`, and `user` values. This option can be useful for multibyte values.	Boolean	`false`
`ifMatch`	The call completes successfully only if the ETag of the corresponding object matches one of the values specified in this property. You can use this property to ensure that the operation succeeds only if the object wasn't modified after you last requested it.	String	None
`ifModifiedSince`	Date and time in HTTP-date format. For example, `Mon, 30 Jun 2014 19:43:31 GMT`. The request completes successfully only if the object was modified after the date specified in property.	Date	None

Property	Description	Type	Default
`ifNoneMatch`	The call completes successfully only if the ETag of the corresponding object does not match one of the values specified by this property.	String	None
`ifUnmodifiedSince`	Date and time in HTTP-date format. For example, `Mon, 30 Jun 2014 19:43:31 GMT`. The request completes successfully only if the object wasn't modified after the date specified in this property. You can use this property to ensure that the operation succeeds only if no one modified the object after that time.	Date	None
`user`	This is the ID (not the user name) of a user. This query parameter allows a user with READ_ALL/ READ_WRITE_ALL permission to access another user's isolated space. A user with READ/READ_WRITE permission may access only their own space.	String	If you are removing an object in a shared collection, there is no default. If you removing an object in an isolated collection, and you have READ_ALL/ READ_WRITE_ALL permission, then the signed-in user is assumed unless you include this property. If you have READ_ALL/ READ_WRITE_ALL permission for an isolated collection, you must include this property to remove objects from another user's space.

Example

This example removes an object from the `attachments` collection:

```
service.delete('/mobile/custom/incidentreport/attachments/:id',
  function (req, res) {
    req.oracleMobile.storage.remove('attachments', req.params.id).then(
      function (result) {
        res.status(result.statusCode).send(result.result);
      },
```

```
    function (error) {
      res.status(error.statusCode).send(error.error);
    }
  );
});
```

storage.store(collectionId, object, options, httpOptions)

This method lets you store an object and have an identifier automatically assigned to it.

Arguments

`collectionId`: Required. String. The name of the collection. When you look at the metadata for the collection, this value corresponds to the metadata's `id` value.

`object`: Required. Text, JSON object, file, or binary object. The object to store.

`options`: Optional. JSON object. This object can have the following properties in addition to those listed in Common options Argument Properties. Note that the `contentType` property plays an important role for Storage, because that also specifies the mediat type to when the object is requested. If you don't include the `content`, then the content-type defaults to application/octet-stream.

Property	Description	Type	Default
`contentLength`	The size of the object in bytes.	Number	If the object is a string or a buffer, then the default is `object.length`. Otherwise, the default is the sum of its members' lengths.
`contentType`	The media type of object being stored. This property also specifies the media type to return when the object is requested.	String	If the `inType` is `json`, then the `Content-Type` header is set to `application/json` automatically. Otherwise, the default is`application/octet-stream`.
`encodeURI`	Set to `true` to URI-encode the `collectionId`, `mobileName`, and `user` values. This option can be useful for multibyte values.	Boolean	`false`
`mobileName`	The display name for the object. If you don't include the display name, the name is set to the object identifier that this method generates automatically.	String	None

Property	Description	Type	Default
`user`	This is the ID (not the user name) of a user. This query parameter allows a user with READ_ALL/ READ_WRITE_ALL permission to access another user's isolated space. A user with READ/READ_WRITE permission may access only their own space.	String	If you are storing an object in a shared collection, there is no default. If you storing an object in an isolated collection, and you have READ_ALL/ READ_WRITE_ALL permission, then the signed-in user is assumed unless you include this property. If you have READ_ALL/ READ_WRITE_ALL permission for an isolated collection, you must include this property to store objects in another user's space.

Response

The return value includes this header:

Header	Description	Type
`Location`	The URI that corresponds to the newly created object.	String

The response body is a JSON object that follows the schema shown for the response body for the `POST /mobile/platform/storage/collections/{collection}/objects` operation in REST APIs for Oracle Mobile Cloud Service.

Examples

In this example, requests can contain JSON objects, files, plain text, images, and so forth. If the input is a JSON object then it must set `inType` to `json`, and pass in `req.body` for the object. Otherwise, it sets `inType` to `stream`, and passes in `req` for the object.

```
service.post('/mobile/custom/incidentreport/attachments',
function (req, res) {
  if (req.is('json')) {
    // Must specify JSON because there is no stream to pipe from req
    // as Express has read it into json and put it in req.body.
    req.oracleMobile.storage.store('attachments', req.body,
      {
        mobileName: 'Technician Notes',
        inType: 'json',
        outType: 'stream'
      })
      .on('error', function (error) {
```

```
        res.status(error.statusCode).send(error.message)
      })
      .pipe(res);
  } else {
    // For streaming, send req instead of req.body
    req.oracleMobile.storage.store('attachments', req, {
      mobileName: 'Technician Notes',
      contentType: req.header('content-type'),
      inType: 'stream',
      outType: 'stream'
    })
      .on('error', function (error) {
        res.status(error.statusCode).send(error.message)
      })
      .pipe(res);
  }
});
```

In this example, the request body contains a Base-64 encoded image. The code converts it to a binary image before storing it. The request body would look like this:

```
{
  imageName: 'brokenWaterHose',
  base74EncodedImage: '/9j/4AAQSkZJRg...AFFFFAH/2Q=='
}
```

```
// Base 64
service.post('/mobile/custom/incidentreport/attachments',
  function (req, res) {
    // convert Base-64 encoded image to binary image
    image = new Buffer(req.body.base64EncodedImage);
    req.oracleMobile.storage.store('attachments', image,
      {
        contentType: 'image/jpeg',
        mobileName: req.body.imageName
      }
    ).then(
      function (result) {
        res.status(result.statusCode).send(result.result);
      },
      function (error) {
        res.status(error.statusCode).send(error.error);
      }
    )
  })
```

Here's an example of a response body:

```
{
    "eTag":"\"1\"",
    "id":"a95edb6f-539d-4bac-9ffa-78ff16b20516",
    "createdBy":"jdoe",
    "name":"Technician Notes",
```

```
    "createdOn":"2014-11-20T15:53:05Z",
    "modifiedOn":"2014-11-20T15:53:05Z",
    "modifiedBy":"jdoe",
    "links":[
        {
            "rel":"canonical",
            "href":"/mobile/platform/storage/collections/attachments/
objects/a95edb6f-539d-4bac-9ffa-78ff16b20516"
        },
        {
            "rel":"self",
            "href":"/mobile/platform/storage/collections/attachments/
objects/a95edb6f-539d-4bac-9ffa-78ff16b20516"
        }
    ],
    "contentType":"application/json",
    "contentLength":9377
}
```

storage.storeById(collectionId, objectId, object, options, httpOptions)

This method stores an object based on an ID that you specify. You can use it to add an object using your own ID instead of one that is generated automatically, or to update an existing object.

Arguments

`collectionId`: Required. String. The name of the collection. When you look at the metadata for the collection, this value corresponds to the metadata's `id` value.

`objectId`: Required. String. If you are adding an object, this is the ID to store the object under. If you are updating an object, this is the ID of the object you are replacing.

`object`: Required. Text, JSON object, file, or binary object. This is the object to store.

`options`: Optional. JSON object. This object can have the following properties in addition to those listed in Common options Argument Properties.

Property	Description	Type	Default
`contentLength`	The size of the object in bytes.	Number	If the object is a string or a buffer, then the default is `object.length`. Otherwise, the default is the sum of its members' lengths.
`contentType`	The media type of object being stored. This property also specifies the media type to return when the object is requested.	String	If the `inType` is `json`, then the `Content-Type` header is set to `application/json` automatically. Otherwise, the default is`application/octet-stream`.

Property	Description	Type	Default
`encodeURI`	Set to `true` to URI-encode the `collectionId`, `objectId`, `mobileName`, and `user` values. This option can be useful for multibyte values.	Boolean	`false`
`ifMatch`	The call completes successfully only if the ETag of the corresponding object matches one of the values specified in this property. You can use this property to ensure that the operation succeeds only if the object wasn't modified after you last requested it.	String	None
`ifModifiedSince`	Date and time in HTTP-date format. For example, `Mon, 30 Jun 2014 19:43:31 GMT`. The request completes successfully only if the object was modified after the date specified in property.	Date	None
`ifNoneMatch`	The call completes successfully only if the ETag of the corresponding object does not match one of the values specified by this property.	String	None
`ifUnmodifiedSince`	Date and time in HTTP-date format. For example, `Mon, 30 Jun 2014 19:43:31 GMT`. The request completes successfully only if the object wasn't modified after the date specified in this property. You can use this property to ensure that the operation succeeds only if no one modified the object after that time.	Date	None

Property	Description	Type	Default
`mobileName`	The display name for the object. If you don't include the display name, the name is set to the object identifier.	String	None
`user`	This is the ID (not the user name) of a user. This query parameter allows a user with READ_ALL/ READ_WRITE_ALL permission to access another user's isolated space. A user with READ/READ_WRITE permission may access only their own space.	String	If you are storing an object in a shared collection, there is no default. If you storing an object in an isolated collection, and you have READ_ALL/ READ_WRITE_ALL permission, then the signed-in user is assumed unless you include this property. If you have READ_ALL/ READ_WRITE_ALL permission for an isolated collection, you must include this property to store objects in another user's space.

Response

The response body is a JSON object that follows the schema shown for the response body for the `PUT /mobile/platform/storage/collections/{collection}/objects/{object}` operation in REST APIs for Oracle Mobile Cloud Service.

Examples

In this example, the request can contain JSON objects, files, plain text, images, and so forth. If the input is a JSON object then it must set `inType` to `json`, and pass in `req.body` for the object. Otherwise, it sets `inType` to `stream`, and passes in `req` for the object.

```
service.put('/mobile/custom/incidentreport/attachments/:id',
function (req, res) {
  if (req.is('json')) {
    // Must specify JSON because there is no stream to pipe from req
    // as Express has read it into json and put it in req.body.
    req.oracleMobile.storage.storeById('attachments', req.params.id,
req.body,
      {
        contentLength: req.body.length,
        mobileName: 'Technician Notes',
        inType: 'json',
        outType: 'stream'
      })
      .on('error', function (error) {
```

```
        res.status(error.statusCode).send(error.message)
      })
      .pipe(res);
  } else {
    // For streaming, send req instead of req.body
    req.oracleMobile.storage.storeById('attachments', req.params.id, req, {
      mobileName: 'Technician Notes',
      contentType: req.header('content-type'),
      inType: 'stream',
      outType: 'stream'
    })
      .on('error', function (error) {
        res.status(error.statusCode).send(error.message)
      })
      .pipe(res);
  }
});
```

Here's an example of a response body:

```
{
    "eTag":"\"2\"",
    "id":"incident412-notes",
    "createdBy":"jdoe",
    "name":"Technician Notes",
    "createdOn":"2014-11-20T15:57:04Z",
    "modifiedOn":"2014-11-20T15:58:09Z",
    "modifiedBy":"jdoe",
    "links":[
        {
            "rel":"canonical",
            "href":"/mobile/platform/storage/collections/attachments/
incident412-notes"},
        {
            "rel":"self",
            "href":"/mobile/platform/storage/collections/attachments/
incident412-notes"}
    ],
    "contentType":"application/json",
    "contentLength":9377
}
```

Accessing the Mobile Users API from Custom Code

The Mobile Users and Mobile Users Extended Operations APIs let you get information about the current mobile, virtual, or social user. In addition, the Mobile Users API lets you update a mobile user's custom properties. Custom properties are properties that a mobile cloud administrator has added to the user's realm.

This API has the following methods:

- ums.getUser(options, httpOptions): Retrieves information about the current user.
- ums.getUserExtended(options, httpOptions): Retrieves information about the current user. In addition, for mobile and virtual users, retrieves the user's roles.

- ums.updateUser(fields, options, httpOptions): Updates the current mobile user's custom properties.

ums.getUser(options, httpOptions)

This method lets you retrieve the information about the current user.

- When the user is a mobile user, this operation retrieves the user name, first name, last name, and email address as well as the custom properties that were added to the realm that the user belongs to.
- When the user is a virtual user, this operation retrieves the user name. To learn about virtual users, see Configuring SAML Tokens for Virtual Users.
- When the user is a social user (that is, signed in using social identity), this operation retrieves the user's ID, identity provider, and access token. To learn about social users and social identity, see Facebook Login in MCS.

Arguments

`options`: Optional. JSON object. For mobile users, this object can have the following property in addition to those listed in Common options Argument Properties:

Property	Description	Type	Default
`fields`	Specifies which user properties to get. For example, you can set `options.fields` to `firstName,lastName` to retrieve just those two values. This property is ignored if the current user signed in using virtual or social identity.	String	None

Response

If the current user is a social user, then the response body includes the generated `username` as well as the `mobileExtended.identityProvider` properties, as shown in this example. To learn more about social identity see Facebook Login in MCS.

```
"username": "1 :623:165",
"mobileExtended": {
    "identityProvider": {
        "facebook": {
            "accessToken":"CAAI...YZD"
        }
    }
}
```

If the current user is a virtual user, then the response body includes the `username`, as shown in this example.

```
"username": "a24x"
```

In all other cases, the response body is a JSON object that contains one or more of the following properties, depending on the value of the request's `options.fields` property.

- `id`
- `email`
- `firstName`
- `lastName`
- `username`
- Custom properties that have been added to the realm that the user belongs to.

Examples

Here's an example of calling this method to get the user's first and last name. In this example, the user is a mobile user:

```
req.oracleMobile.ums.getUser({fields: 'firstName,lastName'}).then(
  function(result){
    res.send(result.statusCode, result.result);
  },
  function(error){
    res.send(error.statusCode, error.error);
  }
);
```

This example shows the response that you get when you set the `options.fields` property to `firstname,lastname`:

```
{
  "firstName": "Joe",
  "lastName": "Doe"
}
```

Here's an example of calling this method to get all the fields for a mobile user:

```
req.oracleMobile.ums.getUser().then(
  function(result){
    res.send(result.statusCode, result.result);
  },
  function(error){
    res.send(error.statusCode, error.error);
  }
);
```

Here's an example of a response body for this request:

```
{
  "id":"295e450a-63f0-41fa-be43-cd2dbcb21598",
  "username":"joe",
  "email":"joe@example.com",
  "firstName":"Joe",
```

```
  "lastName":"Doe",
  "links":[
    {
      "rel":"canonical",
      "href":"/mobile/platform/users/joe"
    },
    {
      "rel": "self",
      "href": "/mobile/platform/users/joe"
    }
  ]
}
```

ums.getUserExtended(options, httpOptions)

This method lets you retrieve the information about the authorized user, including the user's roles.

- When the user is a mobile user, this operation retrieves the user name, first name, last name, roles, and email address as well as the custom properties that were added to the realm that the user belongs to.
- When the user is a virtual user, this operation retrieves the user name and roles. To learn about virtual users, see Configuring SAML Tokens for Virtual Users.
- When the user is a social user (that is, signed in using social identity), this operation retrieves the user's ID, identity provider, and access token. To learn about social users and social identity, see Facebook Login in MCS.

Arguments

`options`: Optional. JSON object. For mobile users, this object can have the following property in addition to those listed in Common options Argument Properties:

Property	Description	Type	Default
`fields`	Specifies which user properties to get. For example, you can set `options.fields` to `firstName,lastNam e` to retrieve just those two values. This property is ignored if the current user signed in using virtual or social identity.	String	None

Response

If the current user is a social user, then the response body includes the generated `username` as well as the `mobileExtended.identityProvider` properties, as shown in this example.

```
    "username": "1 :623:165",
    "mobileExtended": {
        "identityProvider": {
```

```
            "facebook": {
                "accessToken":"CAAI...YZD"
            }
        }
    }
```

If the current user is a virtual user, then the response body includes the `username` and `roles` properties, as shown in this example.

```
    "username": "a24x",
    "roles": [
      "Customer", "Trial"
    ]
```

In all other cases, the response body is a JSON object that contains one or more of the following properties, depending on the value of the request's `options.fields` property.

- `id`
- `email`
- `firstName`
- `lastName`
- `username`
- `roles` (array)
- Custom properties that have been added to the realm that the user belongs to.

The response body also contains links to the API endpoint for the resource.

Examples

Here's an example of calling this method to get a mobile user's first and last name:

```
req.oracleMobile.ums.getUserExtended({fields: 'firstName,lastName'}).then(
  function(result){
    res.send(result.statusCode, result.result);
  },
  function(error){
    res.send(error.statusCode, error.error);
  }
);
```

This example shows the response that you get when you set the `options.fields` property to `firstname,lastname`:

```
{
  "firstName": "Joe",
  "lastName": "Doe"
}
```

Here's an example of calling this method to get all the fields for a mobile user.

```
req.oracleMobile.ums.getUserExtended().then(
  function(result){
    res.send(result.statusCode, result.result);
  },
  function(error){
    res.send(error.statusCode, error.error);
  }
);
```

Here's an example of a response body for this request:

```
{
  "id":"295e450a-63f0-41fa-be43-cd2dbcb21598",
  "username":"joe",
  "email":"joe@example.com",
  "firstName":"Joe",
  "lastName":"Doe",
  "roles": [
    "Customer", "Trial"
  ],
  "links":[
    {
      "rel":"canonical",
      "href":"/mobile/platform/users/joe"
    },
    {
      "rel": "self",
      "href": "/mobile/platform/users/joe"
    }
  ]
}
```

ums.updateUser(fields, options, httpOptions)

This method lets you update the custom properties that were added to the realm that the mobile backend is associated with. Note that you can't use this API to update the built-in properties, such as `username`.

Arguments

`fields`: Required. A JSON object that contains name/value for custom fields to be updated. For example:`{birthdate: '07/21/71', language: 'en'}`. Only primitive data types are supported.

`options`: Optional. A JSON object as described in Common options Argument Properties.

Response

The response body is a JSON object where the name/value pairs represent the user properties. It also contains links to the API endpoint for the resource.

Examples

Here's an example of calling this method to update the custom property `key`. In this example, the request body would look like this: `{key: 'Ax47Y'}`.

```
service.put(
  '/mobile/custom/incidentreport/key',
  function (req, res) {
    req.oracleMobile.ums.updateUser({key: req.body.key}).then(
      function (result) {
        res.send(result.statusCode, result.result);
      },
      function (error) {
        res.send(error.statusCode, error.error);
      }
    );
  });
```

Here's an example of a response body:

```
{
  "id":"295e450a-63f0-41fa-be43-cd2dbcb21598",
  "username":"joe",
  "email":"joe@example.com",
  "firstName":"Joe",
  "lastName":"Doe",
  "key":"Ax47Y",
  "links":[
    {
      "rel":"canonical",
      "href":"/mobile/platform/users/joe"
    },
    {
      "rel": "self",
      "href": "/mobile/platform/users/joe"
    }
  ]
}
```

Calling Custom APIs from Custom Code

The custom code SDK provides two namespaces for sending requests to other custom APIs:

- `oracleMobile.custom.<apiName>`: To use the methods in this namespace, you must explicitly declare in `package.json` a dependency on the custom API.
- `oracleMobile.custom`: To use the methods in this namespace, you don't need to explicitly declare in `package.json` a dependency on the custom API.

There are several reasons for declaring the dependency in `package.json`, such as making it easier to track dependencies, and ensuring that dependent APIs are published when you publish your API. To learn how to declare a dependency in

`package.json` and the advantages for doing so, see Specifying the API Version in Calls to Custom and Connector APIs.

The optional `options` argument can have this property in addition to those listed in Common options Argument Properties.

Property	Description	Type	Default
`versionToInvoke`	The version of the custom API. When you use the `oracleMobile.custom` namespace, you must include this option if the API version is not declared in `package.json`. When you use the `oracleMobile.custom.<apiName>` namespace, the API version must be declared in `package.json`, and you optionally can use this property to override that version.	String	The version that is declared in the `package.json` file.

Both namespaces provide methods for each HTTP operation, as shown in this table:

HTTP Operation	oracleMobile.custom Method	oracleMobile.custom.<apiName> Method
`GET`	`get(apiName, resourceName, options, httpOptions)`	`get(resourceName, options, httpOptions)`
`PUT`	`put(apiName, resourceName, object, options, httpOptions)`	`put(resourceName, object, options, httpOptions)`
`POST`	`post(apiName, resourceName, object, options, httpOptions)`	`post(resourceName, object, options, httpOptions)`
`DELETE`	`del(apiName, resourceName, options, httpOptions)`	`del(resourceName, options, httpOptions)`
`HEAD`	`head(apiName, resourceName, options, httpOptions)`	`head(resourceName, options, httpOptions)`
`OPTIONS`	`options(apiName, resourceName, options, httpOptions)`	`options(resourceName, options, httpOptions)`

HTTP Operation	oracleMobile.custom Method	oracleMobile.custom.<apiName> Method
PATCH	`patch(apiName, resourceName, object, options, httpOptions)`	`patch(resourceName, object, options, httpOptions)`

Here are examples of how to call another custom API from custom code using both namespaces . These examples call the `motd` custom API, and send a POST request to its `years/{year}/months/{month}/days` resource.

```
/**
 *  oracle.Mobile.custom.<apiName> namespace example:
 *
 *  <namespace>.post(<resource>, <body>, <options>)
 *
 *  Note: Because it uses the
 *  oracleMobile.custom.<apiName> namespace,
 *  the dependency on the motd API must
 *  be specified in package.json.
 *  options.versionToInvoke isn't required. You can use
 *  it to override the version that is declared in
 *  package.json.
 */
req.oracleMobile.custom.motd.post(
  'years/2018/months/1/days',
  req.body,
  {inType: 'json'}).then(
  function (result) {
    res.status(result.statusCode).send(result.result);
  },
  function (error) {
    res.status(error.statusCode).send(error.error);
  }
);


/**
 *  oracle.Mobile.custom namespace example:
 *
 *  post(<namespace>, <resource>, <body>, <options>)
 *
 *  You must include the versionToInvoke option if
 *  the API isn't declared in package.json.
 */
req.oracleMobile.custom.post(
  'motd',
  'years/2018/months/1/days',
  req.body,
  {versionToInvoke: '1.0', inType: 'json'}).then(
  function (result) {
    res.status(result.statusCode).send(result.result);
  },
```

```
    function (error) {
      res.status(error.statusCode).send(error.error);
    }
  );
```

Calling Connector APIs from Custom Code

To use a connector, you must create a custom API and implement code that calls the SDK's connector methods. Here's information about how to call a connector from custom code.

> **Tip:**
>
> If your connector is a REST API that you created using a valid descriptor, then you can create the custom API and its implementation automatically, as described in How Do I Generate a Custom API from a Connector. If you use the automatic-generation feature, you typically don't need to know how to use the SDK's connector methods described here unless you are using the customizer method that is in the generated code. For example, you might need to use a customizer to pass `options.externalAuthorization` . Sometimes, you might need to replace a call to the `callConnector` method with your own code, such as when you need to send multipart form data or the `http` options object.

The custom code SDK provides two namespaces for sending requests to connectors:

- `oracleMobile.connectors.<connector>`: To use the methods in this namespace, you must explicitly declare in `package.json` a dependency on the connector. The automatically generated implementations use this namespace.
- `oracleMobile.connectors`: To use the methods in this namespace, you don't need to explicitly declare in `package.json` a dependency on the connector.

There are several reasons for declaring the dependency in `package.json`, such as making it easier to track dependencies, and ensuring that dependent APIs are published when you publish your API. To learn how to declare a dependency in `package.json` and the advantages for doing so, see Specifying the API Version in Calls to Custom and Connector APIs.

The optional `options` argument can have these properties in addition to those listed in Common options Argument Properties.

Property	Description	Type	Default
`externalAuthorization`	If you haven't configured a security policy for the connector, then put the `Authorization` value for the external service in the `options.externalAuthorization` property. When this property is present, the connector sets the outgoing `Authorization` header with the value in `options.externalAuthorization` property before it sends the request to the external service.	String	None
`versionToInvoke`	The version of the connector. When you use the `oracleMobile.connectors` namespace, you must include this option if the API version is not declared in `package.json`. When you use the `oracleMobile.connectors.<connector>` namespace, the API version must be declared in `package.json`, and you optionally can use this property to override that version.	String	The version that is declared in the `package.json` file. When you use the `oracleMobile.connectors.<connector>` namespace, the API version must be declared in `package.json`.

Both namespaces provide methods for each HTTP operation, as shown in this table:

HTTP Method	oracleMobile.connectors Signature	oracleMobile.connectors.<connector> Signature
`GET`	`get(connector, resourceName, options, httpOptions)`	`get(resourceName, options, httpOptions)`
`PUT`	`put(connector, resourceName, object, options, httpOptions)`	`put(resourceName, object, options, httpOptions)`
`POST`	`post(connector, resourceName, object, options, httpOptions)`	`post(resourceName, object, options, httpOptions)`
`DELETE`	`del(connector, resourceName, options, httpOptions)`	`del(resourceName, options, httpOptions)`
`HEAD`	`head(connector, resourceName, options, httpOptions)`	`head(resourceName, options, httpOptions)`
`OPTIONS`	`options(connector, resourceName, options, httpOptions)`	`options(resourceName, options, httpOptions)`

HTTP Method	oracleMobile.connectors Signature	oracleMobile.connectors.<connector> Signature
PATCH	patch(connector, resourceName, object, options, httpOptions)	patch(resourceName, object, options, httpOptions)

Note:

You use the Network_HttpPatch environment policy to control the behavior of PATCH requests.

- HEADER sends a POST request with an X-HTTP-Method-Override header set to PATCH. This enables you to send PATCH requests when the target server doesn't support the PATCH method.
- LEGACY sends a PATCH request with an X-HTTP-Method-Override header set to PATCH. This is consistent with the behavior of environments that were provisioned before 18.2.3.
- METHOD sends a PATCH request without an X-HTTP-Method-Override header set to PATCH.

For environments that were provisioned before 18.2.3, the default is LEGACY. For environments that were provisioned on or after 18.2.3, the default is METHOD. Here's an example of using a policy setting to change the policy for MyRESTConnector:

```
*.connector/MyRESTConnector(1.0).Network_HttpPatch=HEADER
```

To learn about viewing and changing environment policies, see Modifying an Environment Policy.

Here's an example of calling the /mobile/connector/globalweather connector using the oracleMobile.connectors namespace:

```
req.oracleMobile.connectors.post('globalweather', 'GetWeather', body,
{inType: 'json', versionToInvoke: '1.0'}).then(
  function (result) {
    console.info("result is: " + result.statusCode);
    res.status(result.statusCode).send(result.result);
  },
  function (error) {
    console.info("error is: " + error.statusCode);
    res.status(error.statusCode).send(error.error);
  }
);
```

Here's an example of calling the `/mobile/connector/globalweather` connector using the `oracleMobile.connectors.<connector>` namespace.

```
req.oracleMobile.connectors.globalweather.post('GetWeather', body,
{inType: 'json'}).then(
  function (result) {
    res.status(result.statusCode).send(result.result);
  },
  function (error) {
    res.status(error.statusCode).send(error.error);
  }
);
```

Calling a Connector to a REST Web Service

You need the connector name and the resource name to call a REST API connector. You form the resource name by removing the base URI from the endpoint. Say, for example, that your `git` connector maps to `https://example.com`. To call `https://example.com/{owner}/{repo}/contents/{path}`, set the `resourceName` to `{owner}/{repo}/contents/{path}`.

You also need to pass the authorization in either `options.externalAuthorization` or `httpOptions.headers['oracle-mobile-external-authorization']` .

Here's an example of sending a PUT request to a REST connector:

```
service.put('/mobile/custom/incidentreport/connectors/git/:owner/:repo/
contents/:path',
  function (req, res) {
    req.oracleMobile.connectors.idmsamples.put(
      'repos/' + req.params.owner + '/' + req.params.repo + '/contents/' +
req.params.path,
      req.body,
      {externalAuthorization: req.header('external-authorization'),
inType: 'json'},
      null).then(
        function (result) {
          // include the target service's response headers
          res.set(result.headers);
          res.status(result.statusCode).send(result.result);
        },
        function (error) {
          res.status(error.statusCode).send(error.error);
        }
    );
});
```

You use the `httpOptions` object to pass headers and query parameters to a connector.

> **Note:**
>
> A connector to a REST web service can have rules that set default query parameters. If you specify values for those same parameters, then your values take precedence and override the default parameters in the connector rules.

Here's an example of passing query parameters and headers in the `httpOptions` object:

```
service.get('/mobile/custom/incidentreport/connectors/git/:owner/:repo/
contents/:path',
  function (req, res) {
    req.oracleMobile.connectors.idmsamples.get(
      'repos/' + req.params.owner + '/' + req.params.repo + '/contents/' +
req.params.path,
      {externalAuthorization: req.header('external-authorization')},
      {qs: {"branch": req.query.branch}, headers: {"accept":
req.header('accept')}}
       ).then(
        function (result) {
          res.status(result.statusCode).send(result.result);
        },
        function (error) {
          res.status(error.statusCode).send(error.error);
        }
      );
});
```

> **Tip:**
>
> When you use `httpOptions.qs` to pass the query string, you can use `encodeURIComponent(<string>)` for the `qs` value to ensure that your code handles multibyte characters.

To learn how to create a connector to a REST service, see REST Connector APIs.

Calling a Connector to a SOAP Service

The body for a message that you send to a SOAP connector must be in either the XML or JSON form of a SOAP envelope, with an optional `Header`, a required `Body`, and an optional `Fault`.

JSON requests are translated automatically to XML, and XML responses are translated to JSON. This means that you can interact with SOAP services without having to work with XML. See How Does XML Get Translated into JSON? for conditions that you should be aware of when the translation occurs.

If you choose to provide the message in XML, then remember to do the following:

- To request that the response body is in XML format, set `options.accept` to `application/xml`.
- When the request body is in XML format, set `options.contentType` to `application/xml; charset=utf-8`.
- The XML in a request body must be wrapped in a SOAP envelope, which must include any necessary SOAP headers, as shown in this example. If you configured a security policy on a connector that requires a SOAP header to be sent in the message, That header is added automatically so you don't need to include it in your message.

```
<?xml version="1.0" ?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemad.xmlsoap.org/soap/
envelope">

  <SOAP-ENV:Header>
    <!-- Add any SOAP headers here -->
  </SOAP-ENV>

  <SOAP-ENV:Body>
    <!-- Add the Body element here -->
  </SOAP-ENV:Body>

</SOAP-ENV:Envelope>
```

To see a sample message for a connector's operation, go to the Test page for the connector, select the operation, and then click **Examples**.

Note that with SOAP connectors, if your `options.contentType` property doesn't specify the character set, then UTF-8 is assumed.

Here's an example of calling a connector to a SOAP service. In this example, the request body is in JSON format:

```
service.get('/mobile/custom/incidentreport/connectors/
numberConvert/:number/words',
function (req, res) {
  var body = {
    Header: null,
    Body: {
       "NumberToWords": {
          "ubiNum": req.params.number
      }
    }
  };
  req.oracleMobile.connectors.post('numberConvert', 'words', body,
  {inType: 'json', versionToInvoke: '1.0'}).then(
    function (result) {
      res.status(result.statusCode).send(result.result);
    },
    function (error) {
      res.status(error.statusCode).send(error.error);
    }
  );
```

```
});
```

To learn how to create a connector to a SOAP service, see SOAP Connector APIs.

Calling Connectors that Require Form Data

If a connector's operation requires a content type of `multipart/form-data`, use Multer to pass the form data to the connector. Multer is a library for Node.js that handles multipart form data.

To call a connector with a request body of type `multipart/form-data`:

1. Add `multer` as a dependency in `package.json`, as shown in the following example, and then run `npm install`.

   ```
   {
     "name": "sendformdata",
     "version": "1.0.0",
     "description": "Sends form data to a connector API.",
     "main": "sendformdata.js",
     "dependencies": {
       "multer": "latest"
     },
     ...
   }
   ```

2. In the custom code, add the following statements:

   ```
   var multer = require('multer');
   var storage = multer.memoryStorage();
   var upload = multer({storage: storage});
   ```

   Multer adds the following objects to the incoming request body when it is of type `multipart/form-data`:

   - `body`: Contains the text fields that are in the form.
   - `files`: Contains the files that are uploaded using the form.

3. In the method for the operation, pass `upload.array` as the second argument and provide the name of the form's file parameter and the maximum number of uploaded files. For example:

   ```
   service.post('/mobile/custom/SendFormData/upload',
   upload.array("avatar", 12), function (req, res)
   ```

4. Extract the content from the `body` and `files` objects and pass it to the connector via the `httpOptions.formData` object. Note that you must make the file object look like a stream.

Here's an example. In this example, the `POST /mobile/custom/SendFormData/upload` operation requires the following form parameters:

- `username`, which is of type `text`.

- avatar, which is of type `file`.

```
var multer = require('multer');
var storage = multer.memoryStorage();
var upload = multer({storage: storage});

module.exports = function (service) {

   service.post('/mobile/custom/SendFormData/upload',
upload.array("avatar", 12), function (req, res) {

    // Because the uploaded file is a buffer in memory, you must modify it
    // to look like a stream before you send it to the connector.
    var uploadedFile = {
      value: req.files[0].buffer,
      options: {
        filename: req.files[0].originalname,
        contentType: req.files[0].mimetype
      }
    };

    var formData = {
      username: req.body.username,
      avatar: uploadedFile
    };

    // FormData is the name of the connector.
    // The formData object is passed in the httpOptions argument.
    // The options.contentType is set to multipart/form-data automatically.
    req.oracleMobile.connectors.FormData.post("upload", null, null, {
      formData: formData
    }).then(
      function (result) {
        res.status(result.statusCode).send(result.result);
      },
      function (error) {
        res.status(error.statusCode).send(error.error);
      });
  });
};
```

For information about Multer, see https://www.npmjs.com/package/multer.

Passing Headers to the Target Service

With the exception of the following headers, you must use `httpOptions.headers` to pass headers and their values:

- `Authorization`: If the connector doesn't have a connector `Authorization` header rule, or if you don't want to use the rule's default value, then you must pass the authorization information in either the `options.externalAuthorization` property or the `httpOptions.headers['oracle-mobile-external-authorization']` property, as shown here. See Security and REST Connector APIs.
- `Connection`: Don't set this header.

- `Content-Length`: Don't set this header.
- `Host`: Don't set this header.
- `User-Agent`: Don't set this header.

> **Note:**
>
> The original request's `Accept` value isn't passed to the target service. To pass the value to the target service, use either the `httpOptions.headers.accept` property or the `options.accept` property.

The headers that you pass in your request override any related default values that are set by connector rules.

Here's an example that passes headers to the target service:

```
  var httpOptions={'headers':{}};
  // You must pass the Accept header if you don't want to use the target
server's default.
  if (req.header('accept')) {
    // You can pass the accept value using options.accept or
httpOptions.header, as shown here:
    httpOptions.headers.accept = req.header('accept');
  };
  // If the connector doesn't have an Authorization rule,
  // or if you don't want to use the rule's default,
  // pass the authorization information using
options.externalAuthorization or
  // httpOptions.headers.oracle-mobile-external-authorization.
  // Note the ['']syntax to prevent the hyphen from being interpreted as a
minus.
  if (req.header('external-authorization')) {
      httpOptions.headers['oracle-mobile-external-authorization'] =
        req.header('external-authorization');
  };
  // Pass any custom headers
  if (req.header('if-none-match')) {
     httpOptions.headers['if-none-match'] = req.header('if-none-match');
  };
  req.oracleMobile.connectors.git.get('repos/fixItFast/incidentreport/
contents/README.md',
    null,
    httpOptions).then(
    function (result) {
      // include the target service's headers
      res.set(result.headers);
      res.status(result.statusCode).send(result.result);
    },
    function (error) {
      res.status(error.statusCode).send(error.error);
    }
  );
```

Overriding SSL Settings for Connectors

You might encounter issues with external services, such as the service has an invalid SSL certificate or it redirects the request but it doesn't preserve the cookies over the redirect.

To resolve these issues, you use the `options` argument to customize the outgoing HTTP requests, which go through a proxy. You can get the proxy from `req.oracleMobile.proxy.httpProxy`. Here's an example of how to override the strictSSL setting in order to ignore SSL validation issues.

```
var res = {};
var options = {
  uri: req.body.externalURI,
  strictSSL: false,
  proxy: 'http://' + req.oracleMobile.proxy.httpProxy
}
req(options).pipe(res);
```

To learn more about request options, see https://github.com/request/request#requestoptions-callback.

Specifying the API Version in Calls to Custom and Connector APIs

When you call connector APIs or other custom APIs, you must always specify the API version. You can specify the API version in one of the following ways:

- Explicitly state the version dependency in the implementation's `package.json` file, as shown here. You must do this if you are using methods in the `oracleMobile.connectors.<connector>` or `oracleMobile.custom.<apiName>` namespace.

  ```
  {
    "name" : "incidentreports",
    "version" : "1.0.0",
    "description" : "FixItFast Incident Reports API",
    "main" : "incidentreports.js",
    "oracleMobile" : {
      "dependencies" : {
        "apis" : {"/mobile/custom/motd" : "1.0"},
        "connectors" : {"/mobile/connector/geocoder": "1.0"}
      }
    }
  }
  ```

  In this example, a call to any method in the `oracleMobile.custom.motd` namespace uses version 1.0 by default.

  For more information, see package.json Contents.

- Include the `options.versionToInvoke` property in the request and set it to the version that you want to use (represented as a string). If you specify the version number this way, then it overrides what you may have specified in the `package.json` file.

```
req.oracleMobile.custom.post(
    'motd',
    'years/2018/months/1/days',
    req.body,
    {versionToInvoke: '1.0', inType: 'json'}).then(
    function (result) {
      res.status(result.statusCode).send(result.result);
    },
    function (error) {
      res.status(error.statusCode).send(error.error);
    }
  );
```

> **Note:**
>
> If you are using a method from the generic `oracleMobile.rest` namespace, then put the version in the `Oracle-Mobile-API-Version` header instead of the `options.versionToInvoke` property.

When you declare dependencies using the `package.json` file, then it's easier to keep track of those dependencies than when you use the `options.versionToInvoke` property to declare dependencies. When you use `package.json` for this purpose, the API Designer displays the dependencies in a table below the list of implementations. When you prepare to publish your API, you're prompted to publish any unpublished dependent APIs.

However, if you use the `options.versionToInvoke` property to declare the version of a dependent API, the API Designer won't be aware of that dependency and won't prompt you with information when you publish the calling API. In this case, you'll need to remember to publish the dependent API yourself.

Using Generic REST Methods to Access APIs

Earlier versions of the custom code SDK used `oracleMobile.rest` methods to access custom, platform, and connector APIs. To ensure backwards compatibility, these methods continue to be available.

The legacy methods take two options: `optionsList`, which you use to pass request details, and `handler`, which is an optional function to be executed by the method. If you don't include the `handler` argument, then the method returns a promise. A promise represents the result of an asynchronous request. At the time it is issued, the request may or may not have completed. You typically use a promise with the `then` function.

If the `handler` function makes calls to other custom, platform, or connector APIs, then you must follow Request.js conventions as described at https://github.com/request/request.

This API has legacy and asynchronous methods for each HTTP operation, as shown in the next table. The difference between the legacy and asynchronous methods is that asynchronous methods don't have a `handler` argument. They always return a promise.

HTTP Operation	oracleMobile.rest Methods
GET	`get(optionsList, handler)` `getAsync(optionsList)`
PUT	`put(optionsList, handler)` `putAsync(optionsList)`
POST	`post(optionsList, handler)` `postAsync(optionsList)`
DELETE	`del(optionsList, handler)` `delAsync(optionsList)`
HEAD	`head(optionsList, handler)` `headAsync(optionsList)`
OPTIONS	`options(optionsList, handler)` `optionsAsync(optionsList)`
PATCH	`patch(optionsList, handler)` `patchAsync(optionsList)`

Here's an example of using an `oracleMobile.rest` method to access the Database Service API. Notice how it uses `optionsList` to pass in the URI and query string, and to convert the request body to JSON.

```
// The request body looks like this
// {title:'Water heater is leaking', technician:'jwhite',customer:'Lynn
Smith'}
service.post('/mobile/custom/incidentreport/incidents',
function (req, res) {

  var optionsList = {
    uri: '/mobile/platform/database/objects/FIF_Incidents',
    qs: req.query,
    json: req.body,
    headers: {
      'Oracle-Mobile-Extra-Fields': 'createdBy,createdOn'
    }
  };

  req.oracleMobile.rest.post(optionsList, function (error, response, body)
{
    var message = error ? error.message : body;
    res.status(response.statusCode).send(message);
  });
});
```

optionsList Argument

You use the `optionsList` argument to pass request details in `oracleMobile.rest` calls, such as the URI, the body, and the headers. Here are some examples of the options that you can configure:

body
This option contains the body for a `patch`, `post`, or `put` request. The value must be a `Buffer` or a `String` unless `OptionsList.json` is set to `true`. If `OptionsList.json` is `true`, then the body must be a JSON-serializable object. See also the `json` option in this list.

headers
This option contains a list of HTTP headers. For example:

```
optionsList.headers=
{Content-Type : 'application/json;charset=UTF-8'};
```

> **Note:**
>
> When you use the `json` option, you do not need to provide the Content-Type header. For all other cases, when the request has a body, include this header and specify the charset.

json
This option can be used in two ways:

- To hold a JavaScript object. In this case, when the request is sent, the object is converted to JSON and put in the HTTP body, and the `Content-Type: application/json` header is added automatically.
- To indicate, by setting the value to `true`, that the `optionsList.body` value is a JavaScript object. In this case, when the request is sent, the `optionsList.body` value is converted to JSON and put in the HTTP body, and the `Content-Type: application/json;charset=UTF-8` header is added automatically.

timeout
This option specifies the number of milliseconds to wait for a request to respond before terminating the request. If you don't provide this option, then the timeout value defaults to the time out that's specified by the `Network_HttpRequestTimeout` environment policy.
The value shouldn't be greater than the `Network_HttpRequestTimeout` environment policy for the environment that the implementation is deployed to. Ask your cloud administrator for the value of this policy setting.
If the target URI is a connector, then the value should be greater than the `Network_HttpConnectTimeout` and `Network_HttpReadTimeout` policies for the connector. These values are displayed on the connector's configuration page.

uri
This required option contains the URL fragment that uniquely identifies the API to call. For example:

```
/mobile/platform/storage/collections/coll1/objects
```

In addition to the options listed here, you can provide any of the options that are specified by the Request.js API. Go to the API documentation at `https://github.com/mikeal/request` and scroll down to the section entitled "`request(options, callback)`".

Learning About Platform, Custom, and Connector APIs

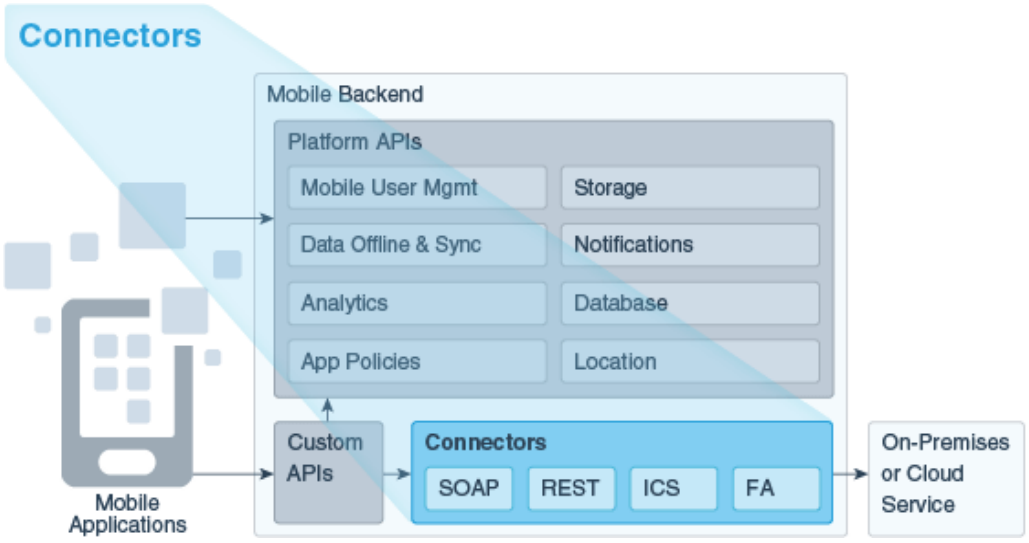
You can use the API catalog to learn about the platform, custom, and connector APIs.

To access the API catalog, click  to open the side menu and then select **APIs**.

- To see the endpoints for a platform API, scroll to the bottom of the API Catalog, and then select the API.
- To see the endpoints for a custom API or connector API, open a custom API, click **Implementations**, and then click **Custom Code API Catalog**. From the **Show** list, select **Connector APIs** or select **Mobile APIs** depending on the API type, and then select the API to view its endpoints.

In addition to the API Catalog, REST APIs for Oracle Mobile Cloud Service provides information about the platform APIs. For example, it provides cURL examples as well as details about request and response bodies and headers.

Part V
Connector APIs



This part contains the following chapters:

- REST Connector APIs
- SOAP Connector APIs
- ICS Connector APIs
- Fusion Applications Connector APIs

24

REST Connector APIs

Oracle Mobile Cloud Service (MCS) enables you to create connector APIs to connect to external REST services. You can then call these connector APIs from the implementations of your custom APIs.

How REST Connector APIs Work

A REST connector API is an intermediary API for calling REST endpoints in enterprise systems or third-part APIs. The connector API takes the form of a configuration that gives your apps a standard way to connect to these REST services and take advantage of the security, diagnostics, and other features provided by MCS.

The connector communicates and passes information between the client and the server using the HTTPS protocol. The information passed can be in the form of XML or JSON (but only in JSON for services based on Swagger descriptors).

The REST Connector API wizard walks you through creating REST Connector APIs, from specifying a remote service and setting security policies to testing your endpoints.

REST Connector API Design Process

Here's the process for designing a REST connector API:

1. **Create REST Connector API**. You create an API with the REST Connector API wizard in MCS.
2. **Authenticate Access to Descriptor Instances with Design Time Credentials**. The design time credentials are saved as a Credentials Store Framework (CSF) key in MCS.

3. **Connect to Descriptor Instance**. If you provided a Swagger 2.0 descriptor URL, a connection to the descriptor is made.

   If you provide a remote service URL, you connect to the external service.

4. **Discover Resources**. After the credentials are authenticated, MCS downloads and parses the descriptor and retrieves the metadata.

   > **Note:**
   >
   > Currently only Swagger in JSON format is supported. Swagger metadata in YAML format can't be parsed.

5. **Set Rules**. Optionally, you can set rules after you have selected the desired resources or after you have connected to the external service.
6. **Examine and Set Resources**. If a descriptor has been provided, the resources are displayed and the desired resources to access from the custom code are selected.

   No resources are displayed if a remote service URL was provided.

7. **Set Security Policies**. You configure the Oracle Web Services Manager (Oracle WSM) security policy to be used at runtime.
8. **Test REST Connector API**. You test the endpoint using mobile user credentials.

Why Use Connectors Instead of Direct Calls to External Resources?

Using a REST Connector API provides you with the following benefits over making direct calls from your app code to external resources:

- Allows for simplified declarative connection and policy configuration.
- With a Swagger descriptor, determines the available resources and creates endpoints for you.
- Provides you with extensive diagnostic information as its tightly integrated with the MCS diagnostics framework. Any outbound REST calls made through connector APIs are logged, which greatly helps with debugging.
- Allows for tracking and analytics on remote API usage.
- Lets you define interaction with the service at design time when you test the validity of your endpoints so that the terms of that interaction aren't dependent on user input at runtime. This protects both the end system and your mobile backend from harm.
- Provides a consistent design approach among multiple connector types for interacting with external services.
- With any change in the interface for a service, lets you can handle any necessary updates, testing, and migration in one place.

Creating a REST Connector API

Use the REST Connector API wizard to create, configure, and test your connector API.

To get a basic working connector API, you can provide as little as a name for the connector API and a URL to the external service.

From there, you can:

- Define rules to form specific requests or responses for the data that you want to access.
- Configure client-side security policies for the service that you're accessing.
- Test the connection and test the results of calls made to the connection.

You must create a custom API and implementation to enable your apps to call the connector APIs. To generate the API and implementation automatically, see Generating Custom APIs for Connectors. If you want to do this manually, create a custom API with the appropriate resources, and then implement the custom code as described at Calling Connector APIs from Custom Code.

Basic Connector Setup

You can create a functioning connector by completing the first two pages in the REST Connector API wizard.

1. Make sure that you're in the environment for which you want to create the REST Connector API.
2. Click  and select **Applications > APIs** from the side menu.
3. Click **REST** (if this is the first connector API to be created) or **New Connector** and from the drop-down list, select **REST**.
4. Identify your new REST Connector API by providing the following:
   a. **API Display Name:** The name as it will appear in the list of connector APIs.
   b. **API Name:** The unique name for your connector API.

      By default, this name is appended to the relative base URI as the resource name for the connector API. You can see the base URI below the API Name field.

      Other than a new version of this connector API, no other connector API can have the same resource name.
   c. **Short Description:** This description will be displayed on the Connectors page when this API is selected.
5. Click **Create**.
6. In the General page of the REST Connector API dialog, set the timeout values:
   - **HTTP Read Timeout:** The maximum time (in milliseconds) that can be spent on waiting to read the data. If you don't provide a value, the default value of 20 seconds is applied.
   - **HTTP Connection Timeout:** The time (in milliseconds) spent connecting to the remote URL. A value of 0mms means an infinite timeout is permitted.

The HTTP timeout values must be less than the `Network_HttpRequestTimeout` environment policy, which has a default value of 40,000 ms. To learn more about environment policies, see Environment Policies.

> **Note:**
>
> If you have a mobile cloud administrator role in addition to your service developer role, you can open the `policies.properties` file to see the value for the network policies for the current environment from the Administrator view. Otherwise, ask your mobile cloud administrator for the values.

7. Click **Descriptor** and enter the connection info for the service.

   If you provide a Swagger descriptor URL, the available resources are identified and displayed, and you can select which ones you want.

   > **Note:**
   >
   > Only standard internet access ports 80 and 443 are supported. Connection to a service can't be made using a custom port.

8. Click **Save**.
9. Optionally, click **Test**, select authentication credentials, and make test calls to the service.

From there, you can further configure the connector in the following ways:

- (If you have provided a descriptor on the Descriptor page) navigate to the Resources page and select the methods for the exposed resources.
- Define rules.
- Set security policies.

To be sure your connector API configuration is valid, you should test it thoroughly (not just from the Connector API Test page) before publishing it. That is, you should also test the custom API (with its implementation) that uses this connector API. See Testing and Debugging Custom Code. Essentially, if you're ready to publish the connector API, you should also be ready to publish the custom API that calls it.

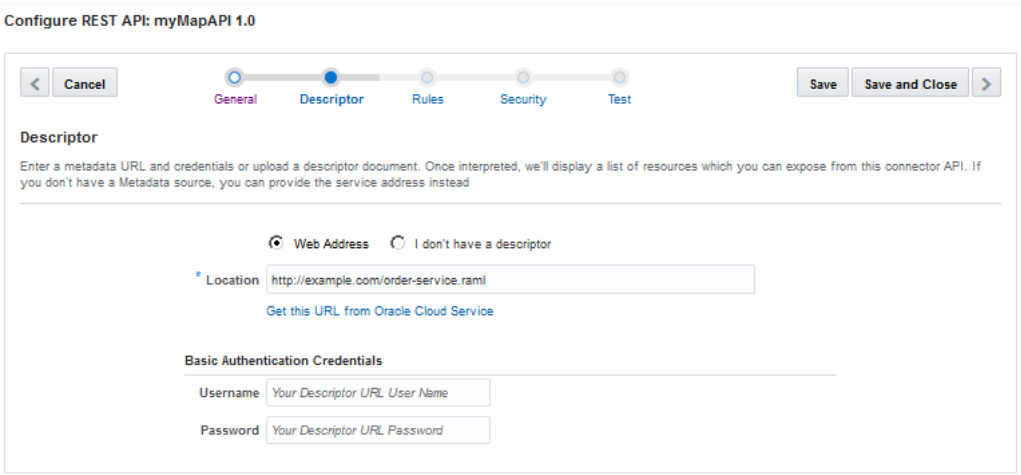
If you've already published the connector API and then find that you need to change it, you must create a new version of it. See Creating a New Version of a Connector.

Providing the Descriptor

If you provide a Swagger descriptor URL, the REST Connector API wizard can examine the descriptive metadata and obtain resources and fields from it.

> **Note:**
>
> Only Swagger metadata in JSON format is currently supported.

If you don't have a descriptor, simply select that option and enter the remote URL of the external service.



1. On the Descriptor page, select the means by which the REST Connector wizard reads the Swagger metadata:

   - **Web Address.** Select this option to enter the URL of the Swagger metadata.

     Click **Oracle Cloud Service REST API Catalog** to get a descriptor from the Oracle Cloud Service REST API catalog. Copy the address for the descriptor you want to the clipboard and paste it into the Location field.

   - **I don't have a descriptor.** Select this option to enter the URL of the external REST service.

     The remote URL is the address of the resource for the external service that this connector API calls.

     You can save time by verifying that the URL you're providing is trusted at trustedsource.org, otherwise, even if your connector API is configured correctly, the connection will fail. See Common Custom Code Errors.

     The HTTPS protocol is used most often, but you can use HTTP if the web service is on a nonsecure site.

     > **Note:**
     >
     > When specifying a port, only standard internet access ports 80 and 443 are supported. Connection to a service can't be made using a custom port.

     You can optionally enter query parameters in the URL, for example:

     ```
     https://maps.googleapis.com/maps/api/directions
     https://maps.googleapis.com/maps/api/directions/json
     https://maps.googleapis.com/maps/api/directions/location?
     origin=Pasadena
     ```

Typically, you set parameters in rules instead of in the Remote URL field, but both ways are possible. See Setting Query Parameters in Remote URLs. To learn more about setting rules, see Rules.

2. If your provided a descriptor URL, enter your basic authentication credentials (user name and password) to access the descriptor if you selected the Web Address option. Then you can proceed to the Resources page.

   If you provided a remote URL, your next step is to set rules (optional) or select a security policy.

   > **Note:**
   >
   > If need to edit your configuration, the descriptor URL and design time credentials you provided are preserved. However, if you provide a different descriptor URL, you will need to enter the credentials to access that descriptor instance.

3. Click **Next** to proceed.

Rules

You set rules to define the interactions between your mobile app and a service. Rules provide a way for you to add default parameter values for all calls to resources on the service, calls to a specific proxy path, and calls for certain types of operations (verbs). This helps enforce consistent syntax of the URL string, saves the custom code developer from having to insert these values, and makes it possible to track the different calls through analytics.

You can create one or more rules. Each rule can have one or more parameters of type `Query` and `Header`.

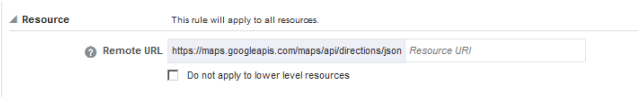
If no rules are applied, all calls are passed through the proxy to the existing service.

1. (If the connector is not already open) click  and select **Applications > APIs** from the side menu.
2. Select the connector API that you want to edit and click **Open**.
3. Select **Roles**.
4. Click **New Rule**.
5. Click **Add Parameter** and select a **Query** or **Header** parameter type and enter the query or header name, and its value.

> **Note:**
>
> Although you can define rules to set certain headers by default, the rules aren't applied if the client that called the connector directly through custom code or indirectly, such as from a web browser or mobile app, has already set the same headers.
>
> In particular, setting the format of the request body is usually done in the custom code with the `Content-Type` header, not as a REST Connector rule. Similarly, setting the format of the response body is also done in the custom code with the `Accept` header, not as a REST Connector rule.

You can add as many parameters to a rule as you want but it's better not to overload a rule with too many operations. A simpler rule construct is easier to troubleshoot.

6. Expand **Resources** and edit the remote URL to provide a resource for the rule to be applied to. The base URL value is what you entered in the setting basic information step and it can't be edited.



7. Select **Do not apply to lower level resources** if you want the rules applied only to the resource level specified in the Remote URL.
8. (Optional) Unselect the HTTP methods that you don't want applied to rules that you just defined. By default, all methods are selected.
9. (Optional) Click **New Rule** to create another rule.

> **Note:**
>
> If you define a rule that conflicts with another rule, the first rule applied takes precedence and the conflicting rule is ignored.

When you're done, click **Save** and then **Next** (**>**) to go to the next step in configuring your connector API.

The description of the rule that you just defined is shown in the Rule banner just above the Default Parameters section. For example, let's say the following values have been provided:

- Remote URL = `https://maps.googleapis.com/maps/api/directions/json?origin=los+angeles&destination=seattle`
- Local URI = `myMapAPI`
- Rule with the following parameter: `Query:key:A3FAEAJ903022`
- `GET` and `PUT` HTTP methods

The rule description would read as follows:

For `GET` to `https://maps.googleapis.com/maps/api/directions/json?origin=los+angeles&destination=seattle` available at `myMapAPI/directions`, Include `Query:key=A3FAEAJ903022`.

If no rules were created, the description would simply read:

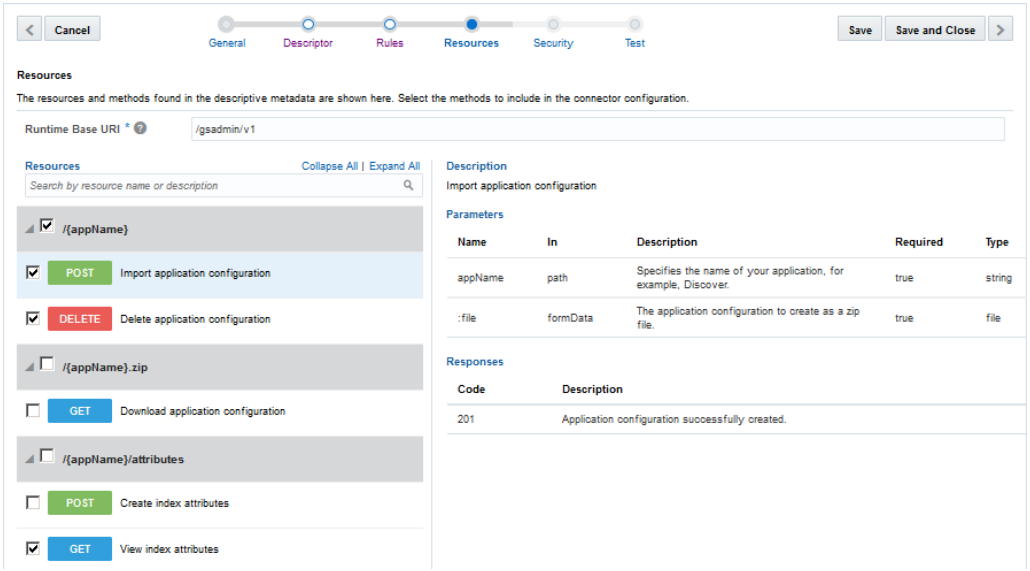
For ALL METHODS to `https://maps.googleapis.com/maps/api/directions` available at `myMapAPI`, No default parameters will be applied.

Now you have a base URI that maps to the existing service. Using our example:

`mobile/connector/myMapAPI/directions/json?origin=los+angeles&destination=seattle` maps to `https://maps.googleapis.com/maps/api/directions/json?origin=los+angeles&destination=seattle`

Selecting Endpoints

If you provided a descriptor, you'll have access to the Resources page. Here's where you'll be able to examine details about the resources that are included in your connector configuration and select the endpoints that you want in your connector configuration.



1. Review the runtime base URI of the external REST service.

   This is the address of the runtime server that you're executing against. It consists of the runtime server port and the base path of the Swagger descriptor. For example, if /documents/api/1.1 is the base path, then the URI would look like:

   `http://server.port/documents/api/1.1`

> **Note:**
>
> This URI is extracted from the metadata if present. If the metadata doesn't contain the address, you must provide it or you won't be able to proceed to the next step in the connector configuration.

2. Select an endpoint to add it to your configuration.

   Resources methods aren't selected by default. If you want a particular endpoint, select it. Un-selecting the resource, un-selects all of its methods.

   When a method is selected, you can view its details, including its associated methods. The Details panel displays the following information for the selected resource:

   - **Description**. The text content of the Swagger description value for the method.
   - **Parameters**. A list of the required and optional parameters to use when calling the method.
   - **Responses**. A list of the available responses that are returned when calling the method.

   Use the **Filter** field to locate a resource based on its name or description.

   Click **Select All** to select all the resources. To start over, click **Clear All**.

   Click **Expand All** to display all the associated methods for every resource.

3. Click **Next** to set the runtime security policy.

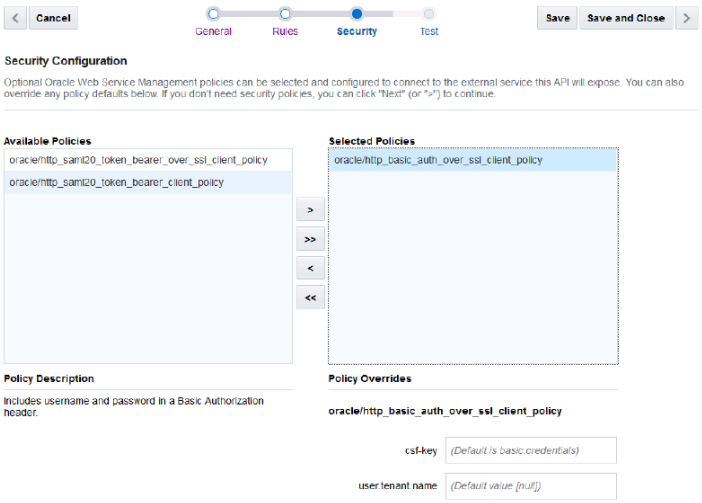
Security Policies and Overriding Properties

Before you finalize your connector API, you should consider how to handle its security. You can use either security policies or authorization headers. Selecting a security policy that describes the authentication scheme of the service to which you're connecting to is the recommended approach.
If you want to use headers, see Security and REST Connector APIs.

Every security policy has properties, called overrides, which you can configure. One reason to override a policy configuration property is to limit the number of policies that you have to maintain: rather than creating multiple policies with slightly varied configurations, you can use the same generic policy and override specific values to meet your requirements.

To select a security policy and set the policy overrides:

1. (If the connector is not already open) click  and select **Applications > APIs** from the side menu.
2. Select the connector API that you want to edit and click **Open**.
3. Select **Security**.



4. Select the security policy from the list of available policies and click the right arrow to move it to the **Selected Policies** list.

   Select only a single policy for your connector API. A description of a selected policy is displayed below the list. To find out more about the supported security policy types for the REST Connector API, see Security Policy Types for REST Connector APIs.

5. Specify overrides, if applicable, to the selected policy if you don't want to use the default values.

   To override a property, enter or select a value other than the default. For a description of policy properties, see Security Policy Properties.
   To set a Credential Store Framework (CSF) Key value, see Setting a CSF Key. To learn about credential keys and certificates, see CSF Keys and Web Service Certificates.

6. Click **Save** to save your work or **Save and Close** to save your work and exit the REST Connector API wizard.

7. Click **Next** (**>**) to go to the next step, testing the connector, Testing in Advanced Mode.

Setting a CSF Key

If you want to authenticate the user, you must set the `csf-key` property. You must set the `csf-key` property if you've selected `http_basic_auth_over_ssl_client_policy`, `http_samle20_token_bearer_client_policy`, or `http_samle20_token_bearer__over_ssl_client_policy`.

> **Note:**
>
> If you set the `csf-key` and the security policy has a `subject.precedence` property, that property should be set to `false`. If you need to set `subject.precedence` to `true`, you must also set the `propagate.identity.context` property. In the latter case, don't set `csf-key`.

Click **Keys** in the **csf-key** field in the Security Overrides section to open the Select or Create a New API Key dialog.

Provide a CSF Key in one of the following ways:

- Select an existing key from the **Available Keys** list.

  When you select the key, its name appears in the **Key Name** field. Click **Select** to add the key. The other fields in the CSF Key Details pane are used only when creating a key.

- Create a new basic (CSF) credentials key.

To create a new CSF key:

1. Click **New Key**.
2. Enter a key name that is descriptive and easy-to-read. Note that after you create the key, you can't change the key name.
3. Enter a brief description of the key's purpose.
4. Enter the user name and password (the user credentials) for the service to which you are connecting.

   Repeat the password in the confirmation field.
5. Click **Save** to add the key to the Available Keys list.

   The key name value will appear as the override value on the Security page.

If you want to edit some aspect of an existing CSF key, select it from the **Available Keys** list and modify the fields as needed. To learn more about CSF keys, see CSF Keys and Web Service Certificates.

Testing the REST Connector API

Now that you've defined your REST Connector API and saved the configuration, you'll want to verify that you're able to actually send a request and receive the expected results from the web service. Testing a connection is also an optional step but can save you time by identifying and fixing problems now before you finalize the connector API. The Test page lets you test one endpoint at a time.

If you provided a descriptor, you have two testing modes to choose from:

- Standard testing

  If you provided descriptor metadata, the standard testing mode is displayed in which the request and response bodies are generated from the descriptive metadata and displayed in the Request and Response tabs. All you have to do is select the parameters to test with for GET methods and include any HTTP headers that you want to test with. See Testing in Standard Mode.

- Advanced testing

  Alternatively, you can refine your testing by selecting **Testing in Advanced Mode** (the test mode you enter if you provided a remote service URL). Without descriptive metadata, you select the method and resource to test, include any HTTP headers you want to include, and manually create the JSON body. See Testing in Advanced Mode.

Testing in Standard Mode

If you provided descriptor metadata, you'll automatically get the standard test page, which lists all the available query parameters for you as well any example bodies or schemas for the request and response payloads (if present in the metadata).
To test your connector with the endpoints exposed for you:

1. Click the **Test** navigation link.
2. Select the endpoint you want to test.

   Endpoints are listed on the left side of the page. Enter a partial resource name in the filter field to narrow the list to make it easier to find the endpoint you want. When you select an endpoint, the method, the resource name, and the URI of service is displayed on right side of the page.
3. Set the default test credentials if you're in the design phase and just want to see if your endpoints are valid, or if you want to test multiple endpoints during the session.

   Otherwise, skip this step and fill out the fields in the Authentication section for each method you test.

   a. Click **Default Test Credentials** at the top of the page.
   b. Select a mobile backend to associate the API with and the version of the mobile backend.
   c. If both OAuth and HTTP Basic Authentication are enabled for the mobile backend, select one in the Authentication Method field to use for testing.
   d. Click **Save** to apply the credentials.
4. Click **Request** and expand **Parameters**.

   When you select a `GET` method, all the available query parameters are displayed on the Request tab.

   a. For a GET method, enter a parameter value.
   b. (Optional) Click **Example** to view the example body, if one was provided. For methods other than GET, enter an alternate example to test with by clicking **Use Example.** The provided example body is copied into the text bod. You edit the example as needed.
   c. (Optional) Click **Schema** to view the request body schema if one was provided.
5. Expand HTTP Headers and click **Add HTTP Header** to add a header.

   Select the header that you want to include for testing purposes and provide a value in the text field.
6. Expand **Authentication**, select the mobile backend and its version that are associated with this API, and enter your mobile user credentials. If both OAuth and HTTP Basic Authentication are enabled for the mobile backend, select one in the Authentication Method field to use for testing.
7. Click **Response**, expand the status code and click **Example** or **Schema** to review the example or schema for the response body, if one was provided.
8. Click **Test Endpoint**.

Test Endpoint toggles to **Cancel Test** when you click it. If you want to stop the test for any reason, then click **Cancel Test**.

If you want to make changes to the testing parameters, click **Reset** to clear all the fields.

To be sure your connector API configuration is valid, test it thoroughly (not just from the Connector API Test page) before publishing it. You should also test the custom API (with its implementation) that uses this connector API. Essentially, if you're ready to publish the connector API, then you should also be ready to publish the custom API that calls it.
If you need to make changes to a connector API that's in the Published state, then create a new version of it. For information on creating a new version, see Creating a New Version of a Connector.

Testing in Advanced Mode

The advanced test page lets you manually set path parameters, add headers, and the request and response payloads.

To manually configure a connector test:

1. Click the **Test** navigation link.
2. If you provided a descriptor, turn **Test in Advanced Mode** to On.

   The advanced test page displays automatically if you provided a remote service url.
3. Select the HTTP method that you want to test from the drop-down list.
4. Specify any resource path parameters in the Local URI field as needed for testing purposes. For example:

   ```
   directions/json?origin=los+angeles&destination=seattle
   ```

   The field is automatically prefixed with the local URI that you defined when you entered an API name. Following our example, the full contents of the field would look like this:

   ```
   myMapAPI /directions/json?origin=los+angeles&destination=seattle
   ```

   Notice that if you defined any rules, the Rules Applied field (below the Body field) displays numbers that correspond to the rules that are applicable for the selected operation. The Remote URL field shows the exact string that will be passed to the service for the test.
5. Add one or more request or response HTTP headers as needed.

   These headers are for testing purposes only and won't be added to your REST Connector API configuration.
6. Click in the **HTTP Body** field to create your message body (the payload) in the source editor.

   For example:

   ```
   {
     "status":"ZERO_RESULTS",
   ```

```
  "routes":[ ]
}
```

Keep the content of the message body relevant to the purpose of the connector, that is, don't bloat the message by adding extraneous data. Including only pertinent data in the message body facilitates quick transmission of the request or response.

7. If the service that you're connecting to requires authentication, open the **Authentication** section and enter your mobile user credentials for each method you test. If you're using default test credentials, you can skip this step.

   With SAML-based security policies, the identity of the user making the call is propagated to the external service. For other security policies such as HTTP Basic Authentication and username token, the credentials used to authenticate with the external service are provided in the policy overrides as CSF keys. Depending on the operation that you've defined, you may have to enter specific credentials for each operation or you might be able to use one set of credentials for all the methods to authenticate your connector with the service.

8. Click **Save as current mobile backend default credentials** to save the user name and password that you provide as the default.

9. If you're in the design phase of creating your connector and you just want to see if your endpoints are valid, click **Default API Designer Test Credentials** and select a mobile backend that you're registered with and its version number.

   Optionally, you can enter your mobile user credentials (user name and password). These default test credentials are persistent across all the methods that you test. They remain valid during the current MCS session.

10. Click **Test Endpoint**.

    Test Endpoint toggles to **Cancel Test** when you click it. If you want to stop the test for any reason, click **Cancel Test**.

    Click **Reset** to clear the fields and modify the test parameters.

11. Click **Done** when you've finished testing your endpoints.

Getting the Test Results

Test results are displayed at the bottom of the Test REST API page. The result indicator is the response status:

- 2*xx*: indicates a successful connection
- 3*xx*: indicates a redirection occurred
- 4*xx*: indicates a user error occurred
- 5*xx*: indicates a server error occurred

Here's a list of the more common status codes that you'll want to use:

Code	Description
200 OK	Successful connection.
201 CREATED	Successful creation through either a PUT or POST operation.

Code	Description
204 NO CONTENT	Successful connection but no response body (used for DELETE and UPDATE operations).
400 BAD REQUEST	General error when fulfilling the request, causing an invalid state, such as missing data or a validation error.
401 UNAUTHORIZED	Error due to missing or invalid authentication token.
403 FORBIDDEN	Error due to user not having authorization or if the resource is unavailable.
404 NOT FOUND	Error due to the resource not being found.
405 METHOD NOT ALLOWED	Error that although the requested URL exists, the HTTP method isn't applicable.
409 CONFLICT	Error due to potential resource conflict caused, for example, by duplicate entries
500 INTERNAL SERVER ERROR	General error when an exception is thrown on the server side.

Click **Request** to see the metadata for the transaction, such as header information and the body of the request.

Click **Response** to see the details of the response returned.

Test each of your operations and modify them as needed to validate your endpoints.

After your connector API is tested, published, and deployed, you can go to the Connectors page to see analytical information about it, such as how often the connector is being called and what apps are using the connector. See Managing a Connector.

Getting Diagnostic Information

You can view the response code and returned data to determine if your endpoints are valid. A response status other than 2*xx* doesn't necessarily mean the test failed. If the operation was supposed to return a null response, a response should show a 4*xx* code.

By examining multiple messages, you can more easily determine where issues occur. For every message that you send, MCS tags it with a correlation ID. A correlation ID associates your request with other logging data. The correlation ID includes an Execution Context ID (ECID) that's unique for each request. With the ECID and the Relationship ID (RID), you can use the log files to correlate messages across Oracle Fusion Middleware components. Click **Logs** on the Administration page to view logging data. You can also retrieve records from Oracle Fusion Middleware Logging using the call's ECID.

Depending on your MCS access permissions, you or your mobile cloud administrator can view the client and server HTTP error codes for your API's endpoints on the Request History page, allowing you to see the context of the message status when you're trying to trace the cause of an error. Every message sent has a set of attributes such as the time the event occurred, the message ID, the Relationship ID (RID), and the Execution Context ID (ECID).

To obtain and understand diagnostic data, see Diagnostics.

Security and REST Connector APIs

MCS gives you the flexibility to configure a secure connection to external services through the use of security policies or authorization headers.

Here are the different ways that you can configure a REST Connector API to communicate with a secured service:

- Configure a security policy.

  On the Security tab of the REST Connector UI, decide which policies describe how the external service that you're communicating with is secured, and configure it as necessary. Configuring a security policy is the recommended practice and takes precedence over setting or configuring authorization headers.

- Set the `Oracle-Mobile-External-Authorization` header on each request.

  If you decide not to configure a security policy, then the next best course of action is to set the `Oracle-Mobile-External-Authorization` header for every request that the connector makes. When calling a connector API through custom code, an MCS-specific authorization header is automatically set as the `Authorization` header. This original `Authorization` header that's set on the connector API request is used to pass only MCS authorization and is never passed through to the external service call. If you set `Oracle-Mobile-External-Authorization` on the request, the value of this header will be set as `Authorization` on the request to the external service. Set an `Oracle-Mobile-External-Authorization` header only when the service that you're connecting to is secured in a way that isn't described by an existing security policy. It won't take effect if one is configured. Passing the `Oracle-Mobile-External-Authorization` header in the connector request takes precedence over an Authorization header rule.

  When setting this header, include `BASIC` to denote HTTP Basic Authorization or `BEARER` to denote OAuth. For OAuth, setting this header is applicable in cases where the OAuth token is passed by way of the Authorization header, such as in the following cases:

  - A REST connector is used to call another Oracle Cloud service. The same access token that was used to authenticate with MCS is reused to authenticate with the other service.
  - An access token generated by a service is passed to an MCS custom code call and set on a REST connector call to obtain the information about the individual who received the access token as part of an enterprise mashup.
  - A person logs on to Facebook and obtains a Facebook access token. The token is passed to an MCS custom code call and set on a REST connector call to retrieve the person's friends list.

- Configure a rule for the `Authorization` header.

  Lastly, when the `Authorization` header isn't already being set by other means, you can create a rule to apply a default `Authorization` header. On the Rules tab of the REST Connector UI, create a rule of type `Header` for `Authorization` and provide a value. This approach isn't recommended as usually the `Authorization` header is dynamic or contains sensitive information (passwords). All sensitive information should be stored in a CSF key, which is why you should configure a security policy when possible.

Security Policy Types for REST Connector APIs

You'll need to set a security policy to protect the information you want to send or receive unless the service you're accessing isn't a secure service or doesn't support security policies, in which case, you can't set a security policy for the connector. When determining what policies to set, consider whether the connection to the service involves transmitting proprietary or sensitive information. Adding a security policy ensures the authentication and authorization of the data transmitted.

From the Security page, you can select one or more Oracle Web Services Manager (Oracle WSM) security policies, including OAuth2, SAML, and HTTP Basic Authentication.

Security Policy Type	Description
OAuth2 and the Client Credential Flow	MCS supports OAuth2, a system where an Authentication server acts as a broker between a resource owner and the client who wants to access that resources. Of the different flows (security protocols) offered by OAuth2, the Client Credentials Grant Flow is used in MCS to secure REST connections. This flow is used when the client owns the resources (that is, the client is the resource owner).
HTTP Basic Authentication	HTTP Basic authentication allows an HTTP user agent to pass a user name and password with a request. It's often used with stateless clients, which pass their credentials on each request. It isn't the strongest form of security though as basic authentication transmits the password as plain text so it should only be used over an encrypted transport layer such as HTTPS.
Security Assertion Markup Language (SAML)	SAML is an XML-based open standard data format that allows the exchange of authentication and authorization credentials among a client, an identity provider, and a service provider. The client makes a request of the service provider. The service provider verifies the identity of the client from the identity provider. The identity provider obtains credentials from the client and passes an authentication token to the client, which the client then passes to the service provider. The identity provider verifies the validity of the token for the service provider and the service provider responds to the client.

Ask yourself the following questions to determine what kinds of security policies you need:

- What are the basic requirements of your security policy? Do you need to only authenticate or authorize users, or do you need both?
- If you need only authentication, do you need a specific type of token and where will the token be inserted?

For a list of the security policies supported for REST Connector APIs, see Security Policies for REST Connector APIs. For descriptions of security policy properties that can be overridden, see Security Policy Properties.

CSF Keys and Web Service Certificates

Depending on the security policy that you selected, you may be able to override a property that sets a CSF key or a Web Service Certificate. In MCS, the Oracle Credential Store Framework (CSF) is used to manage credentials in a secure form. A credential store is a repository of security data (credentials stored as keys) that certify the authority of users and system components. A credential can hold user name and password combinations, tickets, or public key certificates. This data is used during authentication and authorization.

CSF lets you store, retrieve, update, and delete credentials (security data) for a web service and other apps. A CSF key is a credentials key. It uses simple authentication (composed of the user name and the password for the system to which you're connecting) to generate a unique key value. You can select an existing CSF key or create one through the Select or Create a New API Key dialog. To select or create a CSF key, see Setting a CSF Key.

A Web Service Certificate allows the app to securely communicate with the web service. It can be a trusted certificate (that is, a certificate containing only a public key) or a certificate that contains both public and private key information. You override a certificate key by selecting an alias from the drop-down list. The certificate key available in some security policies for a REST Connector API is the `keystore.sig.csf.key`, which is the alias for this property that's mapped to the alias of the key used for signing.

> **Important:**
>
> For security policies for REST Connector APIs, don't override the default value for the `keystore.sig.csf.key` property. Currently, `orakey` is the only valid value for all certificate keys.

Not all security policies contain the same properties. When you select a policy, you can see which properties are listed in the Policy Overrides. For example, if you selected `http_basic_auth_over_ssl_client_policy`, then you'll see that the policy contains the `csf-key` property but none of the certificate keys. However, if you selected `http_saml20_token_bearer_over_ssl_client_policy`, then you'll see both the `csf-key` and the `keystore.sig.csf.key` certificate key.

> **Note:**
>
> It isn't necessary to set all the overrides for a policy; however, you should be familiar enough with the security policies that you've selected to know which overrides to set for each policy.

CSF keys, certificates, and their respective values are specific to the environment in which they're defined. That is, if there are multiple environments, A and B, and you're working in environment A, then only the CSF keys and certificates for the security

policies in use by artifacts in that environment are listed in the CSF Keys dialog. A different set of keys and certificates will be displayed in environment B. It's also possible for keys with the same key name but with different values to exist in multiple environments.

A CSF key can be deployed to another environment, however, because CSF keys are unique to an environment, only the key name and description are carried over to the target environment. You won't be able to use that key in the new environment until it's been updated with user name and password credentials by the mobile cloud administrator.

Query and Header Parameters

A `Query` parameter is the most common type of parameter. Use it to filter, sort, and search for information. Add a question mark (?) to the end of the URL followed by a name-value pair. For example:

```
/directions/distance?origin=Los+Angeles&destination=Seattle
```

The query specifies that the information wanted is the distance from one location (`origin=Los+Angeles`) to another (`destination=Seattle`).

You can see in the example above that the space in the query parameter, Los Angeles, is encoded by a plus sign, (+). The `Url_PercentEncodeQueryParameterSpaces` policy determines how spaces in query parameters are encoded. If set to `true`, a space is encoded as a percent sign, (%). If set to `false` (the default value), a space is encoded as a plus sign (+).

For example, if `Url_PercentEncodeQueryParameterSpaces` is set to `true` , the outbound URL would be `.../distance?origin=Los%Angeles&destination=Seattle`.

> **Note:**
>
> If you specify a parameter in the custom code and you also specify that same parameter in a REST connector rule, the parameter in the custom code takes precedence and overrides the parameter's value defined in the rule.

`Query` parameters are usually set in rules, however, you can have query parameters in the remote URL. In such cases, there's a precedence order for how the parameters are combined at runtime. See Setting Query Parameters in Remote URLs.

Use a `Header` parameter for outgoing requests. REST headers are a means of providing HTTP metadata. For example, the header, `Expires`, can be used to specify the amount of time after which a response is considered stale.

Setting Query Parameters in Remote URLs

You can add query parameters to the remote URL. If the remote URL contains a query parameter and you're adding query parameters to the runtime resource through rules, then there is a precedence order of how the parameters are combined:

1. If you're adding a remote URL that has a query parameter `U?qp=a` to a runtime resource `/r`, the query parameter should come after the runtime resource.

For example, if you have the remote URL `directions?origin=Pasadena` and want to specify the runtime resource `/zones`, the full URL should be `directions/zones?origin=Pasadena`. Note that a simple concatenation of the URL isn't done.

2. If you're combining a remote URL with a query parameter `U?qp=a` with a default rule `qp=b` , both query parameters should come after the URL.

For example, if you have a remote URL `directions/zones?origin=Pasadena` and you want to add the default rule `destination=Anaheim`, the resulting URL should be `directions/zones?origin=Pasadena&destination=Anaheim`. It's orthogonal to rules.

3. If you're combining a remote URL `U?qp=a` with a runtime request `/r?qp=c`, the request parameter is appended to the URL.

For example, if you add the request `/r?date=2015-04_07T14:30:00.000Z` to the remote URL `directions/zones?origin=Pasadena`, the result is`directions/zones?origin=Pasadena&date=2015-04_07T14:30:00.000Z`.

Adding Parameters

Parameters can be added as part of the URI path as a child (nested) resource or added as a query. There are no hard and fast rules as to whether to add parameters to the URI path or to add the parameters in a query. One possible consideration is whether the parameter is essential to the request. For example, you could use an identifier, `id`, to the `directions` resource in the URI path to get data for a specific area. If you're using the parameter as a filter to narrow down the data, then add it in the query. For example, you could define `office` as a query parameter, `.../directions/zones?office=Inglewood`, to filter locations of offices only in the Inglewood area.

Besides the remote URL, you can set parameters in the following ways:

- Setting a rule
- Defining a request body
- Defining a test endpoint
- Creating custom code

The parameters are considered to be URL-encoded. If a parameter isn't already URL-encoded, it will be encoded when sent to the external service.

Editing a REST Connector API

If you need to change some aspect of a connector API, you can as long as it's in the Draft state. After you publish an API, the API can't be changed. You'll have to create a new version of a published connector and make your changes to the new version.

To edit a REST Connector API:

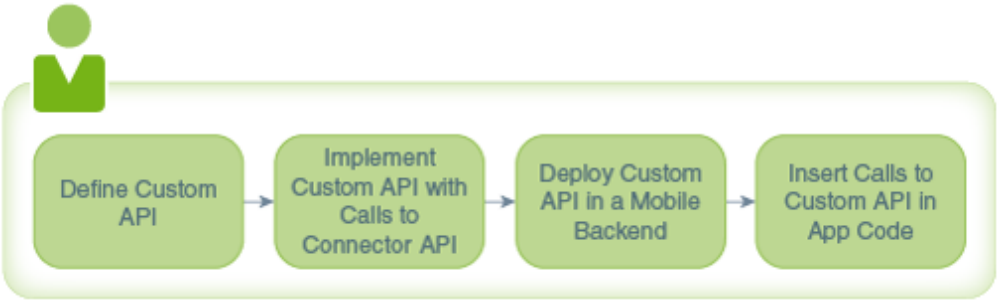
1. Make sure that you're in the environment containing the REST Connector API that you want to edit.
2. Click  and select **Applications > APIs** from the side menu.
3. Select the draft connector API that you want to edit and click **Open**.
4. Click **Refresh** ( ) if you're using the same descriptor and just want to get the latest resources.

5. Click **Save** to test your changes immediately or click **Save and Close** to save your current changes and finish the rest of your changes later.
6. Test your changes.

Using Your Connector API in an App

To use a connector in a mobile app, you need to have a custom API that can call the connector API. Such a custom API could also contain additional logic to process the data returned from the call to the connector.

The syntax for a call to a connector API is the same as you would use when calling any other API from custom API implementation code. See Calling Connector APIs from Custom Code.



When you implement a custom API, you can view the available connectors in the API Catalog tab in the API Designer. While creating your custom API, you might find it beneficial to open the Test page of the connector API so that you can refer to any headers, parameters, and schemas that you've configured for the connector API.

Troubleshooting REST Connector APIs

System message logs are great sources for getting debugging information. Depending on your role, you or your mobile cloud administrator can go to Administration in the side menu and click **Logs** to see any system error messages or click **Request History** to view the client (4*xx*) and server (5*xx*) HTTP error codes for the API's endpoints and the outbound connector calls made within a single mobile backend.

Sometimes a connection fails because the service URL provided is untrusted. You can add the URL to the list of trusted URLs at `trustedsource.org`. To learn more about what happens when you use an untrusted service URL and other common errors that can occur when configuring your connector API, see Common Custom Code Errors.

Issues can also arise when connecting to an external service such as when the service has an invalid SSL certificate or the request is redirected but the cookies aren't preserved over the redirect. You can resolve these issues by using the options argument in custom code to customize the outgoing HTTP requests. See Overriding SSL Settings for Connectors for details.

By default, only TLSv1.1 and TLSv1.2 protocols are used for outbound connections. If you need to use an older version of a SSL protocol to connect to an external system that doesn't support the latest versions of SSL, you can specify the SSL protocol to use for the connector by setting the `Security_TransportSecurityProtocols`

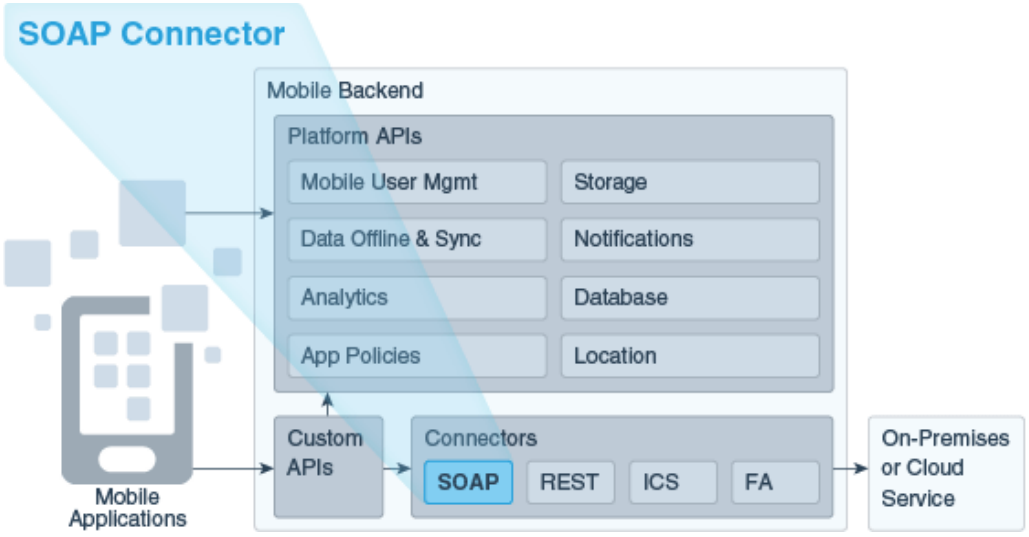
environment policy. The policy takes a comma-separated list of TLS/SSL protocols, for example: TLSv1, TLSv1.1, TLSv1.2. Any extra space around the protocol names is ignored. You can use the SSLv2Hello protocol to debug connectivity issues with legacy systems that don't support any TLS protocol. Note that this policy can't be used to enable SSLv3 endpoints. See Environment Policies and Their Values for a description of the policy and the supported values. Be aware that this policy must be manually added to a `policies.properties` file that you intend to export.

> **Caution:**
>
> Be aware when setting the policy that older protocols are vulnerable to security exploits.

25

SOAP Connector APIs

Oracle Mobile Cloud Service (MCS) enables you to create connector APIs to connect to SOAP services. You can call these connector APIs from the implementations of your custom APIs.



How SOAP Connector APIs Work

A SOAP connector API is an intermediary API for calling SOAP endpoints. The connector API takes the form of a configuration that gives your apps a standard way to connect to these SOAP endpoints and take advantage of the security, diagnostics, and other features provided by MCS.

The key steps to creating a SOAP connector API are establishing a connection to an external system, examining and selecting a set of possible interactions, and then modeling them into a reusable API.

The SOAP Connector API wizard walks you through creating SOAP connector APIs, from specifying the WSDL location of a remote service, setting a port, setting security policies, to testing your endpoints.

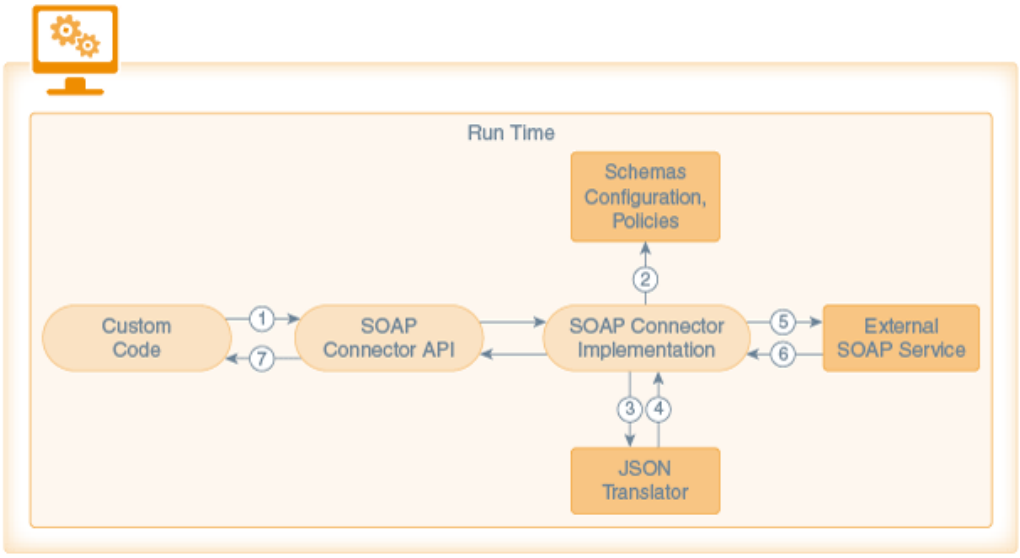
SOAP Connector API Design Process

Here's the process for designing a SOAP connector API:

1. A SOAP Connector API is created in MCS using the SOAP Connector API wizard and is passed to the Asset catalog (the Asset catalog is a repository in MCS where API information is stored). The connector API is added to the list of connector APIs (using the API display name) on the Connectors Manage page on the Development tab.

2. The WSDL location is passed to the WSDL Parser. The WSDL file describes how the service is called, what the expected parameters are, and what data structures are returned. From the data in the WSDL file a sample body is generated.
3. The WSDL Parser goes to the provided WSDL location to obtain the WSDL file.
4. All the available ports for the connector are extracted by the parser and returned to the Asset catalog, after which, the port can be selected and the connector API configurations, such as the endpoint URI and custom operation names, are provided.
5. The Asset Catalog stores the security policies and the request and response schemas.

Here's how the runtime flow goes:



1. Custom code calls the SOAP Connector API. Information is then passed to the connector implementation. The implementation extracts the JSON payload from the request.
2. The schemas, security policies, and API configuration are passed to the Asset catalog.
3. The implementation sends the JSON payload to the JSON translator to translate it to XML using the schemas that are stored as part of the API configuration.
4. The JSON translator returns the payload in XML format.
5. A SOAP message is constructed from the XML, some HTTP headers (like `context-id`) and security-related headers are added and the request is sent to the external service.
6. An XML response is sent by the service back to the connector API. Step 3 and Step 4 are repeated. The response is sent to the JSON translator by the connector implementation to translate the XML response to JSON. The translated response is sent to the connector API.
7. The connector API sends the JSON response back to the custom code.

Why Use SOAP Connectors Instead of Direct Calls to External Resources?

- Allows for simplified declarative connection and policy configuration.
- Allows calls to an external service, along with security policy setup and credentials, to be encapsulated and used consistently across the mobile API.
- Provides automatic translation of JSON requests to XML and XML responses to JSON, enabling you to interact with SOAP services without having to work expressly with XML. In addition, it provides you with the ability to provide the SOAP envelope itself, giving you the choice of using XML or JSON.
- Lets you dynamically modify HTTP timeout properties via the user interface without having to bring down the service. This feature is particularly beneficial when the external SOAP service or network connectivity suffers a slowdown.
- Provides you with extensive diagnostic information as its tightly integrated with the MCS diagnostics framework. Any outbound calls made through connector APIs are logged, which greatly helps with debugging.
- Allows for tracking and analytics on remote API usage.
- Lets you define interaction with the service at design time when you test the validity of your endpoints so that the terms of that interaction aren't dependent on user input at runtime. This protects both the end system and your mobile backend from harm.
- Provides a consistent design approach among multiple connector types for interacting with external services.
- With any change in the interface for a service, lets you can handle any necessary updates, testing, and migration in one place.

Creating a SOAP Connector API

Use the SOAP Connector API wizard to quickly configure your connector API by providing a name and description, specifying a port, setting security policies, and testing it.

Creating a connection to an existing SOAP service can be a simple two-step operation:

1. Name your connector API.
2. Provide the WSDL of the external service.

> **Note:**
>
> A timeout can occur when downloading a large WSDL file or when connecting to a WSDL over high latency networks, which prevents the creation of the SOAP Connector API. To ensure the WSDL is downloaded, set the following environment policies *before* you create the API:
>
> - `*.*.Network_HttpConnectTimeout`
> - `*.*.Network_HttpReadTimeout`
>
> Set these policies in the development environment in which you're creating the SOAP Connector API. A mobile cloud administrator can export the policies file from the Administration view, edit these values, and import the modified file back to the development environment.
>
> These policies affect only the connector APIs during design time. The timeout values that you set while configuring a connector API take effect during runtime.
>
> To edit environment policies, see Modifying an Environment Policy.

You also have the ability to configure client-side security policies for the service that you're accessing and testing and checking the results of your connection.

As soon as it's created, your connector API appears in the list of connector APIs. When at least one connector API exists, you're taken directly to the Connector API landing page when you click **Connectors** from the side menu. From there, you can select the connector API you want and edit it, publish it, create a new version or update an existing version, deploy it if it has a Published state, or move it to the trash. See Connector Lifecycle.

To call a connector API, you can create a custom API and configure the API's implementation to call the connector. See Calling Connector APIs from Custom Code.

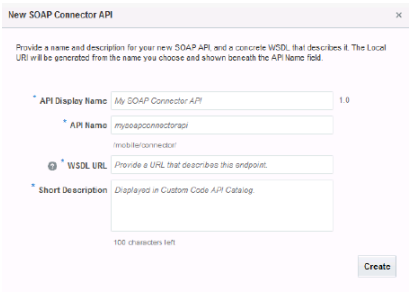
Setting the Basic Information for Your SOAP Connector API

Before you begin configuring your connector, you must provide some initial basic information like the connector API name, the address to the remote service, and a brief description:

1. Make sure that you're in the environment where you want to create the SOAP Connector API.
2. Click  and select**Applications > APIs** from the side menu.

   The Connectors page appears. If no connector APIs have been created yet, you'll see icons for each of the connector APIs that you can create. If at least one connector API exists, you'll see the a list of all the connector APIs. You can filter the list to see only the connector APIs that you're interested in or click **Sort** to reorder the list.
3. Click **SOAP** or **New Connector** and select **SOAP** from the drop-down list.

   Each time you create a SOAP Connector API, the New SOAP Connector API dialog appears. This is where you enter the basic information for your new connector API.



4. Identify your new SOAP Connector API by providing the following:

   a. **API Display Name:** Enter a descriptive name (an API with an easy-to-read name that qualifies the API makes it much simpler to locate in the list of connector APIs).

      For example, `myOrderApi`.

      > **Note:**
      >
      > The names you give to a connector API (the value you enter in the API name field) must be unique among connector APIs.

      For new connectors, a default version of 1.0 is automatically applied when you save the configuration.

   b. **API Name:** Enter a unique name for your connector API.

      For example, `myorderapi`.

      By default, this name is appended to the base URI as the resource name for the connector API. You can see the base URI below the API Name field.

      The connector API name must consist only of lowercase alphanumeric characters. It can't include special characters, wildcards, slashes /, or curly braces {}. A validation error message is displayed if you enter a name that's already in use.

      If you enter a different name for the API here, the change will automatically be made to the resource name in the base URI.

      Other than a new version of this connector API, no other connector API can have the same resource name.

   c. **WSDL Location:** Enter the address of the existing SOAP service that this connector API will call. For example: `http://example.com/incidentreport/reports.wsdl`

      You can also copy and paste a WSDL address into this field. To ensure the WSDL you're using is valid within the scope supported by MCS, see Troubleshooting SOAP Connector APIs.

> **Note:**
>
> When specifying a port in the URL, only standard internet access ports 80 and 443 are supported. Connection to a service can't be made using a custom port.

You can save time by verifying that the URL you're providing is trusted at trustedsource.org, otherwise, even if you're connector API is configured correctly, the connection will fail. See Common Custom Code Errors.

d. **Short Description:** Provide a brief description, including the purpose of this API.

The character count below this field lets you know many characters you can add.

After you've filled in all the required fields, click **Create**, which displays the General page of the SOAP Connector API dialog.

5. Set the timeout values:

- **HTTP Read Timeout:** The maximum time (in milliseconds) that can be spent on waiting to read the data. If you don't provide a value, the default value of 20 seconds is applied.
- **HTTP Connection Timeout:** The time (in milliseconds) spent connecting to the remote URL. A value of 0mms means an infinite timeout is permitted.

  The HTTP timeout values must be less than the `Network_HttpRequestTimeout` environment policy, which has a default value of 40,000 ms. To learn about environment policies, see Environment Policies.

> **Note:**
>
> If you have a mobile cloud administrator role in addition to your service developer role, you can open the `policies.properties` file to see the value for the network policies for the current environment from the Administrator view. Otherwise, ask your mobile cloud administrator for the values.

6. Click **Save** to save your current settings.

   If you want to stop and come back later to finish the configuration, the click **Save and Close**. You can always click **Cancel** at the top of the General, Port, and Security wizard pages to cancel that particular configuration operation. You'll be taken back to the Connector APIs page.

7. Click **Next** (**>**) to go to the next step in configuring your connector API.

After the basic information is provided, you can specify the interaction details for your connector.

You can always edit your configuration when it's in a Draft state; however, after you publish your connector API, no changes can be made to it. You can make changes by creating a new version of an existing connector API. See Creating a New Version of a Connector.

Selecting a Port

The services and their associated ports that are available for the WSDL that you provided are listed on the Port page. A **port** is a set of actions that define the collaboration and interaction with a web service. A **service** defines the operations and structures of the WSDL and exposes those operations as explicit endpoints. Although a WSDL can contain multiple ports, the SOAP Connector API can only use a single port at a time. If you need to expose more than one port, you must create one SOAP Connector API for each port.

On the Port page, you select a single port that lists the available operations for that service. Optionally, you can provide alternate names for those operations to make them more meaningful or easier to read.

1. Click the **Port** navigation link at the top of the SOAP Connector API wizard.
2. Select a port from the service you want in the list.

   You can select only one port. Filter the list by entering a string in the **Filter** field and click the **magnifying glass** .

   The endpoint field is populated with the service and port endpoint (URL) that are extracted from the WSDL. By default, the original operation name of the SOAP service is used to form the REST resource at which the functionality of the operation would be exposed by the SOAP Connector API.

   For example, an operation, `CreateIncident`, of the service, `IncidentReport` and port, `ReportPort`, can be mapped to the REST resource: `/mobile/connector/myIncidentReportAPI/CreateIncident`.

   This is the resource path to which custom code would send requests to. You could expose it differently if you wanted to, for example as the REST resource: `/mobile/connector/myIncidentReportAPI/Create`.

   > **Note:**
   >
   > If you save the connector configuration without explicitly selecting a port, the first available port for the WSDL is selected for you by default. This action ensures your connector configuration is complete and valid for testing purposes. You can always change the port as long as the connector is in Draft state.

3. (Optional) Rename one or more operations to make them more meaningful.

   All the operations available in the selected port are listed.

Each operation is mapped to the relative base URI that you entered. For example: the operation `Create` maps to `Create` resource.

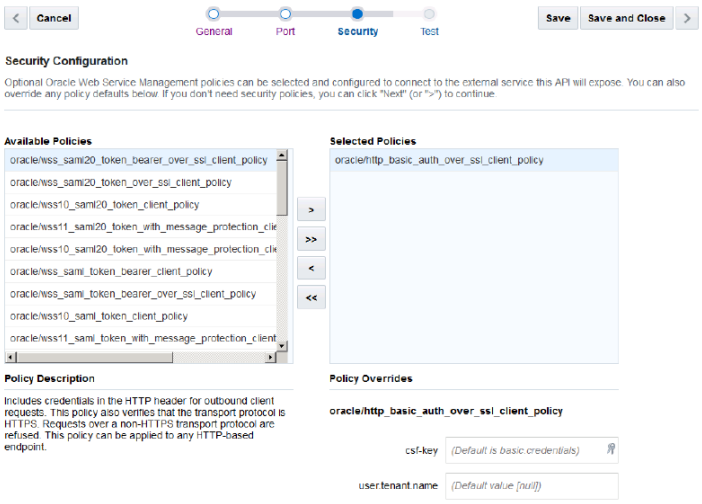
Click **Next** (**>**) to go to the next step in configuring your connector API.

Setting Security Policies and Overriding Properties for SOAP Connector APIs

Select one or more security policies that describe the authentication scheme of the service to which you're connecting. The security policies have properties, called overrides, which you can configure. One reason to override policy configuration properties is to limit the number of policies that you have to maintain: rather than creating multiple policies with slightly varied configurations, you can use the same generic policy and override specific values to meet your requirements.

You don't need to set all the overrides for a policy; however, you should be familiar enough with a security policy to know which overrides to set.

1. Click the **Security** navigation link at the top of the SOAP Connector API wizard.



2. Select one or more security policies from the list of available policies and click the right arrow to move them to the **Selected Policies** list.

   For example, you might want to have `wss10_message_protection_client_policy` for message protection and `wss_username_token_client_policy` for authentication. Although you can move all the policies to the Selected Policies list, it's unlikely that all policies are required for your connector API.
   To learn about supported security policy types for SOAP Connector APIs, see Security Policy Types for SOAP Connector APIs .

3. Select a policy to read its description.

4. Specify any other overrides, if applicable, to the selected policy if you don't want to use the default values.

   To override a policy property, enter or select a value other than the default. For descriptions of policy properties, see Security Policy Properties.
   To set or create a `csf-key` property, see Setting a CSF Key. To learn about credential keys and certificates, see CSF Keys and Web Service Certificates.

5. Click **Save** to save your work or **Save and Close** to save your work and exit the SOAP Connector API wizard.

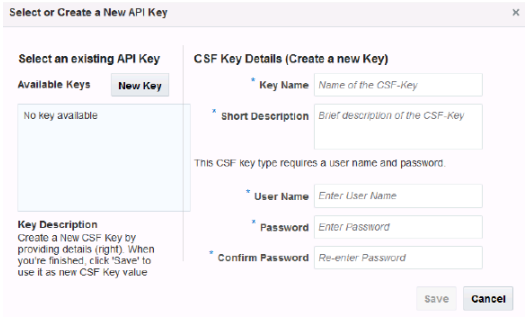
   Before you can test your connection, you must save your configuration. If you proceed to the testing page without saving the API configuration, you'll see a

dialog asking you to save it. You can check the **Always save before testing** option to automatically perform a save operation for you every time you go to the Testing page.

6. Click **Next** (**>**) to go to the next step, testing the connector API.

Setting a CSF Key

Click **Keys** in the **csf-key** field in the Security Overrides section to open the Select or Create a New API Key dialog.



Provide an CSF key in one of the following ways:

- Select an existing key from the **Available Keys** list (a description of the selected key is displayed below the list). The list displays only the basic credentials keys supported by the given policy property.

  When you select the key, its name appears in the **Key Name** field. Click **Select** to add the key. The other fields in the CSF Key Details pane are used only when creating a key.

- Create a new CSF credentials key.

To create a new key:

1. Click **New Key**.
2. Enter a key name that is descriptive and easy-to-read. Note that after you create the key, you can't change the key name.
3. Enter a brief description of the key's purpose.
4. Enter the user name and the password (the user credentials) for the service to which you are connecting. Repeat the password in the confirmation field.
5. Click **Save** to add the key to the Available Keys list. You can create another key by clicking **New Key** or edit an existing one. **Save** toggles to **Select** allowing you to select a key in the list. Click **Cancel** to quit the task.

   The key name value will appear as the override value on the Security page. Note that the value of the key that you create pertains only to the environment in which it's set.

If you want to edit some aspect of an existing credentials (CSF) key, select it from the **Available Keys** list and modify the fields as needed.

Setting a Web Service Certificate

Here the steps for setting the overrides for a Web Service certificate. However, for this release, don't override the values for `keystore.sig.csf.key` and `keystore.enc.csf.key` because `orakey` is the only valid value for all of these certificate keys.

1. Select a security policy.

   The properties for the policy are displayed in the Policy Overrides section.

2. Select an alias from the drop-down list in the field for the certificate key (certificate keys are denoted by the `keystore` prefix) and select an alias.

   Unlike CSF Keys, you can't modify a Web Service certificate. You can only select a different alias. Only mobile cloud administrators can create a new Web Service Certificate. If you don't know the alias for the certificate you want, ask your mobile cloud administrator for the alias.

Testing a SOAP Connector API

Now that you've defined your connector API, you might want to verify your endpoints and ensure that you're able to receive the expected results from the web service. Testing a connection is also an optional step but can save you time by identifying and fixing problems with your endpoints using the mock JSON body provided before you finalize the connector API.

Testing Your Connector

Now its time to validate your connector. The Test page lets you test the connection to a service using sample response data. You'll see a list of all the operations that you defined for the port.

1. Click the **Test** navigation link.
2. Select the operation that you want to test.

   The base URI is displayed below the operation name. If you provided an alternate name for the operation, that name appears, otherwise the default operation name is shown.

3. Click **Examples** to see Request, Response, and Fault payload examples (in JSON format).

   These examples are generated based on the request and response definitions in the WSDL file and can't be edited. The request and response examples display a message body. Fault examples may show one or more faults depending on the operation. They display the error messages returned.

   For example, here is what a sample `GET` request looks like:

```
{
    "Header": null,
    "Body": {
            "GetIncidentById" :  {
                "IncidentId" : 2
            }
```

```
        }
    }
}
```

Here is the request in XML:

```
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:beta="http://xmlns.fixitfast.com/fif/beta">
   <soapenv:Header/>
   <soapenv:Body>
      <beta:GetIncidentById>
         <beta:IncidentId>2</beta:IncidentId>
      </beta:GetIncidentById>
   </soapenv:Body>
</soapenv:Envelope>
```

4. (Optional) Click **Add HTTP Header** to add one or more HTTP headers to apply to the operation.

   You can select a predefined header or a custom header. For each header, select a header name and provide a value.

   These headers are for testing purposes only and won't be added to your SOAP Connector API configuration.

   The default format for the request body and the response body is JSON. You can set the format of one or both to XML if you prefer. See Using XML Instead of JSON.

5. Use the sample JSON body provided to test your connector or create your XML body in the source editor. A JSON sample body that you can edit is generated for you from the operation that you've defined. For example:

```
"Body" : {
    "CreateIncident" :  {
        "Title" : "new title",
        "EmailAddress" : "jack@oracle.com",
        "ImageLink" : "http://example.com/something"
    }
}
```

   For comparison, here's what the body looks like in XML:

```
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:beta="http://xmlns.fixitfast.com/fif/beta">
 <soapenv:Body>
   <beta:CreateIncident>
      <beta:Title>new title</beta:Title>
      <beta:EmailAddress>jack@oracle.com</beta:EmailAddress>
      <beta:ImageLink>something</beta:ImageLink>
   </beta:CreateIncident>
 </soapenv:Body>
</soapenv:Envelope>
```

Click in the editor and enter your own body (in JSON or XML format) if you prefer. To learn about JSON conventions and the mapping between JSON and XML, see How Does XML Get Translated into JSON?

6. If you've selected a SAML-based security policy, open the **Authentication** section and enter your mobile user credentials for each method that you test. If you're using default test credentials (Step 7), you can skip this step.

   With SAML-based security policies, the identity of the user making the call is propagated to the external service. For other security policies such as HTTP Basic Authentication and username token, the credentials used to authenticate with the external service are provided in the policy overrides as CSF keys. Depending on the operation you've defined, you may have to enter specific credentials for each operation or you might be able to use these credentials for all the methods to authenticate your connector with the service.

7. Click **Save as current mobile backend default credentials** to save the user name and password you provide as the default.

8. If you're in the design phase of creating your connector and you just want to see if your endpoints are valid, click **Default API Designer Test Credentials** and select a mobile backend that you're registered with and its version number.

   Optionally, you can enter your mobile user credentials (user name and password). These default test credentials are persistent across all the methods that you test. They remain valid during the current MCS session.

9. Click **Test Endpoint**.

   Test Endpoint toggles to **Cancel Test** when you click it. If you want to stop the test for any reason, click **Cancel Test**.

   Click **Reset** to clear the fields and to change the header types and values and test body.

10. Repeat Steps 1 through 4 for each method.

11. Click **Done** when you've finished testing your endpoints.

    You're returned to the Connector APIs page.

Getting the Test Results

After the test is run, the results are displayed at the bottom of the Test SOAP Connector API page. The result indicator is the response status:

- 2*xx* - indicates a successful connection
- 3*xx* - indicates a redirection occurred
- 4*xx* - indicates a user error occurred
- 500 - indicates an internal server error

Here's a list of the more common status codes that you'll want to use:

Code	Description
`200 OK`	Successful connection.
`401 UNAUTHORIZED`	Error due to missing or invalid authentication token.
`403 FORBIDDEN`	Error due to user not having authorization or if the resource is unavailable.

Code	Description
`500 INTERNAL SERVER ERROR`	General error when an exception is thrown on the server side or when the service returns a SOAP fault response.

Click **Request** to see the metadata for the transaction, such as header information and the body of the request.

Click **Response** to see the details of the response returned. The response code tells you whether the connection was successful.

Test each of your operations and modify them as needed to validate your endpoints. After your connector API is tested, published, and deployed, you can go to the Connectors page to see analytical information about it, such as how often the connector is being called and what apps are using the connector. See Managing a Connector.

Getting Diagnostic Information

You can view the response code and returned data to determine if your endpoints are valid. A response status other than 2*xx* doesn't necessarily mean the test failed. If the operation was supposed to return a null response, a response should show a 4*xx* code.

By examining multiple messages, you can more easily determine where issues occur. For every message that you send, MCS tags it with a correlation ID. A correlation ID associates your request with other logging data. The correlation ID include an Execution Context ID (ECID) that’s unique for each request. With the ECID and the Relationship ID (RID), you can use the log files to correlate messages across Oracle Fusion Middleware components. By examining multiple messages, you can more easily determine where issues occur. For example, you can retrieve records from Oracle Fusion Middleware Logging using the call's ECID. From the Administration page, you can click **Logs** to view logging data: the connector API call received by a single MBE outbound connector API call.

Depending on your MCS access permissions, you or your mobile cloud administrator can view the client and server HTTP error codes for your API's endpoints on the Request History page allowing you to see the context of the message status when you're trying to trace the cause of an error. Every message sent has a set of attributes such as the time the event occurred, the message ID, the Relationship ID (RID), and the Execution Context ID (ECID).

To obtain and understand diagnostic data, see Diagnostics.

SOAP Connector API Design Tips

When you configure your SOAP Connector API, you want to ensure that you have a well-formed API. You want to make a valid SOAP Connector API but you should create an API that can be used and understood by others as well.

Here are some design recommendations to consider when you define a SOAP Connector API:

- Most important, test your connector using the Test page after it’s created and at every update.

- When setting the read and connection timeouts for the connector API, you should set them for a shorter duration than the API timeout. See API Timeouts.
- Provide an HTTPS endpoint wherever possible.
- When calling SOAP services protected with HTTP Basic Authentication, you should configure the appropriate security policies on the Security page and store credentials in a CSF key instead of providing the credentials from custom code.
- While writing custom code to call SOAP Connector APIs, make use of the sample request and response payloads available in the Test page of the SOAP Connector API wizard. See Calling Connector APIs from Custom Code.
- Keep the payload content relevant to the purpose of the connector, that is, don't bloat the payloads by adding extraneous data. Include only pertinent data in the message body to facilitate quick transmission of the request or response.
- When you're working with complex WSDLs, refer to How Does XML Get Translated into JSON? for a discussion of JSON translator limitations.
- Date formats should follow the ISO-8601 International Standard for date and time: `YYYY-MM_DD[THH:mm:ss.sss]Z`. For example: `2014-10-07T18:35:50.123Z` (see Date and Time Formats for a description of the standard).

How Does XML Get Translated into JSON?

The WSDL file, which describes the service that you want to access, is an XML-based protocol. The WSDL contains the XML schemas that define the structure of the SOAP XML requests and responses.

While XML is a standard means of defining SOAP messages, it's cumbersome and not well-suited to data-interchange. JSON is the preferred format because it's a lightweight and easy-to-read and write data interchange format (compared to XML). It's much easier to handle JSON in (Node.js-based) custom code than XML. Here's a comparison of XML and JSON features:

XML	JSON
Human readable	Easier to read and write for developers and machines
Provides a structure to data making it more informative	Same as XML
Easily processed due to simplicity of data structure	Even simpler structure making it even easier to process
Structure of the data must be translated into a document structure	Structure is based on arrays and records

To make the transmission of data via SOAP Connector APIs possible, MCS uses a JSON translator. The JSON translator uses a set of mapping conventions when converting a JSON request into XML prior to passing the information to a remote service and translates the XML response back into JSON to be passed on to the mobile app.

MCS provides sample JSON messages that you can use as a template to construct JSON requests and process JSON responses. A sample payload (body), which gets created for you based on the information in the WSDL, is also translated into JSON.

If you choose to provide your own XML sample payload, then you should adhere to the mapping conventions of XML to JSON to ensure a successful translation. The next section demonstrates those mapping conventions.

XML - JSON Mapping Conventions

Oracle Mobile Cloud Service uses a XML - JSON mapping convention that is based on the Badgerfish convention. The following example shows the mapping of XML elements to JSON object properties:

XML	JSON
`... <Name>John</Name> ...`	`... { "Name" : "John" } ...`

The next example shows how XML attributes are mapped to JSON object properties, with property names starting with the @ symbol:

XML	JSON
`<... archived="true">...</...>`	`... { "@archived" : true } ...`

When elements have attributes defined in the XML schema, text nodes are mapped to an object property with the property name $. This is true even if at runtime the attributes do not occur:

XML	JSON
`... <Name archived="true">John</ Name> ...`	`... { "@archived" : true, "$" : "John" } ...`

Here you can see how nested XML elements become nested JSON objects:

XML	JSON

`...`
`<Address>`
`<City>Bangalore</City>`
`</Address>`
`...`
	`{`
	`...`
	`"Address" : { "City" : "Bangalore" }`
	`...`
	`}`

Here's how XML elements with `maxOccurs > 1` in their schemas (that is, repeating elements) become JSON arrays:

XML	JSON

`...`
`<Name>John</Name>`
`<Name>Susan</Name>`
`...`
	`... { "Name" : [ "John", "Susan" ] } ...`

In the SOAP Connector, the `Envelope` root element is not required in the JSON message body. During the translation to JSON, XML root elements are dropped when

converting to JSON. In the reverse direction, a root element is added when converting JSON to XML. This is done because JSON can have multiple top level object properties which would result in multiple root elements which are not valid in XML:

XML	JSON

<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Header>..</soap:Header>
<soap:Body>..</soap:Body>
</soap:Envelope>
	{
	"Header":{...}
	"Body":{...}
	}

This example shows you how the JSON data types (boolean, string and number) are supported. When converting XML to JSON, based on the type defined in the XML schema, the appropriate JSON type is generated:

XML	JSON

...
<Integer>10</Integer>
<String>string-value</String>
<Boolean>true</Boolean>
...
	{
	...
	"Integer" : 10,
	"String": "string-value",
	"Boolean": true
	...
	}

All namespace information (ns declarations and prefixes) is dropped when converting XML to JSON. On converting the JSON back to XML, the namespace information (obtained from the schema) is added back to the XML:

XML	JSON
<RootElement xmlns="http://xmlns.oracle.com/test"> <Name>John</Name> </RootElement>	{ "Name" : "John" }

If a property in an XML file has an empty value, the same property in the converted JSON file shows an empty string:

XML	JSON

<Customer active="false">
<Name>John</Name>
<Address>
<City/>
<State>AK</State>
</Address>
</Customer>
	{
	"@active":"false",
	"Name":"John",
	"Address":{
	"City":null,
	"State":"AK"
	}
	}

In the reverse scenario, if a JSON file contains a null value, for example `"City":null`, the translation to XML shows an empty value: `<City/>`.

Mapping Limitations

The mapping is comprehensive but isn't quite a one-to-one match. When creating a message body in JSON, there are some conditons that you should be aware of to ensure that the structure of the body is compliant with the JSON-XML mapping convention. The following constructs aren't handled by the JSON translator.

- A choice group with child elements belonging to different namespaces having the same (local) name. This is because JSON doesn't have any namespace information.
- A sequence group with child elements having duplicate local names. For example, `<Parent><ChildA/><ChildB/>...<ChildA/>...</Parent>`. This translates to an object with duplicate property names, which isn't valid.
- XML Schema Instance (`xsi`) attributes aren't supported.

If you want to use a construct that isn't supported by the translator, use XML and be sure to wrap your XML in a SOAP envelope. To learn about JSON, see *Introducing JSON* at http://json.org.

Using XML Instead of JSON

Using JSON isn't required. You might prefer to use XML instead or you might encounter XML schema constructs that aren't supported by the translator. You can still interact with the connector using XML requests and responses.

The response format is determined by the `Accept` header in custom code, which has a default value of `application/json`. To set the format of the request body, add the XML request body and set the `contentType` header in the custom code to `application/xml; charset=utf-8`. If you want the response in XML format, change the `accept` header value to `application/xml`. For example,

```
/**
 *  The following example calls the 'CreateIncident' resource
 *  on a SOAP connector named '/mobile/connector/RightNow'.
 *  The request and response are in XML and not JSON.
 *
 */
var options = {
    contentType: 'appplication/xml;charset=UTF-8',
    accept: 'application/xml'
};

//Here we suppose an XML message has been
//stored in the XML variable
var body = xml;

req.oracleMobile.connectors.RightNow.post('CreateIncident', body,
options).then(
  function(result){
    //result.result  contains  the response XML
    res.status(result.statusCode, result.result);
```

```
  },
  function(error){
    res.status(500, error.error);
  }
);
```

Remember to wrap your XML in a SOAP envelope. Your XML request must contain the entire SOAP envelope (including any SOAP headers):

```
<?xml version="1.0" ?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemad.xmlsoap.org/soap/
envelope">

  <SOAP-ENV:Header>
    <!-- Add any SOAP headers here -->
  </SOAP-ENV>

  <SOAP-ENV:Body>
    <!-- Add the Body element here -->
  </SOAP-ENV:Body>

</SOAP-ENV:Envelope>
```

If you configured a security policy on the connector that requires a SOAP header to be sent in the message, that header is added automatically to the envelope you provide so you don’t need to include it in your message. You can see an example of an XML request wrapped in a SOAP envelope in Testing Your Connector.

Security Policy Types for SOAP Connector APIs

You'll need to set a security policy to protect the information you want to send or receive unless the service you’re accessing isn't a secure service or doesn’t support security policies, in which case, you can’t set a security policy for the connector.

When determining what policies to set, consider whether connection to the service involves transmitting proprietary or sensitive information. A few reasons for adding security policies are:

- Ensuring confidentiality by encrypting messages
- Ensuring the integrity of the data transmitted by using digital signatures
- Authenticating the source or destination

From the Security section, you can select one or more Oracle Web Services Manager (Oracle WSM) security policies, including SAML, Username Token, and HTTP Basic Authentication. Oracle WSM supports a wide range of security standards, including Authentication Policies and Authorization.

Security Policy Type	Description
HTTP Basic Authentication	HTTP Basic authentication allows an HTTP user agent to pass a user name and password with a request. It's often used with stateless clients, which pass their credentials on each request. It isn't the strongest form of security though because basic authentication transmits the password as plain text so it should be used only over an encrypted transport layer such as HTTPS.
Security Assertion Markup Language (SAML)	SAML is an XML-based open standard data format that allows the exchange of authentication and authorization credentials among a client, an identity provider, and a service provider. The client makes a request of the service provider. The service provider verifies the identity of the client from the identity provider. The identity provider obtains credentials from the client and passes an authentication token to the client, which the client then passes to the service provider. The identity provider verifies the validity of the token for the service provider and the service provider responds to the client.
Username Token	A username token is supplied by a web services client as a means of identifying the requestor by using a user name, and optionally by using a password or password-equivalent to the web services provider.

Ask yourself the following questions to determine what kinds of security policies you need:

- What are the basic requirements of your security policy? Do you need to authenticate or authorize users? Do you require only message protection, do you need both?
- If you need only authentication, do you need a specific type of token and where will the token be inserted?
- If you need both authentication and message protection, will message protection be handled in the transport layer?

For a list of supported security policies, see Security Policies for SOAP Connector APIs.

For descriptions of security policy properties that you can override, see Security Policy Properties.

CSF Keys and Web Service Certificates

Depending on the security policy that you selected, you may be able to override a property that sets a CSF key or a Web Service Certificate. In MCS, the Oracle Credential Store Framework (CSF) is used to manage credentials in a secure form. A credential store is a repository of security data (credentials stored as keys) that certify the authority of users and system components. A credential can hold user name and password combinations, tickets, or public key certificates. This data is used during authentication and authorization.

CSF lets you store, retrieve, update, and delete credentials (security data) for a web service and other apps. A CSF key is a credentials key. It uses simple authentication (composed of the user name and the password for the system to which you're connecting) to generate a unique key value. You can select an existing CSF key or

create one through the Select or Create a New API Key dialog. To select or create a CSF key, see Setting a CSF Key.

A Web Service Certificate allows the client to securely communicate with the web service. It can be a trusted certificate (that is, a certificate containing only a public key) or a certificate that contains both public and private key information. Web Service Certificates are stored in the Oracle WSM keystore. You set the overrides by selecting an alias from the drop-down list for the following properties:

- `keystore.recipient.alias`: The alias for this property is used to identify the certificate in the keystore.
- `keystore.sig.csf.key`: The alias for this property is mapped to the alias of the key used for signing. If no value is selected, the default value, `orakey`, is used (for this release, the only valid value for this property is `orakey`).
- `keystore.enc.csf.key`: The alias for this property is mapped to the alias of the private key used for decryption. If no value is selected, the default value, `orakey`, is used (for this release, the only valid value for this property is `orakey`).

Not all security policies contain all three properties. When you select a policy, you can see which properties are listed in the Policy Overrides. For example, if you selected `wss11_username_token_with_message_protection_client_policy`, you'll see that you need to set only `keystore.recipient.alias` . However, if you selected `wss10_username_token_with_message_protection_client_policy`, you'll need to set all three properties.

> **Note:**
>
> It isn't necessary to set all the overrides for a policy; however, you should be familiar enough with the security policies that you've selected to know which overrides to set for each policy.

CSF keys, certificates, and their respective values are specific to the environment in which they're defined. That is, if there are multiple environments, A and B, and you're working in environment A, then only the CSF keys and certificates for the security policies in use by artifacts in that environment are listed in the CSF Keys dialog. A different set of keys and certificates will be displayed in environment B. It is also possible for keys with the same key name but with different values to exist in multiple environments.

A CSF key can be deployed to another environment, however, because CSF keys are unique to an environment, only the key name and description are carried over to the target environment. You won't be able to use that key in the new environment until it's been updated with user name and password credentials by the mobile cloud administrator.

Editing a SOAP Connector API

If you need to change some aspect of a connector API, you can as long as it's in the Draft state. After you publish an API, the API can't be changed.

To edit a SOAP Connector API:

1. Make sure that you're in the environment containing the SOAP Connector API that you want to edit.
2. Click and select**Applications > Connectors** from the side menu.

   Since at least one connector API exists, the Connectors page is displayed.
3. Select the draft SOAP Connector API that you want to edit and click **Open**.

   You can filter the list by version number or status. You can also sort the list alphabetically by name or by last modified date.
4. Edit the fields for general information, ports, and security policies as needed.

   Remember you can always click **Save and Close** to save your current changes and finish the rest of your changes later.
5. Save your changes if you didn't select the option to always save the configuration before testing when you created the API.
6. Test your changes.

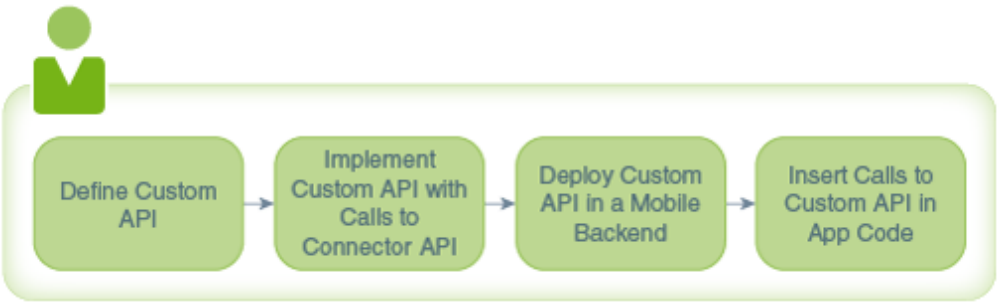
Your edited version is still in a Draft state and you can continue to edit your connector API until you're satisfied with the configuration. At that point, you're ready to publish your connector API. A published connector API can't be changed. If you need to make changes, you can create a new version of the connector API. To create a new version, publish, and deploy your connector API, see Connector Lifecycle.

Your edited version is still in a Draft state and you can continue to edit your connector API until you're satisfied with the configuration. At that point, you're ready to publish your connector API. A published connector API can't be changed. If you need to make changes, you can create a new version of the connector API.

Using Your Connector API in an App

To use a connector in a mobile app, you need to have a custom API that can call the connector API. Such a custom API could also contain additional logic to process the data returned from the call to the connector.

The syntax for a call to a connector API is the same as you would use when calling any other API from custom API implementation code. See Calling Connector APIs from Custom Code.



When you implement a custom API, you can view the available connectors in the API Catalog tab in the API Designer. While creating your custom API, you might find it beneficial to open the Test page of the connector API so that you can refer to any headers, parameters, and schemas that you've configured for the connector API.

Troubleshooting SOAP Connector APIs

System message logs are great sources for getting debugging information. Depending on your role, you or your mobile cloud administrator can go to Administration in the side menu and click **Logs** to see any system error messages or click **Request History** to view the client (4*xx*) and server (5*xx*) HTTP error codes for the API's endpoints and the outbound connector calls made within a single mobile backend.

Sometimes a connection fails because the service URL provided is untrusted. You can add the URL to the list of trusted URLs at `trustedsource.org`. To learn more about what happens if you provide an untrusted URL and other common errors that can occur when configuring your connector API, see Common Custom Code Errors.

By default, only TLSv1.1 and TLSv1.2 protocols are used for outbound connections. If you need to use an older version of a SSL protocol to connect to an external system that doesn't support the latest versions of SSL, you can specify the SSL protocol to use for the connector by setting the `Security_TransportSecurityProtocols` environment policy. The policy takes a comma-separated list of TLS/SSL protocols, for example: TLSv1, TLSv1.1, TLSv1.2. Any extra space around the protocol names is ignored. You can use the SSLv2Hello protocol to debug connectivity issues with legacy systems that don't support any TLS protocol. Note that this policy can't be used to enable SSLv3 endpoints. See Environment Policies and Their Values for a description of the policy and the supported values. Be aware that this policy must be manually added to a `policies.properties` file that you intend to export.

> **Caution:**
>
> Be aware when setting the policy that older protocols are vulnerable to security exploits.

SOAP Connector API Scope

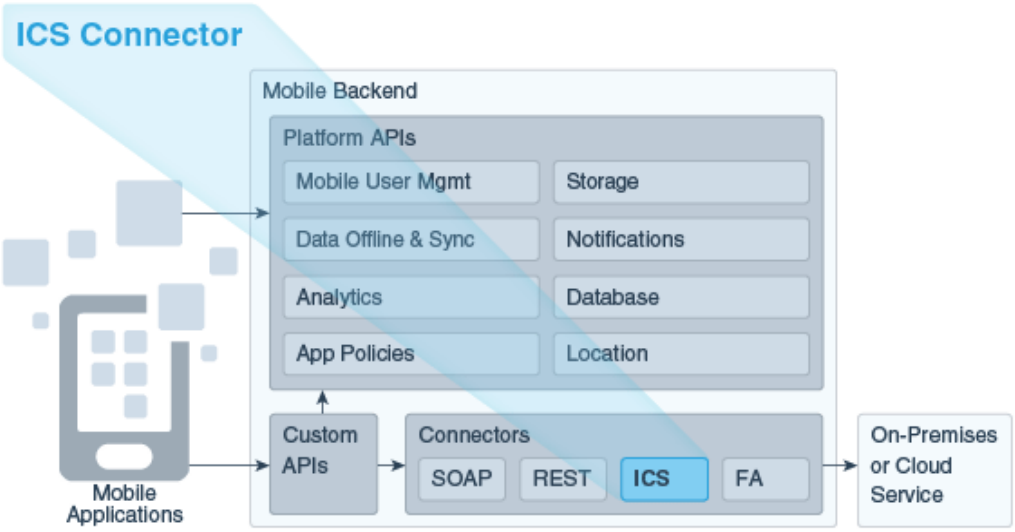
To be sure you're creating a valid SOAP Connector API in MCS, keep in mind the following WSDL constraints:

- Only SOAP version 1.1 and WSDL version 1.2 are supported.
- Only the WS-Security standard is supported. Other WS-* standards, such as WS-RM or WS-AT, aren't supported.
- Only document style and literal encoding are supported.
- Attachments aren't supported.
- Of the possible combinations of input and output message operations, only input-output operations and input-only operations are supported. These operations are described in the Web Services Description Language (WSDL) Version 1.2 specification.

26

ICS Connector APIs

Oracle Mobile Cloud Service (MCS) enables you to create Integration Cloud Service (ICS) connector APIs to access on-premises and cloud services through ICS. You can then call these connector APIs from the implementations of your custom APIs.



You can also use SOAP connector APIs to connect to enterprise services. However, using ICS together with ICS connector APIs has the following advantages:

- You write far less code.
- You connect to services more because the integrations are done for you.
- You let the connector API handle the details of interacting with Oracle Integration Cloud Service.

ICS also makes it easy to map business objects from one application to another. For example, a service can be created that synchronizes data from a purchase order between Oracle Sales Cloud to an Oracle CPQ (Configure, Price, and Quote) Cloud application.

How ICS Connector APIs Work

ICS connector APIS enable you to access services that you have exposed in Integration Cloud Service (ICS).

ICS itself is a service designed to simplify connectivity between your services and applications, both cloud-based and on premises. When you work with ICS, you work with *integrations* that connect applications and map data between them.

You create an ICS connector API with the ICS Connector wizard, in which you enter the SOAP proxy for the integration. Once you have done so, you are shown a list of

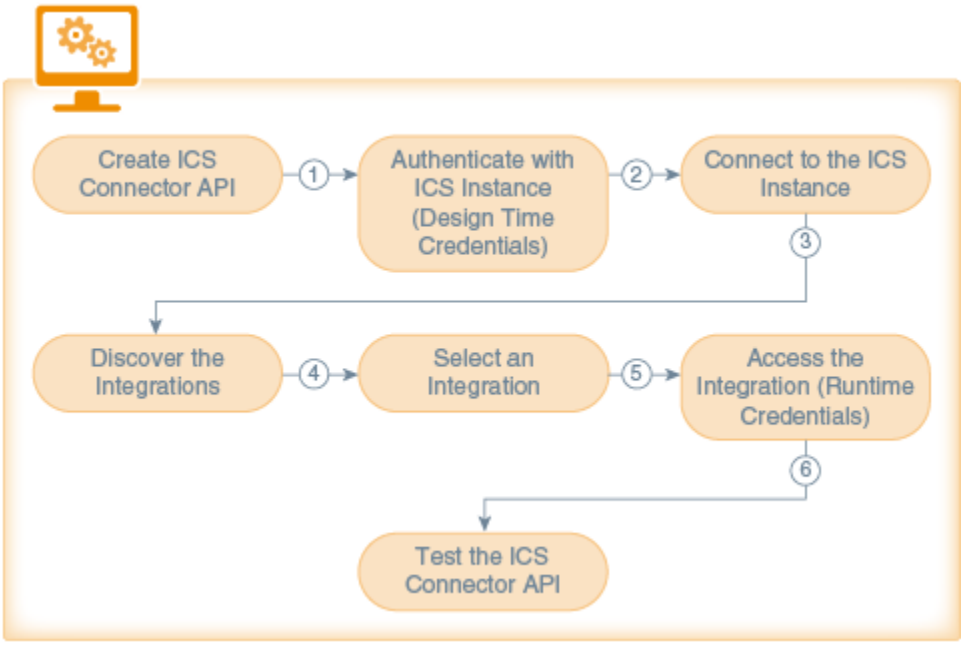
integrations that correspond with that proxy and can select one. For each ICS integration, there is a single operation per endpoint. After you select the integration, you can proceed to test the endpoint.

Once you have created an ICS connector API, you can call it from the implementation of a custom API.

> **Note:**
>
> Only SOAP-based integrations are supported.

ICS Connector API Flow

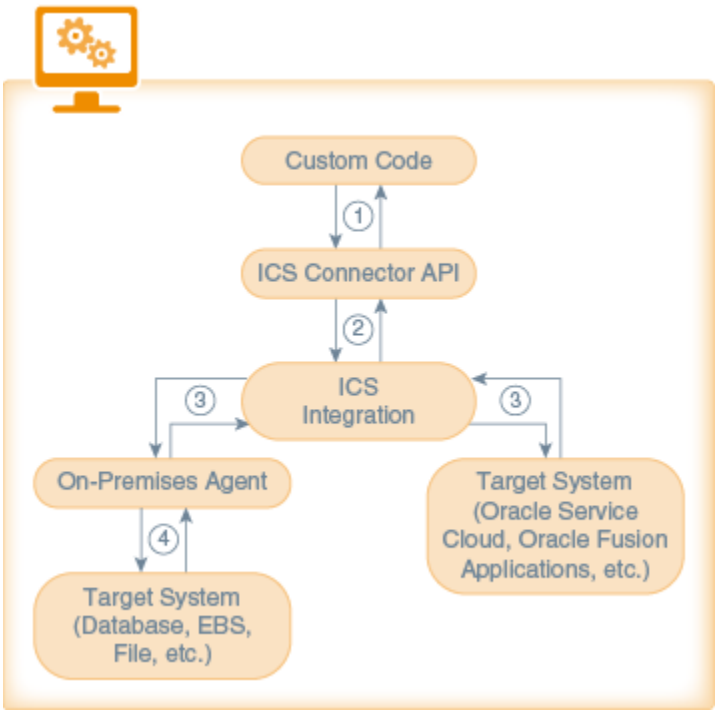
Here's the process for designing an ICS connector API:



1. **Create ICS Connector API**. You create an unbound ICS connector API with the Integration Cloud Service Connector API wizard.
2. **Authenticate with ICS Instance (Design Time Credentials)**. You pass design time credentials to connect to the ICS instance. These credentials are the username and password received when you subscribe to the Oracle Integration Cloud Service.
3. **Connect to the ICS Instance**. MCS locates the ICS instance via the service URL provided.
4. **Discover the Integrations**. When authentication is confirmed, a list of active integrations in the ICS instance is displayed.
5. **Select an Integration**. You select an integration instance from a list of the integrations.

6. **Access the Integration (Runtime Credentials)**. You pass credentials to allow access to the runtime instance of the integration. Runtime credentials are the username and password you received from the ICS administrator that allow you to run the integration.
7. **Test the ICS Connector API**. You test the endpoint using mobile user credentials.

Here's how the connector API works at runtime:



1. The custom code implementation of one of your custom APIs calls the connector API. Information is then passed to the connector implementation, and the implementation extracts the payload from the request.
2. A connection is made to the ICS service via the service URL. The service verifies the design-time credentials passed to it and the active integrations are exposed.
3. Runtime credentials are passed from ICS to either the on-premises agent or to a single cloud service to access the selected service integration.
4. Information is passed back through the integration (and, for on-premises applications, via the on-premises agent) to the connector API and back to the custom API.

How Do I Create an ICS Connector API?

Creating an ICS Connector API consists of four stages:

1. **Creation:** You've named the API and provided a description. Once created the API exists in a Draft state.
2. **Connection:** You've provided the URL to the ICS service and your design time credentials, which give you access to the ICS service.

> **Note:**
>
> The design time credentials can be saved so you only need to do it once per ICS instance. It's important to note that you can only use the credentials that you saved. That is, if other developers want to access this instance, they'll have to enter their own credentials at least once themselves.

3. **Discovery:** MCS locates the ICS service and obtains instances of the active integrations available from the service.
4. **Configure:** You've selected (or created) a CSF key for the security policy and provided your runtime credentials.
5. **Test:** Now you can test your endpoint to validate the connection to the service.

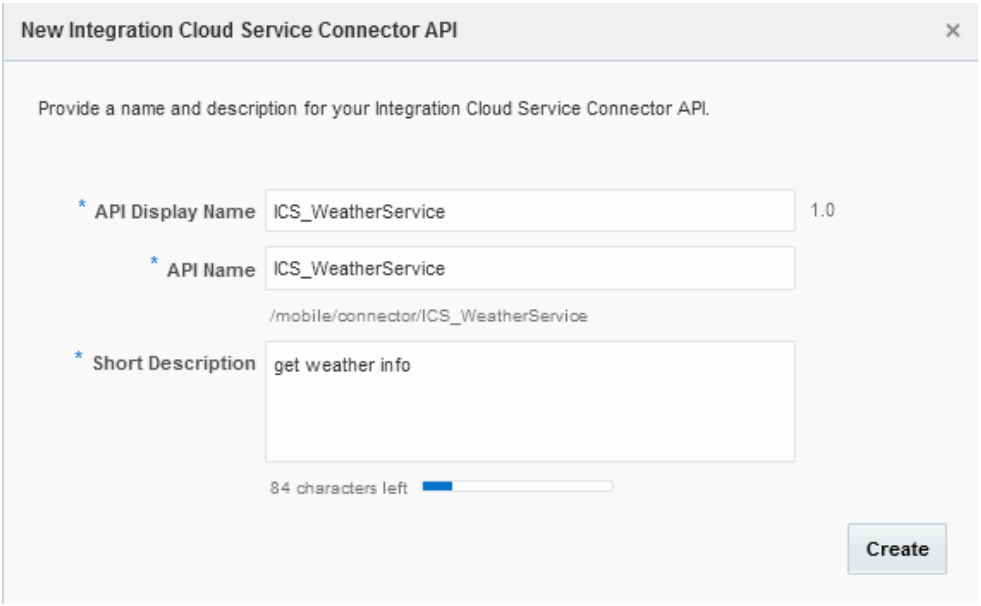
Setting the Basic Information for Your ICS Connector API

Before you begin configuring your connector, you must provide some initial basic information like the connector API name, a brief description, and connection timeout settings.

1. Make sure that you're in the right environment to create the connector.
2. Click  and select **Applications > APIs**.

   The Connectors page appears. If no connector APIs have been created yet, you'll see a REST Connector icon, a SOAP Connector icon, and an ICS Connector icon. If at least one connector API exists, you'll see a list of all the connector APIs. You can filter the list to see only the connector APIs that you're interested in or click Sort to reorder the list.
3. Click **ICS** (if this is the first connector API to be created) or **New Connector** and from the drop-down list, select **ICS**.

   Each time you create an ICS Connector API, the New ICS Connector API dialog appears. This is where you enter the basic information for your new connector API.



4. Identify your new ICS Connector API by providing the following:

   - **API Display Name:** Enter a descriptive name (an API with an easy-to-read name that qualifies the API makes it much simpler to locate in the list of connector APIs).

     For example, `myICSService`.

     For new connectors, a default version of 1.0 is automatically applied when you save the configuration.

   - **API Name:** Enter a unique name for your connector API. The default value is a simplified form of the value that you entered for the API Display Name.

     For example, `myICSService`.

     By default, this name is appended to the relative base URI as the resource name for the connector API. You can see the base URI below the API Name field.

     > **Note:**
     >
     > The connector API name must consist only of alphanumeric characters. It can't include special characters, wildcards, slashes /, or braces {}. A validation error message is displayed if you enter a name that is already in use.

     If you enter a different name for the API here, the change is automatically made to the resource name in the base URI.

     Other than a new version of this connector API, no other connector API can have the same resource name.

   - **Short Description:** Provide a brief description, including the purpose of this API.

This is the description of the API that will be displayed on the Connectors page when this API is selected. The character count below this field lets you know many characters you can add.

5. Click **Create**.

   Tthe General page of the ICS Connector API wizard is displayed.

6. Set the timeout values if needed.

   Connecting to the ICS instance can take several minutes. You can increase the timeout values to reduce the chances of a connection time out but be aware that the values that you apply at design time are also applied at runtime when the connector calls on the instance. If you do set timeout values, be sure to save your edits to the General page before proceeding to the next step of the wizard.

   > **Note:**
   >
   > If you are in a non-development environment, set these values appropriately for the environment that you're working in. Alternatively, don't enter values for these fields and let the environment-level timeout policies take effect.
   >
   > If you're a mobile cloud administrator, you can open the `policies.properties` file to see the value for the network policies for the environment that you're working in from the Administrator page. Otherwise, ask your mobile cloud administrator for the values. To learn about environment policies, see Environment Policies.

   - HTTP Read Timeout: The maximum time (in milliseconds) that can be spent on waiting to read the data. If you don't provide a value, the default value of 20 seconds is applied.
   - HTTP Connection Timeout: The time (in milliseconds) spent connecting to the remote URL. A value of 0 mms means an infinite timeout is permitted.

7. Click **Save** to save your current settings.

   If you want to stop and come back later to finish the configuration, click **Save and Close**. You can always click **Cancel** at the top of the General, Integration, and Runtime Security pages to cancel that particular configuration operation. You'll be taken back to the Connector APIs page.

8. Click **Next** (>) to go to the next step in configuring your connector API.

   After the basic information is provided, you can specify the interaction details for your connector API.

You can always edit your configuration when it's in a Draft state. You can make changes to a connector API that's in the Published state by creating a new version of it. For information on creating a new version, see Creating a New Version of a Connector.

Connecting to an Integration Cloud Service Instance

This is where you select the Integration Cloud Service (ICS) instance that you want or create a connection to an ICS instance. If this is the first time that you're creating an ICS connector API, the Select Connection drop-down list won't be available and you'll have to create a connection to the instance.

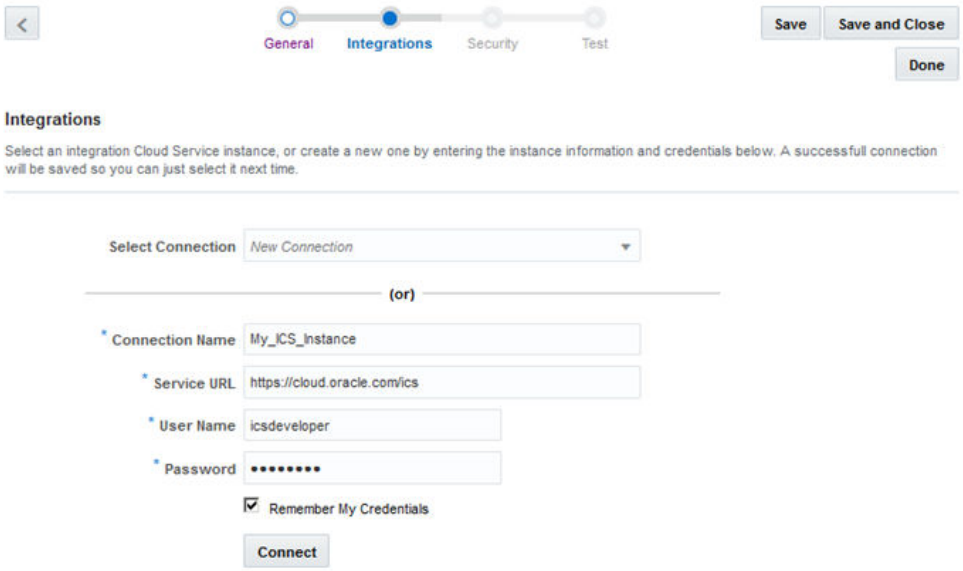
Making a connection consists of the following phases:

- Selecting or creating an ICS instance and authentication
- Connecting to the server hosting the active integrations
- Selecting the active integration

You perform or observe these operations on the Integrations page of the Integration Cloud Service Connector API wizard.

Selecting or Creating an ICS Instance Connection

1. If at least one integration instance exists, select an integration instance from the **Select Connection** drop-down list; otherwise, go to Step 2 to create an instance.



2. Enter a name to identify this Integration Cloud Service instance in the **Connection Name** field.

   This name will be added to the list of integration instances.

3. Enter the address of the server that hosts the integrations in the **Service URL** field.

   You get the URL of the service from the service administrator of the Oracle Cloud Integration Service. The URL takes the form *hostname*`/ics`.

You can save time by verifying that the URL you're providing is trusted at trustedsource.org, otherwise, even if you're connector API is configured correctly, the connection will fail. See Common Custom Code Errors.

4. Enter your user name and password that you were given to access the integration.

   These are the design time credentials that enable you to access the Oracle Integration Cloud Service. These are the user name and password you received when you subscribed to the service.

5. Select **Remember My Credentials** so that the next time you select or create an integration instance, your credentials are already preloaded.

   These credentials are specific to the individual MCS user and aren't provided if another MCS user tries to access the same integration instance.

6. Click **Connect**.

After you've created an integration instance, you'll be able to select it from the **Select Connection** drop-down list the next time you come back to the wizard.

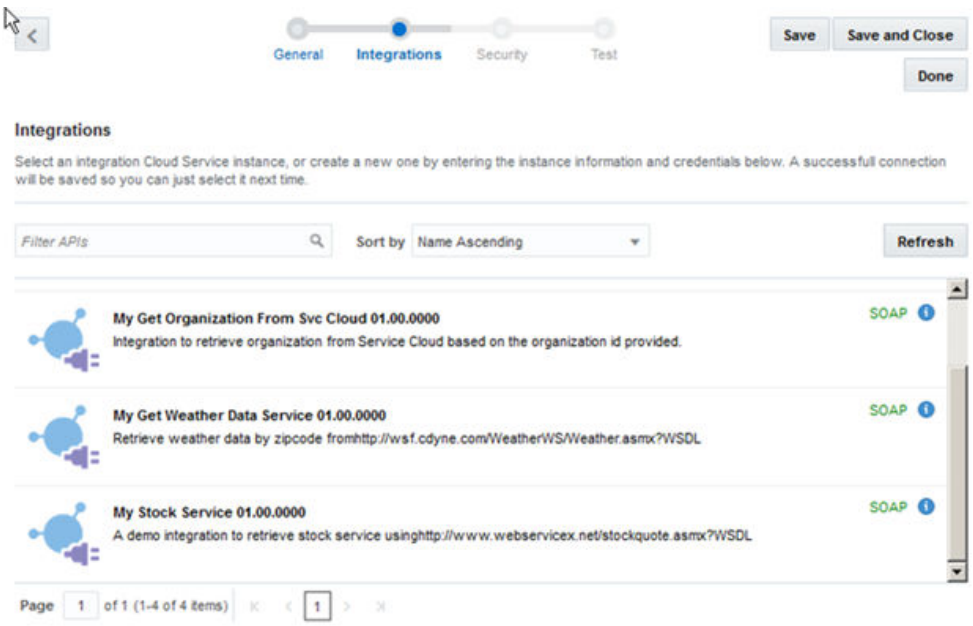
Selecting an Active Integration

When the connection to the server hosting the integrations is made, the Integrations page of the wizard displays all the active integrations where a single cloud service or on-premises solution is exposed as an integration-friendly API. Non-active integrations or integrations that push events from one cloud service or on-premises solution to another aren't listed. Each integration is displayed with its name, version, and description.

1. Filter the list by entering part of its name, description, or integration type.

   You can sort the list in either ascending or descending order based on name, creation date, last update, or type.

2. Select the integration you want.



Click the information icon to see details about the integration including a link to the WSDL for the integration.

> **Note:**
>
> Remember, that currently, only SOAP-based integrations are supported.

3. Click **Save**.
4. Click **Next** (>) to go to the next step in configuring your connector API.

Editing the ICS Connector API

If you go to the RunTime Security page and change your mind about the integration you selected, you can go back and select a different integration. The list of integrations you see might not be the latest available though. If you do go back, be sure to refresh the page before selecting another integration. Also, you'll have to re-authenticate yourself to access the list of integrations if you didn't save your credentials previously.

> **Note:**
>
> Once you've moved on to the Test page, you won't be able to go back to the Integrations page to select a different integration. If you return to the Integrations page from the Test page, you'll see only the integration that you've selected.

1. Click **Integrations** in the navigation links at the top of the wizard.

   The page displays only the integration you originally selected.
2. Click **Refresh** on the Integration page of the wizard.
3. Confirm the refresh action.

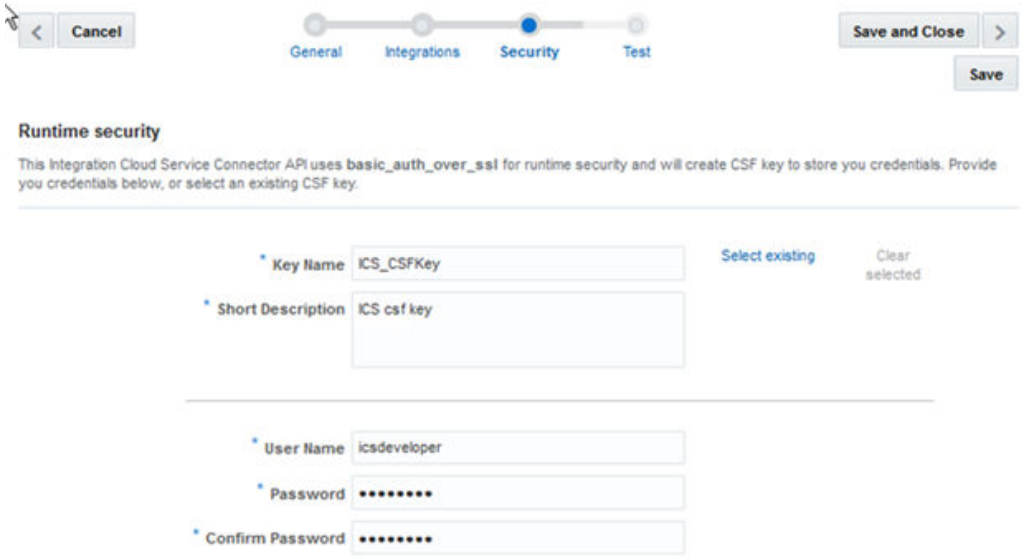
   The Integrations page is displayed at the authentication phase. The connection name and service URL you provided previously are shown as information only.
4. If you previously selected the **Remember My Credentials** option, click **Connect**.

   If you didn't select that option, enter your design time user credentials and click **Connect**.
   Credentials are saved securely in the MCS backend. You only need to save them once for that user's devices and browsers. Note that no sensitive information is stored locally.
5. Select the active integration you want from the list after the connection is completed.

6. Click **Save**.
7. Click **Next** (>) to go to the next step in configuring your connector API.

Setting Runtime Security for the ICS Connector API

You must set the **csf-key** property with your runtime credentials to allow you access and test the active integration.



Provide a CSF Key in one of the following ways:

- Click **Select Existing** and select an existing key from the Available Keys list in the Select or Create a New API Key dialog. A description of the selected key is displayed below the list. The list displays only the keys supported by the client policy, which could be `http_basic_auth_over_ssl_client_policy`, `wss_http_token_over_ssl_client_policy`,or `wss_username_token_over_ssl_client_policy`.

  When you select the key, its name appears in the **Key Name** field. Click **Select** to add the key. The other fields in the CSF Key Details pane are used only when creating a key.
- Create a new basic (CSF) credentials key directly on the Security page.

  For the steps on creating a key, see Creating a New CSF Key. Alternatively, you can click **Select Existing** and create the key in the Select or Create a New API Key dialog.

Regardless of which security policy is used, the ICS adapter API determines the correct authentication mode. Once you've configured the ICS Connector API for a given ICS instance, the runtime credentials that you provided for that instance are remembered the next time you configure an ICS Connector API.

To learn about security policies for the ICS Connector, see Security and ICS Connector APIs.

Creating a New CSF Key

1. Click the **Security** navigation link.
2. Enter a key name that is descriptive and easy-to-read. Note that after you create the key, you can't change the key name.
3. Enter a brief description of the key's purpose.
4. Enter your runtime credentials for the service to which you are connecting.

   Contact your ICS administrator to obtain the credentials used to call the Oracle Integration Cloud Service at runtime. Most likely, you'll only need to do this once per ICS instance (all integrations are called with the same app credentials).
5. Repeat the password in the confirmation field.
6. Click **Save** to continue working in the dialog.

   Click **Save and Close** to save your actions and return to the Security page. Click **Cancel** to quit the task.

The key name value will appear as the override value on the Security page. Note that the value of the key that you create pertains only to the environment in which it's set. If you want to edit some aspect of an existing CSF key, select it from the Available Keys list and modify the fields as needed.

If you've already selected a key but then decide to create a new key, click **Clear Selected** to clear all the fields.

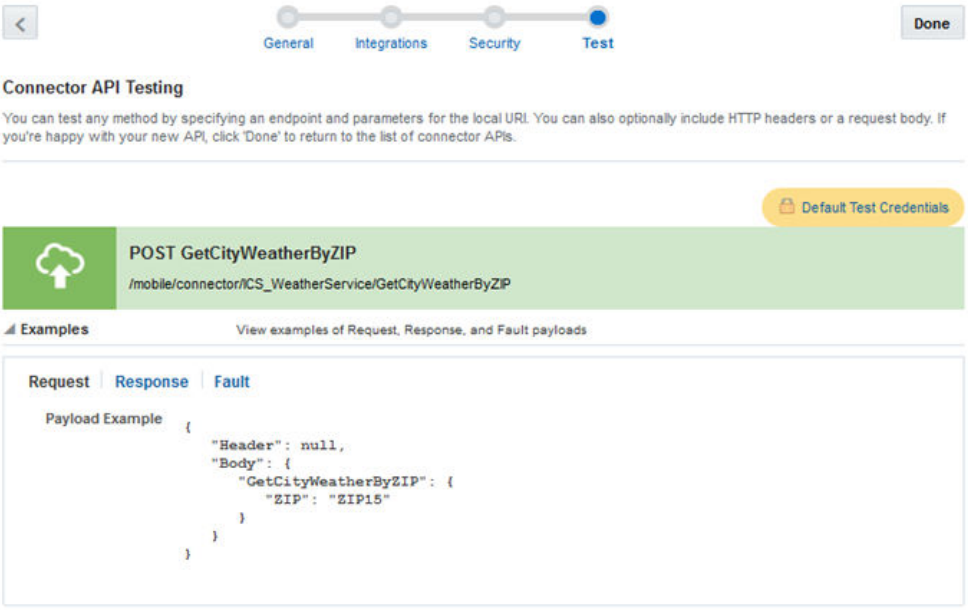
To learn about CSF Keys, see CSF Keys.

Testing the ICS Connector API

When you've finished configuring your ICS Connector API, test the endpoint:

1. Click the **Test** navigation link.

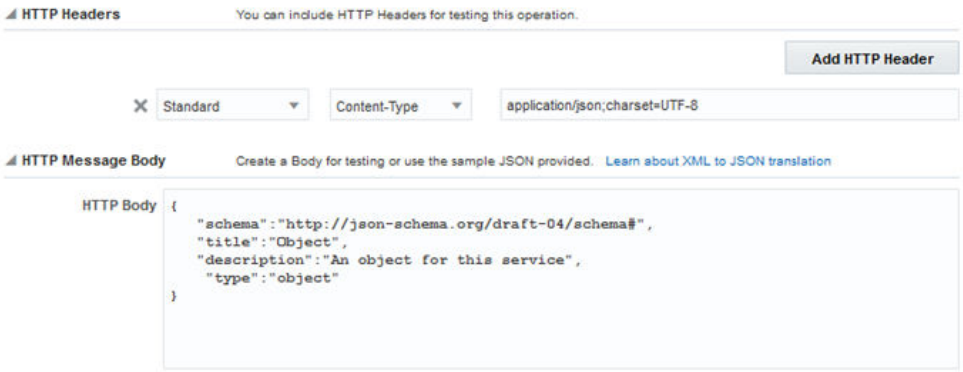
   There is only one endpoint per integration. The resource banner displays the method, the resource name, and the URI of service.
2. Expand **Examples** to see examples of a request, response, and fault payloads that were obtained from the WSDL.



When you select a connection, all the fields on the page are populated with data for that connection with the exception of credentials.

If this is the first time a connection is being created, skip this step and go to Step 3.

3. Add one or more request or response HTTP headers as needed.



4. Click in the HTTP Body field to create your message body (the payload) in the source editor. For example:

```
{
 "$schema":"http://json-schema.org/draft-04/schema#",
 "title":"Object",
 "description":"An object for this service",
 "type":"object"
}
```

5. Provide your runtime credentials for testing this endpoint:

   a. Enter the name of the mobile backend associated with this connector API.

b. Enter the version of the mobile backend.

c. (Optional) Enter your mobile user credentials, that is, your runtime credentials.

6. (Optional) Click **Save as current mobile backend default credentials** to allow the ICS Connection API to remember your credentials. Only your credentials will be stored. These credentials are applied when you test another ICS Connector API, REST or SOAP Connector API, or a custom API.

7. Click Test Endpoint.

Test Endpoint toggles to **Cancel Test**. If you want to stop the test for any reason, click **Cancel Test**.

8. Click **Done** when you've finished testing your endpoint.

   You're returned to the Connectors APIs page.

If you want to make changes to the testing parameters, click **Reset** to clear all the fields.

Getting the Test Results

Test results are displayed at the bottom of the Test ICS API page. The result indicator is the response status:

- 2*xx*: indicates a successful connection
- 4*xx*: indicates a user error occurred
- 5*xx*: indicates a server error occurred

The following table lists the most common status messages you'll see:

Status Code	Description
200 OK	Successful connection.

Status Code	Description
400 BAD REQUEST	General error when fulfilling the request, causing an invalid state, such as missing data or a validation error.
401 UNAUTHORIZED	Error due to missing or invalid authentication token.
403 FORBIDDEN	Error due to user not having authorization or if the resource is unavailable.
500 INTERNAL SERVER ERROR	General error when an exception is thrown on the server side.

Click **Request** to see the metadata for the transaction, such as header information and the body of the request.

Click **Response** to see the details of the response returned. The response code tells you whether or not the connection was successful.

After your connector API is tested, published, and deployed, you can go to the Connectors page to see analytical information about it, such as how often the connector is being called and what apps are using the connector. See Managing a Connector.

Getting Diagnostic Information

You can view the response code and returned data to determine if your endpoints are valid. A response status other than 2xx doesn't necessarily mean the test failed. If the operation was supposed to return a null response, a response should show a 4xx code.

For every message that you send, MCS tags it with a correlation ID. A correlation ID associates your request with other logging data. The correlation ID includes an Execution Context ID (ECID) that’s unique for each request. With the ECID and the Relationship ID (RID), you can use the log files to correlate messages across Oracle Fusion Middleware components. By examining multiple messages, you can more easily determine where issues occur. For example, you can retrieve records from Oracle Fusion Middleware Logging using the call's ECID. From the Administration page, you can click Logs to view logging data: the connector API call received by a single MBE outbound connector API call.

Depending on your MCS access permissions, you or your mobile cloud administrator can view the client and server HTTP error codes for your API's endpoints on the Request History page allowing you to see the context of the message status when you're trying to trace the cause of an error. Every message sent has a set of attributes such as the time the event occurred, the message ID, the Relationship ID (RID), and the Execution Context ID (ECID).

To obtain and understand diagnostic data, see Diagnostics.

Security and ICS Connector APIs

HTTP Basic Authentication is used for runtime security. Basic authentication allows an HTTP user agent to pass a user name and password with a request and is often used with stateless clients, which pass their credentials on each request.

ICS Connector APIs use one of the following security policies:

- `http_basic_auth_over_ssl_client_policy`. It includes the username and password credentials in the HTTP header for outbound client requests. This policy verifies that the transport protocol is HTTPS.
- `wss_http_token_over_ssl_client_policy`. The username and password credentials are included in the HTTP header for outbound client requests. Also a timestamp is sent to the SOAP security header. If the connector detects that the ICS integration that's being connected to is protected by the `wss_http_token_over_ssl_service_policy`, the connector uses the corresponding client policy. This policy verifies that the transport protocol is HTTPS.
- `wss_username_token_over_ssl_client_policy`. The username and password credentials are passed as SOAP headers and are added automatically by the connector. If the security policy is defined in the WSDL for a SOAP-based integration, this is the policy that's used. This policy verifies that the transport protocol is HTTPS.

Although you can set the `Oracle-Mobile-External-Authorization` header in custom code to configure a secure connection, it isn't necessary since authorization to connect to a service is set when configuring the ICS Connector API.

CSF Keys

In MCS, the Oracle Credential Store Framework (CSF) is used to manage credentials in a secure form. A credential store is a repository of security data (credentials stored as keys) that certify the authority of users and system components. CSF lets you store, retrieve, update, and delete credentials (security data) for a web service and other apps.

A CSF key is a credentials key. It uses simple authentication (composed of the user name and the password for the system to which you're connecting) to generate a unique key value. You can select an existing CSF key or create one through the Select or Create a New API Key dialog. To select or create a CSF key, see Creating a New CSF Key.

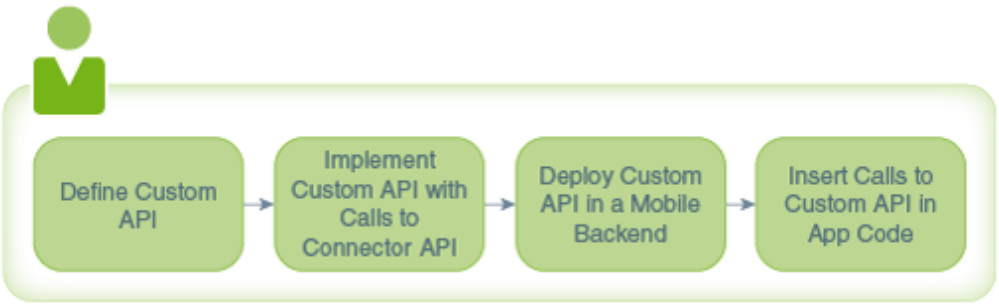
CSF keys and their values are specific to the environment in which they're defined. That is, if the Development environment is selected, then only the CSF keys and certificates for the security policies in use by artifacts in that environment are listed in the CSF Keys dialog. A different set of keys and certificates will be displayed in another environment, such as Staging. It's also possible for keys with the same key name but with different values to exist in multiple environments.

A CSF key can be deployed to another environment, however, because CSF keys are unique to an environment, only the key name and description are carried over to the target environment. You won't be able to use that key in the new environment until it's been updated with user name and password credentials by the mobile cloud administrator.

Using Your Connector API in an App

To use a connector in a mobile app, you need to have a custom API that can call the connector API. Such a custom API could also contain additional logic to process the data returned from the call to the connector.

The syntax for a call to a connector API is the same as you would use when calling any other API from custom API implementation code. See Calling Connector APIs from Custom Code.



When you implement a custom API, you can view the available connectors in the API Catalog tab in the API Designer. While creating your custom API, you might find it beneficial to open the Test page of the connector API so that you can refer to any headers, parameters, and schemas that you've configured for the connector API.

Troubleshooting ICS Connector APIs

System message logs are great sources for getting debugging information. Depending on your role, you or your mobile cloud administrator can go to the Administration view and click **Logs** to see any system error messages or click **Request History** to view the client (4*xx*) and server (5*xx*) HTTP error codes for the API's endpoints and the outbound connector calls made within a single mobile backend.

Here are some areas of particular interest when troubleshooting:

- Security Errors are Occurring

  Take a look at the integration WSDL and see if you can determine what security policy is being used. Use the SOAP connector directly to create a connector API and test with different security policies.

- An Integration Isn't Showing Up

  Go to Oracle Integration Cloud Service and look at your integrations there. The status must be activated, and the source connection type should be SOAP.

- Constructing a Valid ICS Instance URI

  Your instance URI must begin with `https://` and should end in `/ics`. Look for the Email that you received when your user account was provisioned for the ICS instance. From there, you can find the URI to reach the ICS UI. The same URI should be used to create the connection in MCS.

- Identifying Where the Failure Is Occurring

  As with other connectors generally finding where a fault was thrown can be difficult. A 401 or 404 for instance could be returned by the test endpoint, MCS itself, the ICS instance that MCS is connecting to, or the system to which ICS is connecting.

  401 and 404 errors are difficult because they return no message body that might indicate where the error occurred. However, the headers associated with a 401 and 404 error can sometimes act as a signature to indicate where it originated

from. Likewise, trace the end-to-end flow by searching for corresponding log entries at each step in the flow.

- Can’t Make a Connection Using Default Protocols

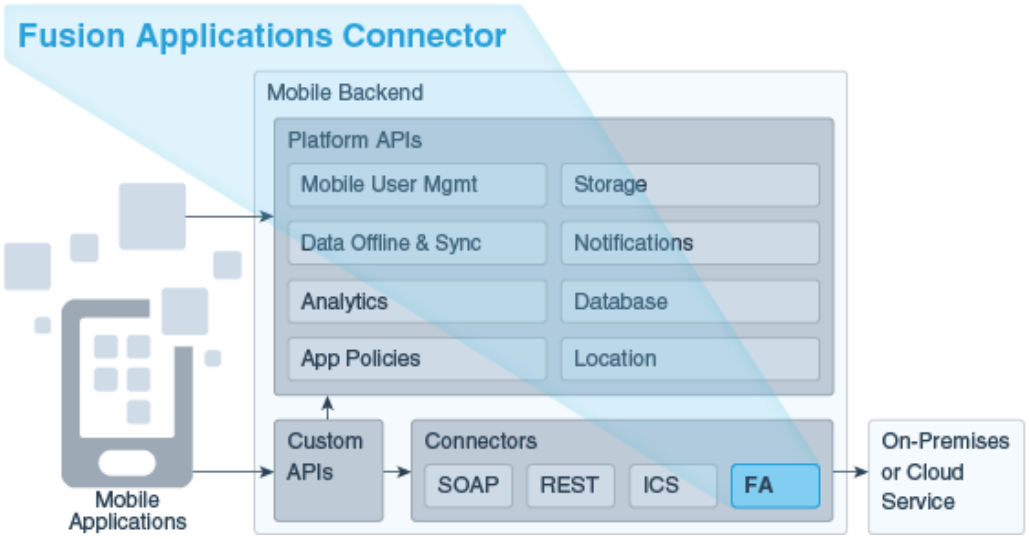
  By default, only TLSv1.1 and TLSv1.2 protocols are used for outbound connections. If you need to use an older version of a SSL protocol to connect to an external system that doesn't support the latest versions of SSL, you can specify the SSL protocol to use for the connector by setting the `Security_TransportSecurityProtocols` environment policy. The policy takes a comma-separated list of TLS/SSL protocols, for example: TLSv1, TLSv1.1, TLSv1.2. Any extra space around the protocol names is ignored. You can use the SSLv2Hello protocol to debug connectivity issues with legacy systems that don't support any TLS protocol. Note that this policy can’t be used to enable SSLv3 endpoints. See Environment Policies and Their Values for a description of the policy and the supported values. Be aware that this policy must be manually added to a `policies.properties` file that you intend to export.

  > **Caution:**
  >
  > Be aware when setting the policy that older protocols are vulnerable to security exploits.

27

Fusion Applications Connector APIs

Oracle Mobile Cloud Service (MCS) enables you to create Fusion Applications (FA) Connector APIs to connect to Oracle Fusion Applications. As a service developer, you can create connector APIs to make it easier to call these external services from the implementations of your custom APIs.



A Fusion Applications Connector API enables a mobile backend to use and expose data from one or more resources available from an Oracle Fusion Applications instance.

When configuring the connector API, you need to enter runtime credentials each time you need to access a Fusion Applications service, but customers of Fusion Applications-based services should only have to sign in once to the mobile app to access the Oracle Cloud application. You can create an app that lets users sign in just once with their identity domain credentials. To see how to set up single sign-in, see Configuring Oracle Cloud Applications as the Identity Provider.

How Fusion Applications Connector APIs Work

A Fusion Applications Connector API enables a mobile backend to use and expose data from resources available from Fusion-based software-as-a-service (SaaS) instances, such as Oracle Human Capital Management Solution (HCM), Oracle Supply Chain Management (SCM), and Oracle Customer Relationship Management Solution (CRM). These suites of modular services help you with customer and employee management, sales and supply chain management, and more.

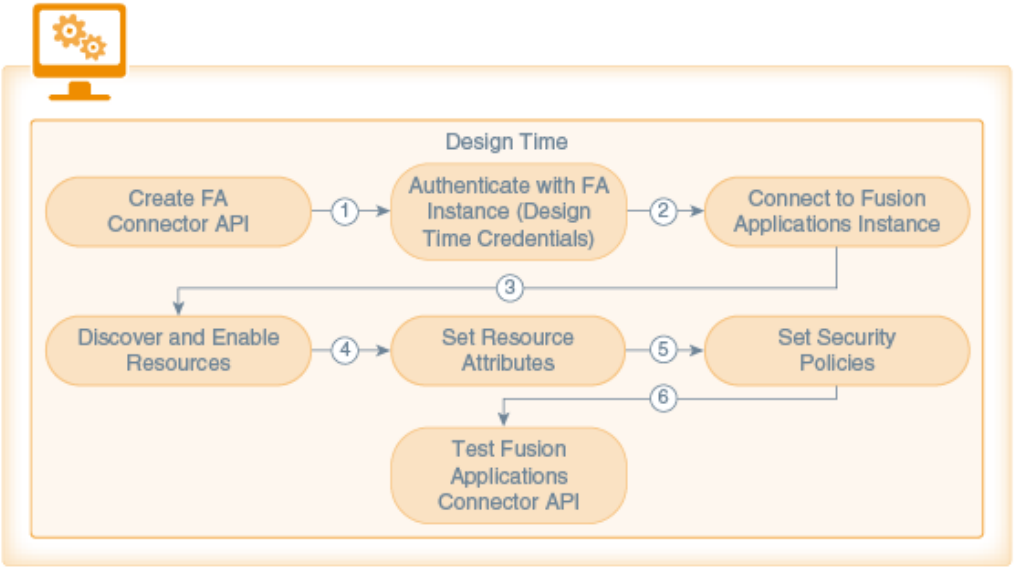
Use the Fusion Applications Connector API wizard to quickly and easily create a connector API with a customized selection of resources from a Fusion Applications service or Fusion-based service.

Here are the some of the advantages to using a Fusion Applications Connector API:

- Makes it easier for customer to explore Fusion-based services through resource discovery.
- Makes it easier for you to see all the resources, child resources, and resource attributes available in a given resource instance.
- Lets you provide easy to identify and comprehend user-friendly names and descriptions for the resources and their attributes in the connector.
- Provides a rich test client that lets you test with Fusion Applications query parameters.

Fusion Applications Connector API Flow

Here's how the design-time flow for a Fusion Applications Connector API design-time goes:



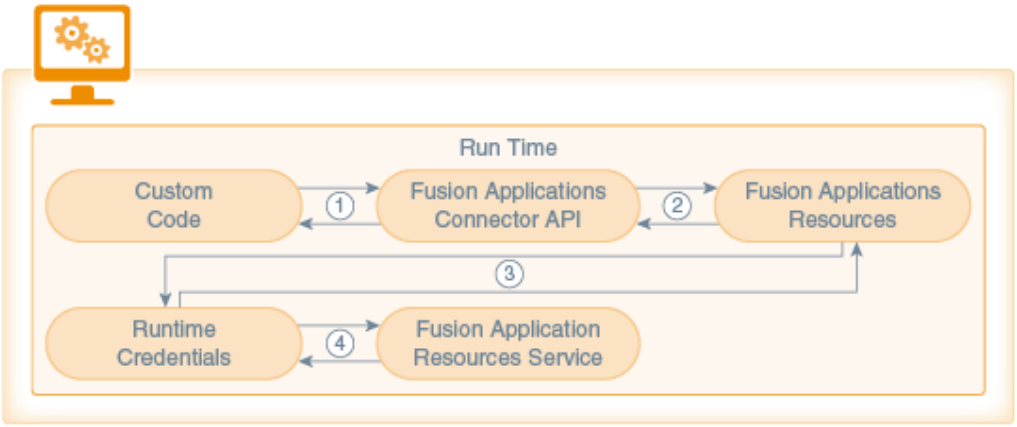
1. **Connector Creation phase.** An unbound Fusion Applications Connector API is created with the Fusion Applications Connector API wizard.
2. **Connection phase**. Design time credentials are passed and a connection to the Fusion Applications instance is made. The design time credentials are saved in the Credentials Store Framework (CSF) in MCS. The Fusion Applications service description, the Fusion Applications Describe, is retrieved from the external service.
3. **Resource Discovery phase**. MCS locates the Fusion Applications instance via the Describe URL provided. When authentication is confirmed, MCS downloads and parses the `Describe` resource and displays the list of resources exposed by the Fusion Applications service. The resources list is examined and the desired resources to access from the custom code are enabled.

   In addition, descriptions for each attribute may be provided. Attribute values are available only at runtime and can't be changed during design time.

Whenever you enable or disable resources or refresh the list of available resources, the changes are time stamped and tracked in a *work area*. Each instance of the connector API has one work area and the contents of that work area are saved as part of the configuration when the connector API is saved.

4. **Attribute Setting phase**. Attributes are selected or de-selected based on the requirements for the connector. Values for resource attributes are modified as needed.
5. **Runtime Security phase**. The Oracle Web Services Manager (Oracle WSM) security policy to be used at runtime is configured.
6. **Testing phase**. The configuration is saved. The enabled resources are displayed on the Test page and tested. Mobile user credentials are provided to test the connector API.

Here's how the runtime flow goes:



1. Custom code calls the Fusion Applications Connector API. Information is then passed to the connector implementation. The implementation extracts the payload from the request.
2. The connector implementation checks whether or not the resource is enabled. If the endpoint is a GET request, a fields query parameter is added to the request so that the attributes returned by the Fusion Applications service are limited to only those attributes that were enabled for the resource at design time.
3. Runtime credentials (which are based on the security policies selected during design time) are added to the request and the request is sent to the Fusion Applications service.
4. Information is passed back from the Fusion Applications service to the connector API and finally back to the custom code.

How Do I Create a Fusion Applications Connector API?

The Fusion Applications Connector API wizard will walk you through the following stages of creating the connector API:

1. **Setting Up the Basics**. Name the API and provide a description. When you click **Create**, the API exists in a Draft state.

2. **Connecting To and Selecting Resources**. Locate the Fusion Applications service through the Describe URL that you provide and select the resources available from the service.
3. **Selecting Attributes**. Choose the attributes for each resource and child resource.
4. **Setting the Runtime Security**. Select the runtime security policies you need to connect to the runtime Fusion Applications instance.
5. **Testing the Connector API**. Test your endpoint to validate the connection to the service.

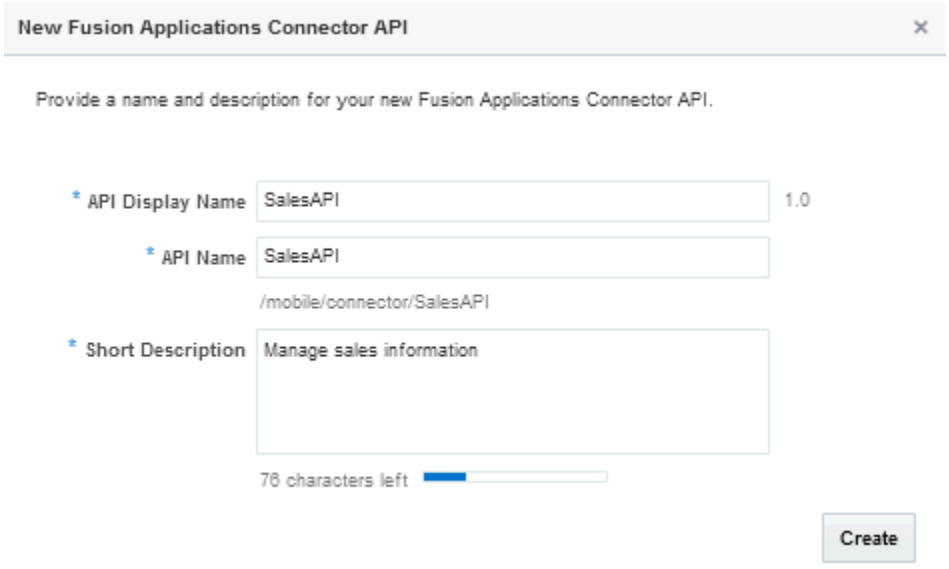
Setting the Basic Information for Your Fusion Applications Connector API

Before you begin configuring your connector, you must provide some initial basic information like the connector API name, a brief description, and a local URI (from which the connector API will available to the custom code):

1. Make sure that you're in the right environment to create the connector.
2. Click  and select**Applications > Connectors** from the side menu.

   The Connectors page appears. If no connector APIs have been created yet, you'll see icons for REST, SOAP, ICS, and Fusion Applications. When at least one connector API exists, you'll see the connector landing page where existing connector APIs are listed. You can filter the list to see only the connector APIs that you're interested in or click **Sort** to reorder the list.
3. Click **Fusion Applications** if this is the first connector API to be created or **New Connector** and select **Fusion Applications**.

   Each time you create a Fusion Applications Connector API, the New Fusion Applications Connector API dialog appears. This is where you enter the basic information for your new connector API.



4. Identify your new Fusion Applications Connector API by providing the following:

a. **API Display Name**: Enter a descriptive name (an API with an easy-to-read name that qualifies the API makes it much simpler to locate in the list of connector APIs).

   For example, `myFAServiceAPI`.
   For new connectors, a default version of 1.0 is automatically applied when you save the configuration.

b. **API Name**: Enter a unique name for your connector API. The default value is a simplified form of the value that you entered for the API Display Name.

   For example, `myFAServiceAPI`.
   By default, this name is appended to the relative base URI as the resource name for the connector API. You can see the base URI below the API Name field.

   > **Note:**
   >
   > The connector API name must consist only of alphanumeric characters. It can't include special characters, wildcards, slashes /, or braces {}. A validation error message is displayed if you enter a name that is already in use.

   If you enter a different name for the API here, the change is automatically made to the resource name in the base URI.

   Other than a new version of this connector API, no other connector API can have the same resource name.

c. **Short Description**: Provide a brief description, including the purpose of this API.

   This is the description of the API that will be displayed on the Connectors page when this API is selected. The character count below this field lets you know many characters you can add.

After you've filled in all the required fields, click **Create**.The connector API is created and the General page of the Fusion Applications Connector API wizard is displayed.

5. Set the timeout values if needed.

   Connecting to the Fusion Applications instance can take several minutes. You can increase the timeout values to reduce the chances of a connection time out but be aware that the values that you apply at design time are also applied at runtime when the connector calls on the instance. If you do set timeout values, be sure to save your edits to the General page before proceeding to the next step of the wizard.

> **Note:**
>
> If you are in a non-development environment, set these values appropriately for the environment that you're working in. Alternatively, don't enter values for these fields and let the environment-level timeout policies take effect.
>
> If you're a mobile cloud administrator, you can open the `policies.properties` file to see the value of the network policies for the environment that you're working in from the Administration page. Otherwise, ask your mobile cloud administrator for the values. To learn about environment policies, see Environment Policies.

- **HTTP Read Timeout**: The maximum time (in milliseconds) that can be spent on waiting to read the data. If you don't provide a value, then the default value (20 seconds) of the environment-level HTTP Read Timeout policy is applied.
- **HTTP Connection Timeout**: The time (in milliseconds) spent connecting to the remote URL. A value of 0mms means an infinite timeout is permitted.

6. Click **Save** to save your current settings.

   If you want to stop and come back later to finish the configuration, click **Save and Close**. You can always edit your configuration when it's in a Draft state. You can always click **Cancel** at the top of the General, Rules, and Security wizard pages to cancel that particular configuration operation. You'll be taken back to the Connector APIs page.

7. Click **Next** (**>**) to go to the next step in configuring your connector API.

Connecting to a Fusion Applications Instance

This is where you specify the Oracle Fusion Applications instance that you want to create a connection to via the `Describe` resource.

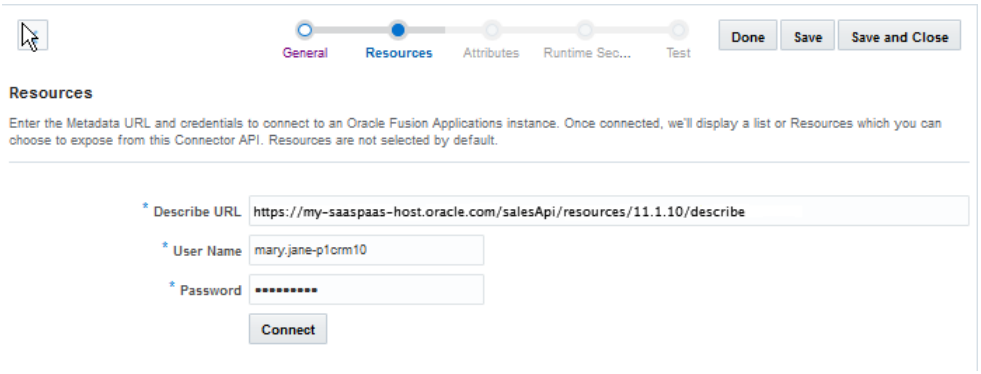
Making a connection consists of the following actions:

- Providing the Describe URL to access the metadata of the Fusion Applications instance that you want
- Providing access authentication (that is, your design time credentials)
- Connecting to the server hosting the resources

You perform these operations on the Resources page of the Fusion Applications Connector API wizard.

Creating a Fusion Applications Instance Connection

1. Click the **Resources** navigation link.



2. In the **Describe URL** field, enter the address of the `describe` resource where the Oracle Fusion Applications instance can be accessed.

   Use the `describe` resource to retrieve the metadata of a resource, which includes the fields and attribute values in the resource, the resource operations, and any child resources.
   You get the Describe URL from the administrator of the Oracle Fusion Applications.

   The URL takes the form `http://`*host*`:`*port*`/`*api-name*`/resources/`*version*`/`*resource-path*`/describe`.

   For example: `https://myhost:8080/CommonAPI/resources/1.1/incidents/describe`.

   You can save time by verifying that the URL you're providing is trusted at trustedsource.org, otherwise, even if you're connector API is configured correctly, the connection will fail. See Common Custom Code Errors.

3. Enter the user name and password that you were given to access the resource.

   These are the design time credentials that enable you to access the Oracle Fusion Applications instance. You should've received these credentials when you registered with Oracle Fusion Applications.

4. Click **Connect**.

   The resources in the Fusion Applications instance are retrieved. Making the connection can take a few minutes. You can stop the connection by clicking **Abort** in the Connecting dialog to stop the process. You'll be returned to the Resources page.

After the connection is made, the Describe URL and your design time credentials are preserved for this connector API.

Selecting Fusion Applications Resources

When the connection to the server hosting the resources is made, the Resources page of the wizard displays a list of all the resources in the given Oracle Fusion Applications resource instance. You create a custom configuration by selecting a combination of top-level resources and child resources. You can see the address of the server hosting

the fusion application services (`http://host:port/api-name/resources/version`) in the **Service Root** field along with the design time credentials user name above the resources.
A list of resources is displayed on one side of the Resources page. All the resources are unselected by default. Select at least one resource to include it in your Fusion Applications Connector API configuration. When you select a resources, its description, resource paths, and any child resources are displayed in the right panel.

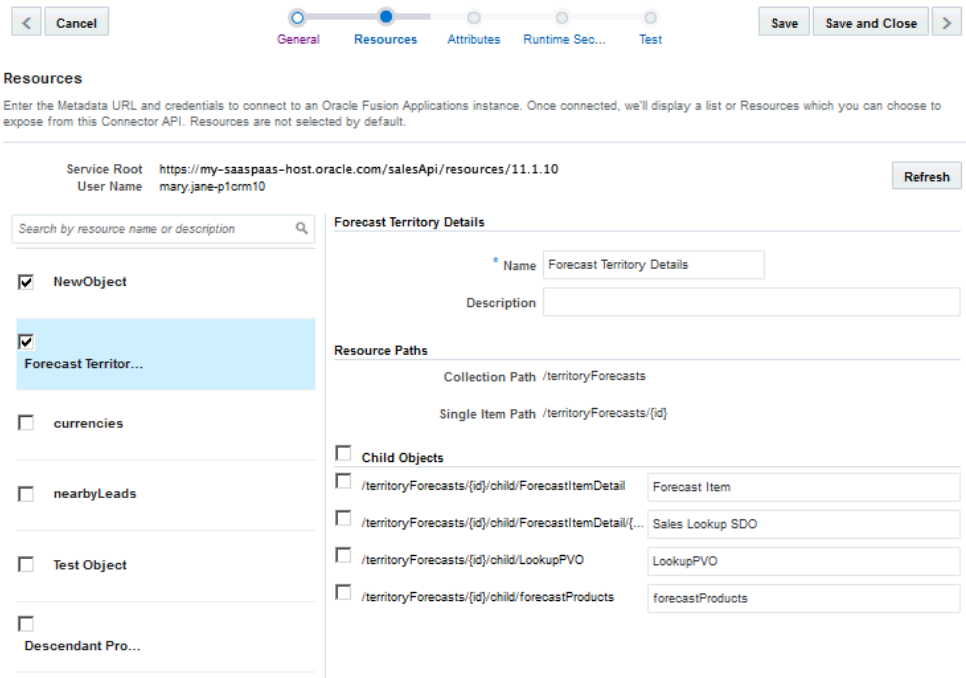
1. Select a resource to enable it and add it to the connector API configuration.

   If the list is long, enter a resource name or its description in the **Search** field to locate a resource.

   When you perform a search and the resource is a child of another resource, it's displayed at the same level as the parent resource in the list. Child resources are displayed in the form *<parent_resource>/<child_resource>*.

   If you change your mind about a selection, you can disable a resource to exclude it by selecting it again. If the resource has child resources, the parent resource and all of its child resources are removed

2. Select a resource to see its details, including any child (nested) resources in the right panel of the page.



The details panel always shows the top-level resource and all of its child resources even if the resource you currently have selected in the resources list is not a top-level resource.

Click **Refresh** to get the most up-to-date list of resources. When you click **Refresh**, the current list of resources is discarded. To get the latest set of resources, MCS must make a connection to the `Describe` resource again. You'll get a confirmation dialog asking you to confirm that you want to discard the current set of resources. If you click **Confirm**, you'll be taken back to the initial display of

the **Resources** page where you'll have to re-enter the Describe URL and your design time credentials.

3. (Optional) Provide a friendly name for the resource or a description in the **Name** field in the Details section.

   Friendly names for resources are displayed on the following Attributes page.

   The Collection and Single Item paths for the top-level resource, which you can see just above the child objects are the relative paths at which the resource collection and the single item resource are available. These paths are relative to the service root shown at the top of the page.

4. (Optional) Select individual child resources to include in your configuration.

   Click **Child Objects** to include all the child resources of the selected top-level resource in your configuration

   All child resources are displayed at the same level. That is, nested child resources are not visibly distinct in the list.

   Each child resource is listed in the form of a relative path of the collection containing the child resource.

   Click **Remove** in the dialog box to continue or **Cancel** to stop the removal.

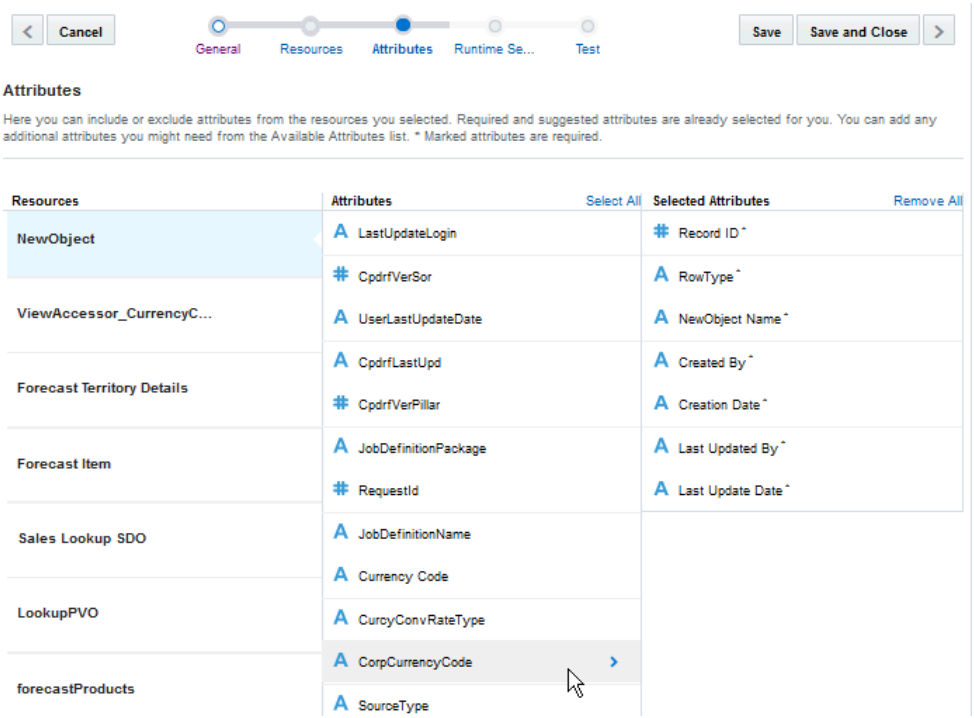
5. (Optional) Provide a friendly (identifiable) name for the child resource in the **Name** field.

6. Click **Next** (**>**) to go to the next step in configuring your connector API.

Setting Resource Attributes

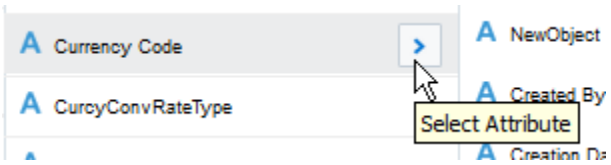
On the Attributes page, you can select the optional attributes you want for each of your selected resources. Any required attributes are automatically added to the configuration. Select a resource from the Resources list, view the available attributes for the resource in the next column, and then select the specific attributes you want to include in the connector configuration:

1. Click the **Attributes** navigation link.



On the **Attributes** page, you'll see three columns. The first column, **Resources**, is the list of resources you previously selected. The second column, **Attributes**, lists all the attributes that you can select for a particular resource. The last column, **Selected Attributes**, lists required and optional attributes that are pre-selected for you. When you select an attribute in the second column, it's added to the list of selected attribute.

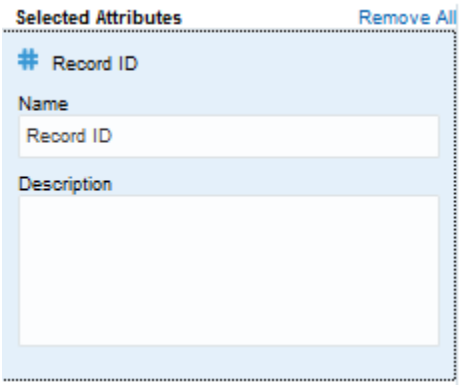
2. Select a resource from the Resources list.
3. Add an attribute for the selected resource in the Attributes to your configuration by clicking **Select Attribute**:



Use your browser's search function to locate specific attributes.

Click **Select All** to move all the attributes to the Selected Attributes list.

4. (Optional) Click an attribute in the **Selected Attributes** list and provide a friendly name and description for it:



Click **Remove All** to clear all attributes except the required ones from the list.

5. Click **Save** to save your configuration.

   If you change your mind about the attributes your want, remove the ones you don't want (don't worry, they'll be added back to the Attributes list) and make new selections.

6. Click **Next** (**>**) to go to the step in configuring your connector API.

Editing the Fusion Applications Connector API

If you know that the resources available through the `describe` resource have changed, you can refresh it to see the most up-to-date list of resources.

> **Note:**
>
> As long as the Fusion Applications connector API is in Draft state, you can edit the connector configuration

1. Click the **Resources** navigation link.

   The page displays only the resources you originally selected.

2. Click **Refresh**.

   When you click **Refresh** on the Resources page, you'll be told that the current resources will be discarded. If you click **Confirm** in the dialog, you'll be taken back to the initial view of the Resources page, where you'll have to re-enter the Describe URL and your design time credentials. The URL is re-queried and the latest resources are then displayed. The refresh action doesn't change any of the resource selections, friendly names, or descriptions that you've already provided. However, if you connect to a different service by entering a different Describe URL, you'll see a completely new set of resources and you'll have to provide friendly names for the ones you select.

3. Confirm the refresh action.

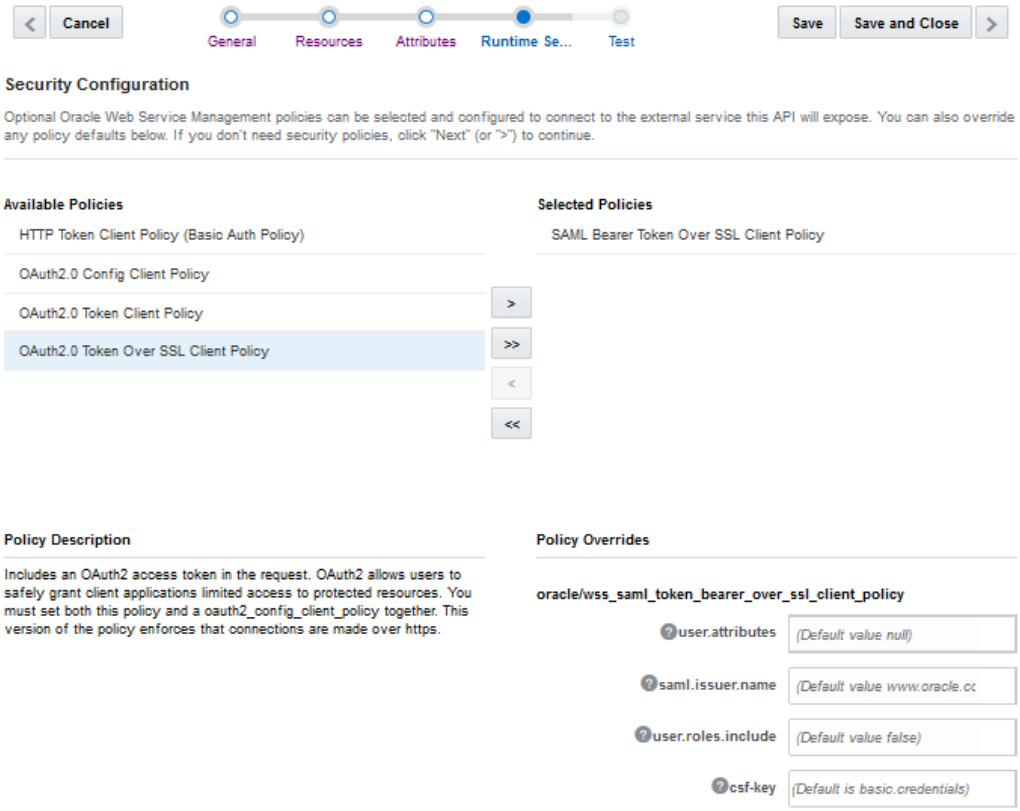
   The Resources page is displayed at the authentication phase. The Describe URL and the design time credentials you provided previously are shown.

4. Click **Connect** to reconnect to the Fusion Applications service or enter a new Describe URL and your design time credentials if you want to change to a different Fusion Applications service.
5. Change the enabled settings for the resources as needed.

   If you reconnected to the same service, your previous selections are kept.
6. Click **Save**.
7. Click **Next** (**>**) to go to the step in configuring your connector API.

Setting Runtime Security for the Fusion Applications Connector API

The Fusion Applications service determines the security policies used by the service. You have the option of selecting the corresponding client policies for the connector API from the Runtime Security page.



The Fusion Applications Connector API supports OAuth Authentication, HTTP Basic Authentication, and Security Assertion Markup Language (SAML). To learn more about these policies, see Security Policy Types for Fusion Applications Connector APIs.

1. Click the **Runtime Security** navigation link.
2. Select one or more security policies and move them to the Selected Policies column.

   When you select a policy, you can see its description below the Available Policies panel.

3. Specify values for the policy overrides for each policy (if applicable) if you don't want to use the default values.

   To override a property, enter or select a value other than the default. For a description of policy properties, see Security Policy Properties.
   To set a Credential Store Framework (CSF) Key value, see Providing a CSF Key.

4. Click **Save** to save your work or **Save and Close** to save your work and exit the Fusion Applications Connector API wizard.

5. Click **Next** (**>**) to go to the next step, testing the connector.

Providing runtime security credentials is necessary when you're configuring the connector, but you don't want customers who run the mobile apps that use Fusion-based services to have to enter credentials to access the services every time they sign in to the app. To see how to allow mobile app users to sign in once, see Configuring Oracle Cloud Applications as the Identity Provider.

Providing a CSF Key

You must set the **csf-key** property with your runtime credentials to allow you access and test the active integration.

Provide a CSF Key in one of the following ways:

- Select an existing key from the Available Keys list in the Select or Create a New API Key dialog. A description of the selected key is displayed below the list.

  When you select the key, its name appears in the Key Name field. Click **Select** to add the key. The other fields in the CSF Key Details pane are used only when creating a key.

- Click **New Key** in the dialog and create a new basic (CSF) credentials key as described in Create a New CSF Key.

To learn about CSF keys, see CSF Keys and Web Service Certificates.

Creating a New CSF Key

1. Click the keys icon in the **csf-key** field.

2. Click **New Key** in the Select or Create a New API KEy dialog box.

3. Enter a key name that is descriptive and easy-to-read. Note that after you create the key, you can't change the key name.

4. Enter a brief description of the key's purpose.

5. Enter your runtime credentials for the service to which you are connecting.

   Contact your Fusion Applications administrator to obtain the credentials used to call the Oracle Fusion Applications service at runtime. Most likely, you'll only need to do this once for each Fusion Applications instance (all services are called with the same app credentials).

6. Repeat the password in the confirmation field.

7. Click **Save** to continue working in the dialog.

   The key name value appears as the override value on the Security page. Note that the value of the key that you create pertains only to the environment in which it's set.

If you want to edit some aspect of an existing CSF key, then select it from the Available Keys list and modify the fields as needed. To learn about CSF Keys, see CSF Keys and Web Service Certificates.

Setting a Web Service Certificate

Here the steps for setting the overrides for a Web Service certificate. However, for this release, don't override the values for `keystore.sig.csf.key` because `orakey` is the only valid value for all of these certificate keys.

1. Select a security policy.

   The properties for the policy are displayed in the Policy Overrides section.

2. Select an alias from the drop-down list in the field for the certificate key (certificate keys are denoted by the keystore prefix) and select an alias.

   Unlike CSF Keys, you can't modify a Web Service certificate. You can only select a different alias.

Only mobile cloud administrators can create a new Web Service Certificate. If you don't know the alias for the certificate you want, ask your mobile cloud administrator for the alias. To set CSF keys and certificates from the Administration page, see CSF Keys and Certificates..

Testing the Fusion Applications Connector API

When you've finished configuring your Fusion Applications Connector API, test the endpoints. You test one endpoint at a time.

1. Click the **Test** navigation link.
2. Select the endpoint you want to test.

   Endpoints are listed on the left side of the page. Enter a partial resource name in the filter field to narrow the list to make it easier to find the endpoint you want. When you select an endpoint, the method, the resource name, and the URI of service is displayed on right side of the page.

3. Set the default test credentials if you're in the design phase and just want to see if your endpoints are valid, or if you want to test multiple endpoints during the session. Otherwise, skip this step and fill out the fields in the Authentication section for each method you test.

   a. Click **Default Test Credentials** at the top of the page.

   b. Select a mobile backend to associate the API with and the version of the mobile backend.

   c. If both OAuth and HTTP Basic Authentication are enabled for the mobile backend, select one in the Authentication Method field to use for testing.

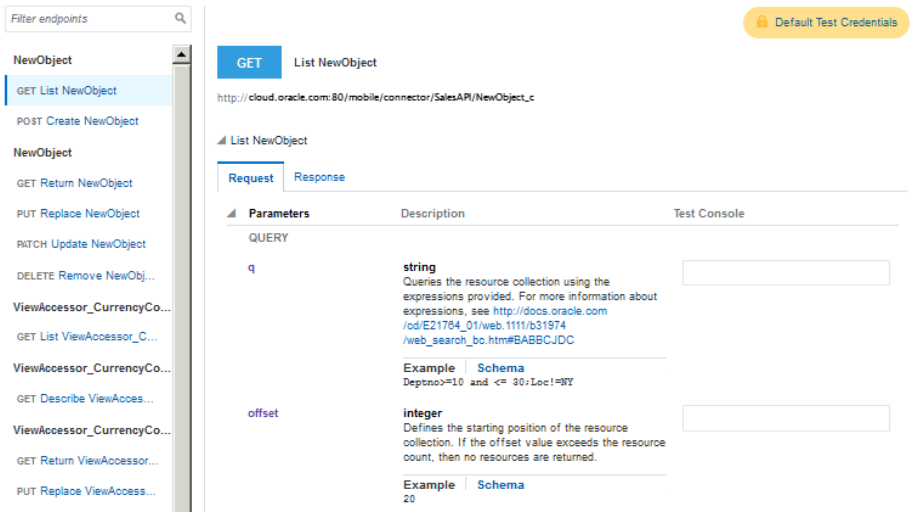
   d. Click **Save** to apply the credentials.

4. Click **Request** and expand **Parameters**.

   When you select a `GET` method, all the available query parameters are displayed on the Request tab.

   a. For a `GET` method, enter a parameter value.

      You can enter a value in the empty field next to the parameter description to test with or use the value, if any, provided in the example.



Ordinarily, when invoking Fusion Application services, you could use the `expand` parameter to include the data for a child resource in a response when querying the parent resource. However, in the Fusion Applications connector, `field` parameters are implicitly added to the requests sent to the Fusion Application service.

Note that the service is unable to handle the `field` parameters in the request and the `expand` parameter when both are used together.

To ensure that data for both the parent and child resources are included in the response, you must add field parameters that explicitly list the attributes for both parent and child. For example, let's say you had a parent resource, `employee`, with the attributes `FirstName` and `LastName` and the child resources, `directReports`, `assignments`, and `photo` with the respective attributes, `PersondId`, `AssignmentName`, and `Image`. You'd add a field parameter with the following values:

```
fields=FirstName, LastName; directReports:PersonId;
assignments:AssignmentName; photo:Image
```

If you do use the `field` parameter, be aware that the values that you provide in the parameter override the selections you made on the Attributes page.

b. (Optional) Click **Example** to view the example body, if one was provided. For methods other than `GET`, enter an alternate example to test with by clicking **Use Example**. The provided example body is copied into the text box. You can edit the example as needed.

c. (Optional) Click **Schema** to view the request body schema if one was provided.

5. Expand HTTP Headers and click **Add HTTP Header** to add a header.

   Select the header that you want to include for testing purposes and provide a value in the text field.

6. Expand **Authentication**, select the mobile backend and its version that are associated with this API, and enter your mobile user credentials. If both OAuth and

Http Basic Authentication are enabled for the mobile backend, select one in the Authentication Method field to use for testing.

7. Click **Response**.
8. Expand the status code and click **Example** or **Schema** to review the example or schema for the response body, if you provided one.
9. Click **Test Endpoint**.

   Test Endpoint toggles to **Cancel Test** when you click it. If you want to stop the test for any reason, then click **Cancel Test**.
   If you want to make changes to the testing parameters, click **Reset** to clear all the fields.

To be sure your connector API configuration is valid, you should test it thoroughly (not just from the Connector API Test page) before publishing it. You should also test the custom API (with its implementation) that uses this connector API. Essentially, if you're ready to publish the connector API, then you should also be ready to publish the custom API that calls it.
If you need to make changes to a connector API that's in the Published state, create a new version of it. For information on creating a new version, see Creating a New Version of a Connector.

Getting the Test Results

Test results are displayed at the bottom of the Test page. The result indicator is the response status:

- 2*xx*: indicates a successful connection
- 4*xx*: indicates a user error occurred
- 5*xx*: indicates a server error occurred

Status Code	Description
200 OK	Successful connection.
400 BAD REQUEST	General error when fulfilling the request, causing an invalid state, such as missing data or a validation error.
401 UNAUTHORIZED	Error due to missing or invalid authentication token.
404 NOT FOUND	Error due to an invalid connector ID. An associated connector with the given ID couldn't be found.
500 INTERNAL SERVER ERROR	General error when an exception is thrown on the server side.

Security Policy Types for Fusion Applications Connector APIs

You'll need to set a security policy to protect the information you want to send or receive. When determining what policies to set, consider whether the connection to the service involves transmitting proprietary or sensitive information. Adding a security policy ensures the authentication and authorization of the data transmitted.

From the Security page, you can select one or more Oracle Web Services Manager (Oracle WSM) security policies, including OAuth2, SAML, and HTTP Basic Authentication.

Security Policy Type	Description
OAuth2 and the Client Credential Flow	MCS supports OAuth2, a system where an Authentication server acts as a broker between a resource owner and the client who wants to access that resources. Of the different flows (security protocols) offered by OAuth2, the Client Credentials Grant Flow is used in MCS to secure connections. This flow is used when the client owns the resources (that is, the client is the resource owner).
HTTP Basic Authentication	HTTP Basic authentication allows an HTTP user agent to pass a user name and password with a request. It's often used with stateless clients, which pass their credentials on each request. It isn't the strongest form of security though as basic authentication transmits the password as plain text so it should only be used over an encrypted transport layer such as HTTPS.
Security Assertion Markup Language (SAML)	SAML is an XML-based open standard data format that allows the exchange of authentication and authorization credentials among a client, an identity provider, and a service provider. The client makes a request of the service provider. The service provider verifies the identity of the client from the identity provider. The identity provider obtains credentials from the client and passes an authentication token to the client, which the client then passes to the service provider. The identity provider verifies the validity of the token for the service provider and the service provider responds to the client.

For a list of the security policies supported for Fusion Applications Connector APIs, see Security Policies for Fusion Applications Connector APIs. For descriptions of security policy properties that can be overridden, see Security Policy Properties.

CSF Keys and Web Service Certificates

In MCS, the Oracle Credential Store Framework (CSF) is used to manage credentials in a secure form. A credential store is a repository of security data (credentials stored as keys) that certify the authority of users and system components. A credential can hold user name and password combinations, tickets, or public key certificates. This data is used during authentication and authorization.

CSF lets you store, retrieve, update, and delete credentials (security data) for a web service and other apps. A CSF key is a credentials key. It uses simple authentication (composed of the user name and the password for the system to which you're connecting) to generate a unique key value. You can select an existing CSF key or create one through the Select or Create a New API Key dialog. To select or create a CSF key, see Providing a CSF Key.

A Web Service Certificate allows the client to securely communicate with the web service. It can be a trusted certificate (that is, a certificate containing only a public key) or a certificate that contains both public and private key information. Web Service Certificates are stored in the Oracle WSM keystore. You set the overrides by selecting an alias from the drop-down list for the property, `keystore.sig.csf.key`. The alias for this property is mapped to the alias of the key used for signing. If no value is selected, the default value, `orakey`, is used (for this release, the only valid value for this property is `orakey`).

When you select a policy, you can see which properties are listed in the Policy Overrides.

> **Note:**
>
> It isn't necessary to set all the overrides for a policy; however, you should be familiar enough with the security policies that you've selected to know which overrides to set for each policy.

CSF keys, certificates, and their respective values are specific to the environment in which they're defined. That is, if there are multiple environments, A and B, and you're working in environment A, then only the CSF keys and certificates for the security policies in use by artifacts in that environment are listed in the CSF Keys dialog. A different set of keys and certificates will be displayed in environment B. It is also possible for keys with the same key name but with different values to exist in multiple environments.

A CSF key can be deployed to another environment, however, because CSF keys are unique to an environment, only the key name and description are carried over to the target environment. You won't be able to use that key in the new environment until it's been updated with user name and password credentials by the mobile cloud administrator.

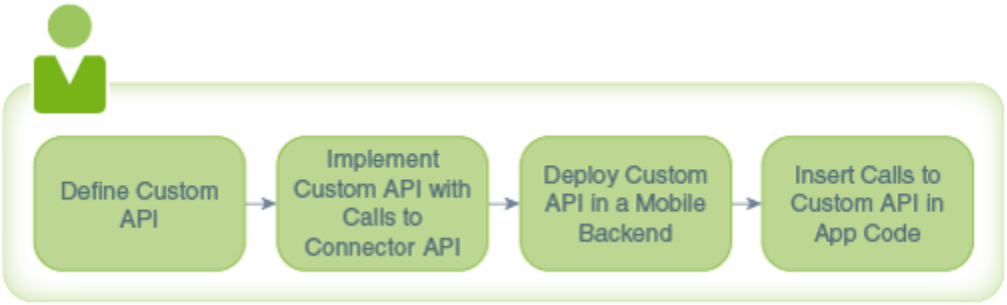
To set CSF keys and certificates from the Administration page, see CSF Keys and Certificates.

Using Your Fusion Application Connector API in an App

To use a connector in a mobile app, you first have to wrap calls to the connector API in a custom API and deploy that API. Such a custom API could also contain additional logic to process the data returned from the call to the connector.

This allows the app to access the connector's functionality by calling the custom API. The syntax for a call to a connector API is the same as you would use when calling any other API from custom API implementation code. See Calling Connector APIs from Custom Code.

Alternatively, you can do this automatically. See Generating Custom APIs for Connectors.



You make calls to connector APIs using JavaScript code in the custom API's implementation. When you implement a custom API, you can view the available connectors and their details in a special version of the API Catalog that's available to custom APIs. (The API Catalog that's available to client apps doesn't contain connector APIs.)

Troubleshooting Fusion Applications Connector APIs

A great source of debugging information are the system message logs. Depending on your role, you or your mobile cloud administrator can go to the Administration view and click **Logs** to see any system error messages or click **Request History** to view the client (4*xx*) and server (5*xx*) HTTP error codes for the API's endpoints and the outbound connector calls made within a single mobile backend.

By default, only TLSv1.1 and TLSv1.2 protocols are used for outbound connections. If you need to use an older version of a SSL protocol to connect to an external system that doesn't support the latest versions of SSL, you can specify the SSL protocol to use for the connector by setting the `Security_TransportSecurityProtocols` environment policy. The policy takes a comma-separated list of TLS/SSL protocols, for example: TLSv1, TLSv1.1, TLSv1.2. Any extra space around the protocol names is ignored. You can use the SSLv2Hello protocol to debug connectivity issues with legacy systems that don't support any TLS protocol. Note that this policy can't be used to enable SSLv3 endpoints. See Environment Policies and Their Values for a description of the policy and the supported values. Be aware that this policy must be manually added to a `policies.properties` file that you intend to export.

> **Caution:**
>
> Be aware when setting the policy that older protocols are vulnerable to security exploits.

You won't be able to test a Fusion Applications connector that hasn't been modified since June 2017 unless you save the connector first. Saving the connector regenerates the RAML from the descriptor. You can see when the connector was last modified by selecting it on the Connectors page and expanding the **History** panel.

Part VI
Deployment and Lifecycle

This part contains the following chapters:

28

MCS Environments

Your team's Oracle Mobile Cloud Service (MCS) subscription consists of at least one environment and offers distinct roles for team members.

As a mobile cloud administrator, you manage these environment service instances, their associated policies, and the permissions that team members have within them. To create and structure environments, see Create Mobile Environment Service Instances.

Team Members

In most cases, developing a mobile application is a team effort. Mobile application developers work closely with service developers and support personnel such as program managers and administrators. MCS makes the complexities of a large software development project easier to manage.

Depending on their assigned roles within an environment, team members can develop mobile backends, custom code, custom APIs, and use built-in services, such as Notifications, Storage, Analytics, and more. You can give team members wide access to features, environments, and user information, or restrict them to a small set of permissions. You must be an Oracle Cloud identity domain administrator to manage MCS team member roles. To control the access of your team members and manage the steps of your product development process using environments and roles, see Assign MCS Team Member Roles.

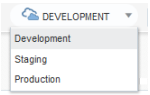
To see how team members access the MCS UI, see What is My Environment? As a mobile cloud administrator, you can manage artifacts and permissions in the environment from the Administration UI as described in Administration View.

> **Note:**
>
> Team members are users with access to MCS development features; these users are different from mobile users, the end users that access mobile applications running on MCS. To manage mobile users, see Set Up Mobile Users, Realms and Roles.

What is My Environment?

An **environment** is a predefined arena for working in MCS. Depending on your implementation of MCS and your role, you have access to one or more environments.

If you have access to multiple environments, you can always tell which environment you're in by looking at the environment field at the top of the page:



To access environments, see Your Work Environment.

Each environment has its own set of policies that govern the behavior of artifacts within that specific environment. To learn about environment policies, see Environment Policies. CSF keys and security certificates are also unique to each environment, see CSF Keys.

Your Work Environment

You work in one environment at a time. The first time you log in to MCS, you're put in the default environment set by the mobile cloud administrator. If you don't have permission to access the default environment but you do have access to another environment, that environment is displayed instead. If MCS can't determine which environment to open, an environment selector dialog opens that lets you select the environment.

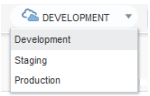
After you're logged in to MCS, you can select the environment you want to work in from the environment drop-down list. The list only includes environments that you have permission to access. To switch environments, see Changing Environments.

When you open the MCS side menu (click ☰) and select an artifact from Applications (for example Mobile Backends or Collections), you go to the top-level page for that artifact type where you'll see a list of those artifacts in the current environment. When you select an artifact in the list, details for that artifact are displayed. See Managing an Artifact's Lifecycle.

When you log in to MCS again, you're put in the environment that you were in when your previous session was closed.

Changing Environments

You can always see which environment you're in by looking at the top of the MCS page. If you have access to multiple environments, you'll see a drop-down list next to the environment name.



If you need access to an environment that you don't see in the list, contact your mobile cloud administrator. See Changing Environment Views for the Administrator.

To change environments, click the environment name in the drop-down list. Be sure to save any unfinished work before switching environments.

What Happens When You Change Environments?

This discussion presumes your instance of MCS has more than one environment. When you switch to another environment, you'll be moved to the same page that

you're currently viewing. That is, if you were on the Collections page in Environment A (the source environment), you'll be on the Collections page when you move to Environment B (the target environment).

You won't be able to access an artifact in another environment if one of the following conditions exist:

- The target environment you want to switch to has been deleted.

  For example, you've got an old bookmark to the details page for `collection123` in Environment B. When you go to the bookmark, you're told Environment B can't be located. You're no longer in any environment. Don't worry, you've got some options:

  – You can select another environment. The Alternative environment field lists all the environments for which you have access permission.

    If `collection123` exists in the environment that you select, you'll be taken to the details page of `collection123` in that environment.

  – You can click **Home** in the side menu to go back to the default environment.

    In the default environment, you can click  and select **Applications > Storage** from the side menu to view the list of collections. If you don't see Storage under Applications, you need to request access permission to collections in the default environment from your mobile cloud administrator.

- You don't have access permission for the target environment.

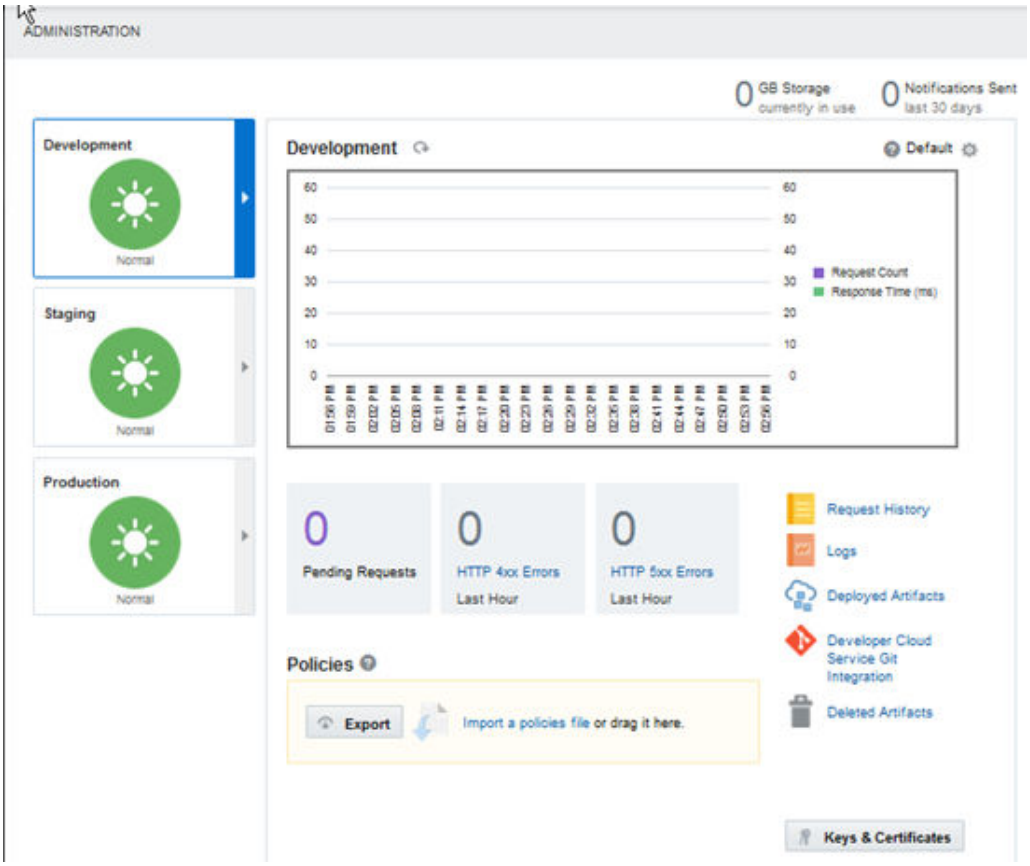
  For example, you get a link to view the details for `collection123` in Environment B but you're told that you can't access the environment. You can select an alternative environment from the list of environments that you can access. Otherwise, contact your mobile cloud administrator to get access permission for the target environment.

- You don't have permission to view the artifact in the target environment.

  For example, in Environment A you might be viewing analytical details on new users. You switch to Environment B to see analytic details for new users in that environment, but you're told that you don't have permissions to view that information in Environment B. You need to request read permission from your mobile cloud administrator for the artifact in the target environment or go to another environment.

Administration View

If you're a mobile cloud administrator, you can go to the Administration view by clicking  and selecting **Administration** from the side menu. From the Administration view, you can examine the details of an environment, such as its policies, CSF keys and certificates, logs, and more. If you have a multi-environment instance of MCS, you can click on a environment tab to see its details. If you're viewing the default environment, you'll see `Default` in the upper right corner above the chart.



The green (healthy), amber (severe), and red (adverse) traffic-light indicators give you a bird's eye view of how quickly and successfully requests are processed within each environment over the last minute.

When you select an environment, the page plots a grouped bar chart comparing the number of requests against response times. It also displays the number of client (4xx) and server (5xx) errors and the number of pending requests for the selected environment. When the indicators signal that the health of an environment has declined from healthy to severe or even adverse, you can drill down from this page to quickly diagnose the cause by viewing detailed logging information and error messages. For details about diagnostics, see Diagnostics.

Each environment has its own set of policies and permissions. Policies govern the behavior of artifacts within a specific environment. You can view the environment policy settings for a particular environment by clicking **Export** in a selected environment. See Environment Policies.

Permissions let you control which team members have access to which environments and to specific functionality in those environments. You can assign access permissions when you create a role. See Assign MCS Team Member Roles.

CSF keys and security certificates are also unique to each environment. See CSF Keys and Certificates.

Changing Environment Views for the Administrator

If you have a multi-environment instance of MCS, you can easily and quickly view details specific to an environment from the Administration view.

1. Click and select **Administration** from the side menu.
2. Click an environment tab to see information about it such as analytics or policies.

You can quickly verify whether the environment you're currently viewing is the default environment by looking at the upper right corner of the Administration page.

Setting the Default Environment

If you have a multi-environment instance of MCS, you can set a default environment for users when they access MCS for the first time:

1. Click and select **Administration** from the MCS side menu.
2. Select an environment.
3. Click **Settings** next to the Default indicator.
4. (Optional) Enter a name if you want to change the current environment name.

   The name should be 30 characters or less. You can use uppercase and lowercase letters, the integers 1 through 9, and spaces.
5. Select **Set as default for first-time user access**.

Environment Policies

Environment policies are environment settings and artifact properties that are specific to a particular environment. They let you override given values per environment.

Some things to note about environment policies:

- Policies can always be modified.
- Policies can never be published.
- Policies can't be deployed across environments. The same policy can exist in multiple environments at the same time but each instance of that policy is unique to the environment containing it. For example, if your instance of MCS had Development, Staging, and Production environments, they can all have a `Network_HttpRequestTimeout` policy, but the policy in the Staging environment is applicable only to the APIs in the Staging environment.

Even though an artifact can't be changed after it's published, its behavior can still be affected after it's deployed to another environment. If you're a mobile cloud administrator, you have the flexibility to adjust policy values to better fit the required behavior of artifacts in a particular environment. For instance, if you're in a development environment, you may be interested only in verifying that a connector's endpoints are valid. The timeout values aren't much of a concern during the experimental phase, but when you deploy that connector to the runtime environment, it's available for wide use and realistic timeout settings are required. You'll want to adjust the value of the `Network_HttpRequestTimeout` policy in the runtime environment accordingly.

Only mobile cloud administrators can deploy an artifact and, therefore are the only ones who can modify environment policies. You can modify policies when you deploy an artifact (mobile backend, custom API, API implementation, connector, collection, or realm) or by editing the `policies.properties` file for an environment directly from the Administration view. To change policy settings, see Modifying an Environment Policy. For descriptions of environment policies and their values, see Oracle Mobile Cloud Service Environment Policies.

In general, you should use the policy's default settings. Changing the setting of a policy that has an environment scope can have adverse results. When you set a policy at the environment level, the value for that policy is applied to the relevant artifacts in that environment. For example, if you set the value of the `Sync_CollectionTimeToLive` policy to a value other than the default in the runtime environment, that value is applied to all mobile backends in that environment.

For information on a policy's scope, see Environment Policy Scopes.

Environment Policy Names

A policy is simply a name-value pair. A policy name has the format `mbeArtifactIdentity.invocableArtifactIdentity.policyPropertyName`, and each part of the name has the following function:

- The first `mbeArtifactIdentity` binds the policy value to a specific mobile backend.
- The second `invocableArtifactIdentity` binds the policy to an API, an API implementation, or connector.
- The `policyPropertyName` is the short name of the policy, which usually indicates the function of the policy, for example, `Logging_Level` or `User_DefaultUserRole` (note that property names are initial-capped and preceded by a category).

You can set `mbeArtifactIdentity` and `invocableArtifactIdentity` in one of the following ways:

- The artifact name and version, which matches the artifact of a particular type with the given name and its version, for example, `myMobileBackend(1.0.0)`.
- The artifact name alone, which matches all artifacts of the same type with the given name and any version, for example, `myMobileBackend`.
- A wildcard denoted by an asterisk (*), which matches all artifacts of the particular type.

Qualifying API Types

When you define a policy that affects an API, you must use a fully qualified name (which consists of an API category and the API name). The API category can be one of `custom`, `connector`, `platform`, or `system`. For example:

- `*.custom/myAPI(1.0)`
- `*.connector/myAPI(1.0)`
- `*.platform/myAPI(1.0)`
- `*.system/myAPI(1.0)`

Policy names for API implementations do not include category types. For example, `*.myAPIImpl(1.0)`.

Examples of Policy Names

Here are some examples of policy names and their meanings:

- `myMBE(1.0).custom/myAPI(1.0).someProperty` is applied to calls of a custom API or API implementation named `myAPI` that has a version of 1.0 within the context of a mobile backend named `myMBE` with a version of 1.0.
- `myMBE(1.0).connector/myAPI(1.0).someProperty` is applied to calls of a connector API or API implementation named `myAPI` that has a version of 1.0 within the context of a mobile backend named `myMBE` with a version of 1.0.
- `myMBE.custom/myAPI(1.0).someProperty` is applied to calls of a custom API or API implementation named `myAPI` that has a version of 1.0 within the context of a mobile backend named `myMBE` of any version.
- `*.custom/myAPI(1.0).someProperty` is applied to calls of a custom API or API implementation named `myAPI` that has a version of 1.0 within the context of any mobile backend.
- `*.myAPIImpl(1.0).someProperty` is applied to calls of an API implementation named `myAPIImpl` that has a version of 1.0 within the context of any mobile backend.
- `*.*.someProperty` is applied to calls of any API or API Implementation within the context of any mobile backend. This name format denotes an environment wide policy.

Policy Property Names

Property names should follow the format `Category_PropertyName`. Property names should consist only of alphabetic characters and multi–part names should use CamelCase capitals, for example, `Network_HttpReadTimeout`. Names should not include the scope or an object type or unit of measure.

Environment Policy Scopes

When setting a policy, you should be aware of its scope. A policy's scope refers to the artifacts to which a policy setting applies. Each policy has a unique set of scopes for which it's valid. While some policies can be set at the environment-level and at the artifact-level, it may make more sense to set them at one particular level. For instance, the `Connector_Endpoint` policy stores the endpoint URL of a specific connector. This

policy must be set on an individual connector basis and should be set at the artifact-level. The `Network_HttpRequestTimeout` policy on the other hand affects APIs and connectors. It needs to be set at the environment-level so that it's applied to all APIs and connectors.

How you name a policy can determine its scope. Use the `mbeAssetIdentity` and the `invocableAssedIdentity` parts of the name to set whether a policy has an environment or artifact-level scope:

- **Environment scope:** Set both `mbeAssetIdentity` and the `invocableAssedIdentity` as wildcards so that the policy is applied globally across the environment.

  Example: `*.*.Logging_Level=800`

- **Mobile Backend scope:** Set `mbeAssetIdentity` to a specific mobile backend and set the `invocableAssedIdentity` as a wildcard so that the policy is applied to all APIs and connectors called in the context of the given mobile backend. You make this more specific by including the version of the mobile backend.

  Example: `MyMobilBackend(1.3).*.Logging_Level=800`

- **API scope:** Set `mbeAssetIdentity` to a wildcard and set the `invocableAssedIdentity` to a specific API so that the policy is applied to that particular API when called in the context of the any mobile backend in that environment. You make this more specific by including the version of the API.

  Example: `*.custom/MyApi(2.0).Logging_Level=800`

- **API Implementation scope:** Set `mbeAssetIdentity` to a wildcard and set the `invocableAssedIdentity` to a specific API implementation so that the policy is applied to that particular implementation when called in the context of the any mobile backend in that environment. You make this more specific by including the version of the API.

  Example: `*.MyApiImpl(1.0).Logging_Level=800`

- **Connector scope:** Set `mbeAssetIdentity` to a wildcard and set the `invocableAssedIdentity` to a specific connector so that the policy is applied to that particular connector when called in the context of the any mobile backend in that environment. You make this more specific by including the version of the API.

  Example: `*.connector/MyConnector(2.2).Logging_Level=800`

- **Fully-qualified API scope:** Set the `mbeAssetIdentity` to a specific mobile backend and set the `invocableAssedIdentity` to a specific API so that the policy is scoped at the fully qualified API level whenever the API is called within the scope of the given mobile backend. You make this more specific by including the versions of the mobile back and the API.

  Example: `MyMBE(1.3).custom/MyApi(2.0).Logging_Level=800`

- **Fully-qualified API Implementation scope:** Set the `mbeAssetIdentity` to a specific mobile backend and set the `invocableAssedIdentity` to a specific API Implementation so that the policy is scoped at the fully qualified API implementation level whenever the implementation is called within the scope of the given mobile backend. You make this more specific by including the versions of the mobile back and the API implementation.

  Example: `MyMobileBackend(1.3).MyApiImpl(1.0).Logging_Level=800`

- **Fully-qualified Connector scope:** Set the `mbeAssetIdentity` to a specific mobile backend and set the `invocableAssedIdentity` to a specific connector so that the policy is scoped at the fully qualified connector level whenever the connector is called within the scope of the given mobile backend. You make this more specific by including the versions of the mobile back and the connector.

  Example: `MyMobileBackend(1.3).connector/MyConnector(2.2).Logging_Level=800`

For details about policy name formats, see Environment Policy Names.

Modifying an Environment Policy

MCS generates a `policies.properties` file for you. If you're a mobile cloud administrator, you can modify an environment policy by editing its value, saving your changes, and importing the modified file from the Administration view.

1. Click  and select **Administration** from the side menu.
2. Click the environment for which you want to edit policy values.
3. Click **Export** under Policies.
4. Edit the policies as needed for the selected environment and save the file.
5. Import the modified `policies.properties`.

Here's an example of a `policies.properties` file in a development environment:

```
##--------------------------------------------------------
# MCS Policies. Snapshot from: development
#
#Thu Mar 19 17:40:19 EDT 2015
*.*.Asset_DefaultInitialVersion=1.0
*.*.CCC_Log_Body=false
*.*.CCC_Log_Body_MaxLength=512
*.*.CCC_ReclaimHighWaterMark=30
*.*.CCC_ReclaimLowWaterMark=20
*.*.Database_CreateTablesPolicy=allow
*.*.Database_MaxRows=1000
*.*.Database_QueryTimeout=20
*.*.Diagnostics_AverageRequestTimeErrorThreshold=6000
*.*.Diagnostics_AverageRequestTimeWarningThreshold=3000
*.*.Diagnostics_ExcludedHttpHeadersInLogs=Authorization,Cookie
*.*.Diagnostics_LongRequestCountErrorThreshold=10
*.*.Diagnostics_LongRequestCountWarningThreshold=0
*.*.Diagnostics_LongRequestThreshold=8000
*.*.Diagnostics_PendingRequestErrorThreshold=30
*.*.Diagnostics_PendingRequestWarningThreshold=15
*.*.Diagnostics_RequestCountErrorThreshold=10
*.*.Diagnostics_RequestCountWarningThreshold=0
*.*.Logging_Level=800
*.*.Network_HttpConnectTimeout=20000
*.*.Network_HttpReadTimeout=20000
*.*.Network_HttpRequestTimeout=40000
*.*.Notifications_DeviceCountWarningThreshold=70.0
*.*.Routing_DefaultImplementation=MockService(1.0)
*.*.Routing_MbeVersionFidelity=major.minor.qualifier.version
*.*.Sync_CollectionTimeToLive=86400
*.*.User_AllowDynamicUserSchema=true
*.*.User_DefaultUserRealm=Default(1.0)
*.RightNow(1.0).Routing_BindApiToImpl=RightNow(1.0)
*.RightNow(1.0).Security_OwsmPolicy=[]
*.UiTestApi(1.0).Routing_BindApiToImpl=uitestapis(1.0.0)
*.analytics(1.0).Routing_BindApiToImpl=analytics(1.0)
*.database(1.0).Routing_BindApiToImpl=database(1.0)
*.databaseManagement
(1.0).Routing_BindApiToImpl=databaseManagement(1.0)
*.devices(1.0).Routing_BindApiToImpl=devices(1.0)
*.incidentreport(1.0).Routing_BindApiToImpl=incidentreports
(1.0.0)
*.incidentreport(1.1).Routing_BindApiToImpl=incidentreports11
(1.0.0)
*.notifications(1.0).Routing_BindApiToImpl=notification(1.0)
*.storage(1.0).Routing_BindApiToImpl=storage(1.0)
*.sync(1.0).Routing_BindApiToImpl=sync(1.0)
*.ums(1.0).Routing_BindApiToImpl=ums(1.0)
*.users(1.0).Routing_BindApiToImpl=users(1.0)
```

Removing Environment Policies

Let's say you created an artifact some time ago that became obsolete and has been purged. You want to reduce clutter in the `policies.properties` file by removing policies defined for the obsolete artifact. You can delete the policies, however, the process involves deleting all the policies within a given environment and replacing them with the policies from the modified `policies.properties` file that you import. Be very careful when deleting policies to ensure you don't remove the wrong ones.

To remove environment policies:

1. Click  and select **Administration** from the side menu.
2. Be sure you're in the environment containing the policies you want to delete.
3. Click **Export**.
4. Make a copy of the exported `policies.properties` file.

   Important: If, after you've imported the modified file, you find you've accidentally deleted policies that you need, you can restore the environment back to its previous state by deleting the all policies and importing the backup copy of the file.
5. Delete the policies you want to remove and save the file.
6. Select **Delete all policies before import**.
7. Import the modified `policies.properties` file.

All the previous environment policies are deleted and the environment contains only the policies imported from your modified file.

CSF Keys and Certificates

MCS uses the Credential Store Framework (CSF) to manage credentials in a secure form. CSF lets you store, retrieve, update, and delete credentials for a web service and other apps.

CSF keys are credentials that certify the authority of users and system components that are used during authentication and authorization. A CSF key uses basic authentication (user name and password) to generate a unique key value.

Certificates, which are electronic documents that are used authenticate an individual or organization, are stored in different keystores by type. Other certificates that you can create using the CSF Keys & Certificates dialog are:

- Web Service Certificates, which can be trusted certificates (that is, a certificate containing only a public key) or a certificate that contains both public and private key information. Web Service Certificates are stored in the Oracle WSM Keystore.
- Token (Signing) Certificates, which are standard X509 certificates that are used to securely sign all tokens issued by a federation server.
- Secure Sockets Layer (SSL) Certificates, which are trusted certificates that you use to establish SSL communication with the external service. SSL Certificates are stored in the Trust Keystore.

You can also create a *token issuer*, which allows identity tokens provided by trusted third parties to be accepted and verified for authenticating mobile users. In the case of

virtual users, identity tokens provided by third-party providers are used to both authenticate and authorize mobile users.

CSF keys, certificates, and their values are specific to the environment in which they are defined. It's possible for keys with the same key name to exist with different values in multiple environments. Only the CSF keys and certificates in use in the selected environment are listed in the CSF Keys & Certificates dialog. A different set of keys and certificates can be displayed when this dialog is opened in another environment.

Token Certificates and Token Issuers

If you have identity tokens that are provided by third parties, you can add Token Issuer Certificate and Signing Authority Certificate information, along with any intermediate certificates, to establish a chain of trust. This lets MCS accept and verify identity tokens issued by third parties for authenticating mobile users.

Through the CSF Keys & Certificates dialog, you can do the following:

- Create a token certificate.
- Create a token issuer and associate certificates with that issuer.
- Create a data bound rule, a setting that specifies how third-party tokens are processed per certificate.

Viewing Available CSF Keys, Certificates, and Token Issuers

As administrator, you manage the credential keys and certificates used by service developers when setting security policies. You can view the details of CSF keys and the definitions of certificates from the CSF Keys & Certificates dialog. You can also edit the details of a CSF key (except for the key name and alias) and review the list of available token issuers.

To view a key, certificate, or token issuer:

1. Click  and select **Administration** from the side menu.
2. Click **Keys & Certificates**.
3. Click the **CSF Keys**, **Certificates**, or **Token Issuer** tab.
4. Select an alias in the **Available Keys** or **Available Certificates** list to view the details of the key or certificate.
    a. For CSF Keys, select **Show only referenced keys with null values** to see only keys that are referenced by artifacts that have no credentials values.

       If you edit the description, user name, or password of a CSF key (the key name can't be changed), click **Save** to save your changes and continue working in the dialog. Click **Save and Close** to save your actions and return to the main menu. Click **Cancel** to close the dialog without saving your changes.

    b. For Web Service, Token, and SSL Certificates, click **Export** to save the selected certificate to a file. You can then import the certificate for use in another instance.

To add more CSF keys or certificates, see Configuring a CSF Key, Configuring a Web Service or Token Certificate, and Configuring an SSL Certificate.

Configuring a CSF Key

You can configure a new CSF key from the **CSF Keys** tab in the CSF Keys & Certificates dialog.

1. Click the **CSF Keys** tab.
2. Click **Add** and provide the following values:
   - Unique key name. This name can't be changed after the key is created.
   - User name and password for the external system that requires this key for access.
3. Save the key.

Configuring a Web Service or Token Certificate

You can configure a new Web Service or Token Certificate from the **Web Service and Token Certificates** tab in the CSF Keys & Certificates dialog. You can't edit a certificate after you've created it.

1. Click the **Web Service and Token Certificates** tab.
2. Click **Add** and provide the following information:
   - **Alias** — Enter a unique name for the certificate.
   - **Content** — Copy the certificate definition into the text field. You can get Web Service certificate content from the system administrator of the service, or token certificate content from the remote identity provider.
3. Save the certificate.

> **Note:**
>
> When a certificate is uploaded, it takes a few seconds before the certificate is available. Token certificates can take up to ten minutes.

To delete a certificate, click **X** by the selected alias in the list of **Available Certificates**. You can only delete certificates that you created.

Configuring an SSL Certificate

You can configure a new SSL Certificate from the **SSL Certificates** tab in the CSF Keys & Certificates dialog. You can't edit a certificate after you've created it.

1. Click the **SSL Certificates** tab.
2. Click **Add** and provide the following information:
   - **Alias** — Enter a unique name for the certificate.
   - **Content** — Copy the certificate definition into the text field. You can get Web Service certificate content from the system administrator of the service, or token certificate content from the remote identity provider.
3. Save the certificate.

> **Note:**
>
> When a certificate is uploaded, it takes a few seconds before the certificate is available.

To delete a certificate, click **X** by the selected alias in the list of **Available Certificates**. You can only delete certificates that you created.

Disabling SSL Hostname Verification

Testing connectors can be difficult when they call an outbound service over SSL. If the SSL certificate has an incorrect or missing hostname, the developer might not be able to create the connector or might just have problems with testing.

You can make it easier to test a connector by turning off hostname verification for outbound SSL connections through the `Security_IgnoreHostnameVerification` policy.

> **Caution:**
>
> Turning off hostname verification is a security risk. Setting this policy to `true` should be limited to development. When testing is complete, set the policy back to its default value of `false`.

This policy is set globally (`*.*.Security_IgnoreHostnameVerification`) and will affect all connectors. Setting the scope for a specific backend or connector is not supported.

For more information on configuring policies, see Environment Policies.

> **Note:**
>
> Even if SSL hostname verification is disabled, you still need to import the SSL certificate if it's self-signed.

Adding a Token Issuer

To authenticate users with third-party tokens, you need to register the token issuers and associate them with their certificates.

After you've added at least one token certificate, use the steps below to add a token issuer from the **Keys & Certificates** dialog:

1. Click the **Token Issuers** tab.
2. Click **New Issuer**.
3. Enter the name of the token issuer in the **Name** field under **Issuer Details**.
4. Click **Add** (**+**) and select at least one name from the Select Certificate Subject Names dialog. All the certificates that have been uploaded are listed.

5. Save the token issuer.
6. If the list on the Token Issuers tab doesn't include your new issuer, click **Save** in the tab to update the list.
7. (Optional) Click **Rules** to configure a rule for a certificate subject name.

Rules for Certificate Subject Names

You set rules to define filters, which determine whether or not a given token is considered valid. The type of rules you can select depends on whether virtual users are enabled or not.

If virtual users are disabled, you can set a User Mapping rule that specifies how the token subject content is used to identify the user record in SIM.

If virtual users are enabled, you can set a Default Roles rules that lists one or more default roles that are applied to all requests for all users. You can also define a Role Attribute rule that can contain one or more attribute names. If the token role names are different from MCS role names, you can additionally set role mapping rules to match the token roles names to the MCS role names.

Filter rules can be applied regardless of whether virtual users are enabled or not. Filter rules accept or reject a token based on whether its content matches the information in the token certificate. To get a full description of all the rules, see Rule Types. Filter rules can be applied regardless of whether virtual users are enabled or not. Filter rules accept or reject a token based on whether its content matches the filter rules associated with the token certificate.

Configuring Rules

Rules govern how tokens provided by token issuers are processed. If the token provided by a token issuer doesn't meet the criteria specified by the rule, the request is rejected.

After you've added at least one token certificate and created a token issuer, you can configure a rule for the certificate subject name from the Token Issuer tab in the **Keys & Certificates** dialog.

1. On the Token Issuers tab, select a certificate subject name from the list.
2. Click **Rules**. As you add rules, the current number of rules is indicated on the Rules button.
3. Select **Enable Virtual User** if you're configuring rules for users that aren't registered.

   With virtual users enabled, a token identifies a user with a record in Oracle Cloud, but roles are associated with the user based on the default content in the token, instead of on information in that account.
4. Under **Add a New Rule**, select the rule type.
5. Enter the required values for the rule type.
6. Click **Add**.

If you need to change a rule, just select it, make the updates and click **Done**. To delete a rule, select the rule and click **X**.

Rule Types

Filter Rule

The Filter rule consists of a token attribute and at least one value that must match the value associated with the token. The `name-id` attribute represents the username identified in the token, while the `user.tenant.name` attribute represents the tenant name associated with the token.

Use a comma-separated list to enter multiple attribute values for either attribute. If none of the values match, the token is deemed invalid. A value can contain a wildcard (*) character.

For example:

- `name-id=jack, jill, ann`
- `user.tenant.name=testing, development`

You can configure only one Filter rule per token attribute (that is, you configure one Filter rule with the `name-id` attribute and one Filter rule with the `user.tenant.name` attribute).

User Mapping Rule

The User Mapping rule defines how tokens are mapped to users, either by user name or email address. This rule is applicable only to JWT tokens, only if virtual users are disabled.

The rule consists of a token attribute, `name-id`, that represents the username identified in the token, and a user attribute name value of either `uid` or `mail`:

- `uid` is the user's username in the associated Cloud Account (default behavior)
- `mail` is the user's email address in the associated Cloud Account

You can configure only one User Mapping rule per issuer certificate name. If you don't configure a User Mapping rule, name matching is used (the default behavior).

> **Note:**
>
> For SAML tokens the User Mapping rule type is ignored and the default behavior is to map the username in the token to the username in the associated record.

Default Role Rule

The Default Role rule defines a list of roles to associate with users. This rule is applicable only if virtual users are enabled.

The rule consists of a list of role names that are assigned to all users presenting tokens verified using the corresponding token certificate. Use a comma-separated list to enter multiple attribute values.

For example:`role=technician, manager, tester`

You can configure only one Default Role rule per issuer certificate name. If you don't configure a Default Role rule, no roles are assigned to the requesting user unless you've configured a Role Attribute rule.

Role Attribute Rule

Use the Role Attribute rule to determine which roles to assign by examining the attributes in the token. If a Role Attribute role is defined, the token is searched for attributes with names that match any of the values defined in the rule. If matches are detected, the values of those token attributes are interpreted as roles and assigned to the virtual user. This rule is applicable only if virtual users are enabled.

The rule consists of a comma-separated list of token attribute names used to derive the roles that are assigned to users.

For example: `employeelevel, QAgroup`

You can configure only one Role Attribute rule per issuer certificate name, but you can use this rule in combination with the Role Mapping rule. If you don't configure a Role Attribute rule, no roles are assigned to the requesting user unless you've configured a Default Role rule.

> **Note:**
>
> If you configure both the Default Role rule and the Role Attribute rule and the role attribute you defined is present in the token, the Default Role rule is ignored. However, if the defined role attribute isn't present, the roles specified in the Default Roles rule are applied to the virtual user. Role Mapping rules can also define which roles to use when no matches are found.

Role Mapping Rule

The Role Mapping rule associates roles with role attributes in the token identified by the Role Attribute rule. This rule is applicable only if virtual users are enabled.

The rule consists of an external role name, which is the value that should be found in one or more token attributes, and a list of roles to which the external role names are mapped. Use a comma-separated list to enter multiple attribute values.

For example: `employee=technician, manager, tester`

This example maps the external role name, `employee`, to the existing roles, `technician`, `manager`, `tester`.

> **Note:**
>
> Role Mapping rules only work in conjunction with Role Attribute rules. If no Role Attribute rule is defined, Role Mapping rules are ignored. If the names of the token attributes configured in the Role Attributes rule don't match the external role names configured in the Role Mapping rule, the token attributes are treated as role names and are assigned to the requesting user. If the role names defined in the rule don't correspond to any existing roles, the value is ignored.

You can configure as many Role Mapping rules per issuer certificate as you need, but only one rule can be configured for each external role name. To map one external role to multiple roles, use a single rule and include all the roles in a comma-separated list, as shown in the example above.

Rule Examples

Here are some examples of setting different types of rules for certificate subject names.

User Mapping Rule with a JWT Token, and a SIM User

The setup: You have a SIM user and a JWT token. The JWT token contains an attribute called Subject with the value, `mary.keane@fixitfast.com`.

In the SIM, there's a corresponding entry for the SIM user with the following attributes and values:

uid	mail	roles
`mkeane`	`mary.keane@fixitfast.com`	`manager`

You configure a User Mapping Rule with a token attribute value of `name-id` and user attribute value of `mail`.

What happens: The token is mapped to the SIM entry, `mary.keane@fixitfast.com`, and is assigned the role of `manager`.

User Mapping Rule, a SAML Token, and a SIM User

The setup: You have a SIM user and a SAML token. The SAML token contains an attribute called Subject with the value, `mary.keane@fixitfast.com`.

In the SIM, there's a corresponding entry for the SIM user with the following attributes and values:

uid	mail	roles
`mkeane`	`mary.keane@fixitfast.com`	`manager`

You configure a User Mapping Rule with a token attribute value of `name-id` and user attribute value of `mail`.

What happens: Remember, for SAML tokens, the User Mapping Rule is ignored and the default behavior of name matching is used, that is, you can only map the `name-id` value in the token to the `uid` attribute in the SIM. The `name-id` is mapped to `mkeane` and is assigned the role of `manager`.

Default Roles Rule and a Virtual User

The setup: You have a virtual user and a SAML token. The token contains a Subject attribute with the value `mary.keane`.

You configure a Default Role rule with the values `director` and `technician`, which are roles that have been defined in MCS.

Rule	Values
Default Role	director, technician

What happens: The virtual user is assigned the roles of `director` and `technician`.

Role Attributes Rule and a Virtual User

The setup: You have a virtual user and a SAML token. The token contains a Subject attribute with the value, `mary.keane` and a `Custom-Roles` attribute with the values, `director` and `technician`.

You configure a Role Attributes rule with a token attribute called `Custom-Roles`.

What happens: A match is found between the token role attribute and the MCS role you configured. The virtual user is assigned the roles of `director` and `technician`.

Role Attributes Rule with a Role Mapping Rule and a Virtual User

The setup: You have a virtual user and a SAML token. The token contains a Subject attribute with the value, `mary.keane` and a token attribute called `Custom-Roles` with the values, `director` and `technician`.

Rule	Values
Custom-Roles	director, technician

You configure a Role Attribute rule with a token attribute name also called`Custom-Roles` .

In addition, you configure a two Role Mapping rules, one with the external role name `director` with the values `supervisor` and `manager`, and another with the external role name of `technician` and the MCS role `developer`.

External Role Name	MCS Roles
director	supervisor, manager
technician	developer

What happens: The virtual user is assigned the roles of `supervisor`, `manager`, and `developer`.

What if you had configured only one Role Mapping rule, with the role name of `director` with the MCS roles of `supervisor` and `manager`?That is, you didn't configure a Role Mapping rule for the external role, `technician`.

What happens: The virtual user is assigned the roles of `supervisor` and `manager`. The `technician` role is ignored.

Native Builds

The mobile apps built with MAX can run within the container of the MAX App, or, after Mobile OS Native Build Service packages them as APK or IPA files, they can run as native apps as well.

Oracle Developer Cloud Service (ODCS) packages the mobile app metadata as APK and IPA files. Unlike the apps that run within the MAX APP, these apps run natively on iOS phones and Android devices. Despite this difference, users download these apps in the same way that they download the apps that run within the container: by scanning the QR code that the MAX Designer generates when it completes a test or production build. Whether you distribute the app through the Apple Store or Google Play (or in this case, through QR codes), you always need to get the platform-specific trust certificates that enable the app to run on the device. You can find out more about what you need to do to prepare your apps for release and distribute them in the Apple and Google documentation.

Besides the credentials for app distribution, you also need the following:

- The URL of your ODCS instance.
- The name of the native build service project in ODCS.
- The name and password that belong to the member of the build service project.

The general process setting up the user and the project are:

1. Logging into your Shared Identity Management (SIM) system as an administrator and then creating the user for the native build service. For example, create a user called *MAXBuildsUser*.

   > **Note:**
   >
   > This is not an actual user (meaning that this account doesn't belong to a human). This user is a safeguard that guarantees the availability of the native build service. If this were a real user, the build service would fail if the account is deactivated.

2. Adding the `DEVELOPER_USER` role to this user.
3. Logging into ODCS as an administrator and then creating a new project with a name like *MAXBuilds*.
4. Adding the build service user as a member of this project.

With this information and the platform-specific credentials in hand, you can now add native builds to each app that's created using MAX. That means that in addition to the QR code for the MAX App, MAX presents another set of QR codes for the native builds.

How Do I Enable Native Builds?

You enable MAX users to build and distribute native apps by configuring the Native Build Service as follows:

1. In the Admin page, click **Mobile OS Native Build Service**.
2. Enter the ODCS information:
   - Enter the base URL for your ODCS instance. For example, enter a URL like `https://developer.us.oraclecloud.com/xx-xx`.
   - Enter the project name.
   - Enter the project member name.
   - Enter the password.

3. Click **Test Connection**.
4. Add the platform security certificates.
    a. For Android:
        - Upload the keystore file.
        - Enter the Keystore password, Keystore Alias, and the Alias Password.
    b. For iOS:
        - Add the P12 Certificate.
        - Enter the P12 Certificate password.
        - Add the Provisioning Profile.
        - Enter the Certificate Common Name (CN).
5. Click **Save**. From this point forward, the builds will include QR codes for the native platforms.

29
Diagnostics

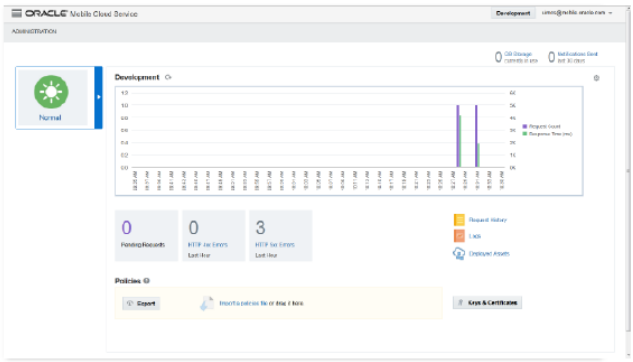
The Diagnostics features of MCS provide live performance data and quick access to detailed log messages for each API and connector request. If you are an administrator, you can use these features to monitor performance and error rates and to debug any problems that arise. If you are a developer, these features help you debug your code.

What Can I Do with Diagnostics?

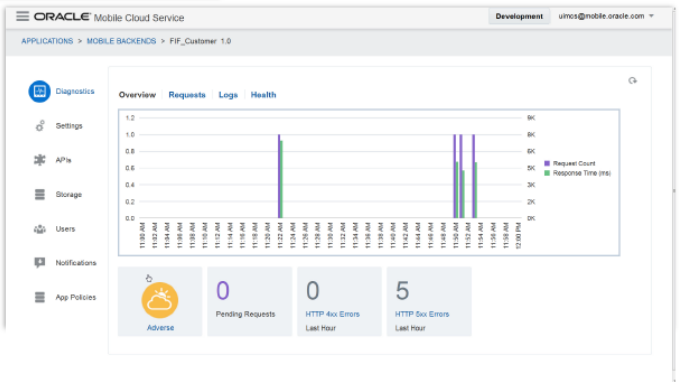
Whether you're a developer tracing errors in custom code, or an administrator who notices a flurry of 5xx responses, Diagnostics lets you easily find out what's going on by providing you with increasingly detailed levels of logging messages.

Diagnostics presents request and error data in two different views: the Environments page, used by mobile cloud administrators, and the Diagnostics page that is specific to a single mobile backend, which developers use.

Administrator View



Developer View



Both the Diagnostics page and Environments page enable troubleshooting for app responsiveness and errors. They provide a high-level view that includes traffic-light indicators that convey overall environmental health, a timeline that plots requests and

responses, and also counters to tally the failing requests resulting in HTTP 4xx and HTTP 5xx errors. These pages provide the entry point to more detailed levels of analysis, because you can drill down from an indicator or an error counter to identify which requests are failing and view log records that are associated with them. For more about going from these top-level pages to specific logs, see Viewing Underperforming Requests.

If you're a mobile cloud administrator who wants to go to the Environments page, click  and select **Administration** from the side menu.

If you're a developer who wants to go to the Diagnostics page, click  and select **Applications > Mobile Backends** from the side menu, then select your mobile backend and click **Open**.

Although the Environments page (administration view) and Diagnostics page (developer view) appear to be similar, they're used differently. As pointed out in Monitoring Environments for a Selected Mobile Backend, developers typically use a mobile backend's Diagnostics page as the starting point in their debugging efforts. To get an idea how developers go through their paces starting with this page, see Use Case: Using Correlation to Diagnose Custom Code. While developers focus on one mobile backend, mobile cloud administrators instead monitor all of the environments in the system. The same performance and error metrics that are displayed in a mobile backend's Diagnostics page are also available from the Environments page, though here they comprise the behavior of all of the backends deployed to a selected environment. For an example of how a mobile cloud administrator goes from this page to access logging data, see Use Case: Using Correlation to Diagnose Connector Issues.

Viewing Environment Health

The green, amber, and red traffic-light indicators in the Environments and Diagnostics pages depict the overall health of an environment for the last minute. MCS bases this at-a-glance view on the fine-grained health metrics for that environment. When the number of errors and current request or response times exceed configured thresholds, the traffic-light indicator changes from green (normal) to amber (adverse) or red (severe).

> **Note:**
>
> The throughput (and number of errors) varies by environment: development environments typically have more errors and lower throughput. In contrast, production environments are characterized by fewer errors and higher throughput. See Adjusting the Performance Threshold Configurations.

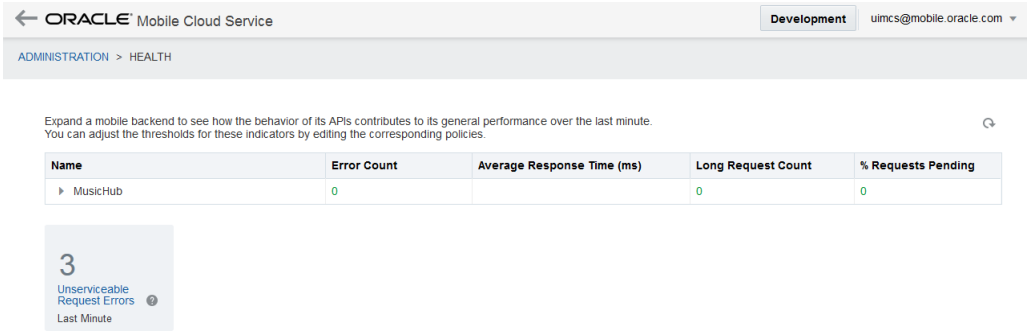
Viewing Server Load

As part of the overall portrait of health at any given moment, the Environments and Diagnostics pages include a timeline that plots a recent history of the number of requests and response times. The pages also include the number of pending requests.

Viewing Errors

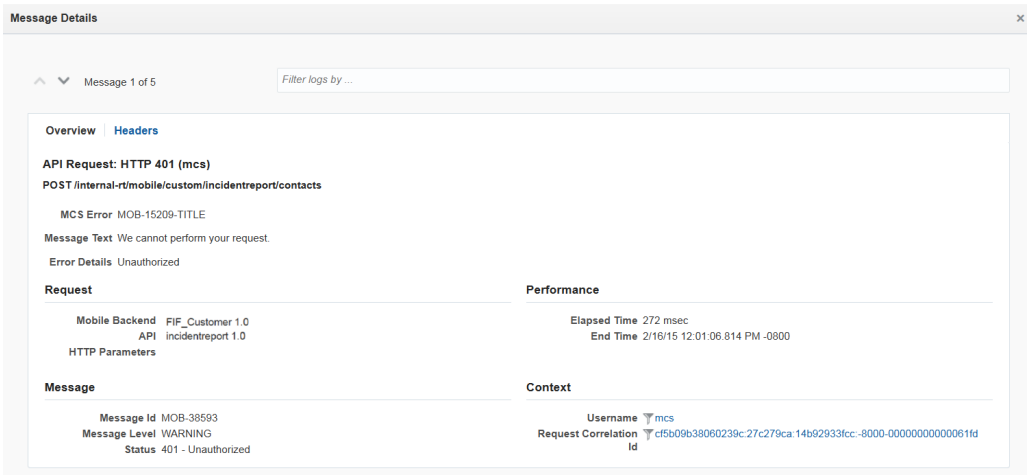
The Environments and Diagnostics pages note the number of client (4xx) and server (5xx) errors that have occurred within the last hour.

For the overall environment, MCS includes the number of unserviceable requests, errors which occur when requests fail to identify any mobile backend or API endpoint.



> **Note:**
>
> Unserviceable requests aren't associated with mobile backends. They are instead associated with an environment.

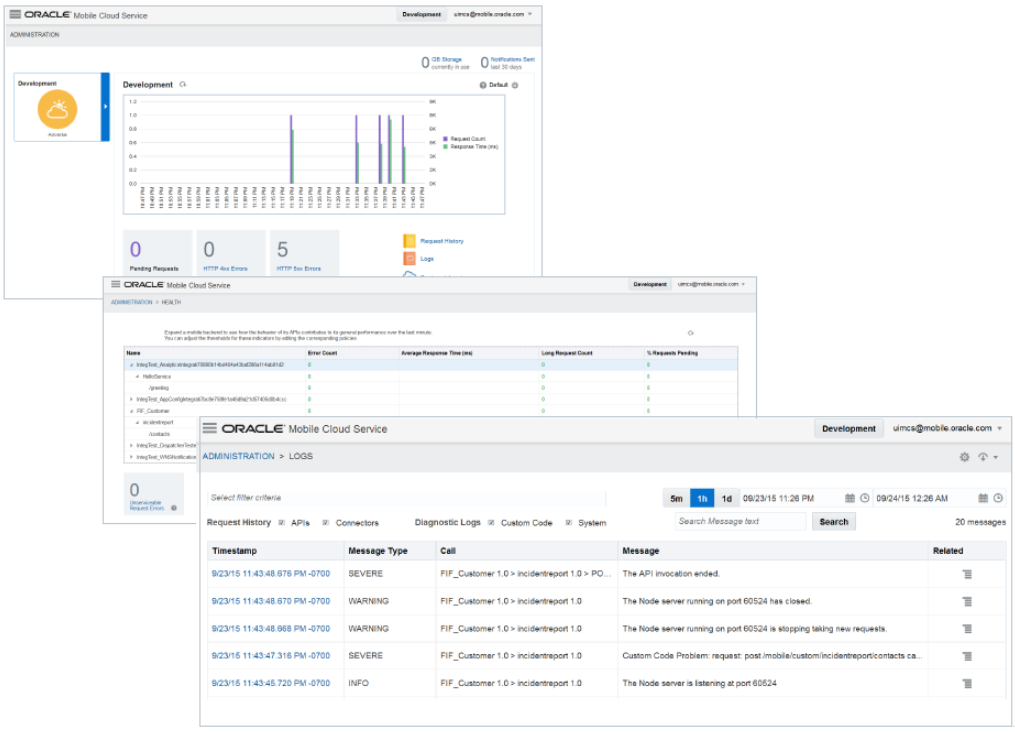
Increases in unserviceable requests may reflect a poor user experience, such as mobile apps sending requests to nonexistent mobile backends or APIs. These requests typically occur in development environments because the code is undergoing continual changes and has yet to be fully debugged or tested. As mobile backends are promoted to a production environments, however, fewer unserviceable requests are recorded by the system because the mobile backends' end-to-end code has been thoroughly tested. When unserviceable requests arise (in any environment), the MCS logging data identifies the unsupported API, endpoint, or incorrect mobile backend.



To find out more about request messages, see Viewing Status Codes for API Calls and Outbound Connector Calls.

Viewing Underperforming Requests

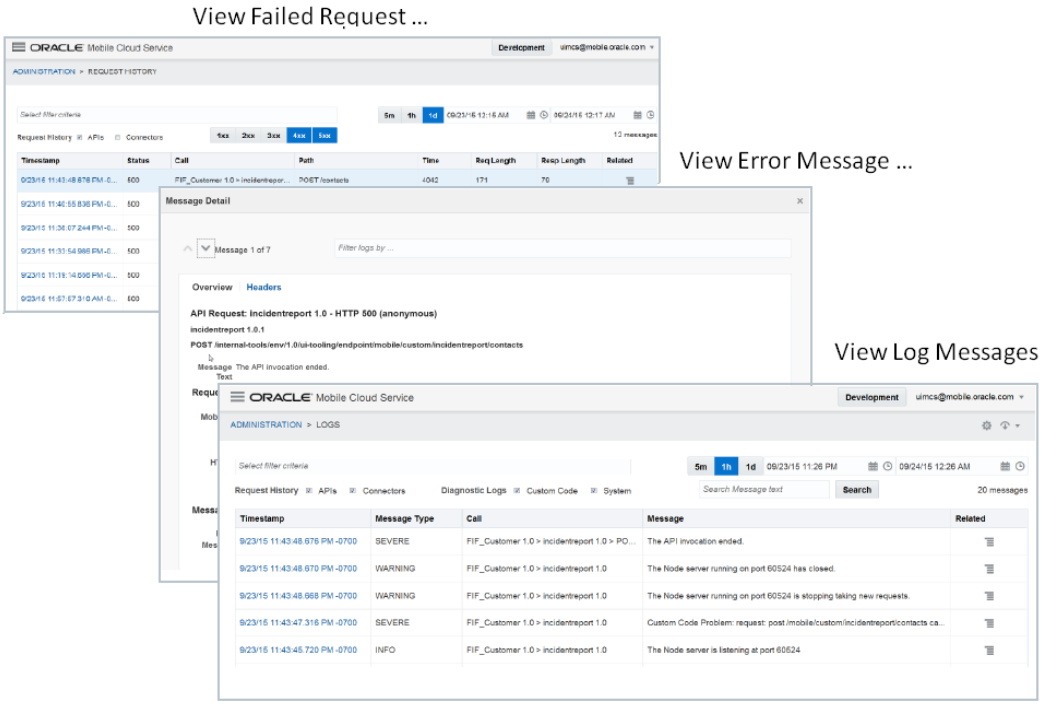
The high-level data presented on the Environments and Diagnostics pages is the entry point for increasingly detailed levels of analysis. When you hover over an indicator, the console lists metrics derived from the last minute of the system's behavior: the number of errors, the average duration of requests, the number of pending requests, and the number of long-running requests. The console highlights the severity of the problematic metric by color, from green (healthy) to red (severe). From here, you can evaluate the root cause by clicking the traffic light to investigate problematic requests or APIs. For example, if the environment indicator shows a high number of errors, then you can investigate the cause by drilling down to the error logs for information on each request that results in an error. When the number of slow requests increases, you can drill down from the environment indicator to see which API requests are slow. You then can view the API history log data and get a breakdown of the requests and any child requests. See also Viewing Log Messages Related to a Request.



Viewing Log Messages Related to a Request

Rather than using various `grep` commands to find log records between time stamps in the logs,MCS uses **correlation** to associate log messages to a specific API request to help you locate the pertinent records from the API request history. If you're troubleshooting, then correlation lets you quickly find the root cause by presenting detailed information, such as invalid JavaScript code or an unavailable resource called by a connector. For more information, see Relating Log Messages. For more

information about the various logs generated by Diagnostics (such as the API History, Connector History, Custom Code, and System logs), see Viewing Log Messages.

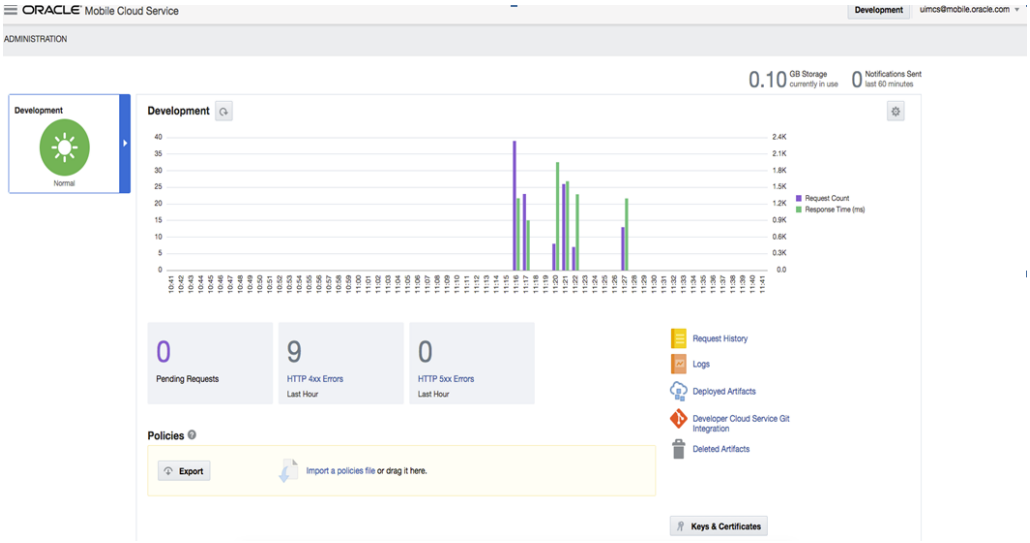


Viewing Storage Usage

In addition to showing API request data, the Environments page shows you information for your environment, in particular:

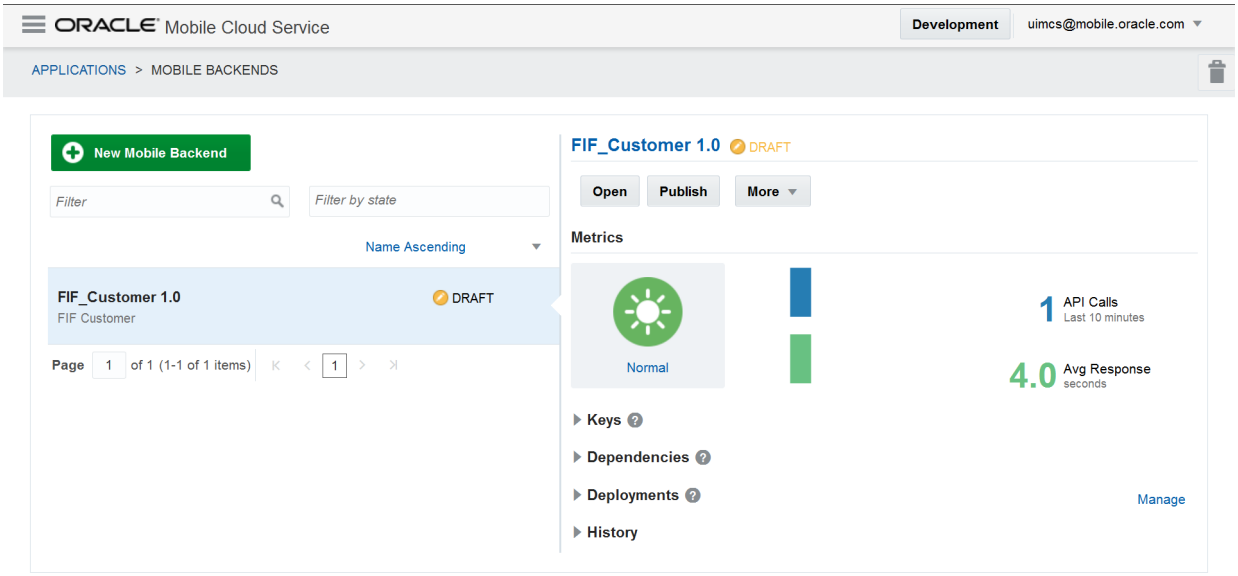
- Storage: How much database storage, shown in gigabytes, the environment is currently using
- Notifications: How many notifications have been sent in the past 60 minutes

You can see this information in the top right corner of the Environments page.

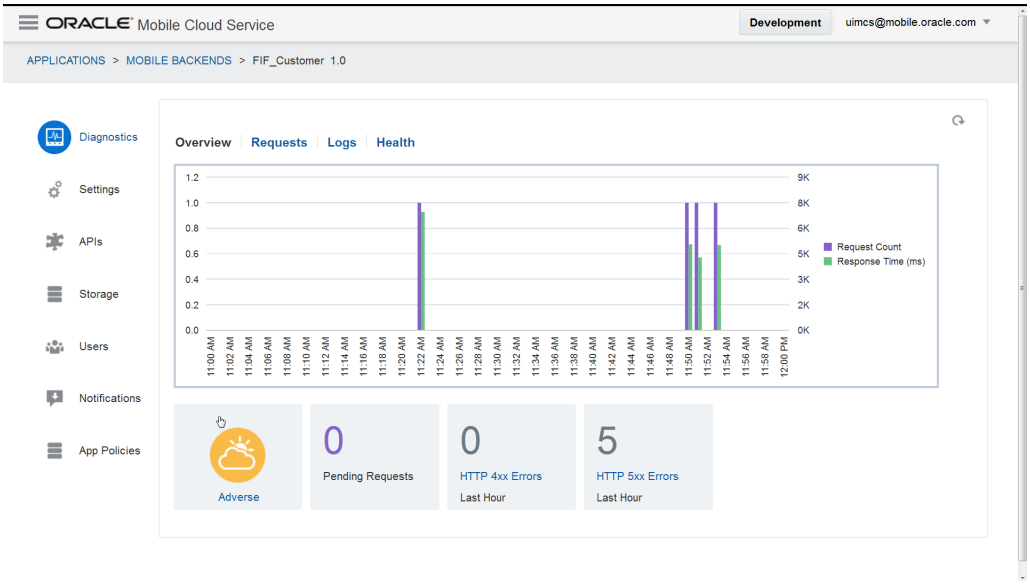


Monitoring a Selected Backend

The backend's summary page gives you a snapshot of the current health of its environment. You can take a deeper look at request and response processing and error handling by selecting the backend and then clicking **Open**.



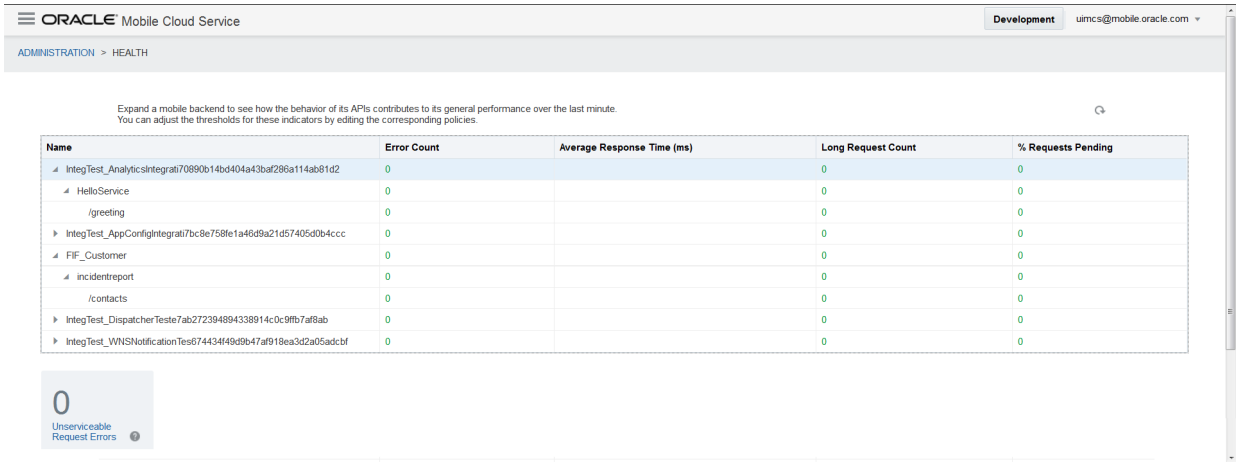
The Overview page displays the number of the requests and responses, plots them on a timeline, and notes the number of client and server (4xx and 5xx) errors. Because this page gives you a snapshot of the overall health of a mobile backend, you can focus your attention where its needed: on specific performance issues or problems with the API implementations and connectors used by the mobile backend.



While you can drill down through the Overview page to specific endpoint data, you can also view detailed API request and error information using the Health, Request History, and Logs pages.

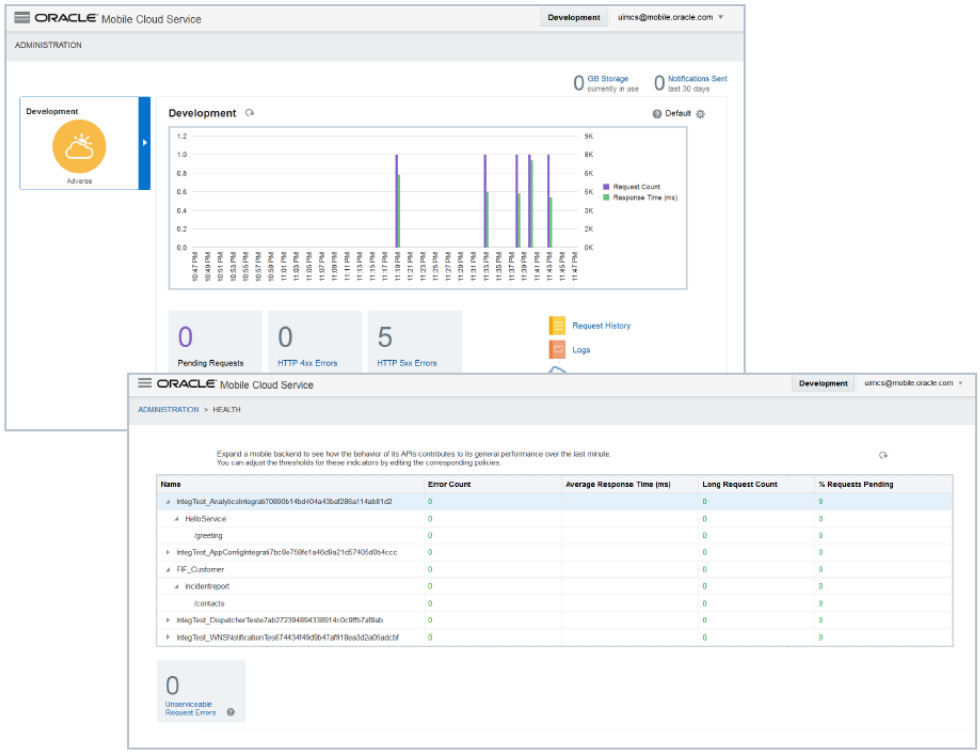
Viewing API Performance

You can find out how the performance of a specific API contributes to the overall health of a mobile backend, or to an entire environment. For each API, MCS records the same error and request handling metrics that it applies to mobile backends and environments. You can drill down to see how the API endpoints behave in terms of these performance metrics.



In the Development page, you can view the APIs for a selected mobile backend using the Health page. You can also open this page by clicking the traffic indicator on the Overview page. If the traffic indicator is amber or red, then you can quickly investigate the cause by going to this page. Similarly, if an indicator for an environment has

changed to amber or red in the mobile cloud administrator's Environments page, then you can open the Health Details page in a single click to find the problematic API.



What Do the Health Indicator Thresholds Mean?

Threshold	Amber	Red	Comments
Average Response Time	When the average response time within the last minute is greater than (or equal to) 3 seconds (3000 ms).	When the average response time within the last minute is greater than (or equal to) 6 seconds (6000 ms).	This threshold is applied across all requests (successful and failed) over the last minute. Even though the average response times may indicate a healthy environment, the Long Request Count might indicate that some of the requests aren't behaving well.

Threshold	Amber	Red	Comments
Long Requests	When any (that is, more than 0) long-running requests occur in the last minute. A long-running request to an endpoint server has a duration that's greater than (or equal to) 8 seconds (8000 ms). Depending on the environment, the default configuration may not reflect an adverse (amber) warning. While the default configuration triggers an adverse warning when a long request exceeds 8 seconds and the number of long requests has increased from 0 within the last minute, you might instead want to define an adverse warning when more than 10 long-running requests (which exceed 4 seconds) occur in the last minute.	When 10 or more long-running requests occur in the last minute.	To find out why requests may be running long or failing, review the custom code or the system at the far end of an outbound connector.

Threshold	Amber	Red	Comments
Percentage of Requests Pending	When the number of pending requests (expressed as a proportion of all requests over the last minute) is greater than (or equal to) 25%.	When the number of pending requests (expressed as a proportion of all requests over the last minute) is greater than (or equal to) 30%.	Pending requests represent the ratio of in-flight requests to the number of in-flight requests, as well as successful, and failed requests within the last minute. Pending requests don't necessarily indicate problems. They occur in both normally functioning, evenly loaded systems and also in erratic systems that are characterized by spikes in active requests. Evenly loaded and erratic systems require different proportions of pending requests. The proportion threshold that indicates a particular level of health in the request backlog may differ for these two types of systems; in evenly loaded systems, for example, MCS displays an adverse warning if the request backlog is 25% of active requests. While this default setting may be too lenient for erratically loaded systems, a reduction to 5% may be too severe because of spikes in active requests that jump to 10% may alternate with periods when no active requests are present. The thresholds set for the backlog depend on both the average request duration and the request density function over time. Remember that a high proportion of pending requests in erratically loaded systems doesn't signify problems as long as they're handled in a timely manner.
Failed Requests	When any failed requests occur in the last minute.	When 10 or more failed requests occur in the last minute.	When applying thresholds, Diagnostics counts of unserviceable requests are counted alongside failed requests.

Threshold	Amber	Red	Comments
Errors	When the error count in the last minute is greater than 0.	When there are 10 (or more) errors in the last minute.	
Unserviceable Requests	When any unserviceable requests occur in the last minute.		Diagnostics factors the thresholds for both failed requests and unserviceable requests into its assessment of the overall health of an environment. When unserviceable requests occur, review the requests made by the mobile app. Diagnostics might classify a request as unserviceable because it uses an incorrect URL. See also Viewing Status Codes for API Calls and Outbound Connector Calls.

Tip:

Ask your mobile cloud administrator to redefine these thresholds if they don't apply to your mobile backend or environment. For more information on the Diagnostics policies, see Oracle Cloud Service Environment Policies. See also Adjusting the Performance Threshold Configurations.

Note:

MCS provides preconfigured thresholds to determine the API health within the context of both environments and mobile backends.

These thresholds don’t apply to connector (endpoint server) requests.

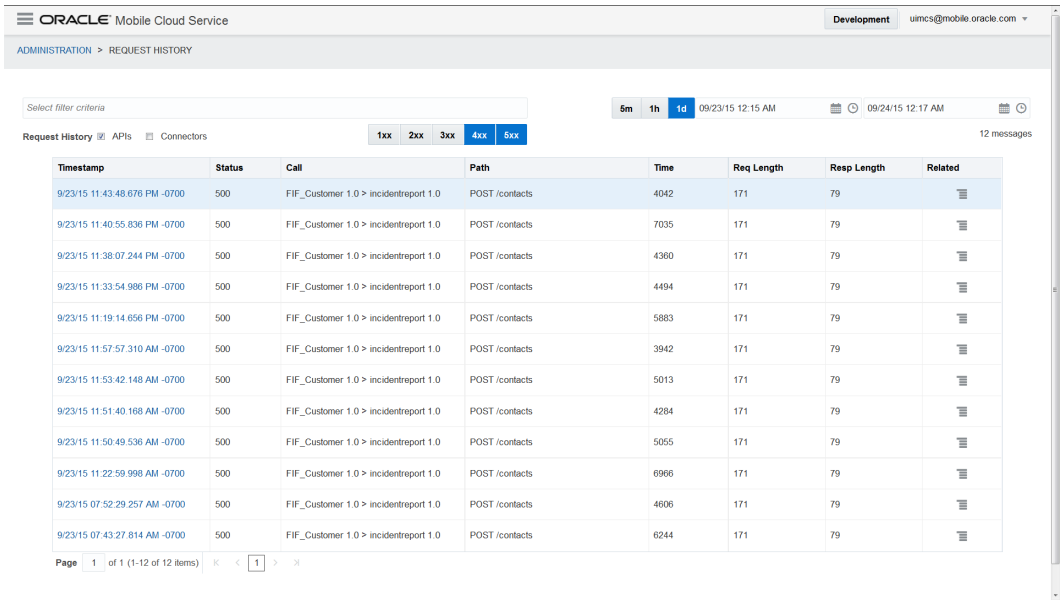
Adjusting the Performance Threshold Configurations

The default thresholds may not apply at all phases of the mobile backend's lifecycle and may not always reflect your interpretation of a healthy environment. For example, a development environment might be more tolerant of unserviceable requests than a production environment. Using the Environments page, mobile cloud administrators can obtain the policies file that contains the default configurations by clicking **Export**. After they adjust the thresholds, they can import the file by dragging it into the Policies pane.

See also Environment Policies.

Viewing Status Codes for API Calls and Outbound Connector Calls

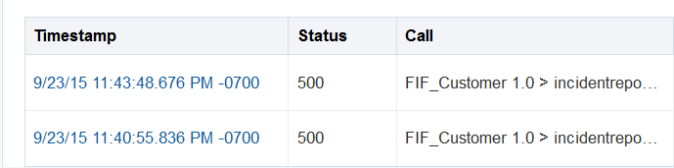
When you open the Request History page, its 4xx and 5xx status code buttons are selected by default, displaying the client (4xx) and server (5xx) HTTP status codes for the API's endpoints and the outbound connector calls made within a single backend (if you're a developer) or across all backends (if you're an administrator). This page gives you a glimpse into the context of the status code, letting you trace the causes for various status codes.



The Request History page displays a time stamp that indicates when the connector or API request was made and the resulting status code.

Tips:

- Clicking the time stamp opens the message itself.



Timestamp	Status	Call
9/23/15 11:43:48.676 PM -0700	500	FIF_Customer 1.0 > incidentrepo…
9/23/15 11:40:55.836 PM -0700	500	FIF_Customer 1.0 > incidentrepo…

See Viewing Message Details

You can learn more about the API call or outbound connector request by looking at the page's Call and Path columns, which show you a description of the targeted resource as well as the action and object of the request.

The table that lists the calls displays the sizes of the request and response in bytes as well as the response time. If a slow response time might indicate a problem, then you can troubleshoot the issue using correlation. See Viewing Log Messages Related to a Request.

Request Type	Content Displayed in the Call Column	Content Displayed in the Path Column
API requests that are returned 200 (Success)	The backend name, version > API name and version. For example: `FiFTechnician 1.1 > FiFReports 1.1`	The HTTP method with the resource path. For example: `GET /reports/{report}`
API requests that are returned 5xx (Unserviceable Requests) status codes	The backend name, version > API name and version (if available); otherwise this column is blank. `FiFCustomer 1.0 > incidentreports`	The HTTP method and information about the resource path. For example: `POST /contacts`
Outbound Call from a SOAP Connector	The endpoint URL, such as: `http://myhost.us.example.com:7002/mobilesvc/IncidentService`	The operation name. For example: `GET /incidents/{id}`
Outbound Call from a REST Connector	The host, such as:`maps.somecompanyapis.com`	The method with the resource path.

You can filter the display of error messages using any combination of the page's status code buttons and sort them in chronological or reverse-chronological order. While the default 4xx and 5xx buttons are toggled by default to display error codes, you can also view messages for informational (1xx), success (2xx) and redirection (3xx) codes. Common 4xx and 5xx codes include:

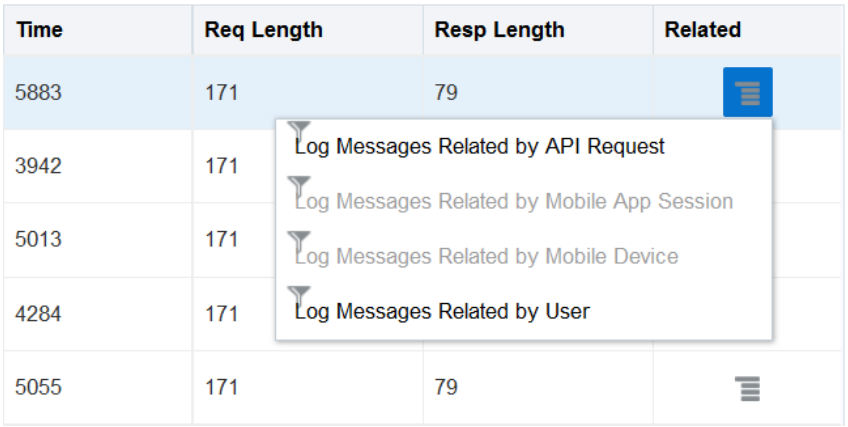
- `400 - Bad Request`

- 404 - Not Found
- 408 - Request Time Out
- 500 - Internal Server Error
- 501 - Not Implemented
- 503 - Service Unavailable

  For a complete list of HTTP status code definitions, see http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html.

Relating Log Messages

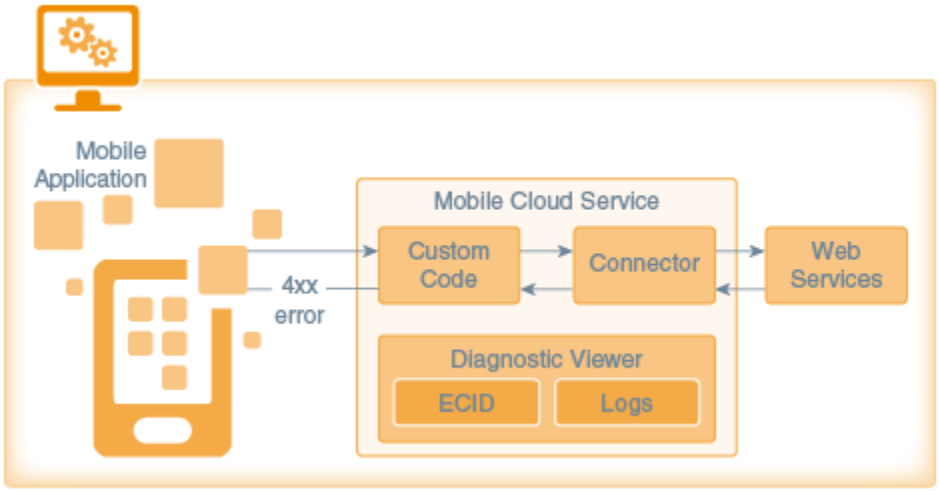
For each request, you can use correlation to get the logging data to a request by using the options in the Related Logs column. You can correlate log records by app session, mobile device, user, and API request.



To query a list of log records that are tagged with the correlation ID for the request, select **Log Messages Related by API Requests**. After you select this option, the **Filters** field is populated by the request's correlation ID. The messages displayed in the Logs page were generated during the servicing of the request.

> **Tip:**
>
> You can also generate a list of request-related messages by clicking the **funnel** next to **Request Correlation ID** in the Message Details page. See Viewing Message Details.

This ID provides additional correlation when you use the Oracle stack. For example, if you run systems on Oracle Fusion Middleware and use connectors to communicate with those systems, then all of the requests made will use the same correlation ID and can therefore be correlated with requests to the MCS server. See Diagnosing Custom Code.



How Client SDK Headers Enable Device and Session Diagnostics

When you use the client SDK for your mobile platform in your apps, the SDK injects the mobile diagnostic session ID (`M_DSID`) into request headers. Because the client SDK is optional, app developers can override this behavior by setting their own headers.

The `Oracle-Mobile-DEVICE-ID` and `Oracle-Mobile-SESSION-ID` headers, described in SDK Headers, enable Diagnostics to correlate records when you select the **Log Messages by Mobile Device** and **Log Records by Mobile App Session** options. While the server automatically generates the correlation ID for each request, the mobile app adds diagnostic capabilities by providing the session and device IDs. App developers can define how sessions are expressed. For example, they can group requests as a single session. App developers can also define the device ID to distinguish requests. A device ID isn't the device manufacturer ID, but rather an ID assigned by the developer to the user's device.

> **Note:**
>
> A single user can operate multiple devices that run the same app. The app may exhibit problems on only one of the devices.

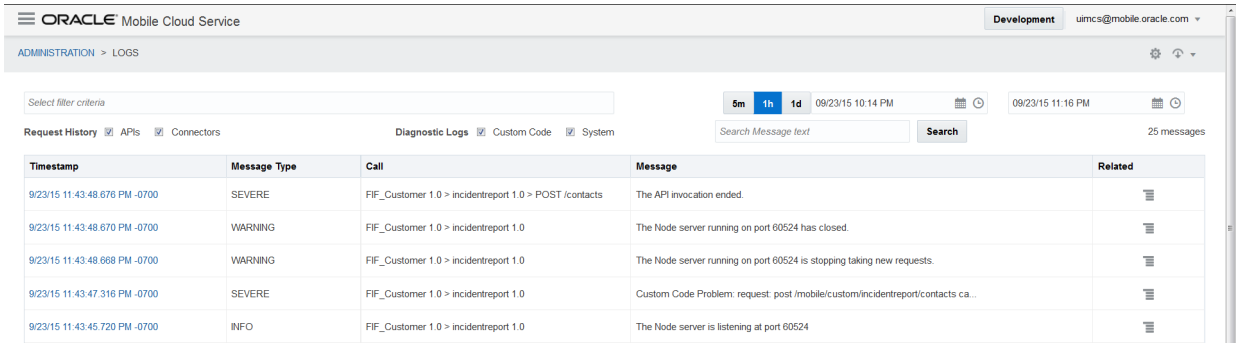
Administrators can use this ID to differentiate a request message that's specific to an app user's device amid thousands of other messages. Without this header, administrators can still correlate records by a user because users are established through authenticated requests.

Viewing Log Messages

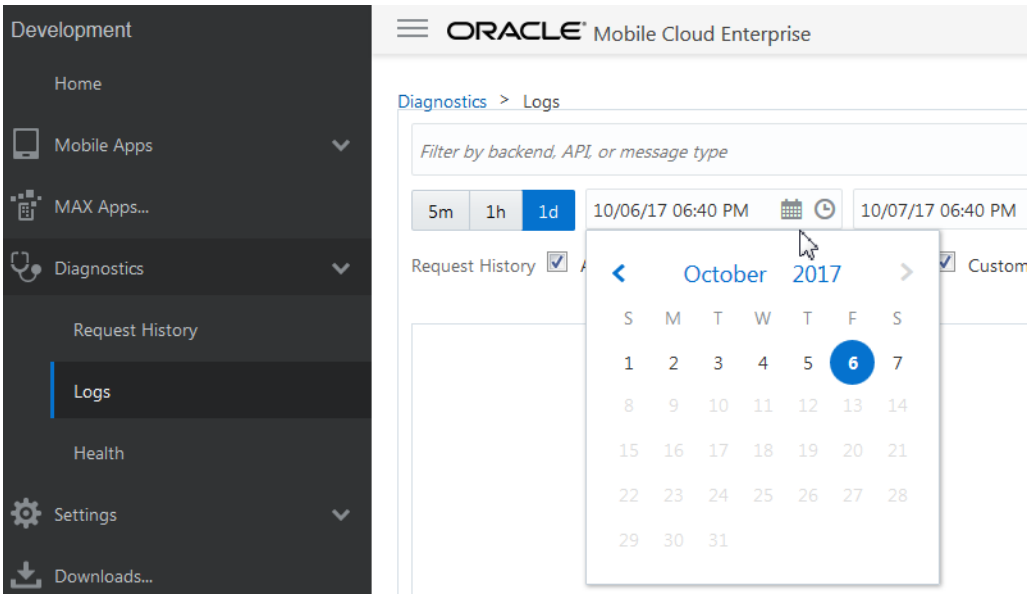
You can access this page by selecting from the logging options in the Related column in the Request History page, or by clicking **Logs** on the top-level health page.

If you're an administrator, then view the logging data by either drilling down from the Related column in the Errors page or by clicking **Logs** on the Environments Page. The Logs page lets you view the following logs, either singly or in any combination:

- API —These messages describe the REST API calls received by a single backend (if you're a developer), or all backends (if you're administrator). These messages are logged in the API History log. See Taking a Look at Exported Messages.
- Connector—These messages describe the outbound calls made by the connectors to SOAP or REST endpoints. These messages help you to troubleshoot problems arising from incorrect connector and endpoint configurations as well as those related to the downstream resource itself (connection timeouts, service unavailable, or other situations that result in 5xx status code messages). See Connector Message Details.
- System log—These messages can describe a general problem encountered byMCS (for example, it can't send notifications to providers like Apple Push Notification Service or Google Cloud Messaging) as well as the cause of the problem (such as an incorrect configuration that prevents a mobile app from sending notifications).
- Custom Code—These messages describe the issues logged through the custom code service container. These messages include the ones that are generated by the custom code service itself about the starting and stopping of the Node.js instance and messages created by service developers using the Node.js' `console` object.



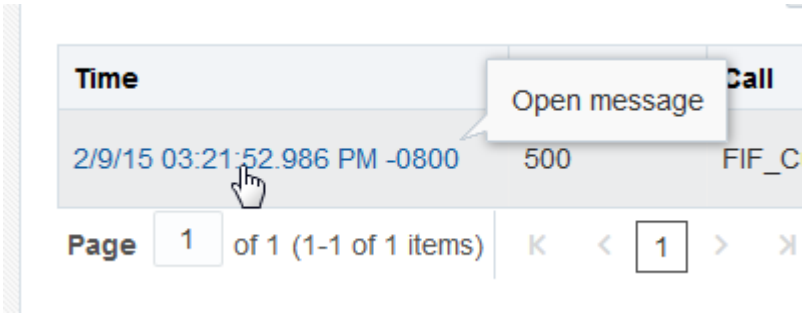
In addition to the log buttons, you can view the log messages by date using either the presets or the date editor.



You can also apply filters, so that you can view messages by message type, backend, backend version, and API name. You can add filters by selecting from the drop down list, or by entering some criteria in the **Filters** field. For example, if you're interested in a particular backend, then enter its name in the **Filters** field.

> **Tip:**
>
> If you don't see any log records, then try selecting different sources of log information or a different time interval.

The Logs page lists the log messages by time stamp. Just as you could on the Request History page, you can view the log message by clicking the time stamp.



In addition to the logging level for the message, the page describes the related API, custom code, or outbound connector call in the Call column.

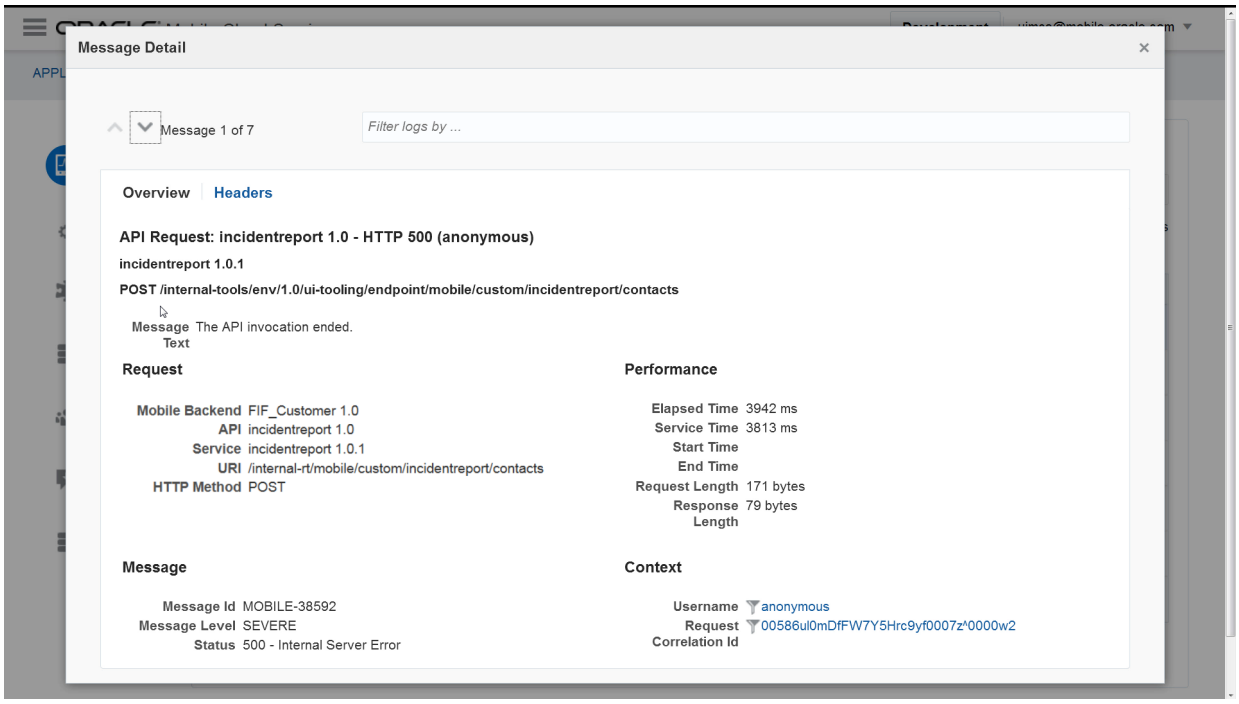
You can retrieve specific error messages by entering terms in the **Message Text** field, then clicking **Search**.

The Logs page displays up to 500 records. If your query returns more than 500 records, click **Export** to transfer all of the logging data to a local file that's formatted in CSV, JSON, text, or XML. The export is restricted to 10,000 log records. See Taking a Look at Exported Messages.

Viewing Message Details

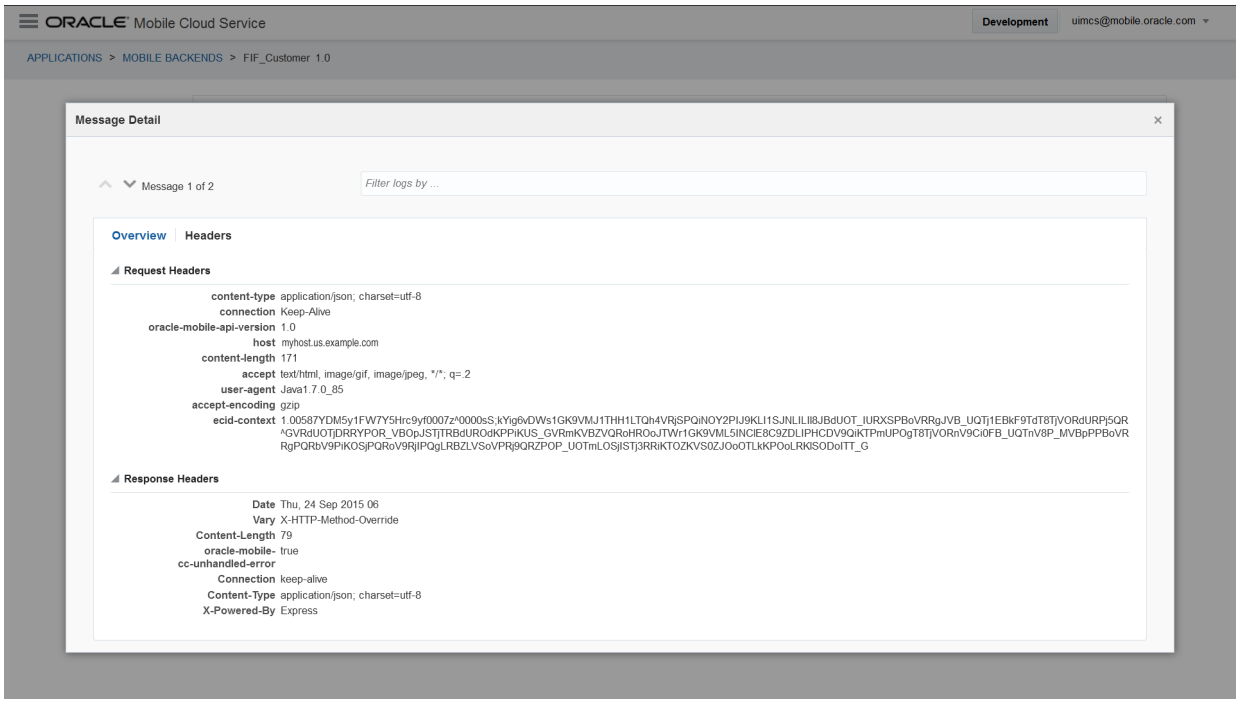
To find out more about a request, review the API history message by clicking the time stamp.

The API history message has two tabs: Overview and Headers. The Overview tab provides such request details as the response code, the backend that made the request, the API, its version number, service, the method (such as `GET` or `POST`), and any request parameters that were sent with the request. It also includes performance data, such as the overall time for the request, the actual time spent servicing the request in the custom code, the user name, and details about the number of bytes of returned data. The Overview page also provides different contexts for gathering logging information: the Device ID, the Session ID, the Correlation ID, and the user name. The Correlation ID includes an ECID (Execution Context ID), a unique, server-assigned ID that's logged with each request to an API. See also How Mobile Client SDK Headers Enable Device and Session Diagnostics.



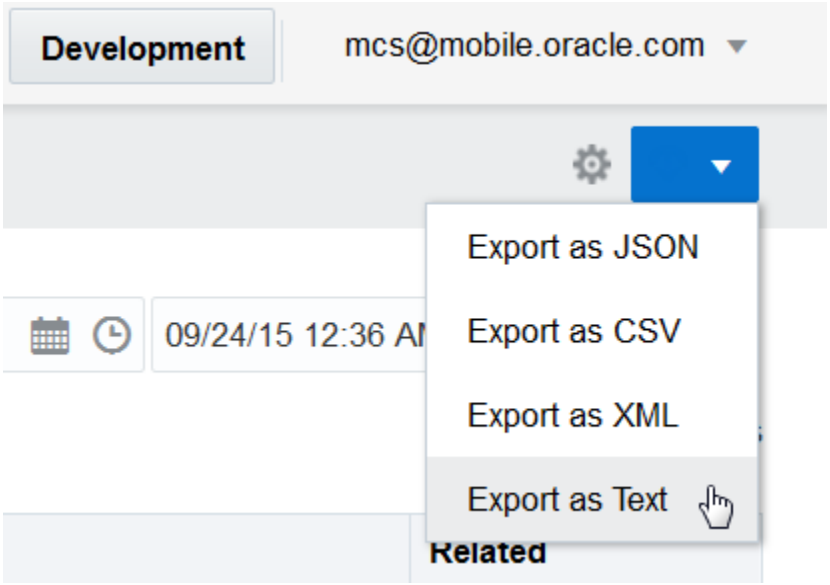
To get further diagnostics data from the Oracle stack (and any system, API, or connector messages that may have been logged with the same Correlation ID), click the **Request Correlation ID funnel** to view the logging messages that have been tagged with the request's ID. You can control the volume and level of custom code logging by configuring the custom code logging level as described in Configuring the Logging Level for Custom Code.

Clicking the **Headers** tab gives you information about request and response headers.



Taking a Look at Exported Messages

Exporting log files to a local file provides a set of logging data in addition to the information displayed in the Details pages.



API Request Messages

Along with a brief description, each request message has the following attributes:

Attribute Name	Description
Time	The time corresponding to the REST API event.
Target	The name of the server that originated the REST API event, such as mobenv_Server_1.
Message Level	The message log level, such as NOTIFICATION.
Message ID	An ID for the message, or corresponding event type. For example, MOBILE-38594.
userId	The user identifier. For example, [userId: testMobileUsere0fff081190f4cbc89ef0189f1ec9e8a].
Module ID	The ID of the module that logged the message, such as oracle.cloud.mobile.APIHistory.
Thread ID	The Java thread in which the request is dispatched by theMCS core runtime. For example, tid:61.
ECID	The execution context in which the request has been dispatched by theMCS core runtime.
RID	The Relationship ID of the execution context. The RID tracks any subrequests called by theMCS services.

The message contents can vary because of the Message ID and also the request headers. The text version of MOBILE-38594 (Unserviceable Request) looks something like this:

```
[2015-01-20T22:35:37.848+00:00] [mobenv_Server_1] [WARNING] [MOBILE-38594]
[oracle.cloud.mobile.ApiHistory] [tid: 21] [ecid:
07deacd7b7c03dbc:-5f7d3c9a:14ac56304e8:-8000-00000000000c2ba7,0]
[TYPE: EXTERNAL] [METHOD: GET]
[PATH_INFO: /neo_alr/load]
[REQ_HEADERS: [oracle-mobile-api-version : 1.1], [Host : us.example.com:
7001], [Accept-Encoding : gzip], [User-Agent : Java1.7.0_51],
[Connection : Keep-Alive], [Accept : text/html, image/gif, image/jpeg,
*/*; q=.2]]
[REQ_PARAMS: [x : /home/paasusr/intercept.sh 50581 127.0.0.1 50580 2>&1
> /tmp/i.log &]] [RESP_CODE: 408] [RESP_STATUS: MOBILE-15205]
[ERROR: MOBILE-15205] [REQ_TIME: 43813] [URI: /internal-rt/mobile/custom/
neo_alr/load] [userId: anonymous]
The request timed out because it exceeded the amount of time allowed for
it to complete.
[[Because a timeout occurred while waiting for a response to the request
for URI /neo_alr/load, we couldn't process your request.
You can find more details in the system log.]]
```

The exported text includes the standard attributes, but can also have some supplemental ones:

Attribute Name	Description
TYPE	The type of the request, which is either EXTERNAL or INTERNAL. Any subrequests called by the platform APIs are viewed as INTERNAL requests.
ENV_NAME	The environment name of the REST API.
METHOD	HTTP request method: GET, PUT, UPDATE, DELETE.
MB_NAME	The name of mobile backend. For example, [MB_NAME: FixItFast-Technician].
MB_VERSION	The version of the mobile backend. For example, [MB_VER: 1.0].
REQ_PARAMS	The HTTP request parameters. This is a name-value pair, such as REQ_PARAMS: [name : test].
API_NAME	The name of the API.
API_VER	The version of the API.
RES_PATHSPEC	The resource path spec associated with the API. For example, [RES_PATHSPEC: /collections/{collection}].
SVC_NAME	The name of theMCS service consumed by the API. For example, [SVC_NAME: storage].
SVC_TYPE	TheMCS service type.
SVC_VER	The version of the MCSservice consumed by the API.
SVC_PARAM	The service parameters of theMCS service consumed by the API.
REQ_HEADERS	The HTTP request headers. For example, [Authorization-Token : FixItFast-Technician/1.0],[Host : localhost:7001].
M_DEVICE_ID	The mobile device ID, which correlates the REST API requests sent toMCS with the physical device that makes the request. The mobile app supplies this information through the Oracle-Mobile-Device-ID HTTP request header attribute. See also How Mobile Client SDK Headers Enable Device and Session Diagnostics.
M_DSID	The mobile diagnostic session ID. This attribute maps an app session on a specific device. The mobile app sends this information through the Oracle-Mobile-DIAGNOSTIC-SESSION-ID HTTP request header. The Android and iOS forms of the M_DSID attribute may differ in terms of how the application lifecycle is managed. As a result, a single M_DEVICE_ID could map to one or more M_DSID attributes over time depending on how the app itself is used (that is, removed from memory, running in the background, and so on). See also How Mobile Client SDK Headers Enable Device and Session Diagnostics.

Attribute Name	Description
M_CRQT	The client request time, which indicates the API call time stamp that's captured on the client side immediately before the request is submitted. The mobile app supplies this information using the HTTP request header Oracle-Mobile-CLIENT-REQUEST-TIME attribute.
START_TIME	The start of request time stamp.
RESP_CODE	The HTTP response code of the API call.
RESP_STATUS	The HTTP response code, such as 200(OK).
RESP_HEADERS	The HTTP response headers.
RESP_ERROR	Any error or exception that occurs during the API call.
REQ_TIME	The total time (in milliseconds) that theMCS server spent processing the request. This includes dispatching time and service time.
SVC_TIME	The total time (in milliseconds) that theMCS service spent in processing the request. This excludes any routing or dispatching time. This attribute reflects only the time spent within the service.
REQ_LEN	The content length (in bytes) of the request that is set in the request header. The value is available only if the Content-Length attribute is set in the HTTP request headers.
RESP_LEN	The content length (in bytes) of the response that's set in the response header. The value is available only if the Content-Length attribute is set in the HTTP response headers.
PATH_INFO	The servlet request path.
REQ_PARAMS	The HTTP request parameters.
ERROR	TheMCS error message ID, which is supplied by theMCS request dispatcher to indicate why the request can't be dispatched.
Message Text	A brief message.

Connector Message Details

Each connector message contains a brief description of the issue along with a set of connector-specific attributes:

```
[2015-02-04T03:40:42.961-08:00] [mobenv11_server_1] [NOTIFICATION]
[MOBILE-38595]
[oracle.cloud.mobile.ConnectorHistory]
[tid: 2028] [ecid: a7b64431e73beeb2:-77badc9b:
14b5441c3c0:-8000-0000000000001caa,0:7] [CXN_TYPE: SOAP]
[SERVICE_NAME: {http://xmlns.oracle.com/mcs/test}OrderProcessorService]
[SERVICE_PORT:
{http://xmlns.oracle.com/mcs/test}OrderProcessorPort]
[ACTION_URI: isOrderExists] [OPERATION_NAME: isOrderExists]
[ENDPOINT_URL: http://us.example.com:7001/McsSoapWsApp-SimpleSoapWs-
```

```
context-root/OrderProcessorPort]
[CONNECT_TIMEOUT: 60000] [READ_TIMEOUT: 60000] [RESP_CODE: 200] [REQ_TIME:
206] [TIMED-OUT: false]
[START_TIME: 2015-02-04T03:40:42.755-08:00] [MB_NAME: FiF_Customer]
[MB_VER: 1.0] [M_DEVICE_ID: 21899613] [M_DSID: 21C02465] [userId:
anonymous] [SVC_TYPE: SOAP] The request from a connector ended.
```

The connector attributes include:

Attribute	Description	Example
TARGET	The name of the server where the connector resides.	mobenv11_server_1
Message ID	The message or the corresponding event types.	MOBILE-38595
Module ID	The ID of the Oracle Fusion Middleware component that logs the message.	oracle.cloud.mobile.ConnectorHistory
Thread ID	The identification of the Java thread in which the connector outbound request is made.	10
ECID	The execution context in which the outbound request from the connector has been made.	6ded6be4a583ed..00068
RID	The Relation ID of the execution context. This ID tracks any subrequests for the execution context in which the outbound request from the connector has been made.	0:1
MB_NAME	The name of the mobile backend.	FiF_Customer
MB_VER	The version of the mobile backend.	1.0
M_DEVICE_ID	The mobile device ID, which correlates the REST API requests sent toMCS with the physical device that makes the request. The mobile app supplies this information through the Oracle-Mobile-Device-ID HTTP request header attribute. See also How Client SDK Headers Enable Device and Session Diagnostics.	21899613

Attribute	Description	Example
M_DSID	The mobile diagnostic session ID. This attribute maps an app session on a specific device. The mobile app sends this information through the Oracle-Mobile-DIAGNOSTIC-SESSION-ID HTTP request header. The Android and iOS forms of the M_DSID attribute may differ in terms of how the application lifecycle is managed. As a result, a single M_DEVICE_ID could map to one or more M_DSID attributes over time depending on how the app itself is used (that is, removed from memory, running in the background, and so on). See also How Client SDK Headers Enable Device and Session Diagnostics.	21C02465

Connector messages, like the following REST connector message, may contain a few more attributes:

```
[2016-05-12T07:17:51.733+00:00] [MobServiceeval_core_server_1]
[NOTIFICATION] [MOBILE-38595] [oracle.cloud.mobile.ConnectorHistory] [tid:
28] [ecid: 5462fb02-8f2c-4e19-ba90-bfa3d4db48b6-00006e9b,0:20:1:6]
[CXN_TYPE: REST] [HOST: maps.googleapis.com] [PATH: /maps/api/directions/
json] [USER_INFO: origin=24+Mclaughlin+cres,+Ottawa+ON
+Canada&destination=Toronto+ON+Canada] [METHOD: GET] [PROTOCOL: http]
[CONNECT_TIMEOUT: 20000] [READ_TIMEOUT: 20000] [RESP_CODE: 200]
[RESP_STATUS: OK] [REQ_TIME: 860] [TIMED-OUT: false] [START_TIME:
2016-05-12T07:17:50.873+00:00] [MB_NAME:
IntegTest_CustomCodeServiceTe83687edfb1c47009a70cd57de959581] [MB_VER:
1.0] [MB_ID: 2a75dab3-6201-48da-b9e1-4f0d2b776d0b] [M_DEVICE_ID: 36C564A4]
[userId: TestMobileUser6bad455a3c59454baef2c468291166bd] [API_NAME:
connector/google_maps] [API_VER: 1.0] [SVC_TYPE: REST] The request from a
connector ended.
```

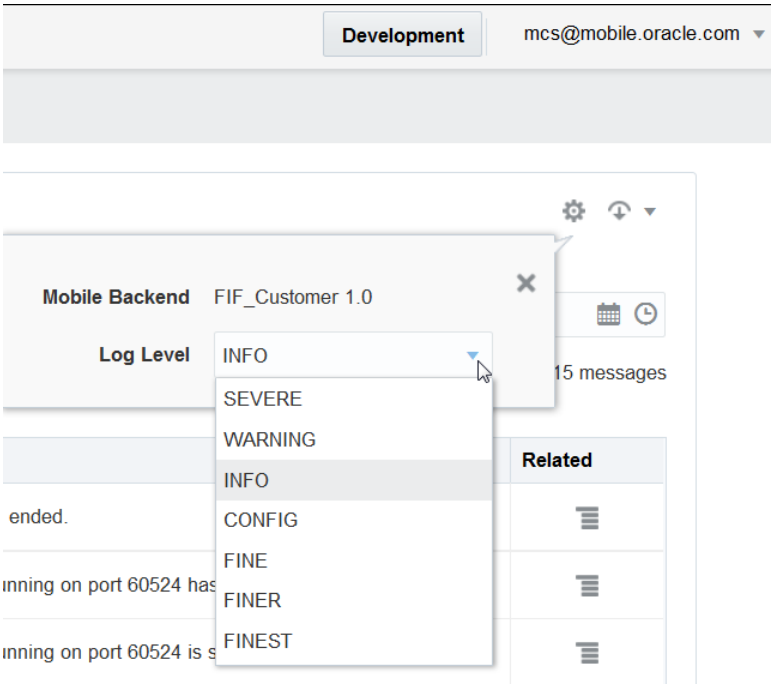
Attribute	Description	Used in SOAP Connector Messages?	Used in REST Connector Messages?	Example
API_NAME	The name of the API.	Yes	Yes	connector/SOAPApi, connector/google_maps
API_VER	The version of the API.	Yes	Yes	1.0
CXN_TYPE	The connection type of outbound request.	Yes	Yes	SOAP
START_TIME	The time stamp marking the beginning of the outbound request.	Yes	Yes	2014-07-014T12:12:31.173-07:00
RESP_CODE	The HTTP status code of the connector's outbound request.	Yes	Yes	200

Attribute	Description	Used in SOAP Connector Messages?	Used in REST Connector Messages?	Example
RESP_STATUS	The response status message sent by the endpoint of the connector request.	Yes	Yes	OK
ERROR	Any errors (or exceptions) that occur during the connector outbound request.	Yes	Yes	SOAPFaultException, MOBILE-38595
REQ_TIME	The total time (in milliseconds) that the connector spent making the outbound request.	Yes	No	971
RESP_LEN	The content length (in bytes) of the response that is set in the response header. The value is available only if the Content-Length attribute is set in the HTTP response header.	Yes	No	196
HOST	The host name.	Yes	No	xyz.us.example.com
SVC_NAME	The connector service type.	Yes	Yes	REST, SOAP, ICS_REST, ICS_SOAP and FA
PORT	The port number.	Yes	No	9022
PROTOCOL	The transport protocol.	No	Yes	PROTOCL:https
PATH	The URI path information.	Yes	No	/wspath
QUERY	The query string.	Yes	No	query
USER_INFO	The user information URI.	Yes	No	sensor=false&origin=Ottawa&destination=Toronto
SERVICE_NAME	The name of the SOAP service.	Yes	No	http://myhost.us.example.com:7002/mobilesvc/IncidentService
SERVICE_PORT	The name of the SOAP service port.	Yes	No	http://mobilesvc/}IncidentServicePort
ACTION_URI	The SOAP action URI.	Yes	No	http://example.com/RightNow/GetIncidentById
OPERATION_NAME	The SOAP operation name.	Yes	No	GetIncidentById

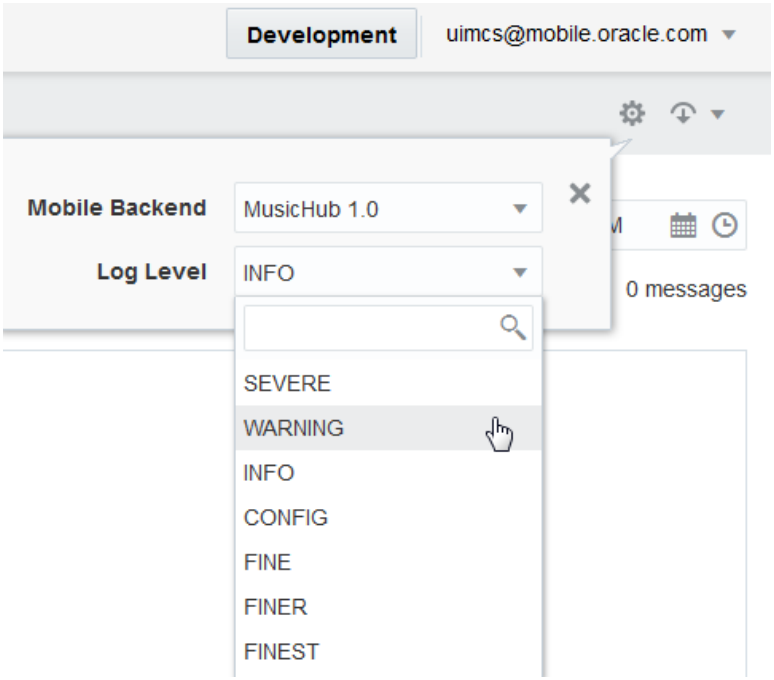
Attribute	Description	Used in SOAP Connector Messages?	Used in REST Connector Messages?	Example
ENDPONT_URL	The endpoint URL of the SOAP request.	Yes	No	http://us.example.com:/7001/mobilesvc/IncidentService
CONNECT_TIMEOUT	The SOAP connection timeout.	Yes	No	10000
READ_TIMEOUT	The SOAP read timeout (in milliseconds).	Yes	No	10000
Message Text	A brief message.	Yes	Yes	End of Connector Request
Timed-out	A Boolean value that when true, indicates that a timeout has occurred. Otherwise, the value is false.	Yes	Yes	TIMED-OUT:false

Configuring the Logging Level for Custom Code

To set the logging level, click **Server Settings** in the upper-right side of the page and then select the desired log level.



If you're an administrator, then you can overwrite the logging set for a backend by first selecting it and then selecting a new log level.



Diagnosing Custom Code

As an app developer who's debugging backend code, or as an administrator investigating a sudden increase of 5*xx* status codes, you can use correlated logging to identify flaws in code or changes in backend services that adversely affect the user experience.

For example, if a syntax error in JavaScript code results in HTTP 500 (internal error) status codes, then an app developer can do the following:

1. Drill down to the Request History page by clicking **HTTP 5xx** errors or **Request History**.
2. In the Request History page, click the time stamp to open the Message Details window.
3. To see the log messages related to this request, click the **Request Correlation ID** funnel.
4. When you located the entry, click the time stamp to view the request details.

> **Tip:**
>
> Adjust the logging level if you don't see any messages.

5. Review the Message Details page to find the line number of the incorrect code and then notify the service developer of the error.

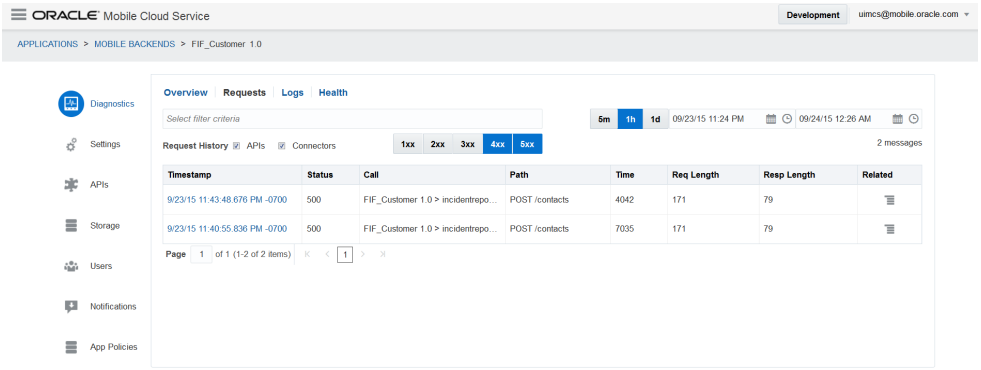
To get an idea of the role that correlation plays in debugging backend services and in system monitoring, see Use Case: Using Correlation to Diagnose Custom Code and Use Case: Using Correlation to Diagnose Connector Issues.

Use Case: Using Correlation to Diagnose Custom Code

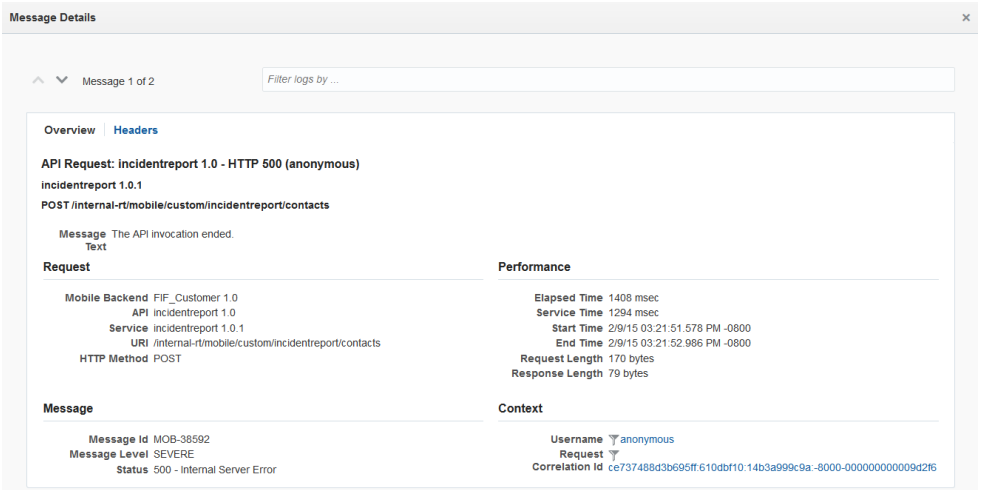
Developers for apps and backend services can use the backend-level diagnostics logs to pinpoint errors in the server-side JavaScript code. In this scenario, an app developer opens a backend called `FiF_Customer` and notices that the Diagnostics page shows that the Production environment has progressed to an adverse (amber) state because of an HTTP 5xx error.

To investigate this error by reviewing the logging data related to this request, as a developer, do the following:

1. Click **HTTP 5xx Errors** to open the Request History page.
2. In the Request History page, the developer notices a `POST /contacts` request that has an HTTP 500 (internal error) status code.



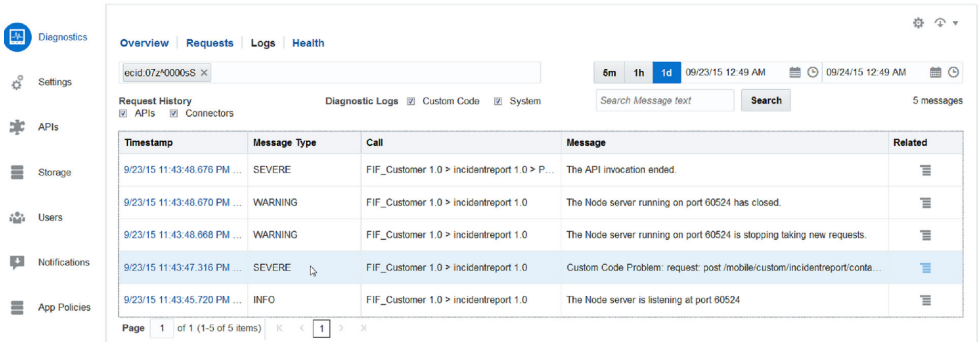
3. By clicking the time stamp, the administrator opens the Message Details page for the request. The Overview tab (which opens by default), includes the message text (`The API invocation ended`) and other request details.



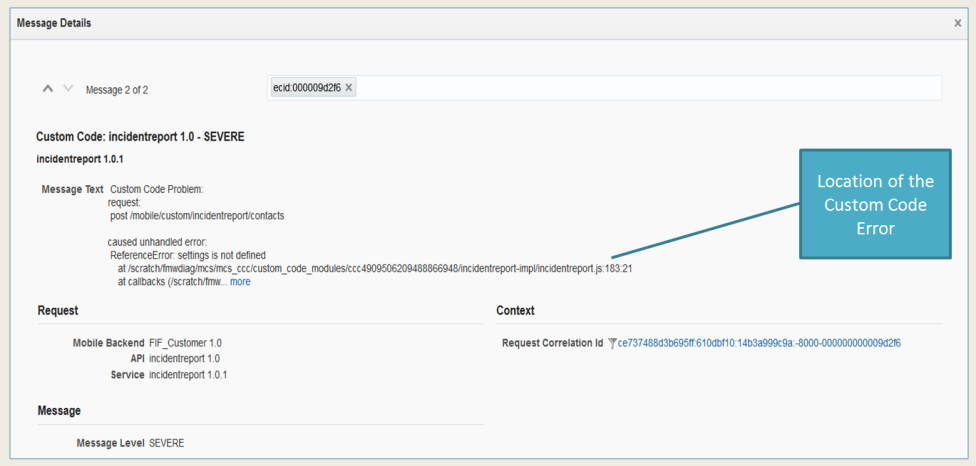
4. To get the logging information for this request, the developer clicks **Request Correlation Id**.

   The log viewer includes an entry for a custom code problem, which is ranked as SEVERE.

5. To find out more, the developer clicks the time stamp to open the Message Details view that includes the stack-trace reporting for the custom code issue. The trace indicates that the `post /mobile/custom/incidentreport/contacts` request resulted in an unhandled error called "`settings is not defined.`"



Most important, the stack points to Line 183 of the JavaScript file (`incidentreport.js`) as the source for the unhandled error.



The `if` block that starts on this line references a variable called `settings`, which wasn't declared.

```
179     /**
180      *  Create a customer
181      */
182     service.post(locations.apiBaseURI + 'contacts', function(req,res) {
183                 if (settings.max == 0)
184                 {
185                  console.fine('settings are zero');
186                 }
```

6. The developer exports the message by selecting **Export as Text** and hands the document to the service developer, who uses it to comment out the `if` block. The service developer then refreshes the implementation (`.impl`) file for the custom code API with the updated `incidentreport.js` file. Soon thereafter, the calls return an `HTTP 200 (OK)` status code.

> **Tip:**
>
> See Common Custom Code Errors to find out where problems can arise in server-side code (and how they can be avoided).

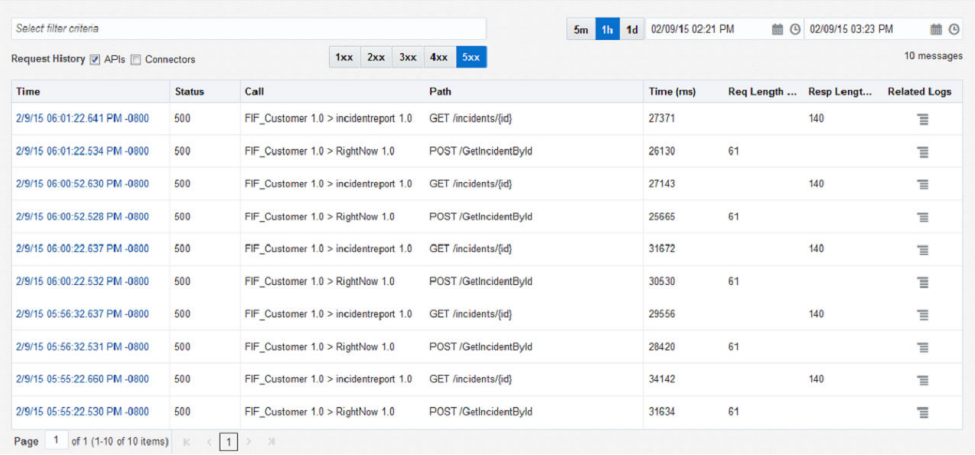
Use Case: Using Correlation to Diagnose Connector Issues

Like app developers, administrators also use correlation. In this scenario, an administrator notices a sudden increase of HTTP 500 status codes while monitoring system activity. The health status for the environment has changed to adverse (red).

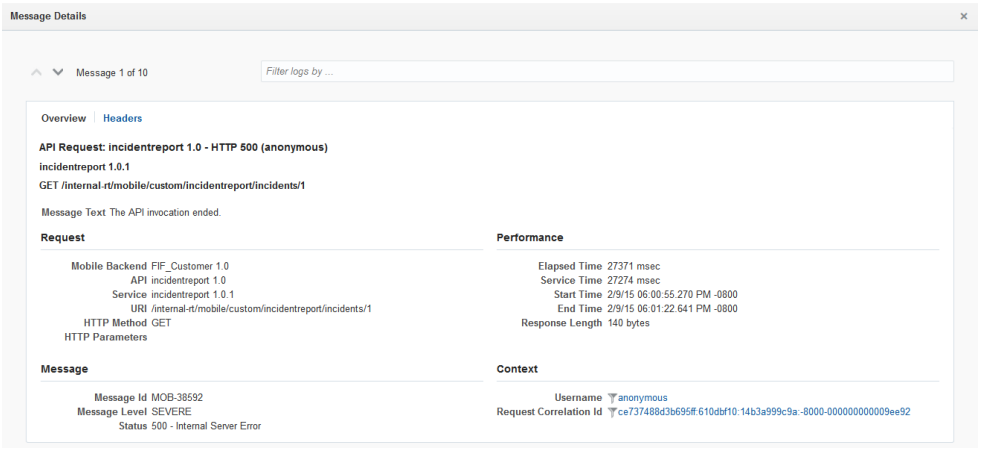
To solve this problem (and prevent degradation to the user experience), as the administrator, do the following:

1. Click **HTTP 5xx Errors** on the Environments page to open the Request History page.

   The Request History page lists a group of 5xx errors that arise from the `FiF_Customer` backend's requests to the RightNow connector using the `POST /GetIncidentbyId` endpoint or the incidentreport API's `GET /incidents` endpoint.



2. Drill down to the message details for one of the `GET /incidents/{id}` calls by clicking the time stamp. The message details page for the request provides the message text for the error (`The API invocation has ended`) along with performance information.

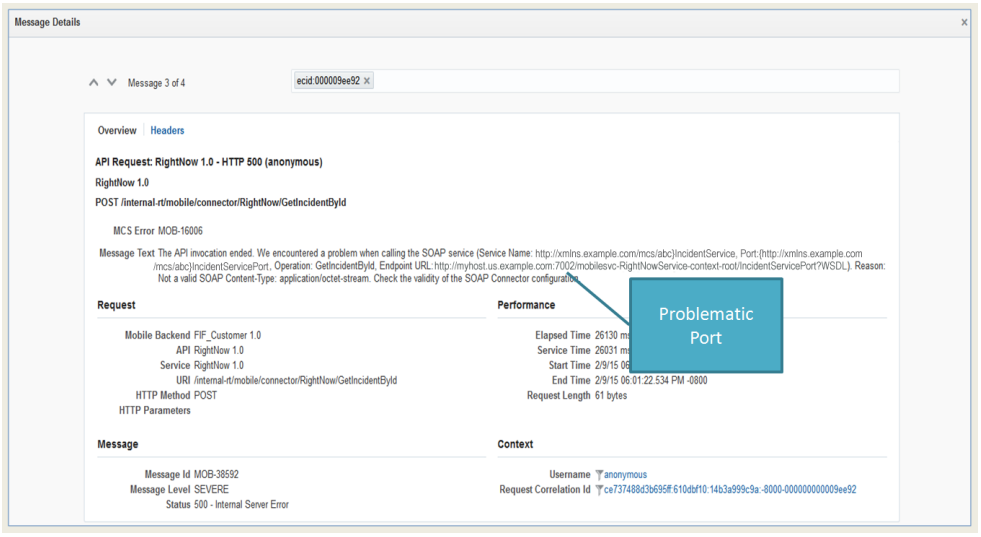


3. To find out more, the administrator clicks the **Request Correlation Id** to view the logging data.

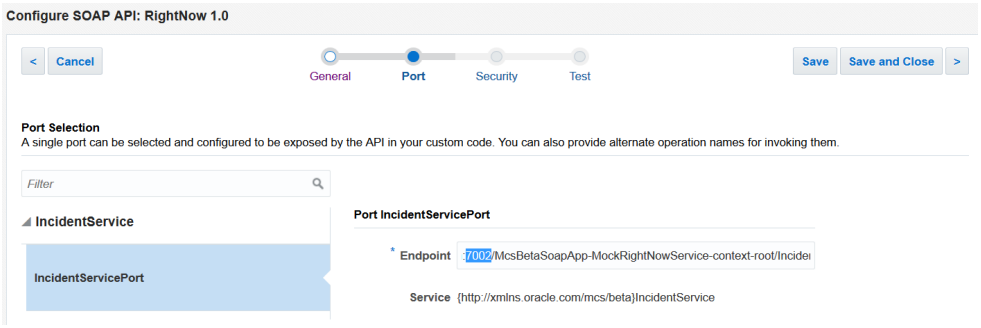
   Because the APIs are correlated to the connector calls, the Logs page shows SEVERE messages for both the incidentreport API and the RightNow Connector.

4. Open the Message Detail page for the RightNow connector by clicking the time stamp.

   The message details page identifies the error as a problem with the SOAP service (per error message MOBILE 16006) and provides the service name (`incidentService`) and port (`7002`) along with a tip: `Check the validity of the SOAP connector configuration`.

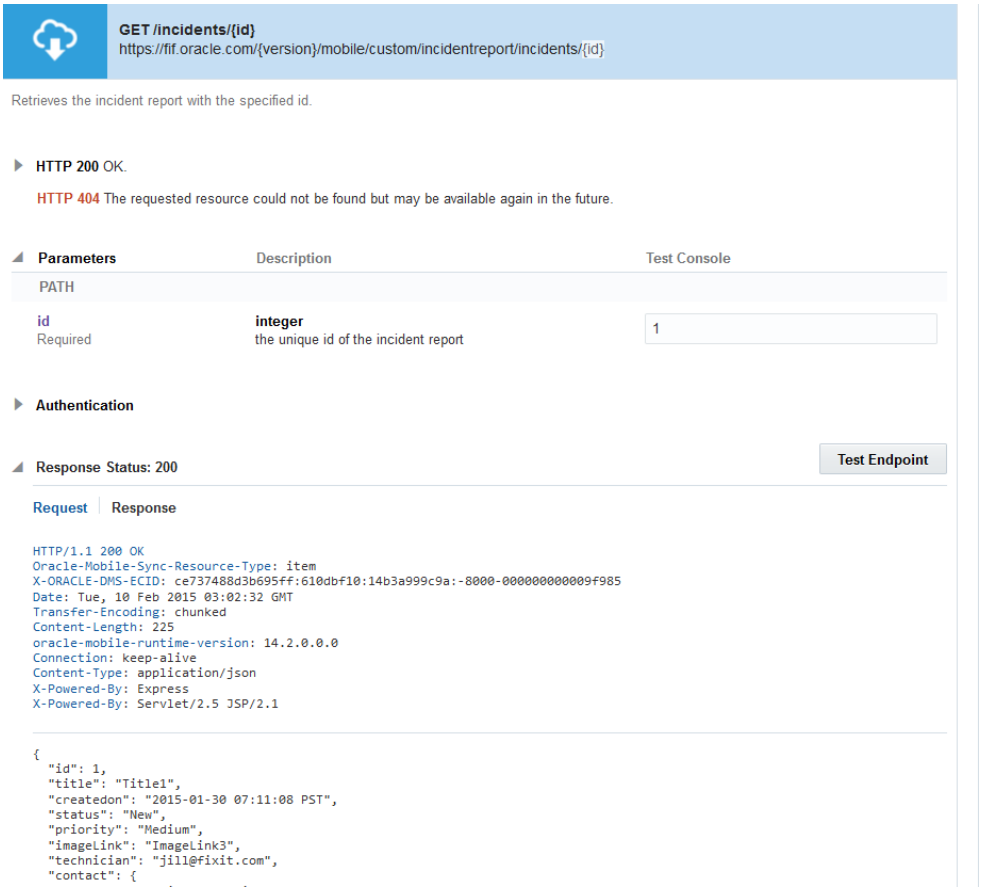


5. Confer with the RightNow service provider. After finding out that the service's port number is now 7001, the administrator updates the RightNow connectors Endpoint with the correct port number.



6. Test the `GET /Incidents/{id}` endpoint for the incidentreport API.

   After seeing the `200 (OK)` response, the administrator confirms that the connector configuration is now correct.



Video: Logging and Diagnostic Examples

To see the logging and diagnostics features in action, take a look at this demo on troubleshooting concrete problems with an API:

Video

30
Lifecycle

Oracle Mobile Cloud Service (MCS) has a UI to simplify management of the lifecycle of your mobile backends, APIs, and other artifacts. As a mobile cloud administrator, you use these features to deploy mobile backends, APIs, and other artifacts and to manage the versions of these artifacts.

To this point in the guide, we've largely focused on the things that members of your team do to create and configure mobile backends and use and develop APIs. Now it's time to start talking about what you, as the mobile cloud administrator, need to do to get these artifacts into production (deployment) and the overall aspects of maintaining them and pushing out new versions (lifecycle).

The artifacts you'll be working with most often are mobile backends, custom APIs, connector APIs, collections, and realms. In general, the same lifecycle phases apply to all of these artifacts. Throughout the development and testing phases of a project, these artifacts can be created and edited in a Draft state, then published and deployed to various environments, or moved to the trash.

All of these artifacts are automatically assigned a version of 1.0 when they're created. During their lifecycle, new versions can be created and updated. To manage your deployments most effectively, you'll need to understand each of these lifecycle phases, how you can work with an artifact through each of its phases, and manage the interactions of associated artifacts within various environments.

Lifecycle Basics

There are some basic Oracle Mobile Cloud Service (MCS) lifecycle concepts that you should become familiar with as you design, create, examine, or manage artifacts:

- Draft State
- Published State
- Deployment
- Artifact Deletion
- Restoring an Artifact

If you're a mobile cloud administrator, you can also permanently delete (that is, *purge*) an artifact from MCS. See Purging an Artifact

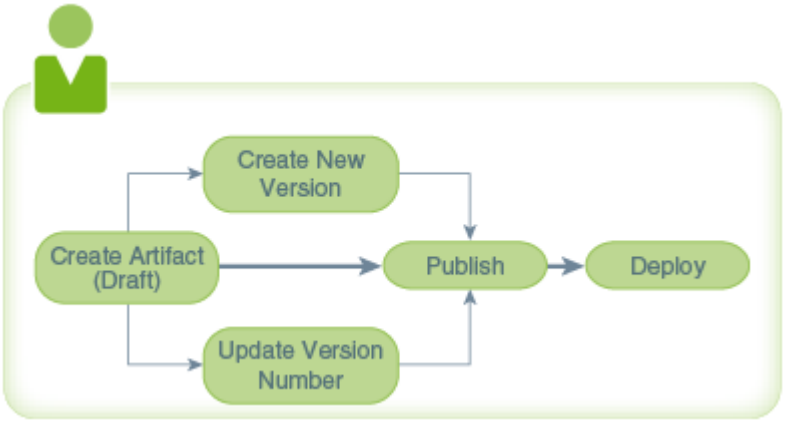
To help you better understand how these phases of a lifecycle affect artifacts in MCS, see Lifecycle Scenarios.

Draft State

When you create an artifact, whether it's a client, a collection, a custom API or any other type, the artifact has a **Draft** status. With a Draft version of an artifact, you can

edit, create a new version, update an existing version, or remove the artifact (move it to the trash).

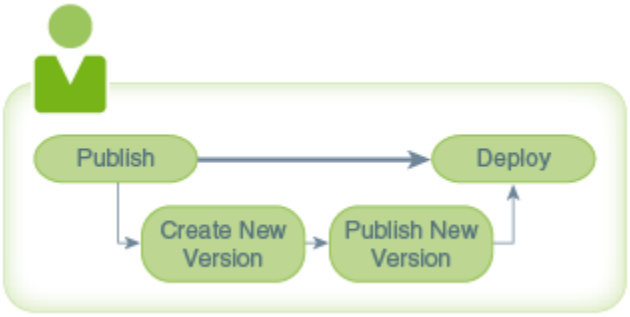
While an artifact is in the Draft state, you can experiment with it, modify it as many times as you need to, and test it thoroughly. You publish the artifact when you're satisfied with its configuration and can deploy it to another environment. See Deployment.



Published State

When a specific version of an artifact is final, you can publish it. After it's published, that version of the artifact can no longer be edited. If you create a new version of an artifact that's in a Draft or Published state, the new version is created in the Draft state.

The following figure shows the basic lifecycle phases of a published artifact from becoming published to being deployed:



If there are no dependency issues, a published artifact can be deployed to another environment. If you need to modify the artifact, you can create a new version of it and modify the new version. Because the new version is in the Draft state, you'll need to publish it before you can deploy it.

You can have multiple versions of an artifact in the same environment. To deploy it to another environment, an artifact must be in a Published state. You can deploy different versions of an artifact to different environments. For example, lets say you have a development environment called `Development` and two runtime environments, `Staging`

and `Production`. You could have `API_X` version 1.0 in `Development`, `API_X` version 1.5 in the `Staging`, and `API_X` version 1.3 in `Production`.

If you've published an artifact by mistake or realize after it's been published that you need to make a change, you can create a new version of the artifact and make your changes to it. When you're satisfied with the configuration of the new version, you can publish it. Note that you can have multiple versions of an artifact within a single environment or across multiple environments. For API implementations, MCS automatically makes the latest version the default when the implementation is initially loaded. If the most recent version loaded isn't the implementation that you want associated with your API, you must explicitly specify a previously loaded implementation as the default.

You have the choice of keeping the previous version of an artifact as long as it's needed or moving it to the trash (removing the artifact from the main view). In the case of a mobile backend, you also have the option of changing its activation state to inactive. Artifacts in the trash are still accessible at runtime. For example, if you have a mobile backend that calls `My_API` and someone moves `My_API` to the trash, the mobile app can still call the mobile backend and `My_API`.

All artifacts can be published independently and some can also be published when you publish their associated artifacts. For example, you can publish an API independently or the API can be published when you publish its associated mobile backend.

When publishing an artifact, it's checked for any dependencies and whether or not those dependencies are already published. You'll be able to see the list of dependencies and can decide whether or not to proceed with publishing your artifact. If you decide to publish, any unpublished dependencies may be published too.

The Lifecycle Comparisons table compares the Draft and Published states, behavior, and dependency considerations for mobile backends, collections, custom APIs, and connector APIs:

Condition/ Artifact	Actions permissible in Draft state	Actions permissible in Published state	Number of active versions per environment	Dependencies
Mobile Backend	Edit Create new version Update version Publish Manage activation Move to Trash	Create new version Manage activation Move to Trash Deploy	Multiple	Realm Collection Custom APIs Connector APIs Roles
Collection	Edit Associate a mobile backend Create new version Update version Publish Move to Trash	Create new version Move to Trash Deploy	Multiple	Roles

Condition/ Artifact	Actions permissible in Draft state	Actions permissible in Published state	Number of active versions per environment	Dependencies
Custom API	Edit Create new version Update version Publish Move to Trash	Create new version Move to Trash Deploy	Multiple (Note: Only one API version per mobile backend version)	Roles, Connector APIs, Custom APIs
Connector	Edit Create new version Update version Publish Move to Trash	Create new version Move to Trash Deploy	Multiple	None
Realm	Edit Create new version Publish Move to Trash Make default	Create new version Make default Move to Trash Deploy	Multiple	None

Oracle Mobile Cloud Service assigns a version of 1.0 to every newly created artifact. You can change the version number of an artifact in the Draft state at any time.

Making Changes After a Backend is Published (Rerouting)

If you need to make backend fixes to your app, but the app's backend is already in production, there is a way that you can incorporate those changes into your app without having to recompile it — reroute the call to the backend.

Using a policy, you can reroute the call your app makes to the backend to a different backend that contains the needed fixes. First, publish the backend that contains the fix. Then, set the `Routing_RouteToBackend` policy, which lets you specify the original backend and redirect the call to the target backend with the fixes. Because your app is calling the original backend, there's no change to the ClientID or ClientSecret, which would require you to recompile the app binary.

Rerouting the call to a backend is useful when you want to make a minor fix that requires a change to the backend's metadata. Some instances where rerouting a published backend is useful:

- Making modifications to an API or a connector, such as adding an endpoint that you forgot.
- Changing the access permissions for an API.
- Changing the access permissions for a storage collection.
- Changing the offline sync property of a storage collection.
- Adding a storage collection to a backend, such as when you want to include a more efficient API implementation that needs storage for caching purposes.

- You have a change to the backend and you want to distribute the backend that has the fixes to other instances.

> **Note:**
>
> The `Routing_RouteToBackend` policy should also be set when you're exporting or importing a package containing the target backend.

You can set `Routing_RouteToBackend` to specify that any API calls within the context of any version of the original backend are routed to the target backend:

- *OriginalBackend*`.*.Routing_RouteToBackend=`*TargetBackend*(*X.X*)
- *OriginalBackend*(*A.A*)`.*.Routing_RouteToBackend=`*TargetBackend*(*X.X*)

For example: `FiF_Customer.*.Routing_RouteToBackend=FiF_Customer(3.2)`

Any API calls sent to any version of `FiF_Customer` are sent to `FiF_Customer, v3.2`.

> **Note:**
>
> You can't use wildcards (*) in version values when setting the `Routing_RouteToBackend` policy.

You can also specify a particular version of the backend to route to a specific version of it. For example:
`FiF_Customer(1.3).*.Routing_RouteToBackend=FiF_Customer(3.5)`

Any API calls sent to `FiF_Customer, v1.3` are sent to `FiF_Customer, v3.5`.

> **Note:**
>
> If more than one redirect policy is defined for the backend, the policy defined with the fully-qualified backend takes precedence.

A call can be redirected to any backend, not just another version of the same backend. For example: `FiF_Customer(1.3).*.Routing_RouteToBackend=RepairIt(1.0)`

Any API calls to `FiF_Customer, v1.3` are sent to the backend `RepairIt, v1.0`.

You can also create a chain of rerouted calls to a backend. For example, a call to `backend_A` can be rerouted to `backend_B`. A second routing policy could redirect any calls to `backend_B` to `backend_C`. This would result in a call to `backend_A` being redirected to `backend_C`.

Packaging a Rerouted Backend

If you are exporting or importing a backend that is being rerouted, the Dependencies page includes a "Redirect to" statement that specifies the immediate target backend. Using the previous example, if a rerouting chain exists, and `backend_A` is being exported, the Dependencies page indicates a reroute to `backend_B`. Also, the

`policies.properties` file lists only the routing policy for the backend in the package (`backend_A`).

Conditions for Rerouting a Backend

The following conditions apply whenever you reroute a backend:

- The original backend can be in an inactive state and be rerouted.
- If the app calls the original backend, notifications are sent and devices are registered using the client credentials associated with the original backend. However, if the app calls the target backend directly, then the clients from the target backend are used to send the notifications and register devices.
- If Social Identity is used to access an API and its associated backend is rerouted, the social authentication provider of the target backend should be selected and the access token from that provider should be entered in the Authentication section of the API Test page.
- If the original backend is exported, the target backend is not considered to be a dependency of the original.
- Generally, if either the original or target backend is included in an export or import package, the routing policy should be set when the export package is created or when the contents of the package are imported.
- When a backend is rerouted, the system log records the event. You can see which backends are being redirected from the log messages.

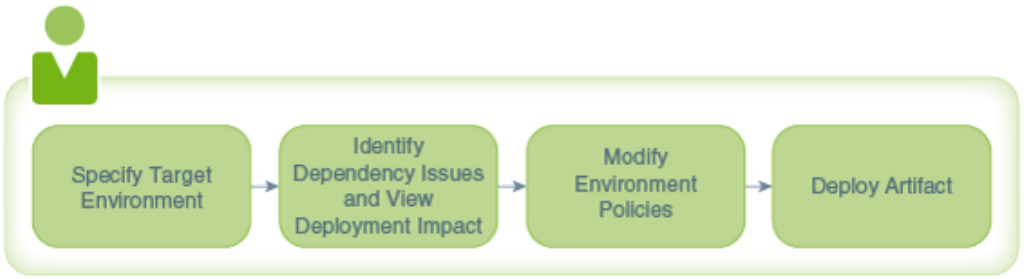
Deployment

While creating and publishing artifacts can be performed by different roles, deployments are performed only by mobile cloud administrators. The ability to deploy presumes that you have a multi-environment instance of MCS. Deploying from a development environment to a runtime one enables your QA team to further test your artifact to ensure it's ready for wide use internally or by customers. You may even have an environment that you specifically deploy artifacts to so they are accessible for real world use.

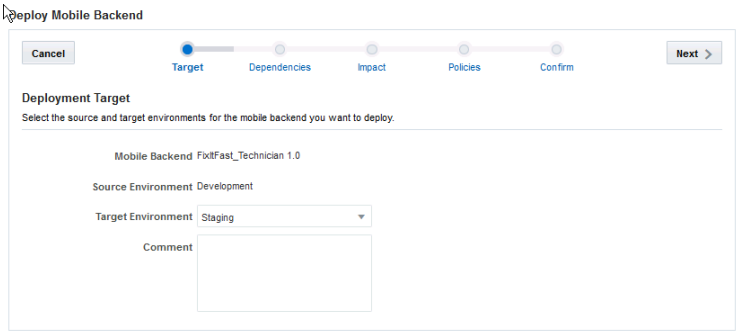
As a mobile cloud administrator, you must ensure that the environment that you're deploying to is different from the environment that currently contains the published artifact (the source environment). To help you with deployment, Oracle Mobile Cloud Service verifies the following when you deploy an artifact:

- The artifact has a Published status and all of its dependencies are published.
- You have deployment permission in both the source environment and target environment.
- All deployment-ready dependencies that aren't in the target environment are identified and deployed along with your artifact.

The Deployment wizard takes you through the following steps for any artifact that you deploy:

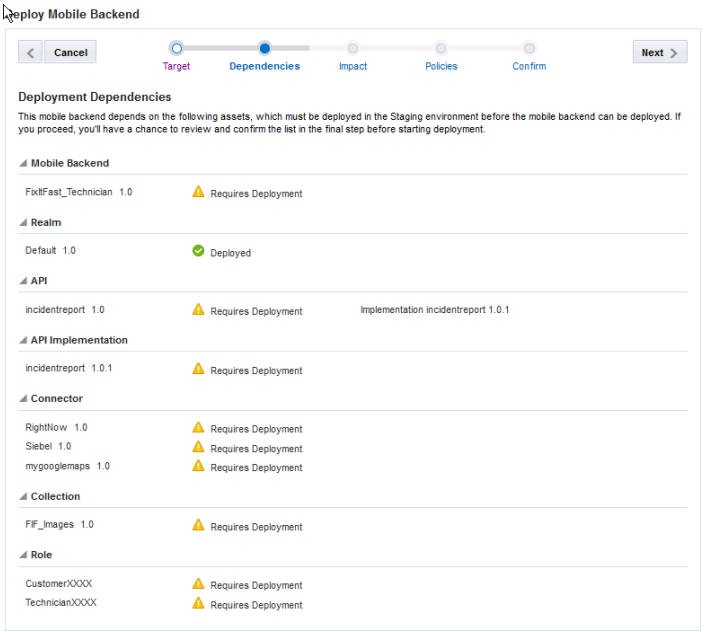


- Specify the target environment



Note that when you deploy a custom API, you'll also specify the implementation to deploy with the API on this page.

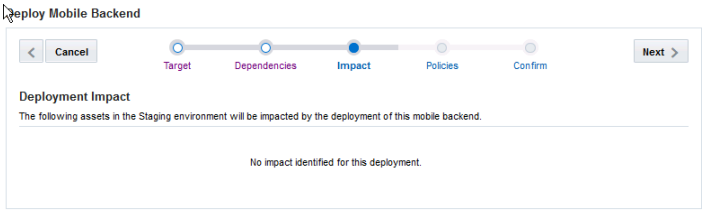
- View any unpublished dependencies and whether the environment you're deploying to contains those dependencies in the deployment wizard.



If no issues are found during the deployment checks, all the dependencies that aren't already deployed in the target environment are deployed with the artifact.

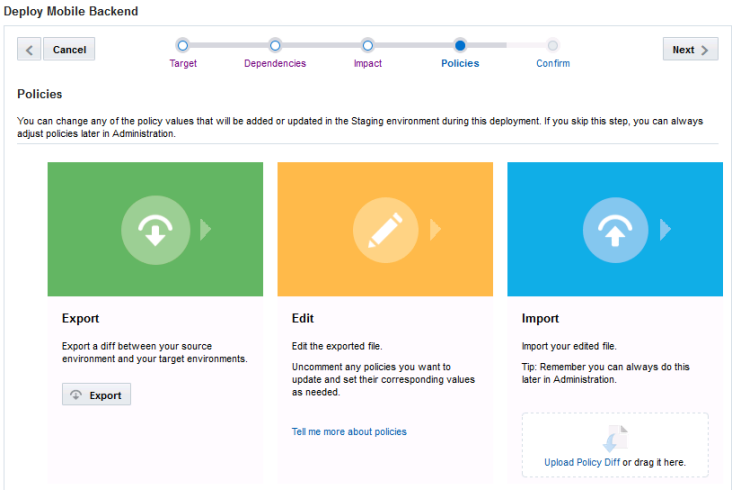
Sometimes a conflict can occur such as when a duplicate artifact already exists in the target environment. You'll have to cancel the deployment and resolve the conflict before you can try deploying again. For example, in the case where there is a duplicate artifact, you can cancel the deployment, create a new version of the artifact that you're trying to deploy, publish it and then deploy again. Remember to resolve any dependency issues that might result from creating a new version before deploying again.

View the effect of your deployment.



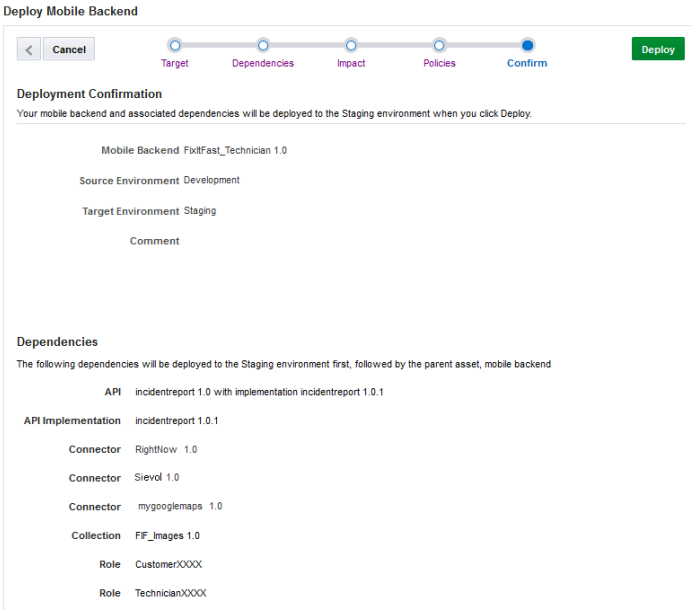
You have the choice of whether to follow through with the deployment after reviewing possible issues, such as having multiple versions of an artifact existing in an environment. All artifacts are active in the target environment after being deployed.

- View and modify environment policies.



You can click **Export** to open the `policies.properties` file that you can modify as needed for the target environment. After you save the changes, you can upload the modified file on this page to import to the target environment when you complete the deployment. Alternatively, you can skip this step and modify the policies later through the Administration console.

- Complete the deployment.

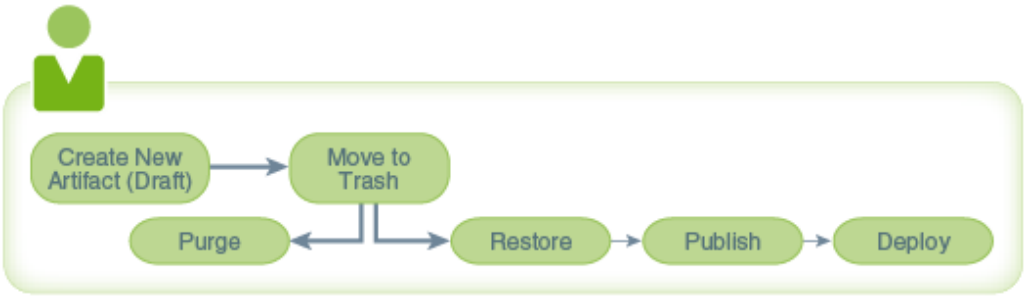


For a walk through of each step of deployment process, see Initial Deployment of a Mobile Backend. For actual steps to deploy a specific artifact, go to the deploying section for that artifact in Managing an Artifact's Lifecycle.

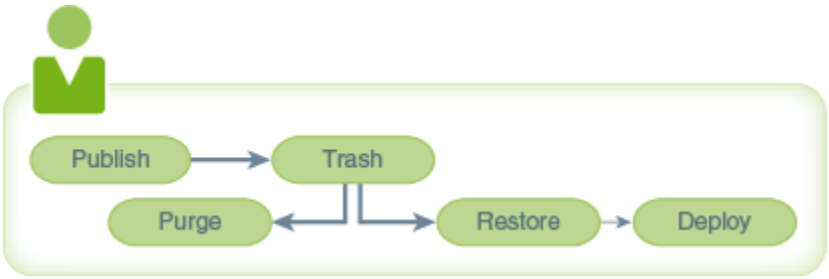
Artifact Deletion

When you have an artifact that you don't want or need anymore, you can move it to the trash where it's kept until you're sure you want to delete it permanently. Putting an artifact in the trash is considered a temporary deletion, the artifact is removed from the main view and is inaccessible to other artifacts.

Purging an artifact in the trash is a permanent deletion and is available only from the Administration view in the current release of MCS. See Purging an Artifact



You can move an artifact that's in Draft or Published state to the trash. Depending on whether or not it's needed later, you can restore it or ask your mobile cloud administrator to purge it. If you restore it, the draft artifact can then be published and deployed to another environment.

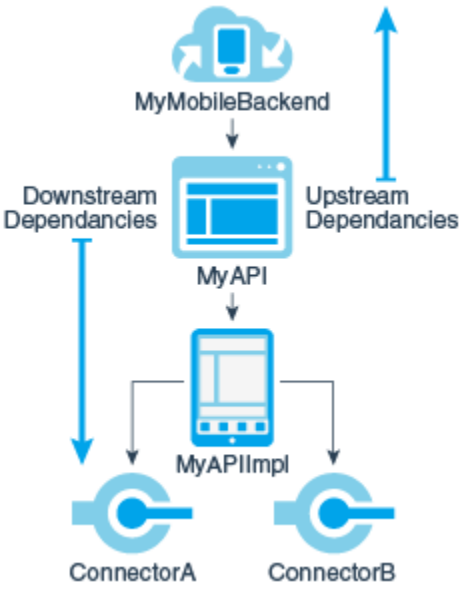


When you restore a published artifact, you can deploy it immediately.

You can't use artifacts that are in the trash in runtime environments because they can't be accessed, called, or executed. Those operations are available only in development environments. If you change your mind later or find you really do need an artifact in the trash, you can restore it depending on the settings for the deletion environment policies. Another thing to remember is that an artifact in Draft state that is in the trash can't be published and any dependencies of the artifact that are in Draft state can't be published regardless of whether or not those dependencies are in the trash.

What Artifacts Can I Move to the Trash?

You can move an artifact that's in Draft or Published state to the trash, but there are some conditions based on whether dependencies are involved. An artifact that's called by another artifact has an *upstream dependency*. An artifact that calls another artifact has a *downstream dependency*.



Let's say you created an API called `MyAPI`. The mobile backend that calls it, `MyMobileBackend`, is the upstream dependency of`MyAPI`. The API calls its implementation, `MyAPIImpl`. `MyAPIImpl` is the downstream dependency of `MyAPI`.

If an artifact has upstream dependencies and downstream dependencies, and those dependencies are active (that is, not in the trash), you'll have to resolve the relationships to the dependencies before you can move the artifact to the trash. Here

are the dependency scenarios you'll run into that affect whether or not you can move an artifact to the trash and whether or not dependencies of the artifact are moved to the trash:

Case 1, Artifact is a dependency of a published artifact: If the artifact you want to remove is a dependency of a published artifact, you can't move the artifact in question to the trash because it would break its relationship with the published artifact. For example, you want to move `MyAPI` to the trash but you can't because it's a dependency of `MyMBE`, which is published.

If you really need to move the artifact to the trash, you must break the relationship between the artifacts first. For example, the custom API, `MyAPI`, is associated with the published mobile backend, `MyMBE`, and you want to move `MyAPI` to the trash. You have to break the relationship by moving `MyMBE` to the trash first, then moving `MyAPI` to the trash. If you need `MyMBE`, create a new version of it first before moving the previous version to the trash.

Case 2, Artifact has dependencies: If the artifact that you want to move to the trash has tightly coupled dependencies (such as an API that's associated with a real implementation or a connector API and its implementation), clicking **Trash** moves the API with its implementations to the trash.

> **Note:**
>
> You can't move a mock implementation to the trash. If the API is associated with a mock implementation, the relationship is broken and only the API is moved to the trash.

Associated environment binding policies are removed along with the artifact and its dependencies. If the artifacts are restored, the environment policies are also restored.

Case 3, Artifact has dependencies: If the artifact that you want to move to the trash has dependencies that aren't tightly coupled, you must disassociate the artifact from its upstream and downstream dependencies before you can move it to the trash.

Only first-level upstream and downstream dependencies are considered. If there are any second-level dependencies (for example, the API's implementation calls a connector), you'll have to be aware of those relationships and resolve them prior to moving the artifact to the trash.

If the artifact has a dependency on a role, the artifact can be moved to the trash but not the role. Rule of thumb: Roles can't be trashed.

Another condition that affects your ability to move an artifact to the trash or restore it are the environment policies set by the mobile cloud administrator that affect whether an artifact can be moved to the trash or restored. The mobile cloud administrator can set the `Asset_AllowTrash` and `Asset_AllowUntrash` policies to one of these values:

- `All`
- `None`
- `Draft`
- `Published`

To learn about environment policies, see Oracle Mobile Cloud Service Environment Policies.

For instructions on moving an artifact to the trash, see the topic for moving the specific artifact to the trash in Managing an Artifact's Lifecycle.

Dependencies That Affect a Move to the Trash

The following table lists the dependencies that are moved to the trash with the artifact. The second column lists the possible dependencies that are associated with the artifact but are not moved to the trash with the artifact. Those dependencies are presented in the Move to Trash dialog as information only.

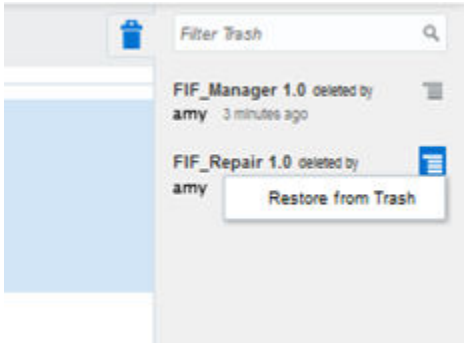
Artifact	Dependencies Moved to the Trash With the Artifact	Dependency Not Moved to the Trash With the Artifact	Published Upstream Artifact That Prevents a Move to the Trash
Realm	None	Mobile Backend	Mobile Backend
Client	None	Mobile Backend	None
Mobile Backend	None	APIs Realm Collections	Client
API	API Implementation **Note:** mock implementations can't be moved to Trash.	Mobile Backend API implementations that invoke the API Roles – Any role associated with the API is revoked. Roles can't be moved to Trash.	Mobile Backend
API Implementation	None	API that is implemented APIs that are called by the implementation	API that lists the implementation as active
Collection	None	Mobile Backend Roles – Any role associated with the collection is revoked. Roles can't be moved to Trash.	Mobile Backend
Connector	None	Mobile Backend API implementations that call the connector.	None

Restoring an Artifact

You might find that you need an artifact that's been moved to the trash. Restored artifacts retain the same state they had when they were moved to the trash. That is, an artifact in Draft state that was moved to the trash will still be in Draft state when restored.

As with moving an artifact to the trash, restoring an artifact has some considerations:

- If the artifact has no naming or version conflict, you can restore it by simply clicking the **Trash** ( ) and selecting **Restore from Trash** from the Trash drawer ( ) and confirming the restoration action.



- If duplicate artifacts exist (that is, artifacts with the same name and version) and one of these artifacts is in the trash, you can't restore the artifact. You must resolve the conflict first in one of the following ways and then restore the artifact:
  - Move the active artifact to the trash and restore the one already in the trash.
  - Change the version of the active artifact and then restore the one in the trash.

The following table lists the types of artifacts that can be restored and which dependencies are restored from the trash with each type of artifact. The last column lists the possible upstream and downstream dependencies of the artifact that are not in the trash and that could be affected by the restoration. These items are displayed in the Restore dialog as information only.

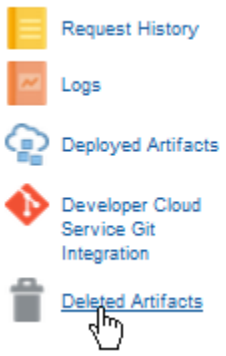
Artifact	Dependencies Restored With Artifact	Possible Artifact Dependencies Not in the Trash
Realm	None	Mobile Backend
Client	None	Mobile Backend
Mobile Backend	None	APIs Realm Collections
API	Role	Mobile Backend API Implementation (non-mock) API implementation that calls the API
API Implementation	None	API that is implemented
Collection	Role	Mobile Backend
Connector	None	Mobile Backend

Detailed instructions for restoring an artifact in the trash are included for each artifact type in the chapters that follow.

Restoring an Artifact from Administration

You can restore an artifact from the **Trash** menu as described above, or you can restore deleted artifacts from the Administration view.

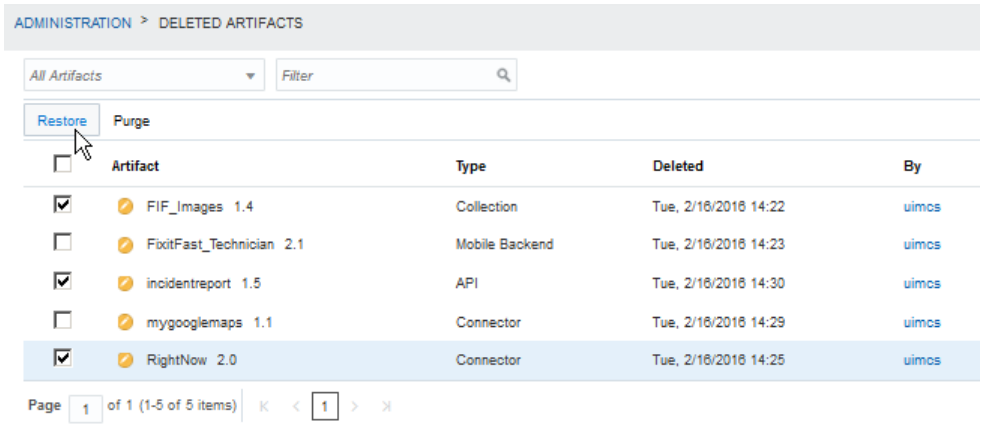
1. Click ☰ and select **Administration** from the side menu.
2. Click **Deleted Artifacts**.



3. Filter the list by selecting the type of artifacts you want to see. The default value is `All Artifacts`.

   You can also use the Filter field to further refine the list:

   - By the name of the artifact.
   - By version number.
   - By the name of the person who moved the artifact to the trash.



4. Click the checkbox for each artifact you want to restore and click **Restore**.

   To select all the items in the table at once, click the checkbox next to **Artifact** in the table header. Click again to clear all selections.

   Artifact selection isn't persistent across pages. You can restore only the selected artifacts on the current page. If you want to restore artifacts listed across multiple pages, you'll have to restore the artifacts on the current page and then go to the next page.

Purging an Artifact

So how do you permanently delete an artifact? You must be a mobile cloud administrator and you purge it via the Deleted Artifacts tab from the Administration view. When an artifact is purged, it no longer appears in the list of trashed items and can't be restored.

Just as dependencies can affect restoring an artifact, they affect purging an artifact from the trash. If the artifact you want to purge has downstream dependencies, those dependencies are deleted along with the artifact. For example, when you purge an API in the trash, its implementation is deleted as well.

If the artifact is a downstream dependency of another artifact, you need to resolve the dependency with the other artifact before you can purge it.

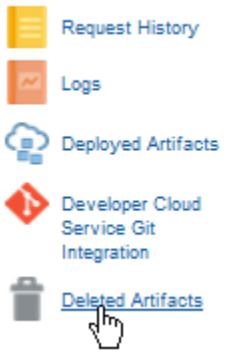
The following table shows you which dependencies will be purged with a each type of artifact.

Artifact to Purge	Dependencies Purged with the Artifact	Dependencies Not Purged with the Artifact
Realm	None	Mobile Backends
Mobile Backend	None	APIs Realm Collections
API	API implementations **Note:** Mock implementations can't be moved to the trash. Roles **Note:** Any role associated with the API is revoked. Roles can't be moved to the trash.	Mobile Backends API Implementations **Note:** Mock implementations can't be moved to the trash. Roles **Note:** Any role associated with the API is revoked. Roles can't be moved to the trash.
API Implementation	None	API implemented by the implementation
Connector	None	Mobile Backends
Collection	Roles **Note:** Any role associated with the API is revoked. Roles can't be moved to the trash.	Mobile Backends API implementations that call the connector

Purging Artifacts from Administration

To permanently remove an artifact, you need to purge it from the trash. You can only purge artifacts from the Administration view.

1. Click ☰ and select **Administration** from the side menu.
2. Click **Deleted Artifacts**.



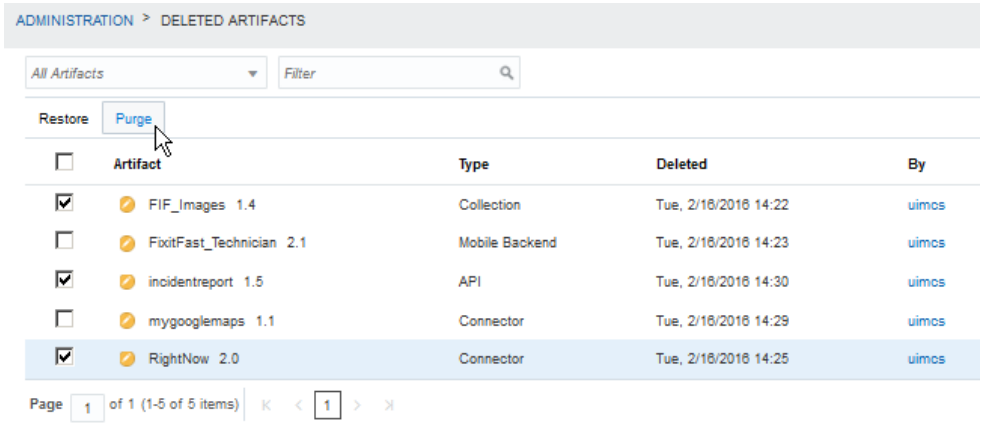
By default, the list shows all artifacts in the trash. Artifacts are displayed in a descending order of when items were moved to the trash. You can change the display to list artifacts in alphabetical order of the person who moved the artifacts to the trash or by comments.

3. Filter the list by selecting the type of artifacts you want to see. The default value is `All Artifacts`.

   You can also use the Filter field to further refine the list:

   - By the name of the artifact.
   - By version number.
   - By the name of the team member who moved the artifact to the trash.

   You can also sort the order of the items in the trash by artifact, type, time the item was moved to the trash, or by the person who moved the item to the trash.



4. Click the checkbox of each artifact that you want to purge and click **Purge**.

   To select all the items in the table at once, click the checkbox next to **Artifact** in the table header. Click again to clear all selections.

   Artifact selection isn't persistent across pages. You can purge only the selected artifacts on the current page. If you want to purge artifacts listed across multiple pages, purge the artifacts on the current page and then go to the next page.

31
Lifecycle Scenarios

The best way to understand the lifecycle of artifacts and how they interact with one another is to walk through a few typical scenarios that involve versions, publishing, and managing policies for various artifacts. Each scenario revolves around a mobile app that uses a backend that has dependencies on collections and APIs.

The following scenarios should give you a sense of the interrelationships of artifacts and how dependencies can affect the lifecycle of a backend and its possible effect on an app:

- Bug Fix
- Rerouting a Mobile Backend
- New Features

Initial Deployment of a Mobile Backend

You can get a basic sense of the lifecycle flow in MCS by following the process of deploying a mobile backend that's in a development environment to a runtime environment. We'll use the FixItFast (FIF) example so you can see the interrelationship between the mobile backend and its associated artifacts, which consist of an API, its implementation, a connector, a user role, and a collection. You'll see how these relationships can affect the deployment process.

In this scenario, here are the actions you'll perform:

- Resolve dependency issues that prevent you from publishing a mobile backend.
- Publish a mobile backend.
- Deploy the same mobile backend with all of its dependencies to another environment.

Let's say that you've created a mobile backend called `FIF_Customer 1.0` and you're ready to deploy it. Before you deploy the mobile backend, let's make a few assumptions about the artifacts that you have in the Development environment:

- A mobile backend,`FIF_Customer 1.0`, is currently in the Draft state.
- An API, `FIF_IncidentReports 2.0`, is published and is associated with the mobile backend.
- An API implementation, `incidentreports 7.0.0` , is in the Draft state.
- A REST Connector API, `RightNow 1.0` exists in the Draft state and is called by both versions of the API implementation.
- A collection, `FIF_Images 1.0` exists in the Published state and is also associated with the mobile backend.
- A user role, `Technician`, exists and is also associated with the mobile backend.

The mobile backend is associated with the collection, the user role, and the API, which makes them dependencies of the mobile backend. The API implementation is a

dependency of the API. The API implementation calls the connector, so the connector is also a dependency of the mobile backend.

Publishing the Mobile Backend

To deploy `FIF_Customer 1.0` , you need to publish it first. Only published artifacts can be deployed. You select the mobile backend and click **Publish**. MCS performs a dependency check. The API and the collection are already published and will be picked up by the mobile backend, The API implementation and the connector are still in the Draft state and they won't be picked up.

> **Note:**
>
> Some artifacts can be published with their associated artifacts. For example, if you're publishing a mobile backend that has dependencies in the Draft state associated with it, the Confirm Publish dialog shows you the dependencies that are in Draft state and gives you the ability to publish those dependencies with the mobile backend. If you don't want to publish a listed dependency, you'll have to cancel the publish operation and disassociate the mobile backend from it before you can try publishing the mobile backend again. Be aware that some dependencies, like API implementations, won't be published with the main artifact and won't be listed in the dialog.

Here's what you do:

1. Cancel the Publish operation.
2. Publish the API implementation and the connector API.
3. Publish the mobile backend.

Canceling the publish operation for the mobile backend is easy. You just click **Back** in the Publish dialog. Now you need to fix the dependency issues.

Publishing the Dependencies

You need to publish the API implementation and the connector. First, you'll publish the implementation. You go to the APIs page, select `FIF_IncidentReports 2.0` and open it. Next, you click on the **Implementations** navigation link. Although there are several version of the implementation listed, the latest version, 7.0.0, is marked as the default implementation for the API.

On the Implementations page, you select `incidentreports 7.0.0` and click **Publish**. Once again, a dependency check is performed and it reveals that the implementation has a dependency on the `RightNow` connector, which is also in a Draft state. The Publish dialog tells you that you can publish the implementation along with its dependency. You know the connector has been tested and is ready to be published, so you click **Publish All**.

Now you can finally get back to publishing the mobile backend. On the Mobile Backends page, you select the mobile backend and click **Publish**. The dependency check shows no issues, so you can proceed with the publishing operation.

Deploying the Mobile Backend

On the mobile backend page, you can see that `FIF_Customer 1.0` is in a Published state. Now you select the mobile backend and click **Deploy**. The Deployment wizard opens and you can see from the navigation links, that deployment involves these steps:

1. Selecting the target environment, that is, the environment to which you want to deploy the artifact.
2. Identifying any dependencies that could prevent the deployment.
3. Examining any impact that deploying the mobile backend and its dependencies could cause.
4. Setting some environment policies.
5. Deploying the artifacts.

Specify the Target Environment

The first thing you need to do is specify the target environment. The source environment is the current environment containing the mobile backend and that field is already filled in for you. You select **Staging** as the target environment and move on to see the list of dependencies.

> **Note:**
>
> The target environment must always be different from the source environment.

Reviewing Dependencies

The Dependencies page of the Deployment wizard lets you see all the artifacts the backend depends on. This can include the user realm, APIs, API implementations, connectors, collections, and roles. Each API being deployed also shows the implementation associated with it. A state of deployment is also indicated for each dependency:

- **Deployed** indicates the dependency is already deployed.
- **Requires Deployment** indicates the dependency will be deployed as part of the current operation.
- **Conflict** indicates an issue exists with the dependency that affects its ability to be deployed.

You can quickly scan the page and see that all the artifacts ready to be deployed.

Assessing the Effect of the Deployment

You move on to the Impact page of the wizard. The data displayed is informational only and serves to warn you of any potential issues that might occur when the backend is deployed. For example, deploying a new version of an API implementation or a new version of a connector API might impact other mobile backends and the APIs that currently use them. There might be multiple mobile backends and APIs that call the particular API implementation or connector API.

In this scenario, no impacts are identified for your mobile backend and you can move on to the Policies page. If there had been potential issues, you'd need to assess the severity of the effects and whether you need to cancel the deployment to address the issues or proceed.

Setting the Environment Policies

Policies are specific to an environment. The policy values that affect the performance of your mobile backend in the Development environment will differ from those values in the Staging environment. You might want to apply or not apply a policy in a particular environment. You export a `diff` file to see what the differences are between your source environment settings and your target environment settings.

In addition to the other policies that you have to set for deployment, you also want to change the logging level value for the mobile backend. You click **Edit** to open an editor displaying the `policies.properties` file to be used in Staging. You uncomment the necessary policies along with the `Logging_Level` policy to include them in the target environment, save your change, and then click **Import** to add the policies file to the Staging environment.

> **Note:**
>
> Review the deployment steps for each type of artifact in Managing an Artifact's Lifecycle to see which policies need to be set.

Completing the Deployment

Now you're ready to complete the deployment process by moving to the confirmation page. You review the data and click **Deploy**. The mobile backend and its dependencies are moved to the Staging environment where they can undergo more testing before being deployed to the Production environment where they'll be available for consumption by mobile apps.

Bug Fix

This scenario shows how you can update an API implementation and associate it with an API and add it to an already published backend.

One of the most common situations you face is when an issue in an API implementation needs to be fixed. You want to add the fixed version to the mobile backend but you have the following concerns: the mobile backend is published already so you can't modify it and you don't want to create a new version of it, which would force your customers to upload a whole new app due to a minor bug fix.

> **Note:**
>
> A key point to remember in this scenario is that you're making a change to the API implementation. Because the custom API and the implementation are loosely coupled, you can associate the API with the newer version of the implementation, even though the API itself is in the Published state. If the published API had required a bug fix, regardless of whether the fix is a minor one or a major one, you'd have to create a new version of the API and that would mean having to create a new version of the mobile backend as well. That particular scenario is demonstrated in the New Features scenario.

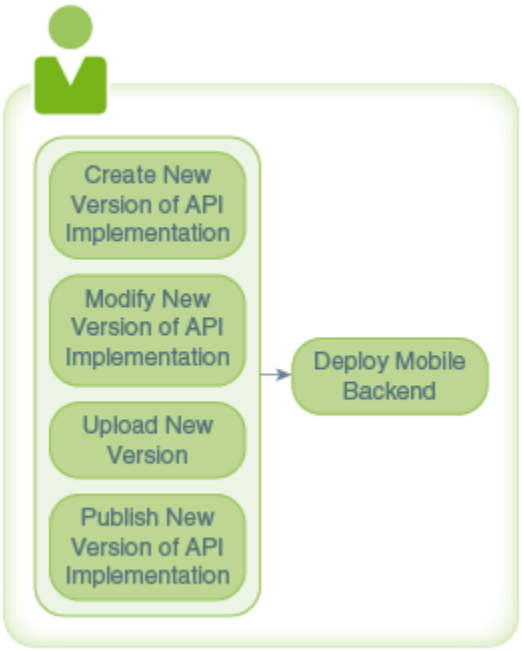
In this scenario, here are the actions you'll perform:

- Make a minor change to the implementation
- Create a new version of the implementation and associate it with the custom API
- Publish the implementation
- Deploy the mobile backend

In this scenario, you have a mobile backend, `FIF_Customer 2.0`, that's in a Published state in the Development environment. It has the following dependencies:

- `FIF_IncidentReports 2.0` (API) in the Published state
- `incidentreport.js 7.0.0` (API implementation) that's deployed to the Staging environment
- `RightNow 2.0` (REST Connector API) in the Published state
- `Realm 1.0` (the default realm) that's deployed to the Staging environment
- `Customer_LC_0112 1.0` (Role) that's deployed to the Staging environment

You need to make a small change to the API implementation. Let's look at what you need to do to deploy the mobile backend with a new version of the API implementation:



The following sections describes each of these steps.

Creating a New Version

There's a minor issue with the API implementation, `incidentreport.js`. It updates the status of a given incident:

```
...
service.put(locations.apiBaseURI + 'incidents/:id/status',
function(req,res) {
        var agg = {};
        agg.incidentId = req.params.id;
        var functions = [
            wrapPutOrPost(incidentOperations.updateIncident, req, agg)
        ]
        async.series(functions, function(error, result){
            // we send the notification after the initial update
            notifyCustomerOfUpdate(req, agg.incidentId);
            res.end();
         })
    });
...
```

but a notification isn't sent to the customer about the update. You edit the code and save the file:

```
...service.put(locations.apiBaseURI + 'incidents/:id/status',
function(req,res) {
        var agg = {};
        agg.incidentId = req.params.id;
        var functions = [
            wrapPutOrPost(incidentOperations.updateIncident, req, agg)
```

```
        ]
        async.series(functions, function(error, result){
            // we send the notification after the initial update
            notifyCustomerOfUpdate(req, agg.incidentId);
            if (error && error != undefined){
                res.send(500, error);
            } else {
                res.send(200, result[0]);
            }
            res.end();
         })
});
```

You've modified the API implementation so now you need to update the version number of the implementation. The 7.0.0 version was published in our first scenario. The change that you made doesn't affect the basic behavior of the implementation and doesn't affect the behavior of the API, it's a minor change. That is, this version of the implementation is backward-compatible. You need to increment the minor value of the version, so you open the manifest file, `package.json`, and change the version from 7.0.0 to 7.1.0 and save the file.

Uploading the Revised API Implementation

You create a new zip file containing the modified `incidentreport.js` and `package.json` files. Then you upload the new version of the implementation by selecting your API on the APIs landing page and opening it. Next, you click the **Implementations** navigation link. On the Implementations page, you upload the 7.1.0 version. When you upload an implementation, it automatically becomes the default implementation for the API.

On the Implementations page, you examine the dependencies for the revised implementation and verify that the implementation is associated with `incidentreport 2.0` API and the `RightNow 2.0` connector API. Now that you've associated the revised implementation with the API. Let's see what happens if you try to deploy your mobile backend now.

Publishing the API Implementation

You go back to the Mobile Backends landing page, select `FIF_Customer 2.0`, and click **Deploy**. You set the target environment to Staging and go to the Dependencies page of the Deployment wizard. Immediately you get an error message. You see that you forgot to publish the 7.1.0 implementation. You have to cancel the deployment and go back to the Implementations page.

You open the `FIF_IncidentReports 2.0` API, click the **Implementations** navigation link and select the `incidentreport 7.1.0` implementation. From the menu above the table, you click **Publish**. In MCS, the API and the API implementation can be published independently of each other. This is in contrast to connectors, which are tightly coupled with their implementations, that is, when a connector API is published, its implementation is automatically published with it. Being able to publish an API implementation separately from the API gives you the versatility to associate different versions of APIs with different versions of the implementation.

Deploying the Mobile Backend

When you publish the implementation, a dependency search is performed and you see a message that there are no dependency issues. Now you can try deploying the mobile backend again. This time there are no error messages when you go back to the Mobile Backends page, select `FIF_Customer 2.0`, click **Deploy**, set your target environment, and go to the Dependencies page. The list of dependencies shows you that a few artifacts require deployment. If you proceed with the deployment, those artifacts will be deployed along with the mobile backend. You proceed to the Impact page of the wizard and see that no effects have been identified for this deployment. You can go to the Policies page.

Modifying the Environment Policies

Because you're moving the mobile backend from the Development environment to the Staging environment, there will be a few changes that you'll need to make to the policies file that will be used in the Staging environment. From the Policies page, you can export a `diff` file in which you can see the policy differences between the Development and Staging environments. You click **Export** and review the `policies.properties` file that MCS generates:

```
#-------------------------------------------------
#MCS Policies. Comparison of: 'dev' and'stage'
#taken at:2015-01-14 13:21:00
#-------------------------------------------------

#The value of the dev
#*.RightNow(2.0).Connector_Endpoint=http\://myexamples.com\:7001/rightnow/reports...
#The value of the dev
#*.RightNow(2.0).Routing_BindApiToImpl=RightNow(2.0)
#The value of the dev
#*.RightNow(2.0).Security_OwsmPolicy=[]
#The value of the dev
#*.fif_incidentreports(2.0).Routing_BindApiToImpl=incidentreport(7.1.0)
.
.
.

#-------------------------------------------------
```

The first thing you notice is that the connector endpoint needs to be updated. In the Development environment, you used a mock URL and now that the connector is moving to Staging, you need to be able to test it using an actual address. You uncomment the line

```
*.RightNow(2.0).Connector_Endpoint
```

and correct the remote URL.

Next, you check that the API is bound to the correct version of the implementation. You see that it's set to the 7.1.0 version. Everything else in the `diff` file looks fine. You save your change. You also open the implementation's manifest file by clicking the `package.json` tab and verify that the implementation version is correct.

Back on the Policies page of the Deployment wizard, you click **Upload Policy Diff** and upload your modified policies file. After that's done, you go to the Confirmation page. You review the deployment information: name of the mobile backend, the source and

target environments, the dependencies that will be deployed to the target environment, and the `policy.properties` file that will be applied to the target environment.

Everything looks right and you click **Deploy**. You see that the deployment is successful. On the Mobile Backends page, you select **FIF_Customer 2.0** and see that Staging is now listed under the Deployments section and that the 7.1.0 API implementation is listed under dependencies.

The minor version change to the API implementation didn't require a change in version for the API, or the mobile backend. The fact that the API was already published wasn't an impediment to the deployment because the implementation could be published independently and associated with the API.

Rerouting a Mobile Backend

As you work on improving your product, you might find that you need to make some changes to it after you've already published your mobile backend. If the changes you want to make affect only the metadata of the mobile backend (that is, you're making minimal changes that won't require upgrading the mobile app that calls it), you can go ahead with those changes by rerouting the call to that mobile backend to a backend that has the updates.

You set the `Routing_RouteToBackend` environment policy to reroute the app's call to the original mobile backend to a new (target) backend. The app still calls the original mobile backend but the call is redirected to a new version of the backend or to an entirely different mobile backend that incorporates the changes. The app binary doesn't need to be recompiled because the app isn't directly associated with the new mobile backend. The app calls the original backend so there's no change to the client ID and client secret.

In this scenario, you'll see how to redirect calls to a mobile backend. Let's say you need to change the permissions for a storage collection. The app calls the mobile backend, `FIF_Customer 4.1` which is already published and deployed to a runtime environment called Production.

Assume you have the following setup:

- `FIF_Customer 4.1`
- `FIF_IncidentReports 2.0`API
- `incidentreports.js 7.2.0` API implementation
- `FIF_Parts 1.0` storage collection
- `RightNow 2.0` REST Connector API
- `Realm 1.1`

All artifacts are published and deployed to a runtime environment.

Updating the Collection and Rerouting the Call to the Mobile Backend

You have a collection, `FIF_Parts 1.0` which stores the images and serial numbers for various parts used to fix appliances. You defined the collection to have a Shared collection type. You've set Read-Only access permission to the Sales Representative role and Read-Write to the Engineer role and associated it `FIF_Customer 4.1` when the mobile backend was still in Draft state.

`FiF_Customer 4.1` has been published and you realize you forgot to give Read-Write permission to the Parts Manager role and `FIF_Parts 1.0` is published.

Here's how you can add the Parts Manager role to the Read-Write permission list and get it associated with a mobile backend that your app can call (without having to recompile your app):

- Create a new version of the `FIF_Parts 1.0` collection, and call it `FIF_Parts 1.5`.
- Set the access permissions for `FIF_Parts 1.5` just like you did for version 1.0 but this time make sure to add the Parts Manager role to the Read-Write permission list.
- Save the collection.
- Create a new version of the mobile backend, call it `FIF_Customer 4.5`.

  The new 4.5 version of the backend will have all the artifact associations of the 4.1 version.
- On the mobile backend landing page, open `FIF_Customer 4.5` , go to the **Storage** page and disassociate it from the old `FIF_Parts 1.0` and select `FIF_Parts 1.5`.
- After thorough testing, publish `FIF_Customer 4.5`. The publishing process lets you publish any unpublished dependencies at the same time.
- Deploy the mobile backend to the Production environment, reviewing and resolving any effects the deployment could have on other artifacts, and edit the routing policy:

  ```
  FIF_Customer(4.1).*.Routing_RouteToBackend=FIF_Customer(4.5)
  ```

When your app calls the mobile backend, it's redirected to the new version of the mobile backend that has the updated collection.

New Features

As you work on improving your product, at some point you're going to add at least one new major feature or make a major change to a feature. A major change will affect the mobile backend, requiring you to create a new version that won't be backward-compatible with previous versions. Changes could consist of adding another dependency or making a major change to the mobile backend or one of its dependencies. This time, deploying your mobile backend will mean that any mobile apps calling on the mobile backend will require that customer upgrade the application.

In this scenario, you introduce a new feature to your `FIF_Customer 2.1` mobile backend. Let's assume you have the following artifacts:

- `FIF_Customer 2.1` mobile backend already deployed to the Production environment
- `FIF_IncidentReports 2.0` API
- `incidentreports.js 7.2.0` API implementation
- `FIF_Images 1.0` storage collection
- `RightNow 2.0` REST Connector API
- `Realm 1.0`
- `Customer_LC_0112 1.0` user role

Upgrading the Mobile Backend

You've been using `FIF_IncidentReports 2.1` and now you've got a *new and improved* version of it that you want the mobile backend to use. You've created a 3.0 version of it that will improve how the incident report data is obtained and you want to make it available to the mobile app. The change in the major value of the version number implies that the functionality of the API isn't backward-compatible. The major change in the API necessitates creating a new version of the API implementation, which could affect any connector APIs that it calls, and definitely means that a new version of the mobile backend is needed.

Here's what you'll need to do to add a new major feature:

- Create a new major version of the API.
- Create a new major version of the API implementation for the new API.
- Test the API and the new implementation.
- If necessary, create a new major or minor version of the connector API and test it also.
- Create a new major version of the mobile backend by selecting the not backward-compatible option, which automatically increments the major version value. (In this scenario, the version will change from 2.1 to 3.0.)
- Select the new 3.0 version of the mobile backend, open it, and click the **APIs** navigation link. You click **X** to remove the association with the old 2.0 version of the API and click **Select APIs** to associate the new 3.0 version of the API.

> **Note:**
>
> You don't need to define new user roles, a new collection, or a realm but you will have to associate the roles and the collection with the new version of the mobile backend.

- Publish the mobile backend after you've thoroughly tested all the components. The publishing process lets you publish all the dependencies at the same time.
- Deploy the mobile backend to Staging, reviewing and resolving any effects the deployment could have on other artifacts, and modifying environment policies as needed.

The mobile apps that use `FIF_Customer` will have to upgrade to use the new version of the mobile backend. Later on, you decide that it isn't necessary to have a running 2.1 version because all the mobile apps have been upgraded to use the new version. You select the **FIF_Customer 2.1** mobile backend from the Mobile Backends page, and select **More > Manage Activation**. In the Manage Activation dialog, you change the state of the mobile backend to **inactive**. A mobile backend that's in the inactive state can't accept requests from mobile apps. The activation state is specific to an environment, so if you've deployed version 2.1 to Staging and then to Production, you'll need to change its activation state for each environment.

32
Managing an Artifact’s Lifecycle

Mobile backends, APIs, and other artifacts in Oracle Mobile Cloud Service (MCS) each have an independent lifecycle. As a mobile cloud administrator, you can manage the versioning, deployment, and dependency management of each.

In most respects, how an artifact is managed after it’s created is the same regardless of whether it’s a client, mobile backend, collection, connector API, or a custom API. You‘ve learned how to create an artifact, then modify it, and test it. Now that you have a viable artifact, it’s time to publish it, perhaps create new versions or update existing versions and eventually deploy it to another environment for others to test and use.

We'll show you how to take each of these artifacts through its lifecycle phases:

- Realm Lifecycle
- Client Lifecycle
- Mobile Backend Lifecycle
- API Lifecycle
- API Implementation Lifecycle
- Connector Lifecycle
- Collection Lifecycle

> **Note:**
>
> Remember, to perform operations on artifacts, such as viewing, creating new versions, editing, and so on in an environment, you need the following:
>
> - Permission to perform the operation on the artifact
> - Permission to access the environment containing the artifact
>
> If you can’t access the environment containing the instance of the artifact you want or if you can’t perform an operation on the artifact, ask your mobile cloud administrator for permission.

Realm Lifecycle

Realms go through lifecycle stages similar to other artifacts. You begin by creating a realm and publishing it.

You learned about realms in Creating Realms. Now it’s time to discover how to take a realm through its lifecycle.

If you think you need a better understanding of how artifacts interrelate in the overall MCS lifecycle before exploring the lifecycle of realms, see Lifecycle.

Publishing a Realm

Realms are created and tested in a development environment. They can then be published and deployed to other environments.

1. Make sure you're in the environment that contains the realm you want to publish.
2. Click  and select **Applications > Mobile User Management** from the side menu.
3. Click the **Realms** navigation link.
4. Select the realm to publish.
5. Click **Publish**.

   (Optional) You can enter a justification for publishing the realm in the **Comment** field.

After a realm is published, the user schema can't be changed. Only data can be updated, including adding or editing mobile user information.

Creating a New Version of a Realm

You can create a new version of a realm, which can be in a Draft or Published state.

1. Make sure you're in the environment containing the realm you want.
2. Click  and select **Applications > Mobile User Management** from the side menu.
3. Click the **Realms** navigation link.
4. Select the realm.
5. In the right section, select **More > New Version**.
6. (Optional) Add a brief description that states what distinguishes this version from the previous one.
7. Click **Create**.

Deploying a Realm

You must have Oracle Cloud identity domain administrator permissions to deploy a realm. Only the user schema is migrated during deployment. No mobile app user data is migrated. In the target environment, the realm won't have any users associated with it.

Sometimes deploying a realm along with its mobile backend and all the dependencies of that mobile backend can result in various permissions issues that can affect the deployment process. You can avoid these issues by deploying just the realm first and then deploying the mobile backends and its dependencies. This ensures that all the dependencies in the underlying security systems are resolved before deploying your mobile backend.

1. Make sure you're in the environment containing the realm you want to deploy.

2. Click  and select **Applications > Mobile User Management** from the side menu.
3. Click the **Realms** navigation link.
4. Select the realm to deploy.
5. In the right section, click **Deploy**.

   The Source Environment field is read-only and automatically defaults to the current environment of the mobile backend.
6. Specify the target environment.

   Only the environments for which you have permission to deploy to are listed. If the artifact you want to deploy has dependencies, you'll have to resolve those dependencies before attempting to deploy.
7. (Optional) Enter a comment about the deployment.
8. Click **Dependencies**.

   The Dependencies page lists the artifacts that the realm is dependent on. You can skip this page because realms don't have dependencies on other artifacts.
9. Click **Impact**.

   The Impact page lists artifacts in the target environment that will be affected when the realm is deployed. The data displayed is for your information only. Assess the effects and determine whether or not to proceed with the deployment.
10. Click **Policies**.

    The Policies page is where you can view the policies in both the source and target environments and edit policy values as needed.

    a. Click **Export** to see a `diff` file of the `policies.properties` file showing the policies in both the source and target environments.

    b. (Optional) Edit a policy value as needed.

       To get descriptions of environment policies, see Oracle Mobile Cloud Service Environment Policies. If you need to know more about environment policies, see Environment Policies.

    c. If you modified the `policies.properties` file, click **Import** to load it into the target environment.

       The `User_DefaultUserRealm` policy sets the realm version associated with a newly created mobile backend. In most cases, you'll want to use the default value, which is 1.0.
       You might want to modify this policy if you create another realm, for example, if you create a second realm and you want new mobile backends to be automatically associated with it. Instead of setting it at deployment, you set this policy at the environment level (that is, go to the **Administrator** view, select the environment and set this policy by clicking **Policies** and editing the `policies.properties` file. The value is applied to all realms created in that environment, therefore, don't set the value for a specific realm.)
11. Click **Confirm** and view the deployment configuration, then click **Deploy**.

To learn about deployment in MCS, see Deployment.

Moving a Realm to the Trash

Remove a realm by moving it to the trash.

1. Make sure you're in the environment containing the realm you want to remove.
2. Click and select **Applications > Mobile User Management** from the side menu.
3. Click the **Realms** navigation link.
4. Select the realm you want to remove.
5. In the right section, select **More > Move to Trash**.
6. Click **Trash** in the confirmation dialog if there are no dependency issues.

   If you think you or someone else might restore it later on, enter a brief comment about why you're putting this item in the trash.

To learn how dependencies can affect moving an artifact to the trash, see Dependencies That Affect a Move to the Trash. To restore a realm that's in the trash, see Restoring a Realm.

Restoring a Realm

1. Make sure that you're in the environment containing the realm you want to restore.
2. Click and select **Applications > Mobile User Management** from the side menu.
3. Click the **Realms** navigation link.
4. Click **Trash** ( ).
5. In the list of items in the trash, click by the realm you want and select **Restore from Trash** from the trash menu.

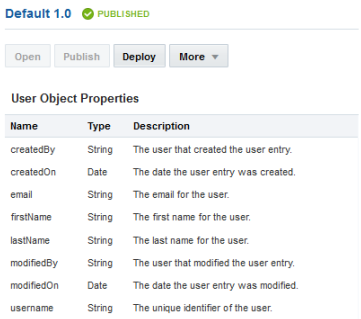
   You can see an example of the trash menu in Restoring an Artifact.
6. Click **Restore** in the confirmation dialog if there are no conflicts.

Restoring an artifact can cause conflicts if a duplicate artifact already exists. To restore an artifact when a duplicate artifact exists, see Restoring an Artifact.

Managing a Realm

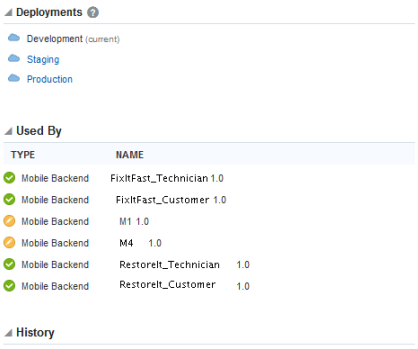
When at least one realm exists, you'll be taken to the Mobile User Management page every time you click and select **Applications > Mobile User Management** from the side menu. On the left side of the page, you see a list of all the mobile backends, their version numbers, and their Draft or Published state (mobile backends in Trash aren't displayed).

On the upper right side of the Realms page, you can open, test, publish, and deploy your realm. You can view the user object properties defined for this realm also:



- Click **Trash** to see which realms are in Trash.
- Click **Open** to see details about the selected realm.
- Click **Publish** to change the state of the realm from Draft to Published.
- Click **Deploy** to deploy the realm to another environment.
- Click **More** to create a new version, set the realm as the default realm, or move the realm to the trash.
- Click **User Object Properties** to see information about the users assigned to the realm.

On the lower side of the page, you can examine deployment, usage, and history details:



- Expand **Deployments** to see the environments that contain the selected the realm. You are only shown the environments that you have permission to access.

  Click on an environment to switch to it.
- Expand **Used By** to see the list of mobile backends that are associated with the realm.
- Expand **History** to quickly see the latest activity for the realm.

Client Lifecycle

If your mobile app uses push notifications or you want to use analytics to examine and improve your app, you need a client. As a mobile developer, you associate a client, which represents a mobile backend binary, with a mobile backend. Clients go through

similar lifecycle phases as other Mobile Cloud Service (MCS) artifacts with a few differences.

MCS can help you manage a client's lifecycle. You can publish, deploy, and export a client. You can modify its version number and move it to the trash when you don't need it anymore. Clients are top-level artifacts and their relationships with mobile backends can affect how clients and mobile backends are deployed, exported, imported, and moved to trash. see Client Management for details on creating clients.

If you want a general understanding of how artifacts interrelate in the overall MCS lifecycle, see Lifecycle.

Publishing a Client

When you're satisfied with a client's configuration, you can publish it but only if that client is associated with a mobile backend.

1. Make sure you're in the environment containing the client you want to publish.
2. Click ☰ and select **Applications > Client Management** from the side menu.
3. Select the client that you want to publish.
4. Click **Publish**.

   Dependencies are checked and if the associated mobile backend is in Draft state, the confirmation dialog lists it and informs you that it will be published with the client. If the mobile backend is already published, no dependencies are shown.
   If the mobile backend has downstream dependencies in Draft state, those dependencies will also be published. For example, `MyClient 1.1` references `MyMobileBackend 1.0`. `MyMobileBackend` has dependencies on published `MyAPI2.2` and unpublished `MyAPI2.4`. When you publish `MyClient 1.1`, the confirmation dialog only lists `MyMobileBackend1.0` as a dependency but `MyAPI2.4` is also published.
5. Click **Publish All**.

   If the mobile backend is in the trash, you won't be able to publish the client. Cancel the publish operation, and either restore the mobile backend or associate the client with a different mobile backend. Then try publishing the client again.

Usually, once an artifact is published it can't be changed. In the case of clients, however, you can add or remove the notification profiles associated with the client even if that client is published.

Updating the Version Number of a Client

When you create a client, you assign it a version number that is usually the version of the mobile app that the client represents. You can update its version number at any time if the client is in a Draft state. This is useful if a change to the binary was made and you need a new version designation for the client.
If you need to modify the version number for a draft client, you just need to open that client and change the value in the Version field.

1. Make sure you're in the environment containing the client you want.
2. Click ☰ and select **Applications > Client Management** from the side menu.
3. Open the client that you want to update from the list.

4. On the Settings page, change the value in the **Version** field.

   You'll get a message letting you know if you enter a duplicate version number (a version number that already exists for another client).

Creating a New Version of a Client

You can create a new version of a client regardless of whether it's in a Draft or Published state. When you create a new version of a client, you're basically cloning the client configuration. You can then make changes to the new version. For example, although a client can be associated with only one instance of a backend, that backend can reference multiple clients. You could create new versions of a client, where each client corresponds to a specific platform of a mobile app (iOS, Android, and Windows), and then edit each client to reference the same backend.
Another reason for creating new versions is to create multiple clients for the same platform if there are multiple mobile app binaries for the same platform that use the same backend.

> **Note:**
>
> Unlike other artifacts, which require that the version number use the *Major.minor* format, the version number for a client should be the same as the mobile app binary that's set by the app store. Depending on the version of the mobile app binary, the version could take the format of *Major.minor* or include an alphanumeric suffix with or without parentheses, a hyphen, space, or full stop. For example:
>
> - 1.2
> - 1.2 build 3452
> - 1.2 (3452)
> - 1.2–3452
> - 1.2.3 (01–Jun-2016)

1. Click  and select **Applications > Client Management** from the side menu.
2. Select the client that you want and then select **More > New Version**.
3. Enter a version number. (The same as the mobile app binary set by the app store.)
4. Click **Save**.

The new version is created in a Draft state.

Deploying Clients

When you have a published client that you're satisfied with, you can deploy it to another MCS environment. The Deployment wizard takes you, the mobile cloud administrator, through the process of specifying your target environment, identifying any dependencies, and alerting you to any dependency issues. You'll have the opportunity to view any possible effects that deploying the client could have on other artifacts.

When you deploy a client, the associated mobile backend is automatically deployed with it.

> **Note:**
>
> When you deploy a client, its associated mobile backend is automatically deployed with it. However, if you're deploying a mobile backend, the client associated with it isn't deployed.
> If you deploy a mobile backend without a client associated with it, you'll have to create a client for it in the target environment.

After the client has been deployed, the copy of the client in the target environment is assigned a new client ID and application key. Also the mobile backend associated with it is given an anonymous access key if HTTP Basic Authentication is enabled or a consumer key if OAuth is enabled. In addition, there's a different base URL for each environment. You'll need to incorporate all of these details into the apps that use this client.

You can view details for the client on the Clients page. The values applicable to that environment are shown in the **Keys** section, which includes the client application key value. To see the authentication and key information for the mobile backend, go to the Mobile Backends page.

On the Clients page, select your published client and click **Deploy** to open the Deployment wizard. Go to the links at the top of the wizard to complete these deployment steps:

- Specifying a Target Environment for the Client
- dependencies
- policies
- Deploying the Client

To learn more about deployment in MCS, see Deployment.

Specifying a Target Environment for the Client

After a client is published, the **Deploy** action is enabled. This step shows you how to designate a target environment.

1. Make sure you're in the environment containing the client you want to deploy.
2. Click  and select **Applications > Client Management** from the side menu.
3. Select the published client.
4. In the right section, click **Deploy**.

   The Source Environment field is read-only and automatically defaults to the current environment of the client.
5. Select the target environment for your deployment.

   Only the environments for which you have deployment permission are listed. If the artifact you want to deploy has dependencies, you'll have to resolve those dependencies before attempting to deploy.

6. (Optional) Enter a descriptive statement about the deployment.

Identifying Dependencies and Deployment Impact

The next two navigation links let you view all the dependencies related to the artifact and whether they're currently deployed in the target environment. You can also see what impact the deployment might have on other artifacts.
When you deploy a client, the mobile backend that it references is also deployed. You can't deploy a client unless it's associated with a mobile backend.

1. Click **Dependencies**.

   Lists artifacts that must already be deployed to the target environment and on which the current artifact depends. You can see potential deployment issues, such as version conflicts, missing implementations (depending on the artifact), and so on. This gives you the opportunity to cancel the deployment and fix any potential issues. Afterward, you can go back and deploy your artifact.

   > **Note:**
   >
   > You won't see profiles listed because profiles aren't deployed with clients. You'll have to manually create the profiles you need in the target environment after you deploy the client.

2. Click **Impact**.

   Shows notifications of possible effects on the artifact. The data displayed here is for your information only. Assess the effects and determine whether or not to proceed with deployment. If there's an issue, you can cancel the deployment process. After the issue is resolved, you can try deploying again.

Setting Environment Policies for Clients

Each environment has policies that govern the behavior of the artifacts within that environment. Policies and policy values in one environment can differ from policies in another environment. For instance, if you're deploying from a development environment to a runtime environment, you might want a higher degree of logging information in the runtime environment because your artifact will be thoroughly tested there prior to being made publicly accessible. The Policies page is where you can view the policies in both the source and target environments and edit policy values as needed.

1. Click **Policies**.
2. Click **Export** to see a diff file of the `policies.properties` file showing the policies in both the source and target environments.

   To get a description of policies and their default values, see Oracle Mobile Cloud Service Environment Policies. See Environment Policies for a general discussion on environment policies.
3. If you modified the `policies.properties` file, click **Import** to load it into the target environment.

To see what policies need to be updated in the target environment, go to the deployment instructions for the specific artifacts.

Deploying the Client

View the basic details of your client deployment: the current environment, the target environment, and any dependencies or policies. You can cancel or go back if you want to make changes to your deployment. If you're satisfied, you can deploy the client.

1. Click **Confirm**.

   The details of the deployment are displayed in a read-only section.

2. Click **Deploy**.

   A confirmation page is displayed that informs you if the deployment succeeded.

After a successful deployment, you can return to the Clients page or go to the Administration tab to review the policy settings in the target environment.
You can manually add notification profiles to the client in the target environment.

Moving a Client to the Trash

Remove a draft or published client by moving it to the trash. If the client is needed later on, you can restore it from the trash.

> **Note:**
>
> Moving a client to the trash does not move the associated mobile backend or any profiles referenced by the client to the trash.

1. Make sure you're in the environment containing the client you want to remove.
2. Click  and select **Applications > Client Management** from the side menu.
3. Select the client, then select **More > Move to Trash**.
4. Click **Trash** in the confirmation dialog if there are no dependency issues.

   If you think you or someone else might restore it later on, enter a brief comment about why you're putting this item in the trash.

To find out how dependencies can affect moving an artifact to the trash, see Dependencies That Affect a Move to the Trash. To restore a client that's in the trash, see Restoring a Client.

Restoring a Client

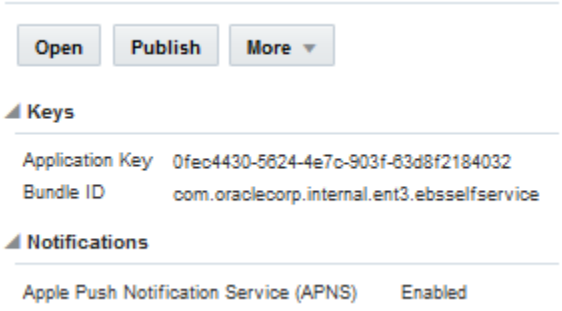
1. Make sure that you're in the environment containing the client that you want to restore.
2. Click  and select **Applications > Client Management** from the side menu.
3. Click Trash ().
4. In the list of items in the trash, click by the client you want and select **Restore from Trash**.
5. Click **Restore** in the confirmation dialog if there are no conflicts.

Restoring an artifact can cause conflicts if a duplicate exists. To find out more about restoring an artifact when a duplicate artifact exists, see Restoring an Artifact.

Managing a Client

When at least one client exists, you'll be taken to the Clients page every time you click and select **Applications > Client Management** from the side menu. On the left side of the page, you see a list of all the clients, their version numbers, and their Draft or Published state (clients in the trash aren't displayed).

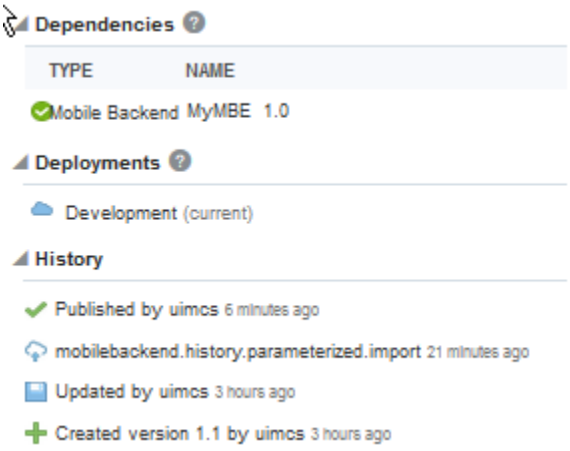
On the Clients page, you can open, test, publish, deploy, see the client dependencies, history, and the environments containing the client:



- Click **Open** to see details about the selected client.
- Click **Publish** to change the state of the client.
- Click **Deploy** to deploy the client to another environment in the same MCS instance.
- Click **More** to create a new version, export the client to another instance of MCS, or move the client to the trash.
- Click **Trash** ( ) to see which clients are in the trash.
- Expand **Keys** to obtain the values for the client ID and the application key.

  To see application key and client ID information for clients deployed to another environment, click on an environment in the Deployments section.
- Expand **Notifications** to see which push notifications, if any, are enabled for this client.

On the lower right side of the page, you can view data about the selected client:



- Expand **Dependencies** to see the mobile backend that this client references.

  > **Note:**
  >
  > Only the mobile backend is listed. If the mobile backend has downstream dependencies, go to **Applications > Mobile Backends** and view them from the Dependencies section of the selected mobile backend.

- Expand **Deployments** to see the environments that contain the client. Note: You're only shown the environments that you have permission to access.

  Click on an environment to switch to it.

- Expand **History** to quickly see the latest activity for the client.

Mobile Backend Lifecycle

You, the mobile developer, have created a mobile backend and now it's time to use it by publishing and deploying it. Remember that after you publish it, it becomes immutable, that is, you can't modify it.

If you want to make a change, you can create a new version of it. Because mobile backends are tightly integrated with custom code, APIs, and other objects in Oracle Mobile Cloud Service, you'll need to consider the relationships and dependencies on those objects.

If you think you need a better understanding of how artifacts interrelate in the overall MCS lifecycle before exploring the lifecycle of mobile backends, see Lifecycle.

Backend Lifecycle States

Relationships with other artifacts create dependencies. For example, your backend might depend on other artifacts, such as collections or APIs. When any artifact changes state, all dependent artifacts must also change states. Oracle Mobile Cloud Service keeps track of any dependencies for you.

Backends have the following activation states that determine whether they can be updated, deleted, or whether or not a new version can be created:

- **Active**: Denotes the version of the backend is valid and active.
- **Quiesce**: Denotes the version of the backend has become quiet, that is, it no longer supports new requests, and after all currently running requests are completed, it's changed to Inactive. This is a transitional state.
- **Inactive**: Denotes the version of the backend that's present but not in an active state (that is, not usable).

  If a user tries to access an API through an inactive backend, a 404 code is returned.
- **Deleted**: Denotes the version of the backend that's been moved to the trash and susceptible to a hard delete (actually removed from the repository).

> **Note:**
>
> Only mobile cloud administrators can purge (that is, permanently delete) an item in the trash.

Publishing a Mobile Backend

Follow these steps to publish a mobile backend. When a backend is published, all dependencies that aren't yet published must also be published.

1. Make sure you're in the environment containing the mobile backend you want to publish.
2. Click  and select **Applications > Mobile Backends** from the side menu.
3. Select the mobile backend that you want to publish.
4. Click **Publish**.

   The Confirm Publish dialog opens:

5. In the Confirm Publish dialog, click **Check Dependencies** to reveal whether or not the backend has dependencies and what those dependencies are so you'll know how to proceed:
   - If you don't have dependencies, a confirmation dialog is displayed. Click **Publish**.

- If any dependencies are found in the trash, they're listed. Cancel the publish operation, restore the dependent items from the trash, and restart the process.
- If there are dependencies in the Draft state, they're listed in the confirmation dialog. You have the option to publish all the dependent artifacts along with your mobile backend. Click **Publish All**.

Published mobile backends can be deployed to your staging server or your production server.

Updating the Version Number of a Backend

If you created a new version of a backend using the New Version dialog, you can update its version number if it's still in a Draft state. This is useful if you need to designate a different version number for it before you publish it or you've made a change to the configuration and you need a new version designation.

1. Make sure you're in the environment containing the backend you want.
2. Click  and select **Applications > Mobile Backends** from the side menu.
3. Select the backend you want to update from the list.
4. In the right section, select **More > Update Version Number**.
5. Enter a version number of the format *Major.minor*.

   The previous version of the backend is displayed next to the field. You'll get a message letting you know if you've entered an existing version number.
6. (Optional) Add a brief description that states what distinguishes this version from the previous one.
7. Click **Update**.

   A confirmation message is displayed. A draft of the new version is added to the list of backends.

Creating a New Version of a Backend

When you create a new backend, the version is automatically set to 1.0. As long as the backend is in a Draft state, you can change any aspect of it. As you develop your backend, you can change the version's major and minor version values as you see fit.

You can use a published backend as a root for a new version.

1. Make sure you're in the environment containing the backend.
2. Click  and select **Applications > Mobile Backends** from the side menu.
3. Select the published backend.
4. In the right section, select **More > New Version**.

The new version is created in a Draft state.

> **Note:**
>
> If the backend is associated with an API, you can't associate another version of that API with the backend, regardless of whether the backend is in a Draft or Published state. You must create a new version of the backend and associate it with the other API version.

Deploying Mobile Backends

The Deployment wizard takes you, the mobile cloud administrator, through the process of specifying your target environment, identifying any dependencies, and alerting you to any dependency issues, such as associated APIs that aren't deployed yet. You'll have the opportunity to view any possible effect that deploying the mobile backend could have on other artifacts. Because you're deploying from one environment to another, the policies applied to the mobile backend in the source environment may need to change in the target environment. You can view and modify these policies before you deploy your mobile backend.

After the mobile backend has been deployed, the copy of the mobile backend in the target environment is assigned a new mobile backend ID and either an anonymous access key if HTTP Basic Authentication is enabled or a consumer key if OAuth is enabled. In addition, there's a different base URL for each environment. You'll want to incorporate all of these details into the apps that use the mobile backend. You can view these details on the Mobile Backends page. Select the mobile backend and click one of the environments listed under the **Deployments** section. The values applicable to that environment are shown in the **Keys** section.

Select your published mobile backend and click **Deploy** to open the Deployment wizard. Go to the links at the top of the wizard to complete these deployment steps:

- Specifying a Target Environment for the Mobile Backend
- Identifying Dependencies and Deployment Effects
- Setting Environment Policies for Mobile Backends
- Deploying the Mobile Backend

To learn about deployment in MCS, see Deployment.

Specifying a Target Environment for the Mobile Backend

After a mobile backend is published, the **Deploy** action is enabled. This step shows you how to designate a target environment.

1. Make sure you're in the environment containing the mobile backend you want to deploy.
2. Click  and select **Applications > Mobile Backends** from the side menu.
3. Select the published mobile backend.
4. In the right section, click **Deploy**.

   The Source Environment field is read-only and automatically defaults to the current environment of the mobile backend.
5. Select the target environment for your deployment.

Only the environments for which you have deployment permission are listed.

If the artifact you want to deploy has dependencies, you'll have to resolve those dependencies before attempting to deploy.

6. (Optional) Enter a descriptive statement about the deployment.

Identifying Dependencies and Deployment Effects

The next two navigation links let you view all the dependencies related to the artifact and whether they're currently deployed in the target environment. You can also see what impact the deployment might have on other artifacts.

1. Click **Dependencies**.

   Lists artifacts that must already be deployed to the target environment and on which the current artifact depends. You can see potential deployment issues, such as version conflicts, missing implementations (depending on the artifact), and so on. This gives you the opportunity to cancel the deployment and fix any potential issues. Afterward, you can go back and deploy your artifact.
   If the call to the mobile backend that's being deployed is rerouted, the name and version of the target mobile backend (as defined in the `Routing_RouteToBackend` policy for the mobile backend being deployed) is shown. The target mobile backend is not a dependency of the original mobile backend, so it won't be automatically deployed. You must manually deploy the target mobile backend to the target environment if it doesn't exist there already.

2. Click **Impact**.

   Shows notifications of possible effects on the artifact. The data displayed here is for your information only. Assess the effects and determine whether or not to proceed with deployment.
   If there's an issue, you can cancel the deployment process. After the issue is resolved, you can try deploying again.

Setting Environment Policies for Mobile Backends

Each environment has policies that govern the behavior of the artifacts within that environment. Policies and policy values in one environment can differ from policies in another environment. For instance, if you're deploying from a development environment to a runtime environment, you might want a higher degree of logging information in the runtime environment because your artifact will be thoroughly tested there prior to being made publicly accessible. The Policies page is where you can view the policies in both the source and target environments and edit policy values as needed.

1. Click **Policies**.
2. Click **Export** to see a `diff` file of the `policies.properties` file showing the policies in both the source and target environments.

   Set the `Sync_CollectionTimeoutToLive` policy to specify the default amount of time you want data in a storage collection to remain in the cache.

   If the call to the mobile backend being deployed is rerouted to another backend, set the `Routing_RouteToBackend` policy to specify the name and version of the mobile backend being deployed (the original mobile backend) and the target backend. The target backend is not automatically deployed with the original mobile backend. Deploy the target mobile backend to the target environment if it doesn't

exist there already. See Making Changes After a Backend is Published (Rerouting).

To get a description of policies and their default values, see Oracle Mobile Cloud Service Environment Policies.

3. If you modified the `policies.properties` file, click **Import** to load it into the target environment.

Deploying the Mobile Backend

View the basic details of your mobile backend deployment: the current environment, the target environment, and any dependencies or policies. You can cancel or go back if you want to make changes to your deployment. If you're satisfied, you can deploy the mobile backend.

1. Click **Confirm**.

   The details of the deployment are displayed in a read-only section.

2. Click **Deploy**.

   A confirmation page is displayed that informs you if the deployment succeeded. For successful deployments, you can choose to return to the Mobile Backends page or go to the Administration tab to review the policy settings in the target environment.

Moving a Backend to the Trash

Remove a backend in a by moving it to the trash. A backend in the trash is no longer listed but it's still viable, that is, it could continue to serve requests. If the backend is needed later on, you can restore it from the trash.

> **Note:**
>
> If a backend is referenced by a client, you can't move that backend to the trash. If the backend is in Draft state, you can disassociate it from the client by opening the backend, selecting **Clients** in the navbar and clicking **Delete** (**X**) for that client. Then you can move the backend to the trash.

An alternative to removing a backend is to deactivate it, in which case it no longer services requests. See Deactivating a Mobile Backend for information.

1. Make sure you're in the environment containing the backend you want to remove.
2. Click  and select **Applications > Mobile Backends** from the side menu.
3. Select the backend.
4. In the right section, select **More > Move to Trash**.
5. Click **Trash** in the confirmation dialog if there are no dependency issues.

   If you think you or someone else might restore it later on, enter a brief comment about why you're putting this item in the trash.

To find out how dependencies can affect moving an artifact to the trash, see Dependencies That Affect a Move to the Trash. To restore a backend that's in the trash, see Restoring a Backend.

If you move a backend to the trash that has been redirected to another backend, the redirection still occurs.

Restoring a Backend

1. Make sure that you're in the environment containing the backend that you want to restore.
2. Click  and select **Applications > Mobile Backends** from the side menu.
3. Click **Trash** ().
4. In the list of items in the trash, click  by the backend you want and select **Restore from Trash**.
5. Click **Restore** in the confirmation dialog if there are no conflicts.

Restoring an artifact can cause conflicts if a duplicate exists. To find out more about restoring an artifact when a duplicate artifact exists, see Restoring an Artifact.

Deactivating a Backend

If you want to stop access to a backend without deleting it, you can do so by deactivating it. A deactivated backend can't service any more requests. Deactivation is most common for backends in a Published state that have been replaced by newer versions and are no longer needed.

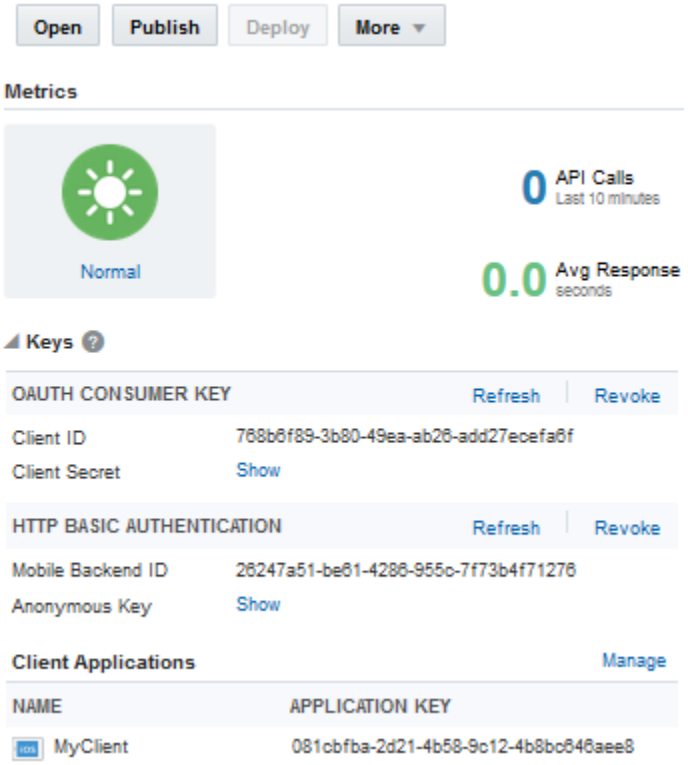
1. Make sure you're in the environment containing the backend you want to deactivate.
2. Click  and select **Applications > Mobile Backends** from the side menu.
3. Select your backend and click **Manage**.
4. In the dialog that appears, select **Inactive** from the drop-down list to deactivate the backend, or **Active** to reactivate an inactive backend.
5. Click **Save**.

If you deactivate a backend that has been redirected to another backend, the redirection still occurs.

Managing a Mobile Backend

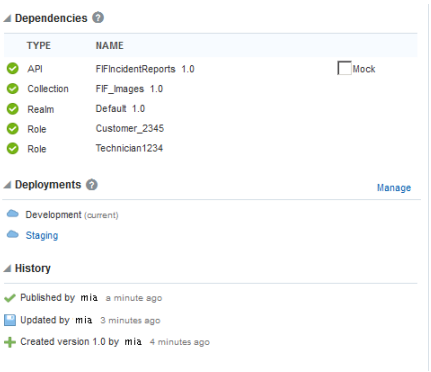
When at least one mobile backend exists, you'll be taken to the Mobile Backends page every time you click  and select **Applications > Mobile Backends** from the side menu. On the left side of the page, you see a list of all the mobile backends, their version numbers, and their Draft or Published state (mobile backends in Trash aren't displayed).

On the upper right side of the Mobile Backends page, you can open, test, publish, deploy, see runtime data about your mobile backend, and get authentication and application key values:



- Click **Trash** to see which mobile backends are in the trash.
- Click **Open** to see details about the selected mobile backend.
- Click **Publish** to change the state of the mobile backend from Draft to Published.
- Click **Deploy** to deploy the mobile backend to another environment.
- Click **More** to create a new version, update an existing version, change the activation state, or move the mobile backend to the trash.
- Look in the **Metrics** section to see the number of calls to the API associated with the mobile backend and the average response time.
- Expand **Keys** to obtain the values for the mobile backend ID, anonymous key (click **Show**), and the application key for the associated client.

On the lower right side of the page, you view data about the selected mobile backend:



- Expand **Dependencies** to see the artifacts the mobile backend is dependent on.
- Expand **Deployments** to see the environments that contain the selected the mobile backend. You are only shown the environments that you have permission to access.

  Click on an environment to switch to it.

  Click **Manage** to change the activation state of the mobile backend.
- Expand **History** to quickly see the latest activity for the selected mobile backend.

To see metrics information for mobile backends in another environment, for example Staging, switch to it by selecting **Staging** from the environment drop-down list (across from Metrics). Select **Staging** from the environment list across from Keys to see the HTTP Basic Authentication values in the Staging environment for the Mobile Backend ID, Anonymous Key, and Application Key fields.

To see metric and key information for mobile backends in the Production environment, choose Production from the environment drop-down lists.

Mobile Client SDK Demo Applications

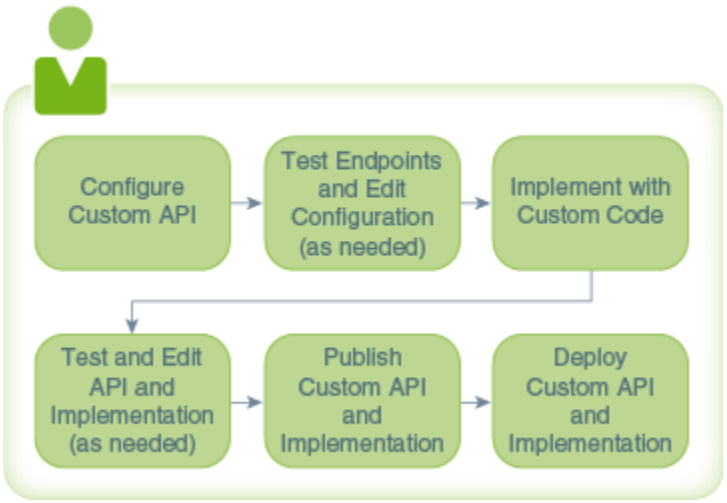
The mobile client SDK provides a single demo application, FixItFast, which runs on each of the MCS-supported vendor platforms, iOS and Android.

FixItFast simulates an enterprise company application very close to what can be found in the real world. Its purpose is to give application developers a good idea of what's possible and how this mobile application can be set up and run with MCS.

API Lifecycle

The lifecycle stages of custom APIs and API implementations are similar. Both artifacts go through a design-time phase where each is created, tested, edited, and then published.

The following figure shows the life stages of custom APIs and their implementations:



When you create a new custom API, its version is automatically set to 1.0 and it's considered to be in a Draft state. In the Draft state, you can test and edit your API as often as needed.

When you're satisfied with your API configuration, publish it with the understanding that a published API can't be changed. APIs are implemented with custom code. To make a change to a published API, create a new version of the API. For custom APIs, you'll also need to create a new implementation for the new version.

As you develop your API, you can change the version's major and minor values as you see fit, that is, creating a new version of your API or updating an existing version. After you've implemented, tested, and published your API, you can deploy it to one or more environments (for example, you can deploy from a development environment to one or more runtime environments if you have multiple environments). Eventually, the API may become obsolete, and you can move it to the trash.

If you think you need a better understanding of how artifacts interrelate in the overall MCS lifecycle before exploring the lifecycle of custom APIs, see Lifecycle.

Publishing a Custom API

Before you can deploy a custom API, you must publish it first. As soon as it's published, the API can't be changed. You can create a new version of it, but you cannot edit it.

> **Note:**
>
> You must have an implementation associated with the API to publish it. A mock implementation is provided by default. To associate an implementation other than the mock implementation, open the API, and click **Implementations** in the left navigation bar. Select the implementation you want and click **Set as Default**.

1. Make sure you're in the environment containing the custom API you want to publish.
2. Click  and select **Applications > APIs** from the side menu.
3. Select the draft API that you want to publish.
4. Click **Publish**.

   You can enter a justification for publishing in the Comment field.

When the API is published, you're returned to the APIs page where you can see the updated status of your API.

> **Note:**
>
> Custom APIs can be published independently of implementations. When you publish an API, the implementation isn't published automatically. To understand the relationship between custom APIs and their implementations, see Custom APIs and API Implementations.

Custom APIs and API Implementations

Oracle Mobile Cloud Service tracks a custom API as it's created, saved, published, deployed, implemented, deactivated, and reactivated. Custom APIs can be published independently or when a related mobile backend is published. The relationship between custom APIs and their implementations is given in the following sections.

Scope and Version Format

Both custom APIs and API implementations have versions that use the format `Major.minor`.

Active Versions

If you have multiple environments, each environment can contain multiple active versions of a custom API.

Though there can be multiple active versions of an API implementation per environment, only a single implementation version is mapped to a specific API version.

Draft and Published States

Both custom APIs and API implementation can have a Draft state or a Published state. A custom API can be published independently or published when a related mobile backend is published.

An API implementation can be published independently.

Actions Tracked by Oracle Mobile Cloud Service

MCS tracks the following operations for custom APIs: Create, Update, Publish, and Move to Trash.

MCS tracks the following operations for API implementations: Create and Save.

Number of APIs Referenced From a Mobile Backend

A mobile backend can reference multiple APIs, with each API having a specific version. That is, only one version of a given API can be referenced by a mobile backend. For example, a mobile backend can't reference both `myAPI1.1` and `myAPI2.0`, but it can reference both `myAPI1.1` and `yourAPI2.0`.

An API implementation isn't referenced directly by a mobile backend. The implementation is referenced by the API version, which is in turn referenced by the mobile backend.

Dependencies

A custom API is dependent on the active API implementation (as determined by the environment policy).

An API implementation is dependent on the API that implements it and other APIs that custom code call (as listed in the file manifest).

In reverse, mobile backends and API implementations are dependent on custom APIs. For an API implementation, it is a dependency of any APIs that list it as the active or default implementation.

Environment Policy Attributes

A custom API is affected by the API version to implementation policy mapping and the default API version setting.

An API implementation is affected by the number of node instances per virtual machine and standard runtime policies such as read-only, log-levels, etc.

At deployment, the API version to implementation policy mapping must be set in the target environment for a custom API. When deploying an API implementation, any policy that is referenced must be defined in the target environment.

For descriptions of environment policies and their values, see Oracle Mobile Cloud Service Environment Policies.

Updating the Version Number of an API

If you created a new version of an API using the New Version dialog, you can update the version number of the API if it's still in a Draft state. This is particularly useful if you need to designate a different version number for it before you publish the API.

1. Make sure you're in the environment containing the custom API you want.
2. Click  and select **Applications > APIs** from the side menu.
3. Select the API you want.
4. Select **More > Update Version Number**.
5. Enter a version number of the format *Major.minor*.

   The previous version of the API is displayed next to the field. You'll get a message letting you know if you enter an existing version number.
6. (Optional) Add a brief description that states what distinguishes this version from the previous one.
7. Click **Update**.

   A confirmation message is displayed. A draft of the new version is added to the list of APIs.

Creating a New Version of an API

You can make a new version of a custom API regardless of whether it's in a Draft or Published state. When you create a new version of a custom API, you are basically cloning the API configuration and making changes to it alone. You can specify the implementation to associate with the new version of the API. You can upgrade your custom API easily by creating a new version of it:

1. Make sure you're in the environment containing the custom API you want.
2. Click  and select **Applications > APIs** from the side menu.
3. Select the API.

   You can create a new version of a custom API whether it's in a Draft or Published state.
4. In the right section, select **More > New Version**.

Oracle Mobile Cloud Service checks for any dependencies on other APIs and for an associated implementation.

5. Enter a version number in the format *Major.minor*.

   If you enter a version number that already exists, you'll get a message letting you know that number is already in use.

6. (Optional) Add a brief description that states what distinguishes this version from the previous one.

7. Click **Create**.

   A confirmation message is displayed. A draft of the new version is created and is visible in the API Catalog.

Deploying APIs

To make a custom API accessible for use, it needs to be deployed to a runtime environment. Generally, a custom API would get deployed at the same time as the mobile backend with which it's associated. When a new version of a custom API is published, it can be deployed to a target environment.

If you deploy an artifact that depends on an API not currently in the target environment, that API is automatically deployed, however, you won't be able to select the implementation for that API. If that API can't be deployed (due to version conflict or not being in Published state, for example), the deployment process stops.

The Deployment wizard takes you, the mobile cloud administrator, through the process of specifying your target environment, identifying any dependencies and alerting you to any dependency issues, such as the implementation associated with the API is a mock implementation. You'll have the opportunity to view any possible effects that deploying the API could have on other artifacts. Because you're deploying from one environment to another, the policies applied to the API in the source environment may need to change in the target environment; you can view and modify these policies before you deploy your API.

Select your published custom API and click **Deploy** to open the Deployment wizard. Go to the links at the top of the wizard to complete these deployment steps:

- Specifying a Target Environment
- Identifying Dependencies and Deployment Effects
- Setting Environment Policies for APIs
- Deploying the API

To learn about deployment in MCS, see Deployment.

Specifying a Target Environment

After a custom API is published, the **Deploy** action is enabled.

1. Make sure you're in the environment containing the custom API you want to deploy.

2. Click  and select **Applications > APIs** from the side menu.

3. Select the published custom API.

4. Click **Deploy**.

   The Deployment wizard opens on the Target page. The Source Environment is read-only and automatically defaults to the current environment of the API.

5. Select the target environment for your deployment.

   Only the environments for which you have deployment permission are listed.

   If the artifact you want to deploy has dependencies, you'll have to resolve those dependencies before attempting to deploy.

6. Specify the implementation to associate with the API.

   The default implementation is provided for you. If you don't want to use the default or if the default is a mock implementation, select another published implementation from the drop-down list.

   You can deploy a custom API only if it's associated with a published real implementation (that is, it's not a mock implementation). The dependencies check in the next step shows the state of the associated implementation. If it isn't published or if you don't have a real implementation associated with the API, you'll have to cancel the deployment and address the issue. Then you can try deploying the API again.

7. Enter a description about the deployment.

Identifying Dependencies and Deployment Effects

The next two navigation links let you view all the dependencies related to the artifact and whether they're currently deployed in the target environment. You can also see what impact the deployment might have on other artifacts.

1. Click **Dependencies**.

   Lists artifacts that must already be deployed to the target environment and on which the current artifact depends. You can see potential deployment issues, such as version conflicts, missing implementations (depending on the artifact), and so on. This gives you the opportunity to cancel the deployment and fix any potential issues. Afterward, you can go back and deploy your artifact.
   If the call to the mobile backend that's being deployed is rerouted, the name and version of the target mobile backend (as defined in the `Routing_RouteToBackend` policy for the mobile backend being deployed) is shown. The target mobile backend is not a dependency of the original mobile backend, so it won't be automatically deployed. You must manually deploy the target mobile backend to the target environment if it doesn't exist there already.

2. Click **Impact**.

   Shows notifications of possible effects on the artifact. The data displayed here is for your information only. Assess the effects and determine whether or not to proceed with deployment.
   If there's an issue, you can cancel the deployment process. After the issue is resolved, you can try deploying again.

Setting Environment Policies for APIs

Each environment has policies that govern the behavior of the artifacts within that environment. Policies and policy values in one environment can differ from policies in another environment. For instance, if you're deploying from a development

environment to a runtime environment, you might want a higher degree of logging information in the runtime environment because your artifact will be thoroughly tested there prior to being made publicly accessible. The Policies page is where you can view the policies in both the source and target environments and edit policy values as needed.

1. Click **Policies**.
2. Click **Export** to see a `diff` file of the `policies.properties` file showing the policies in both the source and target environments.
3. (Optional) Edit a policy value as needed.

   The policies that you'll want to set at deployment are:

   - `Routing_BindAPIToImpl`: Associates an implementation with the API.
   - `Network_HttpRequestTimeout`: Sets the amount of time to read a request before the operation times out. The default value is `40,000 ms`.

   For a description of API policies and their default values, see Oracle Mobile Cloud Service Environment Policies. To learn about environment policies, including scope and naming formats, see Environment Policies.
4. If you modified the `policies.properties` file, click **Import** to load it into the target environment.

Deploying the API

View the basic details of your API deployment: the current environment, the target environment, and any dependencies or policies. You can cancel or go back if you want to make changes to your deployment. If you're satisfied, you can deploy the API.

1. Click **Confirm**.

   The details of the deployment are displayed in a read-only section.
2. Click **Deploy**.

   A confirmation page is displayed that informs you if the deployment succeeded. For successful deployments, you can choose to return to the APIs page or go to the Administration tab to review the policy settings in the target environment.

Moving a Custom API to the Trash

Remove a custom API by moving it to the trash. If the API is needed later, you can restore it from the trash.

1. Make sure that you're in the environment containing the custom API that you want to remove.
2. Click  and select **Applications > APIs** from the side menu.
3. Select the custom API you want to remove.
4. In the right section, select **More > Move to Trash**.
5. Click **Trash** in the confirmation dialog if there are no dependency issues.

   If you think you or someone else might restore it later on, enter a brief comment about why you're putting this item in the trash.

Restoring a Custom API

1. Make sure that you're in the environment containing the custom API that you want to restore.
2. Click and select **Applications > APIs** from the side menu.
3. Click **Trash** ( ).
4. Make sure *APIs* is selected in the trash drawer.
5. In the list of items in the trash, click by the API you want and select **Restore from Trash**.
6. Click **Restore** in the confirmation dialog if there are no conflicts.

When you restore an API, its implementations are not restored with it. You'll have to manually restore the implementations you want and designate an implementation as the default. Open the restored API, click **Implementations** from the navbar, and set an implementation as the default.
Restoring an artifact can cause conflicts if a duplicate artifact already exists. To restore an artifact when a duplicate artifact exists, see Restoring an Artifact.

Managing an API

After you create a custom API, you'll want to edit it, publish it, see what implementations are associated with it, in short, you want to be able to manage the API and examine details of the APIs created by other service developers. The APIs page gives you access to all these features.

When at least one custom API exists, you'll be taken to the APIs page every time you click and select **Applications > APIs** from the side menu. On the left side of the page, you'll see a list of all the custom APIs except for those in the trash. You can see which APIs are in the Draft state and which are in the Published state. Every API is listed by its name and version number.

The right side of the page is where you can open, test, publish, and examine data about your custom API.

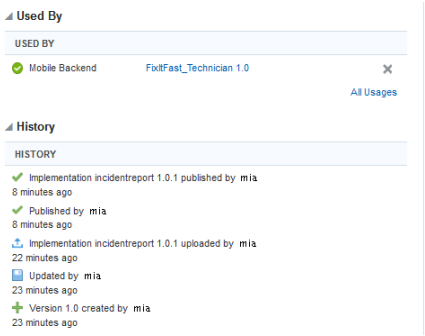
On the upper right side of the APIs page, you can perform the following actions:

- Click **Open** to view details and settings for the selected custom API.
- Click **More** to create a new version, update an existing version, or move an API to the trash.

- Expand **Implementations** to see what implementations are available, along with their version numbers and whether they are in a Draft or Published state. Click **Manage** to go directly to the Implementations page.
- Expand **Deployments** to see the environments that contain the selected the API. You are only shown the environments that you have permission to access.

  Click on an environment to switch to it.

On the lower right side of the page, you view data about the selected API:



- Expand **Used By** to see the list of the backends that call on the API.

  Click **All Usages** to see the complete list.
- Expand the **History** section to quickly see the latest activity for the selected custom API.

API Implementation Lifecycle

After you have an API implementation in a Draft state that's configured and tested, you're ready to publish and deploy it to another environment. API implementations go through the same lifecycle phases as APIs, in addition to being published and deployed, new versions can be created, existing versions can be updated, and obsolete implementations can be moved to the trash.

Remember that after an API implementation is published, it can't be changed. If you're still configuring and testing the implementation, keep it in a Draft state until it's ready for the next phase of the lifecycle.

If you think you need a better understanding of how artifacts interrelate in the overall MCS lifecycle before exploring the lifecycle of API implementations, see Lifecycle.

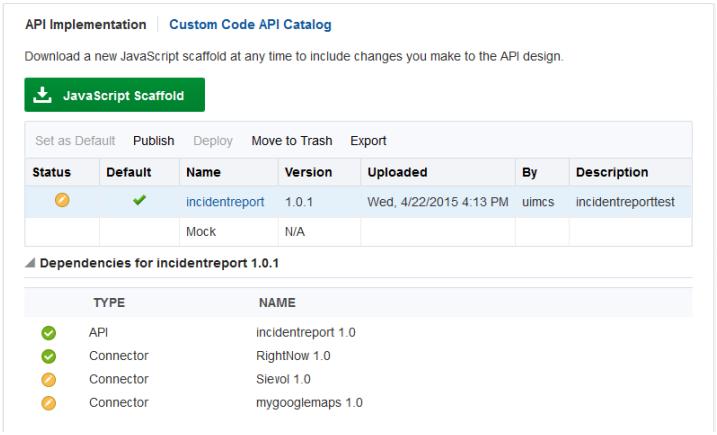
Publishing an API Implementation

You can publish an implementation that contains real, non-mock data from the API Implementation page. Implementations can be published independently of APIs and can have separate versions as well. This lets you make changes to a published implementation, such as minor modifications or bug fixes, without requiring the API itself to be updated.

1. Make sure you're in the environment containing the API implementation that you want to publish.

2. Click and select **Applications > APIs** from the side menu.
3. Select the API associated with the implementation that you want to publish.
4. Expand **Implementations** in the right section and click **Manage**.

   The API Implementation page is displayed:



   You can see the list of dependencies by expanding the **Dependencies** section of the API Implementation page. The API associated with the implementation and any other APIs or connectors that the implementation calls are listed. You can see which dependency is in a Draft state, a Published state, or is unresolved.

5. Select the implementation and click **Publish**.

   A dependency search is performed. If unresolved dependencies are found, the implementation can't be published. Resolve the issue and try publishing the implementation again.

   If any API dependencies are declared through the `Oracle-Mobile-API-Version` header instead of through the `package.json` file, the API Designer isn't aware of dependencies declared through the header and won't prompt you with information when you publish the calling API. In this case, you must remember to publish the dependent API yourself.

6. If unpublished dependencies are found, click **Publish All** to publish all the listed unpublished artifacts.

   If you don't want to publish all the dependencies with your implementation, click **X** to cancel the operation. You can either publish the dependencies individually or edit your implementation to remove them.

When the implementation is published, the **Deploy** command is enabled.

Creating a New Version or Updating the Version of an API Implementation

Implementations can be published independently of APIs and can have separate versions as well. This lets you make changes to a published implementation, such as minor modifications or bug fixes, without requiring the API itself to be updated. You can create a new version of an API Implementation that is in a Draft or Published state. If you want to make changes to a published implementation, you must create a new version of it.

If you have previously uploaded an implementation with a given version specified and that implementation is still in a Draft state, you can replace that version without incrementing the version number. This might be desirable if you've uploaded the implementation and find, after testing the implementation, that there are further changes that you need to make before you can publish the changes. After you've published a version, that version is final.

You can also update the version number of an implementation in a Draft state. The process for both is the same. You set the `version` attribute in the implementation's `package.json` file.

1. Open the `package.json` file and change the `version` attribute. For example, change `"version":"1.0"` to `"version":"1.1"`.
2. Upload a zip file of the modified implementation to the associated API version.

Some key points to know about implementation versions are:

- Implementation versions are maintained independently of API versions. When you publish an API, the implementation isn't published automatically.
- When you upload a new version of an implementation, it becomes the default version (active implementation) for that API. You can change the default version in the API's Implementations page.
- The custom API's `Routing_BindApiToImpl` policy defines the association between an API version and the implementation version.

Deploying an API Implementation

When an API implementation is published, it can be deployed to a target environment. Note that you must be a mobile cloud administrator to deploy an implementation. The Deployment wizard takes you through the process of specifying your target environment, identifying any dependencies and alerting you to any dependency issues, such as connectors that the implementation is associated with. You'll have the opportunity to view any possible effect that deploying the API implementation could have on other artifacts. Because you're deploying from one environment to another, the policies applied to the API implementation in the source environment may need to change in the target environment; you can view and modify these policies before you deploy your implementation.

Select your published custom API and click **Deploy** to open the Deployment wizard. Go to the links at the top of the wizard to complete these deployment steps:

- Specifying a Target Environment for the Implementation

- Identifying Dependencies and Deployment Impact
- Setting Environment Policies for an API Implementation
- Deploying the Implementation

To learn about deploying artifacts in MCS, see Deployment.

Specifying a Target Environment for the Implementation

1. Make sure you're in the environment containing the API Implementation that you want to deploy.
2. Click  and select **Applications > APIs** from the side menu.
3. Select the API associated with the implementation that you want to deploy.
4. Expand **Implementations** on the right and click **Manage**.
5. Select the implementation on the API Implementation page and click **Deploy**.

   The Source Environment field is read-only and automatically defaults to the current environment of the API.
6. Select the target environment.

   Only the environments for which you have deployment permission are listed.
   If the artifact you want to deploy has dependencies, you'll have to resolve those dependencies before attempting to deploy.
7. (Optional) Enter a description about the deployment.

Identifying Dependencies and Deployment Effects

The next two navigation links let you view all the dependencies related to the artifact and whether they're currently deployed in the target environment. You can also see what impact the deployment might have on other artifacts.

1. Click **Dependencies**.

   Lists artifacts that must already be deployed to the target environment and on which the current artifact depends. You can see potential deployment issues, such as version conflicts, missing implementations (depending on the artifact), and so on. This gives you the opportunity to cancel the deployment and fix any potential issues. Afterward, you can go back and deploy your artifact.
   If the call to the mobile backend that's being deployed is rerouted, the name and version of the target mobile backend (as defined in the `Routing_RouteToBackend` policy for the mobile backend being deployed) is shown. The target mobile backend is not a dependency of the original mobile backend, so it won't be automatically deployed. You must manually deploy the target mobile backend to the target environment if it doesn't exist there already.
2. Click **Impact**.

   Shows notifications of possible effects on the artifact. The data displayed here is for your information only. Assess the effects and determine whether or not to proceed with deployment.
   If there's an issue, you can cancel the deployment process. After the issue is resolved, you can try deploying again.

Setting Environment Policies for an API Implementation

You're almost ready to deploy the API implementation. Each environment has policies that govern the behavior of the implementation within that environment. Policies and policy values in one environment can differ from policies in another environment. For instance, if you're deploying from a development environment to a runtime environment, you might want a higher degree of logging information in the runtime environment because your artifact will be thoroughly tested there prior to being made publicly accessible. The Policies page is where you can view the policies in both the source and target environments and edit policy values as needed.

1. Click **Policies**.
2. Click **Export** to see a `diff` file of the `policies.properties` file showing the policies in both the source and target environments.
3. (Optional) Edit a policy value as needed.

   The policies that you'll want to modify when you deploy an implementation are:

   - `CCC_Log_Body`: Logs the request body when set to `true` ; useful for debugging purposes. The default value is `false`.
   - `CCC_Log_Body_MaxLength`: Sets the character limit in the request body to log. The default value is `512`. This policy is used with `CCC_Log_Body`.

   In general, the default values for these policies should be sufficient.

   For a description of environment policies and their default values, see Oracle Mobile Cloud Service Environment Policies. To learn about environment policies, including scope and naming formats, see Environment Policies.
4. If you modified the `policies.properties` file, click **Import** to load it into the target environment.

Deploying the Implementation

View the basic details of your API implementation deployment: the current environment, the target environment, and any dependencies or policies. You can cancel or go back if you want to make changes to your deployment. If you're satisfied, you can deploy the implementation.

1. Click **Confirm**.

   The details of the deployment are displayed in a read-only section.
2. Click **Deploy**.

You are taken back to the APIs page.

Moving an API Implementation to the Trash

Remove an API implementation by moving it to the trash. If the implementation is needed later, you can restore it from the trash.

1. Make sure that you're in the environment containing the API implementation that you want to remove.
2. Click  and select **Applications > APIs** from the side menu.

3. Select the API associated with the implementation.
4. Click **Implementations** in the API navigation bar.
5. Select the draft API implementation to remove.
6. Click **Move to Trash**.

   Only real implementations (not mock implementations) can be moved to the trash. If you're moving the current default implementation to the trash, the next most recent version of the implementation is automatically set to the default. If no other implementations exist, the mock implementation is made the default.
7. Click **Trash** in the confirmation dialog if there are no dependency issues.

   If you think you or someone else might restore it later on, enter a brief comment about why you're putting this item in the trash.

To find out how dependencies can affect moving an artifact to the trash, see Dependencies That Affect a Move to the Trash. To restore an API implementation that's in the trash, see Restoring an API Implementation.

Restoring an API Implementation

1. Make sure that you're in the environment containing the API implementation that you want to restore.
2. Click  and select **Applications > APIs** from the side menu.
3. Select the API associated with the implementation.
4. Click **Trash** ().
5. Select **Implementations** in the trash drawer.
6. In the list of items in the trash, click  by the implementation you want and select **Restore from Trash**.
7. Click **Restore** in the confirmation dialog if there are no conflicts.

If you're restoring an implementation that was used by an API, the implementation won't be restored as the default (active) implementation for the API. You'll have to reset the implementation as the default from the Implementations page (select the API and click **Implementations** in the navbar).
Restoring an artifact can cause conflicts if a duplicate artifact already exists. To restore an artifact when a duplicate artifact exists, see Restoring an Artifact.

Connector Lifecycle

The lifecycle stages of all connectors are the same. Each type of connector goes through a design-time phase where each is created, tested, edited, and then published.

For all connectors, there are the creation phase, the testing and editing phase, the publishing phase, and the deployment phase. When you create a new connector, its version is automatically set to 1.0 and it's considered to be in a Draft state. In the Draft phase, you can test and edit your API as often as needed. When you're satisfied with your connector configuration, publish it with the understanding that a published connector can't be changed.

As you develop your connector, you can change the version's major and minor values as you see fit, that is, creating a new version of your API or updating an existing version. After you've implemented, tested, and published your connector, you can deploy it to one or more environments (for example, you can deploy from a design time environment to one or more runtime environments if you have multiple environments). Eventually, a connector may become obsolete, and you can move it to the trash.

If you think you need a better understanding of how artifacts interrelate in the overall MCS lifecycle before exploring the lifecycle of connectors, see Lifecycle.

Publishing a Connector

Before you can deploy a connector, you must publish it first:

1. Make sure you're in the environment containing the connector you want to publish.
2. Click  and select **Applications > Connectors**.
3. Select the draft connector that you want to publish.
4. Click **Publish**.

   (Optional) You can enter a justification for publishing the connector in the **Comment** field.

When the connector API is published, you're returned to the Connectors page where you can see the updated status of your connector.

Updating the Version Number of a Connector

If you created a new version of a connector using the New Version dialog, you can update the version number of the connector if it's still in a Draft state. This is particularly useful if you want to create an alternate version of the current connector or need to designate a different version number before you publish the connector.

1. Make sure you're in the environment containing the connector you want to update.
2. Click  and select **Applications > Connectors** from the side menu.
3. Select the connector from the list.
4. In the right section, select **More > Update Version Number**.
5. Enter a version number of the format *Major.minor*.

   The previous version of the connector is displayed next to the field. You'll get a message letting you know if you've entered an existing version number.
6. (Optional) Add a brief description that states what distinguishes this version from the previous one.
7. Click **Update**.

   A confirmation message is displayed. A draft of the new version is added to the list of connectors.

Creating a New Version of a Connector

You can make a new version of a connector regardless of whether it's in a Draft or Published state. When you create a new version of a connector, you're basically cloning the connector configuration and making changes to it. You can make minor changes or expand upon already defined functionality to create a backward-compatible API. A major update, however, can result in a disruption of mobile services to your customers due to invalid values being requested or returned, an inability to read the same file formats as the previous version, and so on. Major changes, therefore, aren't backward-compatible.

1. Make sure you're in the environment containing the connector you want.
2. Click  and select **Applications > Connectors** from the side menu.
3. Select a connector from the list.

   You can create a new version of a connector whether it is in a Draft or Published state
4. In the right panel, select **More > New Version**.
5. Select whether the new version of the connector is backward-compatible with the previous version (the default selection).
6. Enter a version number in the format *Major.minor*.

   If you enter a version number that already exists, you'll get a message letting you know that number is already in use.
7. (Optional) Add a brief description that states what distinguishes this version from the previous one.
8. Click **Create**.

   A confirmation message is displayed. A draft of the new version is added to the Connector page.

Deploying Connectors

To make your connector accessible for use, it needs to be deployed to a runtime environment. When a new version of a connector is published, it can be deployed to a target environment. The Deployment wizard takes you, the mobile cloud administrator, through the process of specifying your target environment, identifying any dependencies, and alerting you to any dependency issues, such as an API implementation that calls on the connector. You'll have the opportunity to view any possible effects that deploying the API could have on other artifacts. Because you're deploying from one environment to another, the policies applied to the API in the source environment may need to change in the target environment; you can view and modify these policies before you deploy your connector.

Select your published connector and click **Deploy** to open the Deployment wizard. Go to the links at the top of the wizard to complete these deployment steps:

- Specifying a Target Environment
- Identifying Dependencies and Deployment Impact
- Setting Environment Policies for Connectors

- Deploying the Connector

To learn about deploying artifacts in MCS, see Deployment.

Specifying a Target Environment

After a connector is published, the **Deploy** action is enabled.

1. Make sure you're in the environment containing the connector you want to deploy.
2. Click  and select **Applications > Connectors** from the side menu.
3. Click **Deploy**.

   The Deployment wizard opens on the Target page. The Source Environment field is read-only and automatically defaults to the current environment of the API.
4. Specify the target environment for your deployment.

   Only the environments for which you have deployment permission are listed.

   If the artifact you want to deploy has dependencies, you'll have to resolve those dependencies before attempting to deploy.
5. (Optional) Enter a description about the deployment.

Identifying Dependencies and Deployment Effects

The next two navigation links let you view all the dependencies related to the artifact and whether they're currently deployed in the target environment. You can also see what impact the deployment might have on other artifacts.

1. Click **Dependencies**.

   Lists artifacts that must already be deployed to the target environment and on which the current artifact depends. You can see potential deployment issues, such as version conflicts, missing implementations (depending on the artifact), and so on. This gives you the opportunity to cancel the deployment and fix any potential issues. Afterward, you can go back and deploy your artifact.
   If the call to the mobile backend that's being deployed is rerouted, the name and version of the target mobile backend (as defined in the `Routing_RouteToBackend` policy for the mobile backend being deployed) is shown. The target mobile backend is not a dependency of the original mobile backend, so it won't be automatically deployed. You must manually deploy the target mobile backend to the target environment if it doesn't exist there already.
2. Click **Impact**.

   Shows notifications of possible effects on the artifact. The data displayed here is for your information only. Assess the effects and determine whether or not to proceed with deployment.
   If there's an issue, you can cancel the deployment process. After the issue is resolved, you can try deploying again.

Setting Environment Policies for Connectors

You're almost done with the deployment. Each environment has policies that govern the behavior of the artifacts within that environment. Policies and policy values in one environment can differ from policies in another environment. For instance, if you're deploying from a development environment to a runtime environment, you might want

a higher degree of logging information in the runtime environment because your artifact will be thoroughly tested there prior to being made publicly accessible. The Policies page is where you can view the policies in both the source and target environments and edit policy values as needed.

1. Click **Policies**.
2. Click **Export** to see a `diff` file of the `policies.properties` file showing the policies in both the source and target environments.
3. Uncomment the policies for the connector.

   The policies that you'll want to uncomment when you deploy a connector are:

   - `Connector_Endpoint`: Stores the endpoint URL of the connector.
   - `Network_HttpReadTimeout`: Sets the amount of time spent waiting to read data.
   - `Network_HttpConnectTimeout`: Sets the amount of time spent connecting to the remote service.
   - `Routing_BindApiToImpl`: Specifies the implementation to which the connector API is bound.
   - `Security_OwsmPolicy` : Specifies the security policy used for outbound security.

     Remember that when you deploy a connector, the credentials for any CSF keys defined for the connector's security policies aren't carried over to the target environment. You'll want to tell your mobile cloud administrator to update the CSF keys so they can be used in the target environment.

   The initial values for these policies are set when the connector is created.

```
# The value of the dev
*.mygooglemaps(1.0).Connector_Endpoint=https
\://maps.googleapis.com/maps/api
# The value of the dev
*.mygooglemaps(1.0).Network_HttpConnectTimeout=20000
# The value of the dev
*.mygooglemaps(1.0).Network_HttpReadTimeout=20000
# The value of the dev
*.mygooglemaps(1.0).Routing_BindApiToImpl=mygooglemaps(1.0)
# The value of the dev
*.mygooglemaps(1.0).Security_OwsmPolicy=
[{"name"\:"oracle/http_basic_auth_over_ssl_client_policy","descr
iption"\:null,"overrides"\:[{"propertyName"\:"csf-
key","value"\:"enc-csf-key","description"\:"csf-key"}]}]

#------------------------------------------------------------
```

> **Note:**
>
> Be aware that the `Connector_Endpoint` and the `Security_OwsmPolicy` policies should never be set at the environment level, that is, the values for these policies shouldn't be applied globally to all connectors within an environment. These policies should be set specifically to individual connectors.

For descriptions of environment policies and their default values, see Oracle Mobile Cloud Service Environment Policies.

4. If you modified the `policies.properties` file, click **Import** to load it in the target environment.

To learn about environment policies, including scope and naming formats, see Environment Policies.

Deploying the Connector

View the basic details of your API deployment: the current environment, the target environment, and any dependencies or policies. You can cancel or go back if you want to make changes to your deployment. If you're satisfied, you can deploy the connector.

1. Click **Confirm**.

   The details of the deployment are displayed in a read-only section.

2. Click **Deploy**.

   A confirmation page is displayed that informs you if the deployment succeeded. For successful deployments, you can choose to return to the Connectors page or go to the Administration tab to review the policy settings in the target environment.

Moving a Connector to the Trash

Remove a connector by moving it to the trash. If the connector is needed later, you can restore it from the trash.

1. Make sure that you're in the environment containing the connector you want to remove.
2. Click  and select **Applications > Connectors** from the side menu.
3. Select the connector.
4. In the right section, select **More > Move to Trash**.
5. Click **Trash** in the confirmation dialog if there are no dependency issues.

   If you think you or someone else might restore it later on, enter a brief comment about why you're putting this item in the trash.

To find out how dependencies can affect moving an artifact to the trash, see Dependencies That Affect a Move to the Trash.

To restore a connector that's in the trash, see Restoring a Connector.

Restoring a Connector

1. Make sure that you're in the environment containing the connector that you want to restore.
2. Click  and select **Applications > Connectors** from the side menu.
3. Click **Trash** ().
4. In the list of items in Trash, click  by the connector you want and select **Restore from Trash**.
5. Click **Restore** in the confirmation dialog if there are no conflicts.

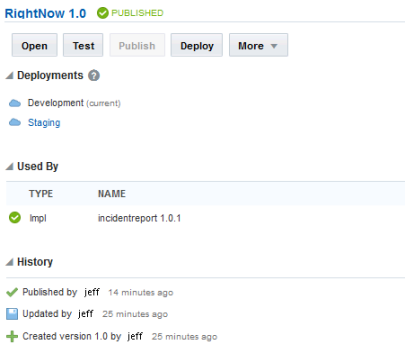
Restoring an artifact can cause conflicts if a duplicate artifact already exists. To restore an artifact when a duplicate artifact exists, see Restoring an Artifact.

Managing a Connector

After you create a connector, you'll want to edit it, publish it, see what artifacts are associated with it, in short, you want to be able to manage the connector and examine details of the connectors created by other service developers. The Connectors page gives you access to all these features.

When at least one connector exists, you'll be taken to the Connectors page every time you click and select **Applications > Connectors** from the side menu. On the left side of the page, you see a list of all the connectors except for those in the trash. You can see which connectors are in the Draft or Published state. Every connector is listed by its name and version number.

The right side of the Connectors page is where you can open, test, publish, or examine data about the connector:



On the right side of the page, you can perform the following actions:

- Click **Open** to see details about the selected connector.
- Click **More** to create a new version, update an existing version, or move an connector to the trash.
- Expand **Deployments** to see the environments that contain the selected the connector. You are only shown the environments that you have permission to access.

  Click on an environment to switch to it.
- Expand **Used By** to see the list of the implementations that call on the connector.
- Expand **History** to quickly see the latest activity for the connector.

Collection Lifecycle

The collection lifecycle involves moving from the Draft state to the Published state and finally deploying to the outside world.

Publishing is the prerequisite to deploying a collection. After you publish a collection, it can't be modified. While you can publish a collection and also create a new version of a collection, you can also remove a collection as described in Moving a Collection to the Trash.

If you think you need a better understanding of how artifacts interrelate in the overall MCS lifecycle before exploring the lifecycle of collections, see Lifecycle.

Publishing a Collection

You create a collection within the context of a backend. When you're satisfied with that collection, you can publish it.

1. Make sure you're in the environment containing the collection that you want to deploy.
2. Click  and select **Applications > Storage** from the side menu.
3. Select the collection you want to publish.
4. In the Details section on the right, click **Publish**.

> **Note:**
>
> You can publish a draft collection whenever you feel that it's complete. After it's published, however, it can't be changed.

A collection can also be published *involuntarily* when a backend associated with a collection is published. If the associated collection isn't yet published, it will be published automatically to support the backend.

When a collection is published:

- Its metadata (its description and access roles) are frozen. To update the collections metadata, you must create a new version.
- The major version (given the version that you arbitrarily defined) is incremented.
- It's no longer in your personal development space. It's available for anyone with the proper permissions to associate with a backend.
- Instance data isn't moved with the collection.

> **Note:**
>
> Instance data (such as actual user objects or actual collection objects stored in collections) is typically created at runtime, or by user scripts or code as part of a configuration. It isn't moved with the collection.

The published collection can be deployed to different environments.

Updating the Version Number of a Collection

When you update a version, the new number is backward-compatible and the collection history continues.

1. Make sure you're in the environment containing the collection that you want to update.

2. Click and select **Applications > Storage** from the side menu.
3. Select a collection.
4. In the Details section, select **More > Update Version Number**.
5. Specify an optional comment and click **Update**.

   The collection history reflects the incremented number.

Creating a New Version of a Collection

You can't copy a collection, but you can save yourself some time by creating a new version of an existing collection. If you create a new version, you'll have the same data. It is possible to rename the collection and reset the version as long as the collection is in a Draft state.

If you want a collection that starts with 1.0 and that has the same data as another collection, you must make a new collection and import the data.

When you create a new version number, an independent collection is spawned from that point with a new history that's unrelated to the previous collection. Any data is carried forward to the new version.

> **Note:**
>
> A collection can't have more than one version of an object.

1. Make sure you're in the environment containing the collection that you want.
2. Click and select **Applications > Storage** from the side menu.
3. Select a collection.
4. In the Details section on the right, select **More > New Version**.
5. Specify an optional comment and click **Update**.

Deploying a Collection

When a collection is published, it can be deployed to a target environment. The Deployment wizard takes you, the mobile cloud administrator, through the process of specifying your target environment, identifying any dependencies, and alerting you to any dependency issues, such as the realm the collection is associated with. You'll have the opportunity to view any possible effect that deploying the collection could have on other artifacts. Because you're deploying from one environment to another, the policies applied to the collection in the source environment may need to change in the target environment; you can view and modify these policies before you deploy your collection.

1. Make sure you're in the environment containing the collection that you want to deploy.
2. Click and select **Applications > Storage** from the side menu.
3. Select the collection that you want to deploy.

4. In the right panel, click **Deploy**.

   The Source Environment field is read-only and automatically defaults to the current environment of the collection.

5. Specify the target environment.

   Only the environments for which you have deployment permission are listed.

   If the artifact you want to deploy has dependencies, you'll have to resolve those dependencies before attempting to deploy.

6. (Optional) Enter a comment about the deployment.

7. Skip **Dependencies** and move on to the Impact page.

   The Dependencies page shows you the dependencies for the artifact being deployed. Because collections have no dependencies, you can skip this page.

8. Click **Impact**.

   The Impact page lists artifacts in the target environment that are affected when the realm is deployed. The data displayed is for your information only. Assess the effects and determine whether or not to proceed with the deployment.

9. Click **Policies**.

   a. Click **Export** to see a `diff` file of the `policies.properties` file showing the policies in both the source and target environments.

   b. (Optional) Edit a policy as needed.

      The policies that you'll want to set at deployment are:

      - `Logging_Level`: Sets the logging level.
      - `Sync_CollectionTimeToLive`: Sets the amount of time that data requested by a mobile application is stored in the cache.

      For a description of environment policies and their default values, see Oracle Mobile Cloud Service Environment Policies. To learn about environment policies, including scope and naming formats, see Environment Policies.

   c. If you modified the `policies.properties` file, click **Import** to load it into the target environment.

10. Click **Confirm** and view the deployment configuration, then click **Deploy**.

Moving a Collection to the Trash

Remove a collection by moving it to the trash. Moving a collection to the trash means it's no longer listed but it's still viable. If the collection is needed later, you can restore it.

1. Make sure that you're in the environment containing the collection you want to remove.

2. Click  and select **Applications > Storage** from the side menu.

3. Select the collection you want to remove.

4. In the Details section on the right, select **More > Move to Trash**.

5. Click **Trash** in the confirmation dialog if there are no dependency issues.

If you think you or someone else might restore it later on, enter a brief comment about why you're putting this item in the trash.

Although only a mobile cloud administrator can purge a collection (eliminate it permanently), you can delete an object in a collection using the command-line operation, `DELETE`.

To find out how dependencies can affect moving an artifact to the trash, see Dependencies That Affect a Move to the Trash.

To restore a collection in the trash, see Restoring a Collection.

Restoring a Collection

1. Make sure that you're in the environment containing the collection that you want to restore.
2. Click and select **Applications > Storage** from the side menu.
3. Click **Trash** ( ).
4. In the list of items in the trash, click by the collection you want and select **Restore from Trash**.
5. Click **Restore** in the confirmation dialog if there are no conflicts.

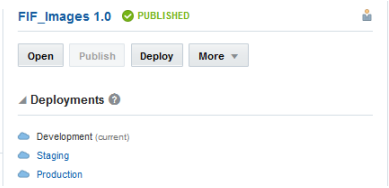
Restoring an artifact can cause conflicts if a duplicate artifact already exists. To restore an artifact when a duplicate artifact exists, see Restoring an Artifact.

Managing a Collection

After you create a collection, you'll want to edit it, publish it, and in short, manage the collection and examine details of collections created by other mobile developers. The Storage page gives you access to all these features.

When at least one collection exists, you'll be taken to the Storage page every time you click and select **Applications > Storage** from the side menu. On the left side of the page, you'll see a list of all the collections except for those in the trash. You can see which collections are in the Draft state and which are in the Published state. Every collection is listed by its name and version number.

The upper right side of the page is where you can open, test, and publish, deploy, and see which environments contain the selected collection:



- Click **Open** to see details about the selected collection.
- Click **Publish** to change the state of your collection from Draft to Published.

- Click **Deploy** to deploy your published collection to another environment.
- Click **More** to create a new version, update an existing version, associate the collection with a backend, or move a collection to the trash.
- Expand **Deployments** to see the environments that contain the selected the collection. You are only shown the environments that you have permission to access.

  Click on an environment to switch to it.

On the lower right of the page, you can examine usage and history details:

- Expand **Used By** to see which backends are associated with the collection. To disassociate the selected collection from an artifact that uses it, click **X** next to the artifact's name.
- Expand the **History** section to quickly see the latest activity for the selected collection.

33
Testing APIs and Mobile Backends

From the time that you begin implementing the specifications for a mobile app project in Oracle Mobile Cloud Service (MCS) to when you deploy its components and beyond, your team members will be involved in testing their work. Here's an overview of the various tests that your team might conduct for each phase of the project. We will also talk about how to diagnose the issues that your tests expose.

Use Case: End-to-End Testing

In this scenario, FixItFast (FiF) is a company that supplies maintenance services for large in-house appliances. To help facilitate speed and quality of service, FiF has requested a set of mobile apps for communicating and coordinating service requests. These mobile apps will be used by customers, customer representatives, and technicians.

Eric, the enterprise architect at FiF, has been assigned the task to design a convenient mobile interface for reporting household appliance issues. The interface must tie in with the existing repair dispatch service and an appliance manual service. While the MCS UI makes it easy for ad hoc testing, Eric puts strong emphasis on well-documented, traceable, repeatable tests that can be easily automated. To encourage his team to pursue his goals, when he designs the overall end-to-end mobile solution, he puts significant effort into defining tests to address business rules, security, scalability, and performance.

Using Eric's specifications, Mia, the mobile app developer, designs the endpoints for the custom API. One of the apps that uses the API is for customers, another is for customer reps, and another is for the FiF technicians. Therefore, Mia must configure the appropriate security for each endpoint. Some endpoints can be used by only one role, and others can be used by two or more roles. She uses Eric's test designs to create the mock data for the API. She then uses the test UI to verify her design, being sure to include security tests for any roles that she has associated with the endpoints.

As soon as the custom API is created with its mock data, Mia creates a mobile backend and associates it with the API. She then uses a REST testing library to implement the test suite that Eric designed for the API, and she performs the initial regression tests, which, at this point, use the API's mock data. From this point forward, Mia will monitor the backend's diagnostics page so that she can address runtime issues early, especially those caused by reconfiguring storage, realms, API roles, security policies, or service policies.

Before he starts implementing the code for the custom API, Jeff, the service developer, creates the connectors for the web services that feed data to the API, such as the repair dispatch service and the repair manual service. Jeff uses the test UI for ad hoc verification that the connectors work as expected. He also wants to implement formal tests for those connectors. These tests do nothing but check that the connectors are in a good state. To build the tests, Jeff creates a custom API for each connector. For each of his test APIs, he creates a module based on Node.js that serves as a pass-through for the HTTP requests. He then creates connector tests and adds them to the test suite to ensure that the connectors provide the expected results.

If issues arise, the tests can help detect whether the problem stems from the web service or the code that uses the connector. After the connectors are moved to to a runtime environment, the tests will be used to ensure that the web services that are used by the connectors still work as expected.

It's Jeff's job to implement the code for the custom API that Mia designed. Before and during custom code implementation, Jeff uses the test UI to perform ad hoc testing. He also creates white box tests to cover code paths in depth. After each upload of the implementation zip file, Jeff makes it a practice to run the original black box tests and the new white box tests to ensure the integrity of the whole implementation. Jeff also makes the tests available to the quality assurance team.

Amanda, the mobile cloud administrator, creates pre- and post-publish checklists for the mobile backend, the associated APIs, the custom code implementation, and the connectors that the APIs depends on. These checklists use the API, connector, and mobile backend tests. Amanda is concerned not only with verifying functionality, but also with verifying other specified characteristics such as timing. She also creates checklists to be performed before and after moving to production, remembering that the differences in environment security policies, service policies, and user management configurations can introduce new issues.

Last, Amanda schedules automated tests to monitor for unexpected failures, such as those caused by larger-than-expected requests or a connector that uses a third-party web service that has become unavailable. These tests, along with the live performance data that the Administration page provides, are part of Amanda's toolbox for ensuring a healthy system.

How Can I Test an API?

The Oracle Mobile Cloud Service (MCS) UI has test pages for testing the endpoints of the platform, connector, and custom APIs. You also can use HTTP tools such as cURL to test the platform and custom APIs remotely.

Testing a Platform or Custom API from the UI

Every platform and custom API has a test page that you can use to send a request to any of the API endpoints and view the response.

Here's how to test the API from a mobile backend:

1. Open the mobile backend.
2. Click **APIs**, and then select the API.
3. From the endpoints list on the left side of the Endpoints page, click the endpoint that you want to test.
4. If the endpoint has parameters, then enter the required parameters and any optional parameters that you want to test.
5. If the endpoint accepts a body, then provide the body or click **Use Example**.
6. In the Authentication section, select the authentication method. Ensure that you use an authentication method that's enabled for mobile backend that you selected. You enable an authentication method on the backend's **Settings** page.
7. If **Login Required** is turned on for the API (that is, anonymous access isn't allowed), then enter the credentials for the user that you want to test.

The test page enforces all security constraints that are configured for the API in the same way that they are enforced for a mobile client request. For example, if a custom API requires enterprise login, then the mobile user must be in the realm that's associated with the mobile backend or the test will fail. In addition, if that API, or the endpoint that you are testing, requires a specific role for access, the mobile user must be granted that role for the test to succeed. Finally, if that API and it's endpoints aren't configured for any roles, and the API is not set to allow anonymous access, then no user can access the API. For more information, see Setting Access to the API.

8. Click **Test Endpoint**.

   The Response Status section displays the status and the response. Click **Request** to see the request URI and headers.

The following topics provide more details about specific API types:

- Testing Your Custom API
- Testing API Endpoints Using Mock Data
- Testing APIs in a Mobile Backend with SSO Login
- Getting a Facebook User Access Token Manually
- Headers Needed for API Calls with Facebook Authentication
- Testing Runtime Operations Using the Endpoints Page for the Storage API

Testing a Connector API from the UI

Every connector has a test page that you can use to send a request to any of the API's endpoints and view its response. This test page enables you to determine if the connector is configured correctly. You can also use it to troubleshoot custom code that uses the connector. If the custom code response is not what you expected, then you can compare it with the test page response to determine whether your code introduced a bug.

To lean how to test a connector from the UI, see Testing the REST Connector API.

Testing Platform and Custom APIs Remotely

The MCS test pages are great for ad hoc testing and the MCS Custom Code Test Tools help with iterative testing and debugging of custom code. However, for formal testing, you need to set up tests that access the platform and custom APIs remotely. By formalizing the tests, you ensure that the entire test suite is exercised every run, and that the expected results are documented. In addition, you can set up the tests to run automatically. Another reason to run the tests remotely is that the MCS test UI automates some actions or exercises them differently, such that the UI tests might not represent true end-to-end simulations of external requests. For example, when you test a custom API from the MCS UI, some of the authentication setup is done automatically. Last, you must test remotely if you need to add or modify a header and the UI does not provide a field for doing so.

The way you remotely access a platform or custom API endpoint depends on the type of authentication that you want to use. See:

- Authenticating with HTTP Basic in Direct REST Calls
- Authenticating with OAuth in Direct REST Calls

- Getting a Single Sign-On OAuth Token through a Browser
- Getting a Facebook User Access Token Manually
- Headers Needed for API Calls with Facebook Authentication

When you create a custom API, MCS creates a mock implementation that you can use for testing before you implement the custom code. You also can use the mock implementation to configure a response for a mobile application test case. After you have uploaded an implementation, you can switch to the mock implementation for testing purposes by making it the default. For more information, see Testing with Mock Data.

If your request is in a test suite, then you can put the name of the test suite in the `Oracle-Mobile-Diagnostic-Session-ID` header. The name appears as the app session ID in the log messages. This lets you filter the log data on the Logs page by entering the test suite name in the **Search** text box. Also, when you are viewing a message's details, you can click the app session ID in the message to view all the messages with that ID. For more information about using the `Oracle-Mobile-Diagnostic-Session-ID` header, see How Client SDK Headers Enable Device and Session Diagnostics.

Troubleshooting Unexpected Test Results

When a test fails for a request, examine the response's HTTP status code and the returned data to identify the issue. Status codes in the 200 range indicate success. Status codes in the 400 range indicate a client error where the calling client has done something the server doesn't expect or won't allow. Depending on the 4XX error, this may require fixing custom code, giving a user the necessary privileges, or reconfiguring the server to allow requests of that type, for example. Status codes in the 500 range indicate that the server encountered a problem that it couldn't resolve. For example, the error might require reconfiguring server settings. Here are some common standard HTTP error codes and their meanings:

Status Code	Description
`400 BAD REQUEST`	General error when fulfilling the request would cause an invalid state, such as missing data or a validation error.
`401 UNAUTHORIZED`	Error due to a missing or invalid authentication token.
`403 FORBIDDEN`	Error due to user not having authorization or if the resource is unavailable.
`404 NOT FOUND`	Error due to the resource not being found.
`405 METHOD NOT ALLOWED`	Error that although the requested URL exists, the HTTP method isn't applicable.
`500 INTERNAL SERVER ERROR`	General error when an exception is thrown on the server side.

To pinpoint where the error occurred, open the mobile backend, click **Diagnostics**, and then click **Requests**. Next, find the request, click **View related log entries** in the **Related** column, and then select **Log Messages Related by API Request**. To see a message's details, click the time stamp. From the Message Details dialog, you can click the up and down arrows to see all the related log messages. Note that if there isn't sufficient information in a request to enable MCS to determine the

associated backend, then the related log messages appear only in the **Logs** page that is available from the **Administration** page.

If you are a mobile cloud administrator, you can view the log from **Administration** page. Click to open the side menu. Next, click **Administration**, and then click **Request History**.

For details about how to use the diagnostic logs, see Viewing Log Messages.

If you don't see any messages that help identify the source of the problem, you can change to a finer level for logging messages. Click **Log Level** in the Logs page, change the log level for the mobile backend, and then rerun the test. If you're troubleshooting custom code, then you can add your own log messages to the custom code, as described in Inserting Logging Into Custom Code, to help identify the code that's causing the problem.

When troubleshooting an unexpected result, consider that the cause might be due to a rerouting of the call to the mobile backend as described in Making Changes After a Backend is Published (Rerouting). If the mobile backend was rerouted, check to see if the following conditions were met:

- If the API was accessed using social identity, then the access token of the provider that was entered in the Authentication header must be the access token of the provider of the **target** mobile backend (that is, the mobile backend to which the original mobile backend was redirected).
- If the API was accessed by a mobile user, then the user must be a member of the realm that is associated with the **target** mobile backend (the mobile backend to which the original mobile backend is being redirected).

> **Tip:**
>
> If, in a request, you set the `Oracle-Mobile-Diagnostic-Session-ID` header to an identifier for the suite, that value is displayed in the message detail as the app session ID. If you click the app session ID in a message detail, then you can then click the up and down arrows to view all the messages for that ID. You can also enter the ID in the **Search** field to display only the log messages with that ID. For more information about using the `Oracle-Mobile-Diagnostic-Session-ID` header, see How Client SDK Headers Enable Device and Session Diagnostics.

These topics contain information about troubleshooting specific APIs:

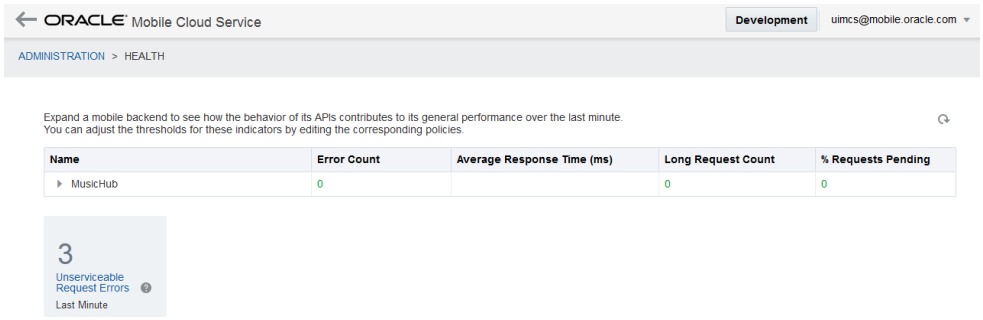
- Troubleshooting Custom APIs
- Troubleshooting Custom API Implementations
- Troubleshooting REST Connector APIs
- Troubleshooting SOAP Connector APIs
- Troubleshooting ICS Connector APIs
- Troubleshooting Fusion Applications Connector APIs
- Troubleshooting Notifications

Monitoring Runtime Issues and System Health

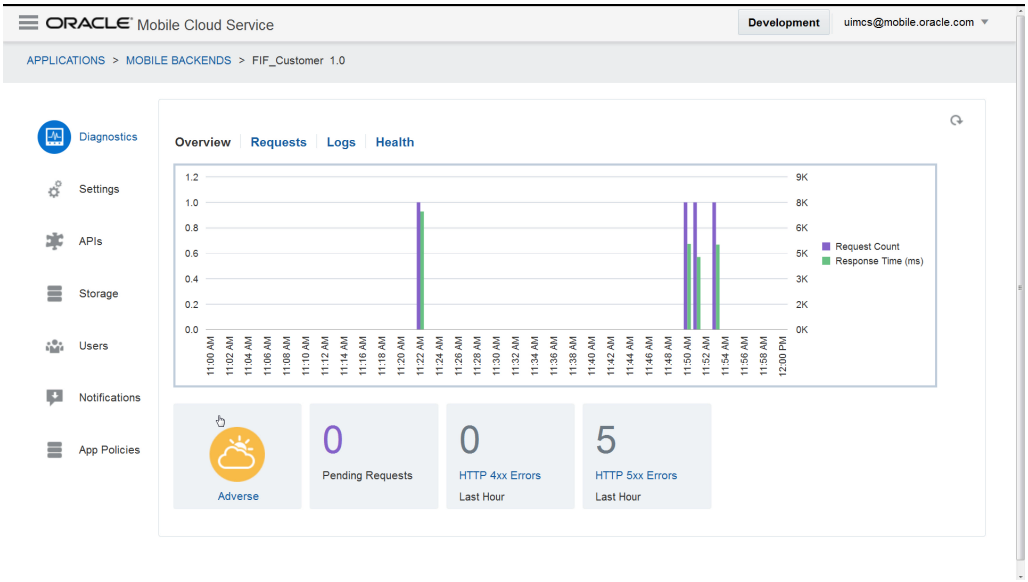
In addition to monitoring the results of white box and black box testing, you, as a mobile app developer or a mobile cloud administrator, will want to see live performance data to immediately address runtime issues, such as poor response times.

MCS provides two types of high-level monitoring consoles that display traffic-light indicators that convey overall environmental health, timelines that plot requests and responses, and counters of requests that result in HTTP 4xx and HTTP 5xx errors.

- The Health page, which is available from the **Administration** page, lets mobile cloud administrators monitor and troubleshoot the overall system health.



- The mobile backend **Diagnostics** page lets mobile developers monitor a mobile backend's health and troubleshoot issues with its associated mobile apps.



For in-depth information about these views and how you can use them, see Monitoring a Selected Backend, Viewing API Performance, and Viewing Status Codes for API Calls and Outbound Connector Calls.

Monitoring Overall Health

Green, amber, and red traffic-light indicators depict the overall health of an environment for the last minute. MCS bases this view on the health metrics for that environment. When the number of errors or the current request or response times exceed configured performance thresholds, the console changes the indicator from green (normal) to amber (adverse) or red (severe). For example, if the error count in the last minute is greater than 0, then the indicator is amber. If there are more than 9 errors in the last minute, then the indicator is red. To learn about the thresholds and how they affect the color of the indicators, see What Do the Health Indicator Thresholds Mean?

To see more detailed metrics for the last minute, such as the number of errors, the average duration of requests, the number of pending requests, and the number of long-running requests, hover over the indicator.

Monitoring Server Load and Request Backlog

To help monitor the server load and request backlog, the timeline plots the request counts and response times that occurred for the last hour, and the view displays the count of pending requests. To investigate a performance issue, click **Requests** or **Request History** to drill down to the request log.

Monitoring Mobile Backend and API Health

To see the error count, average response time, long request count, and percent requests pending for a mobile backend, click the traffic-light indicator to view the Health page. (The Health page shows all active mobile backends, whereas the Diagnostics view for a mobile backend shows just that backend). To see the same information for the mobile backend's APIs and their endpoints, expand the entry for the mobile backend.

To investigate the cause of an error, click **Logs** to drill down to the error log. To investigate a performance issue, click **Requests** or **Request History** to drill down to the request log.

34
Packages

Oracle Mobile Cloud Service (MCS) lets you share and move bundles of related artifacts built in MCS to another instance of MCS. You do this by exporting artifacts along with their dependencies, which creates a package, and importing that package to other instances of MCS.

The export process creates a package file (`package-`*`name`*`.zip`) containing a copy of the artifact, its dependencies, and their local policies. You can also use the package file as an archive for a set of related artifacts and store it outside of MCS. If artifacts in the current instance of MCS are changed or accidentally deleted, you can retrieve their original state from the package.

If you're a mobile or service developer, you can export artifacts such as mobile backends, collections, APIs and API implementations. You or another developer can then import the artifacts into the target environment.

What's a Package?

A package is a container for one or more artifacts. If an artifact has dependencies, they're also included in the package. For example, when you export a mobile backend, a package is created that contains the mobile backend and its dependencies, such as an API and its implementation, the connectors that the implementation calls, and collections. If the artifact you export is an API that has only one dependency, its implementation, then the package would contain just the API and its implementation.

> **Note:**
>
> While you can't explicitly add roles to a package, if an artifact has roles associated with it, they'll be included in the package

Artifacts can be in Draft or Published states. When an artifact is imported, it retains the state it had when the package was created (the source environment). That is, when an artifact in Draft state is imported, it's still in the Draft state in the new instance. The same is true for artifacts in the Published state.

When the import process completes, the artifacts in the package are created in the target environment and can go through all applicable lifecycle phases. For information on artifact lifecycle phases, see Lifecycle.

For information on exporting a package, see Adding Artifacts to the Package. For information on importing a package, see Uploading the Package.

Why Do I Want a Package?

With packages, you can easily share artifacts across different instances of MCS. For example, you might find that you can use the same set of configured artifacts for

different apps. Instead of having to recreate the same set of artifacts with the same configurations in another instance of MCS, you can *export* the artifacts (that is, create a package) in the current instance and import them into the target instance of MCS where work on the other app is being done.

Lets say Jeff, the service developer for Fix It Fast, has created a mobile backend that lets a technician look up the latest service requests and find the location and contact details for each customer. Fix It Fast has a subsidiary business called Restore It Fast, which provides restoration services to customers with fire or water damage. It would be helpful if the team at Restore It Fast could use that same mobile backend.

Jeff exports the mobile backend and all of its dependencies. He then notifies Jane, the service developer at Restore It Fast, that the package is ready to import. Jane locates and imports the package. She edits the environment policies for her MCS environment. She saves significant time by having the essentials of the mobile backend completed. She can begin testing right away and have the app ready to use by Restore It Fast technicians.

Exporting a Package

Use the Export Package wizard to easily create a packaged set of artifacts that you can export to other instances of MCS. The wizard shows you the dependencies associated with artifacts and includes those dependencies in the package for you. In addition to adding artifacts to the package, you'll have the opportunity to modify local environment policies.

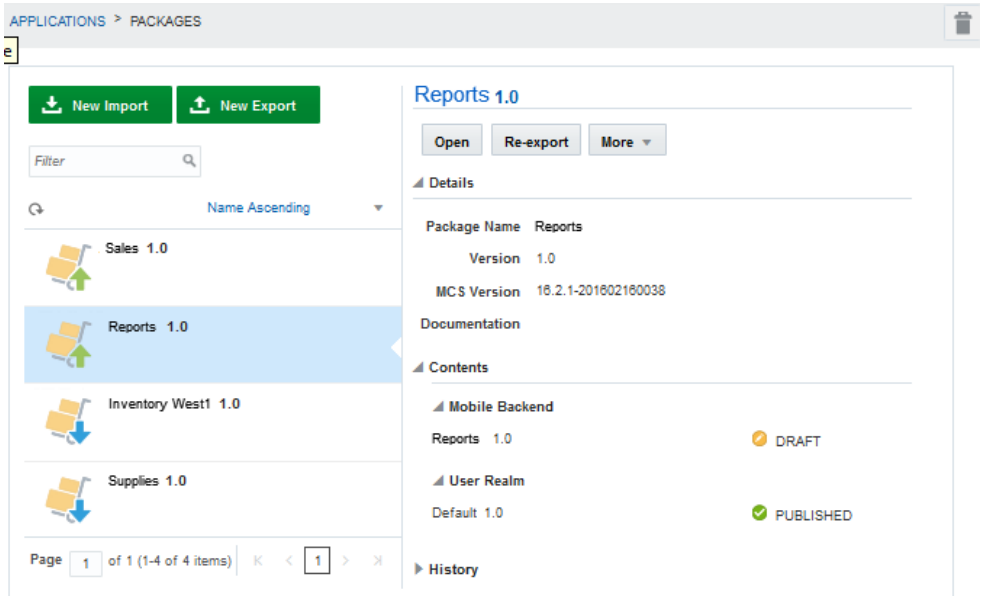
The Export Package wizard walks you through the following steps to export a package:

- Adding Artifacts to the Package
- Reviewing Dependencies During Export
- Setting Environment Policies During Export
- Completing the Export

Adding Artifacts to the Package

1. Make sure you're in the environment where you want to create a package.
2. Click  and select **Applications > Packages** from the side menu.

   If there are existing import and export packages, you'll see a list of packages.

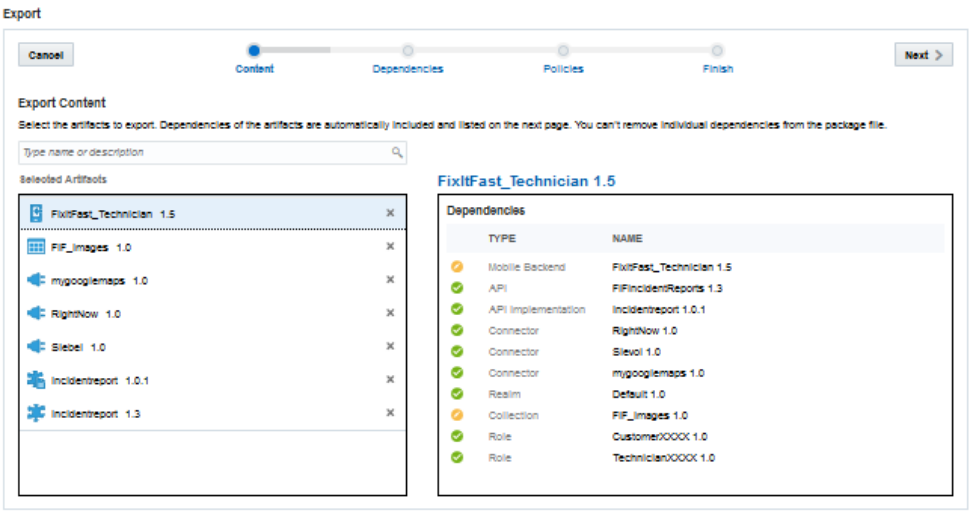


Uup arrow icons denote export packages. Down arrow icons denote import packages.

Alternatively, you can go to an artifact's landing page, select an artifact and choose **More > Export**. That artifact is automatically added to the list of selected artifacts. You can add more artifacts on the Content page of the Export wizard.

3. Click **New Export**.
4. On the Contents page of the Export wizard, click in the artifact **Search** field and select an artifact from drop-down list to add it to the package.

   You can also enter a name in the field. All artifacts with that character string are displayed in the Selected Artifacts list. Click **X** to remove an artifact that you don't want included in the package.



5. Select an artifact to see its dependencies in the right panel.

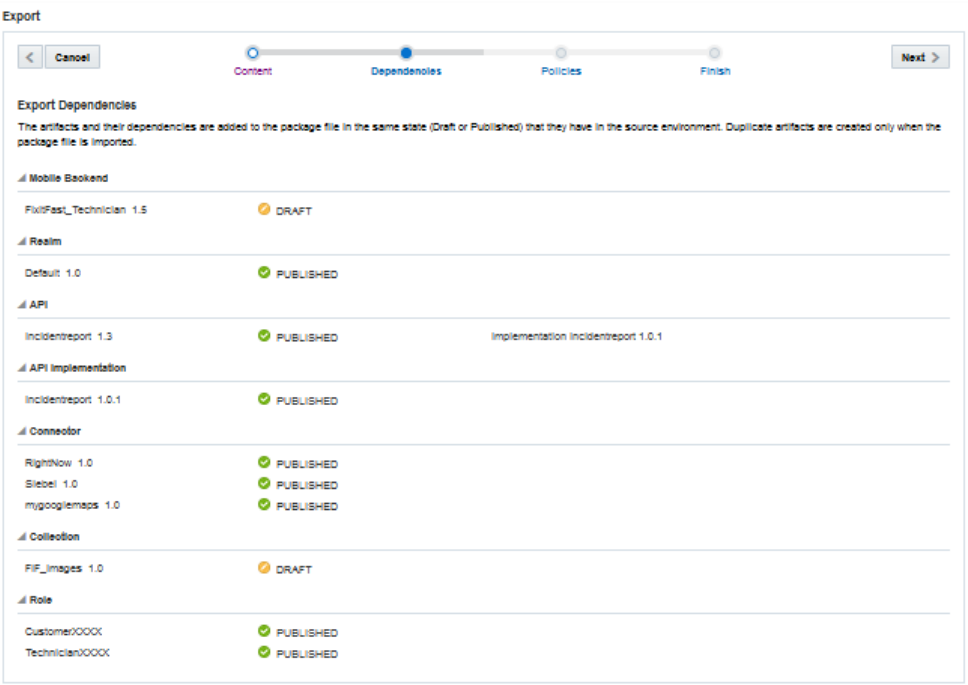
> **Note:**
>
> If you're exporting a client, the mobile backend that it references and any dependencies of the mobile backend are automatically added. However, if you export a mobile backend, the client that references it isn't automatically added. Because a mobile backend can be referenced by multiple clients, you'll have to manually add the client you want by entering its name in the Search and selecting it.
> Also be aware that notification profiles associated with the client are not included in the export or import package. You'll have to manually create the profiles in the target environment and associate them with the client.

6. Click **Next** (>) to go to the next step.

Reviewing Dependencies During Export

Here's where you can examine everything that's included in the export package. You can expand the view of each artifact type to see all the artifacts and their status. All artifacts are displayed under their respective types and top-level (root) artifacts are not distinguished. That is, a custom API that's listed could be a dependency of a mobile backend or a top-level artifact itself.

1. Click **Dependencies** in the navigation links.



If the call to the mobile backend that's being exported is rerouted, the name and version of the target mobile backend (as defined in the `Routing_RouteToBackend` policy for the mobile backend being exported) is shown. The target mobile backend isn't a dependency of the original mobile backend and won't be automatically exported. You must manually export the target mobile backend to the target environment if it doesn't exist there already.

2. If you're exporting APIs, expand **API** to see the associated API implementation for each custom API.
3. Click **Expand All** or **Collapse Al**l to see the full list of artifacts or just the artifact types.
4. Click **Next** (>) to go to the next step.

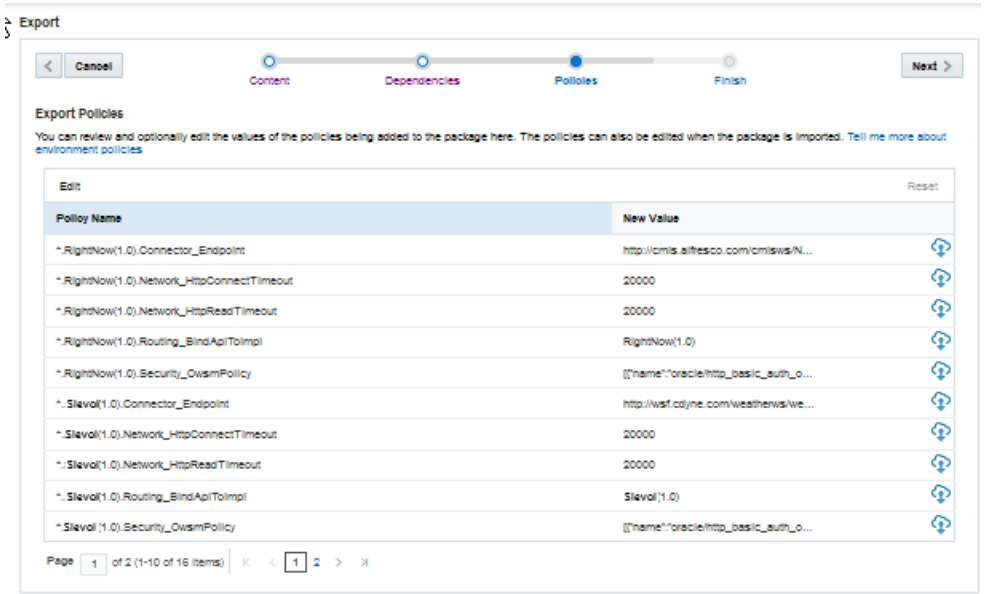
The Draft or Published state of the artifact and its dependencies are retained when the package is imported to the target environment.

Setting Environment Policies During Export

Setting or changing policy values is an optional step during export. You don't have to change policy values here. Policies can be modified during import or from the Administration page afterwards.
You can save some time by setting values now if you know what values will be required. For example, if a connector API is in the package, you may want to change the security policy. If a mobile backend is being exported, you may want to reset the `Sync_CollectionTimeoutToLive` policy. Another example is if the call to the mobile backend that's being exported is rerouted to another mobile backend and you want to ensure the rerouting occurs, you should set the `Routing_RouteToBackend` policy here and specify the name and version of the original and target mobile backends. You'll also want to check if the intended target mobile backend exists; otherwise, you'll need to export it.

> **Note:**
>
> If a policy in the export package doesn't already exist in the target, it will be added during the import.

1. Click **Policies** in the navigation links and review the current policy values for the artifacts in the package.



Policies values with a cloud icon indicate the value is taken from source environment. Pencil icons denote custom values.

2. (Optional) Select a policy and edit its value in one of the following ways:
   - Click **Edit** above the policy table. In the Edit Policy dialog, you can select the value that the policy currently has (**Package file value**) or enter a custom value (**Custom value**). Click **Null** to set the custom value to null. Click **Save** to enact the change.
   - Right-click a policy in the table and select **Set custom value to null** or **Edit** to enter a value in the Custom value field in the Edit Policy dialog.

   Click **Reset** to revert back to the original value for that policy.

   If you change your mind or make a mistake after modifying the policy values, click **Reset All** to revert back to the original policy values.
3. Click **Next** (>) to go to the next step.

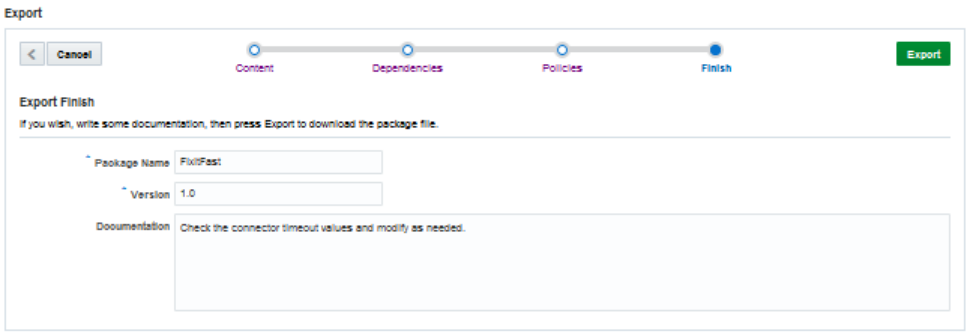
For descriptions of policies, see Oracle Mobile Cloud Service Environment Policies.

Completing the Export

Now that you've selected all the artifacts you want to export (and optionally, set any environment policies), it's time to create the package.

> **Note:**
>
> When you click **Export**, artifacts are added to the package in their current state at that time. For example, if someone publishes an artifact while you're creating the export package, the package will contain the published instance of that artifact.

1. Click **Finish** in the navigation links.



2. Enter a name for your package.

   The default name is the name of the top-level artifact. The package name and version must be a unique combination. No other package name can have the same name and version number.
3. Enter a version number.

   For example, enter 1.0 to designate it as the first version of this package.

4. Enter documentation about this package.

   Add documentation that informs whoever is importing the package about what it contains and what tasks need to be performed before and after the package is imported. The Export wizard automatically enters information about which roles must exist in the target environment before the package can be imported.

   You can manually write documentation for your export package using Markdown syntax in the **Documentation** field or copy and paste your documentation into the field. Markdown syntax lets you write an easy to-read plain text that can easily be converted to structurally valid XHTML for viewing in a browser. See How Do I Write in Markdown?

   Click **Preview** below the field to see the formatted output.

5. Click **Export**.

6. Select the location to place the package from the file chooser.

   You can edit the name of the package here. The file name has the format `package-`*`name`*`.zip`.

Re-exporting a Package

Re-exporting lets you create a new package based on an existing package. Select a package and select **Re-export**, which takes you through the Export Package wizard where you can select more artifacts to include or remove some of the current artifacts.

1. Click  and select **Applications > Packages** from the side menu.

2. Select an export package and click **Re-export**.

3. Follow the steps for exporting a package: selecting artifacts, reviewing dependencies, optionally setting environment polices, naming the package and providing documentation about the package. For steps on creating an export package, see Exporting a Package.

> **Note:**
>
> Remember that the new package must have a unique package name and version combination. That is, if the original package is `MyPackage 1.0`, the new package must have either a different name or version number.

Importing a Package

Importing a package puts copies of the artifacts from the source environment into the target environment. Before you proceed with the import, make sure the package name and version are unique in the target environment. You won't be able to import it if a package with the same name and version already exists. During the import, you'll be able to verify the contents of the package, read the package documentation, and you'll also be able to set the values for policies being added to the target environment or modify existing policies.

Also, if the package contains roles that will be created in the target environment, you must be a team member with Oracle Cloud identity domain administrator permissions

to import the package. Oracle Cloud identity domain administrator permissions are required to create roles in an environment. See Set Up the Service.

The Import Package wizard walks you through the following steps for importing a package:

- Uploading the Package
- Examining the Contents of the Import Package
- Setting Environment Policies During Import

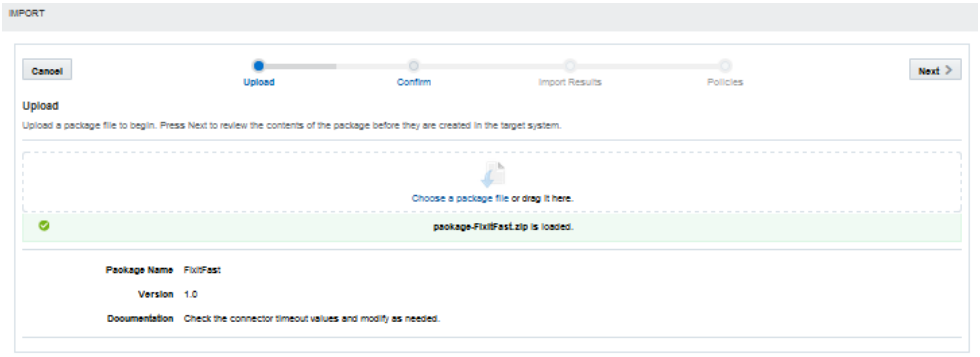
Uploading the Package

When you upload the package, the contents of the package are immediately installed in the target environment unless a conflict or some other error occurs during the import. You can view the contents of the package and whether or not all of the contents were successfully imported on the next page of the Import wizard.

1. Go to the environment where you want to import the package.
2. Click  and select **Applications > Packages** from the side menu.

   If there are existing packages, you'll see them listed here. Packages with a green up arrow denote export packages. Packages with a blue down arrow denote import packages.
3. Click **New Import**.
4. Copy and paste (or drag) the package to the Upload page of the Import wizard.

   After the package is uploaded, you can see the package name, version, and information about the package. If you've uploaded the wrong package, click **Cancel** to exit the import operation.



5. Click **Next** (>) to go to the next step.

Examining the Contents of the Import Package

On the confirmation page, you can see a list of the artifacts being imported and which artifacts already exist in the target environment. You can also see what dependencies are also being imported.

> **Note:**
>
> The notification profiles associated with a client are not included in the import package. If you're importing a client, you'll have to re-create the notification profiles in the target environment and associate them with the client. See Creating a Profile.

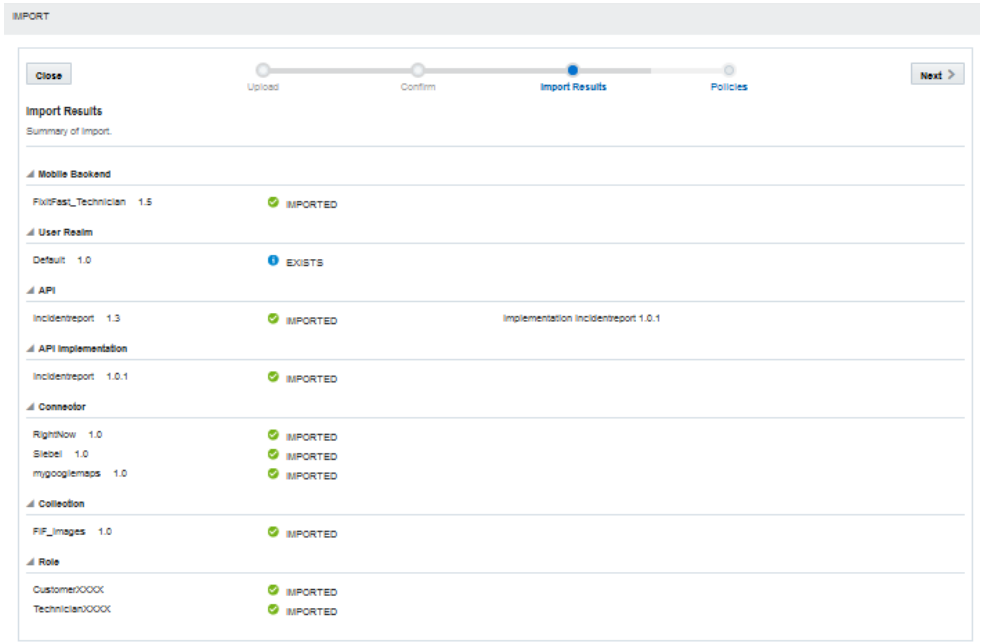
1. Click **Confirm** in the navigation links.
2. Review artifacts the list of artifacts to be installed. Remember if there are roles in the package that will be created in the target environment, you must have Oracle Cloud identity domain administrator permissions to do the import. Only team members with Oracle Cloud identity domain administrator permissions can create roles.

   If you don't want the listed artifacts imported to the target environment, click **Cancel** now. No changes will be made to the target environment.
   If the call to the mobile backend that's being imported is rerouted, the name and version of the target mobile backend (as defined in the `Routing_RouteToBackend` policy for the mobile backend being imported) is shown. The target mobile backend isn't a dependency of the original mobile backend and isn't included in the package. You must manually import the target mobile backend to the target environment if it doesn't exist there already.
3. Click **Next**.

   The process of installing the contents of the package in the target environment begins.

   A conflict occurs when an artifact with the same name and version (but with a different Universally Unique Identifier (UUID) value) exists in both the import package and in the target environment. The import process can't proceed if an error occurs. Close the import wizard and resolve the issue by moving the existing artifact in the target environment to the trash, changing its name or version, and then try importing the package again. Alternatively, you can import the package to a different instance of MCS.

   The Import Results page shows the artifacts that have been installed.



When an artifact in the package has the same name, version, and UUID value as one in the target environment, the artifact is marked as EXISTS on the results page and is not imported.

Setting Environment Policies During Import

Here is where you can set or modify the environment policies in the target environment for the packaged artifacts. Although the mobile cloud administrator can modify these policies later, to ensure that operations can be performed correctly in the target environment, you should update the policies here.
Even if you don't modify values for existing environment policies, any policies associated with the artifacts in the package that are new to the target environment are added for you when you update.

Check the documentation included in the package to see if any recommended values or policies are described. For descriptions of policies, see Oracle Mobile Cloud Service Environment Policies.

1. Click the **Policies** navigation link.

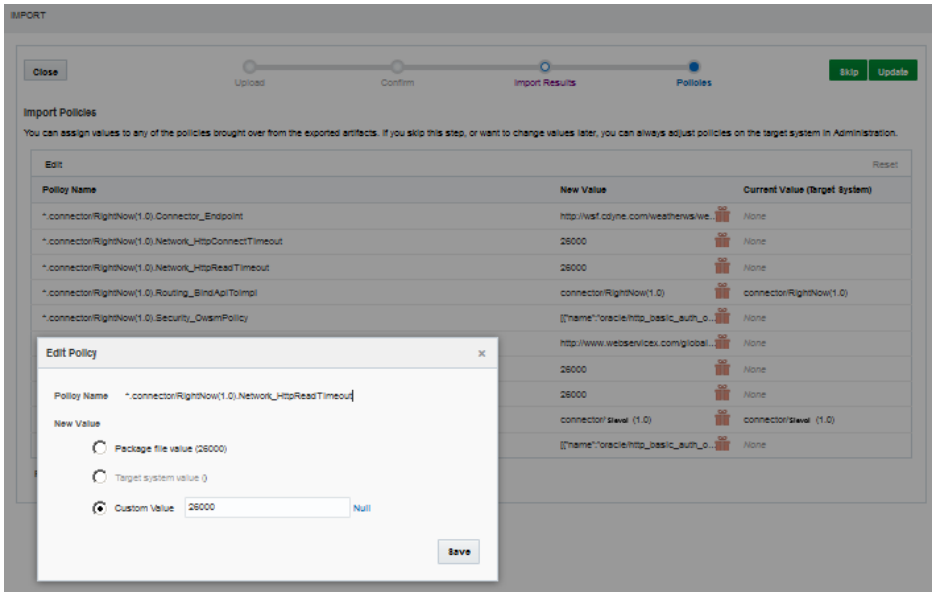
   If you really don't want to modify environment policies, click **Skip**. Be aware though that the import operation completes without updating any policy values or adding any policies to the target environment.

2. Filter the policies displayed by selecting **Mobile Backends** or **API/ Implementations** from the selection list, or enter a policy name in the Search field.

   Select **All Policies** (the default value) to list all the environment policies associated with the artifacts.

3. (Optional) Select a policy and edit its value in one of the following ways:

- Click **Edit** above the policy table. In the Edit Policy dialog, select **Package file value**, **Target system**, or **Custom value**. If you want to set the value to null, click **Null** next to the Custom value field.



  Click **Save** to enact the change.

- Right-click a policy in the table and select **Use value from target system**, **Set custom value to null** or **Edit** to enter a value in the Custom value field in the Edit Policy dialog.

  Click **Reset** to revert back to the original policy value.

If you change your mind or make a mistake, click **Reset** above the table to revert all the policies to their original values. A package icon indicates the policy takes the value it has in the package, a pencil icon indicates the policy has a custom value, and a target icon indicates the policy takes its value from the target environment.

4. Click **Update** to apply the changes to the policies and add any new policies to the target environment.

   Any policies in the policies list that don't already exist in the target environment are added. If you need to change any of the policy values after the import, your mobile cloud administrator can change them through the Administration view.

A blue dot by a policy name indicates that it has been modified. Icons in the Update Value column indicate if the value is taken from the package or if it was manually changed. You can the values of existing policies in the Current Value column.

What Happens When You Import a Package?

Similar to deploying an artifact from one environment to another, when importing artifacts from one instance of MCS to another, conflicts or errors can occur.

Some situations in which you can have a successful import:

- If all the artifacts being imported to the target environment in the new instance of MCS are unique in name and version from any existing artifacts in that environment, the import will be successful.

  For example, a package contains the `MyIncidentReports 1.1` API. The target environment has a `MyIncidentReports 1.5` API. There is no conflict because the two APIs are different and `MyIncidentReports 1.1` is successfully imported.

- Another successful import occurs even if some of the artifacts in the package already exist in the target environment. That is, duplicate artifacts are in the target environment.

  For example, a package contains `RightNow 1.1` connector. During the import process, it's determined that a duplicate connector already exists in the target environment. It has the same name, version, and UUID values. The connector is skipped and the rest of the artifacts are successfully imported

Here are instances where potential problems can occur:

- If a role associated with the artifacts in the package doesn't exist in the target environment, then it is added when the package is imported, but to do so requires that you are a team member with Oracle Cloud identity domain administrator permissions. If you don't have Oracle Cloud identity domain administrator permissions, the import will fail.
- If some of the artifacts in the package are similar to existing artifacts in the target environment, that is they have the same name, version, but different UUID values, the import process can't complete.

  For example, the package contains the published `RightNow 2.0` connector and the target environment also has a published `RightNow 2.0` connector. They both have the same name, version, but have different UUID values. You see a `CONFLICT` message by the artifact and the import operation fails. When an import fails, all changes made to the target environment are rolled back. All artifact attributes and policy values are returned to their original values prior to the import.

  You have two choices. You can create a new version of the connector in the source environment, resolve any dependency issues, export the connector, and then import it to the target environment. Otherwise, you can move the `RightNow 2.0` connector that's in the target environment to the trash and then proceed with the import.

For descriptions of the possible results of importing a package, see Import Results.

Import Results

The import results that can occur are described here:

Import State	Descriptions
Imported	The artifact didn't exist in the target environment and was imported successfully.

Import State	Descriptions
Not Imported	The artifact wasn't imported because of conflict occurred or a missing artifact was detected. The import process was stopped and any changes made prior to the error were rolled back. The target environment is back to its original state before the import.
Exists	A duplicate artifact already exists in the target environment, therefore, the artifact in the package was skipped.
Privileges	A required role or realm didn't exist in the target environment and the current user doesn't have Oracle Cloud identity domain administrator permissions to create the role or realm automatically during import.
Conflict	A similar artifact (same name and version but different UUID) exists in the target environment. The import process was stopped and any changes made prior to the conflict were rolled back. The target environment is back to its original state before the import.

Exporting Updated Artifacts

What happens if you make upgrades to artifacts in your instance of MCS and you want those upgrades in another instance of MCS? Lets say Jeff, at Fix it Fast, makes some changes to `MyIncidentReports1.1` API, which is in Draft state. Samir, who works at Restore It Fast, would like to get the improved API.

When you import updated artifacts, you need to take steps to prevent a conflict. The actions you take depend on the Draft or Published state of the artifacts. That could mean you'll have to move existing artifacts to the trash in the target environment or create a new version of the artifact to export and then resolve any resulting dependency issues with the new version of the artifact.

Following our example, Jeff exports `MyIncidentReports1.1` API and its implementation. However, before Samir can import the package, he moves his Draft instance of `MyIncidentReports1.1` to the trash to avoid a conflict during import.

Examining a Package

You can view the contents of a package from the Packages page. You can also re-export a package, create a new version of an existing package, or move an export package to the trash or the contents of an import.

1. Click  and select **Applications > Packages** from the side menu.
2. Select a package and click **View**.

From the View page, you can look at the details, contents, and policies of a package. You can also see the package details and content information on the packages landing page.

3. Click **Details** to see the package metadata. the contents, policy settings, and the version of MCS that contains the package.

> **Note:**
>
> You can only view the policy settings. You can't change them.

4. Click **Contents** to see the package contents.
5. Click **Policies** to view the environment policies and associated with the package contents and the policy values.
6. On the packages landing page, click **History** to see who created the selected package and when.

Moving a Package to the Trash

When you move an *export* package to the trash, you're moving just the record of the package, to the trash. The artifacts remain in the source environment.
However, when you move an *import* package to the trash, what you're actually doing is moving the package (that is, the record of the package) and all the artifacts in the package to the trash. Even artifacts in the Published state are moved to the trash. You can manually restore each artifact if you need them.

1. Click and select **Applications > Packages** from the side menu.
2. Select a package and then select **More > Move to Trash**.

> **Note:**
>
> Roles can't be deleted. Any roles associated with artifacts in the package are revoked and remain in the target system.

3. Review the information in the confirmation dialog.

   If an artifact is a dependency of several other artifacts, click **More** in the dialog to see the full list.
   You won't be able to deploy any artifacts that have dependencies on an artifact in the package that was moved to the trash.

   Also if an artifact that's in the package is a dependency of a published artifact that's not in the package, the move to the trash operation will fail.

4. Click **Yes** to move the package to the trash.

If you decide you need some or all of the artifacts that you've moved to the trash, you can restore them as needed. Just go to the artifact's landing page (for example, to restore a mobile backend, go to the Mobile Backends page), click on **Trash** ( ) and select the item you want to restore. Select **Restore** from the Trash menu. Your mobile cloud administrator can also restore these items from the Administration view.

Environment Policy Settings for Packaged Artifacts

When you export artifacts, you save their configurations in a portable file (the package) that can be sent to various instances. Only local policies are included in the package. That is, only policies scoped for an artifact are available for editing and exporting. For example, if you're exporting a mobile backend called `FIF_Technician 1.0` and an environment policy has been defined for it that's called `FIF_Technician(1.0).*.Logging_Level`. That policy will be available for editing. Environment-wide policies are not included in the package file. For example, if the mobile backend uses `*.*.Logging_Level`, that policy won't appear on the Policies page. The mobile backend will be subject to the `Logging_Level` policy in the target environment.

The environment policy settings for the artifacts are the values they have in the current instance. Because environment policies are specific to each environment in each instance, you might need to edit some of the policies before they can be used in their new location.

During export and import, you'll have the option to edit these values for the target environment. If someone other than you is performing the import, you should document which policies might need to be modified, and which might be overwritten, and which might need to be added. You might also want to alert them to any roles or realms that are required. To ensure the required policies are added to the target environment.

If a policy that you set during export or import doesn't exist in the target environment, it's added when you import the package.

Any required roles or realms that don't exist in the target environment are automatically created during the import but only if the person performing the import operation is a team member that has been granted an Oracle Cloud identity domain administrator role.

For descriptions of policies, see Oracle Mobile Cloud Service Environment Policies.

Part VII
Reference

This part includes appendices for various pieces of reference material.

A
HTTP Headers

You use headers to provide information (metadata) about the request or response or about the data contained in the message body. Oracle Mobile Cloud Service (MCS) provides custom request and response headers that you can use with the connector APIs and in custom code. The HTTP headers, their descriptions, and the services that use them are described in this chapter.

For detailed descriptions of standard HTTP headers, see Header Field Definitions.

API Headers

The following table lists the custom HTTP headers listed used by Oracle Mobile Cloud Service (MCS) custom APIs and connector APIs.

Header	Description	API
`Oracle-Mobile-API-Version`	The version of the connector or custom API that is called from a custom API implementation. Use this header when the dependency isn't declared in `package.json` or when you need to override the dependency declared in `package.json`. See package.json Contents.	Custom API REST and SOAP Connector APIs
`Oracle-Mobile-Backend-ID`	The ID of the mobile backend issued by MCS, which enables a mobile application to access APIs associated with that mobile backend. This header is required when you are using the HTTP Basic Authentication. The value of the ID (for the given environment) is displayed in the Keys section of the Mobile Backends page.	Custom API

Header	Description	API
`Oracle-Mobile-External-Authorization`	The request header used when a security policy isn't configured for the connector. When this header is set, the value of the header is set as Authorization on the request to the external service. Set the `Oracle-Mobile-External-Authorization` header only when the service you're connecting to is secured in a way that isn't described by an existing security policy. The header won't take effect if a security policy is configured. Setting this header takes precedence over setting an Authorization header and creating a rule for it.	REST Connector API

SDK Headers

The public HTTP headers listed in the following table are used in the iOS and Android SDKs to write calls in your app to mobile backend services.

Header names are case-insensitive and used the same way on both platforms. If you choose to write custom headers, then they must begin with `Oracle-Mobile-`.

Header	Description	Service
`Authorization`	For OAuth and SSO, contains the OAuth token downloaded from the OAuth Server. For HTTP Basic and Facebook, contains the Base64 encoding of the user name and password.	Security
`Oracle-Mobile-Analytics-Session-ID`	The current session to track events.	Analytics
`Oracle-Mobile-Application-Key`	The Application ID that's used to differentiate various applications.	Analytics and Others
`Oracle-Mobile-Backend-ID`	The ID of the mobile backend issued by MCS, which enables a mobile application to access APIs associated with that mobile backend. This header is required when you're using the HTTP Basic authentication or Facebook login. The value of the ID (for the given environment) is displayed in the Keys section of the Mobile Backends page.	Security

Header	Description	Service
`Oracle-Mobile-Canonical-Link`	The canonical link for the object.	Storage
`Oracle-Mobile-Client-Request-Time`	The client timestamp at which the request is made. The timestamp is in UTC in the format yyyy'-'MM'-'dd'-T'HH':'mm':'ss':SSS'Z.	Diagnostics
`Oracle-Mobile-Content-Disposition`	Arequest for the value of the `Content-Disposition` HTTP response header.	Storage
`Oracle-Mobile-Created-By`	The user who initially created the object. Corresponds to the `createdBy` property in the JSON representation of an object.	Storage
`Oracle-Mobile-Created-On`	The `dateTime` when the object was initially created. Corresponds to the `createdOn` property in the JSON representation of an object.	Storage
`Oracle-Mobile-Device-ID`	The Device ID that's used to differentiate various mobile devices.	Storage and Others
`Oracle-Mobile-Diagnostic-Session-ID`	A unique ID to represent a user app session. This is different from an Analytics session in terms of lifetime. The SDK uses the process ID (OS PID) for the header value.	Diagnostics
`Oracle-Mobile-Extra-Fields`	Addition of a set of predefined columns like `createdBy`, `createdOn`, and `modifiedBy`, which you can use to audit mobile users' interactions with the database. See Creating a Table Explicitly.	Database
`Oracle-Mobile-Modified-By`	The user who last modified the object. Corresponds to the `modifiedBy` property in the JSON representation of an object.	Storage
`Oracle-Mobile-Modified-On`	The `dateTime` when the object was last modified. Corresponds to the `modifiedOn` property in the JSON representation of an object.	Storage
`Oracle-Mobile-Name`	The display name for the object. Corresponds to the `name` property in the JSON representation of an object.	Storage

Header	Description	Service
`Oracle-Mobile-Primary-Keys`	Addition of a primary key to implicitly created schema.	Database
`Oracle-Mobile-Self-Link`	The self link for the object.	Storage
`Oracle-Mobile-Social-Access-Token`	For Facebook login, contains the Facebook access token.	Security
`Oracle-Mobile-Social-Identity-Provider`	For Facebook login, contains the value `facebook`.	Security
`Oracle-Mobile-Sync-Evict`	Optional. The specification of when a returned resource should be evicted from the cache, if set. Uses RFC 1123 `SimpleDateFormat`, for example "EEE, dd MMM yyyy HH:mm:ss z"	Synchronization
`Oracle-Mobile-Sync-Expires`	Optional. The specification of when a returned resource should expire in the cache, if set. Uses RFC 1123 `SimpleDateFormat`, for example "EEE, dd MMM yyyy HH:mm:ss z"	Synchronization
`Oracle-Mobile-Sync-No-Store`	If set to `true`, the device doesn't cache the returned resource.	Synchronization
`Oracle-Mobile-Sync-Resource-Type`	An `item` for items or a `collection` for collections; omitted for files. When set to `item` or `collection`, the `Content-Type` header must be `application/json`. For collections, the JSON must conform to the collection envelope structure. This is the custom header defined by the Synchronization service. See Defining Synchronization Policies and Cache Settings in a Response Header for details.	Synchronization
`Oracle-Mobile-Sync-Agent`	Optional. Informs a sync-compatible service (like Storage) to generate compatible collection formats. The value of the header is not critical but the client will set it to `true`.	Synchronization

B

Oracle Mobile Cloud Service Environment Policies

This chapter lists the policies that you can configure for each of your environments (such as Development, Staging, and Production) in Oracle Mobile Cloud Service (MCS). Policies control a variety of things, including logging level, password expiration times, means for restricting user access, and proxies. Policies can affect all artifacts of a specific type within a particular environment when applied at the environment level, or they can affect an individual artifact in the environment in which the policies are set.

> **Note:**
>
> The scope value shown is the narrowest level at which the property can be set.

See Environments and Team Members to learn about environments and environment policies.

Environment Policies and Their Values

Environment policies determine the behavior of various aspects of Oracle Mobile Cloud Service (MCS). If you're a mobile cloud administrator, you can view and modify the environment policies in the `policies.properties` file by exporting the file for a specific environment from the Administration page or by exporting the file when deploying an artifact. See Environment Policies.

Policy	Description	Type	Default Value	Scope / Affects
`Analytics_ApiCallEventCollectionEnabled`	Enables or disables automatic API call analytics event collection.	Boolean	true	Scope: Environment Affects: Analytics
`Analytics_ApiCallEventsAutoShrink`	Enables or disables database compact shrink during the automatic deletion of analytics API call data set by `Analytics_ApiCallEventsDaysRetained`.	Boolean	false	Scope: Environment Affects: Analytics

Policy	Description	Type	Default Value	Scope / Affects
`Analytics_ApiCallEventsDaysRetained`	Determines how many days analytics API call raw event data is retained in the database.	Integer	1	Scope: Environment Affects: Analytics
`Asset_AllowPurge`	Controls whether or not Draft or Published artifacts in the trash can be purged. Valid values are: • All • None • Draft • Published	String	All	Scope: Environment Affects: Realm, Mobile Backend, Custom API, API Implementation, Connector, and Collection
`Asset_AllowTrash`	Controls whether or not Draft or Published artifacts can be moved to the trash. Valid values are: • All • None • Draft • Published	String	All	Scope: Environment Affects: Realm, Mobile Backend, Custom API, API Implementation, Connector, and Collection
`Asset_AllowUntrash`	Controls whether or not Draft or Published artifacts can be restored from the trash. Valid values are: • All • None • Draft • Published	String	All	Scope: Environment Affects: Realm, Mobile Backend, Custom API, API Implementation, Connector, and Collection
`Asset_DefaultInitialVersion`	Sets the default version for all newly created assets.	String	1.0 **Note**: Generally, the default value should be used.	Scope: Environment Affects: all artifacts that have versions

Policy	Description	Type	Default Value	Scope / Affects
`CCC_DefaultNodeConfiguration`	Sets the default Node.js configuration used by the API implementation (custom code). Valid values are: • 8.9: The service uses node.js 8.9.4. • 6.10: The service uses node.js 6.10.10. • 0.10: The service uses node.js 0.10.25. For the related JavaScript library versions, see What's the Foundation for the Custom Code Service?.	String	0.10 for MCS upgrades. 6.10 for new instances of MCS.	Scope: Environment Affects: Custom Code
`CCC_LogBody`	Determines whether to log the body of a request in custom code. Bodies will be logged in the following circumstances: • `Logging level == FINEST` or there is an uncaught exception. • This property is set to true.	Boolean	false	Scope: Mobile Backend Affects: Custom Code
`CCC_LogBodyMaxLength`	Sets the maximum number of characters to log if the custom code is logging the request body.	Integer	512	Scope: Mobile Backend Affects: Custom Code

Policy	Description	Type	Default Value	Scope / Affects
`CCC_MaxLoadPerCPU`	Maximum one minute average load per processor (in nodejs: `os.loadavg()[0] / os.cpus().length`) allowed on custom code VM, or 0 to disable processor load checks. When the load per processor exceeds this threshold: • Requests to custom code are rejected with status 500, Low Resources. • New nodejs containers are not created • Idle nodejs containers are closed faster than normal	Double	1	Scope: Environment Affects: Custom Code
`CCC_MinFreeMemory Megabytes`	Minimal megabytes of free memory (in nodejs: `os.freemem()/ (1024*1024)`) allowed on custom code VM, or 0 to disable minimum free memory checks. When free memory is below this threshold: • Requests to custom code are rejected with status 500, Low Resources. • New nodejs containers are not created • Idle nodejs containers are closed faster than normal	Integer	256	Scope: Environment Affects: Custom Code

Policy	Description	Type	Default Value	Scope / Affects
`CCC_SendStackTraceWithError`	Determines whether or not to send the stack trace from Node.js with the REST response from the custom code container indicating that there is a code problem.	Boolean	false	Scope: Mobile Backend Affects: Custom Code
`Connectors_Endpoint`	Stores the endpoint URL of the particular connector instance. Set this policy when deploying to another environment by uncommenting the policy.	String	There is no default value for this policy. The initial value is set when the connector is created.	Scope: Connector Affects: Connectors
`Connector_ICS_Connections`	Identifies the JSON document representing connections to each configured ICS instance.	String	null	Scope: Environment Affects: ICS Connector
`Database_CreateTablesPolicy`	Controls whether the Database API can create, alter, or drop tables implicitly (`implicitOnly`) using the operations and JSON from custom code calls. Setting this policy to `explicitOnly` enables these operations using the Database Management Service API (and prohibits operations enabled by `implicitOnly`). Setting the policy to `allow` enables calls from custom code that perform implicit operations. Setting this policy to `none` curtails implicit table creation, deletion, and updates.	String	allow	Scope: Environment Affects: Database Service

Policy	Description	Type	Default Value	Scope / Affects
`Database_MaxRows`	Sets the maximum number of rows that can be returned by a single database query.	Integer	1000	Scope: Environment Affects: Database Service
`Database_QueryTimeout`	Sets the number of seconds to wait for a database query to return before canceling it.	Integer	20	Scope: Environment Affects: Database Service
`Diagnostics_AverageRequestTimeErrorThreshold`	Sets the threshold for the average time spent servicing a request. If the average time spent servicing a request equals or exceeds this threshold, then the health of the system is considered severe (red). Set this value higher than the one set for the `Diagnostics_AverageRequestTimeWarningThreshold` policy, which sets the adverse level of system health.	Double	6000.0	Scope: Environment Affects: Administration Console
`Diagnostics_AverageRequestTimeWarningThreshold`	Sets the threshold for the average time spent servicing a request. If the time spent servicing a request equals or exceeds this threshold, then the health of the system is considered adverse (amber).	Double	3000.0	Scope: Environment Affects: Administration Console
`Diagnostics_ExcludedHttpHeadersInLogs`	Creates a list of headers that shouldn't be logged with each API request in the API History log file.	String	Authorization header, cookie name	Scope: Environment Affects: Administration Console

Policy	Description	Type	Default Value	Scope / Affects
`Diagnostics_LongRequestCountErrorThreshold`	Sets the threshold for the number of long-running requests. If the number of long-running requests exceeds this threshold, then the system health is considered severe (red). Set this value higher than the one set for the `Diagnostics_LongRequestCountWarningThreshold` policy, which sets the adverse level of system health.	Integer	10	Scope: Environment Affects: Administration Console
`Diagnostics_LongRequestCountWarningThreshold`	Sets the threshold for the number of long-running requests. If the number of long-running requests exceeds this threshold, then the system health is considered adverse (amber). A long-running request to an endpoint server has a duration that's greater than (or equal to) 8 seconds (8000 ms).	Integer	0	Scope: Environment Affects: Administration Console
`Diagnostics_LongRequestThreshold`	Sets the threshold for the amount of time spent on a request to an endpoint server. If a request to an endpoint server has a duration that is greater than (or equal to) 8 seconds (8000 ms), then it's considered a long-running request.	Integer	8000	Scope: Environment Affects: Administration Console

Policy	Description	Type	Default Value	Scope / Affects
`Diagnostics_PendingRequestErrorThreshold`	Sets the threshold of the proportion of pending requests. If the proportion of pending requests (which is expressed as a percentage) equals or exceeds this threshold, then the system health is considered severe (red). The value should be higher than the one set for the `Diagnostics_PendingRequestWarningThreshold` policy, which sets the adverse level of system health.	Double	30 Generally, the default value should be used.	Scope: Environment Affects: Administration Console
`Diagnostics_PendingRequestWarningThreshold`	Sets the threshold of the proportion of pending requests. If the proportion of pending requests (which is expressed as a percentage) equals or exceeds this threshold, then the system health is considered adverse (amber). Pending requests represent the ratio of in-flight requests to the number of active requests, successful requests, and failed requests within the last minute.	Double	15	Scope: Environment Affects: Administration Console

Policy	Description	Type	Default Value	Scope / Affects
`Diagnostics_RequestCountErrorThreshold`	Sets the threshold of the proportion of failed requests. If the number of failed requests (including unserviceable requests) equals or exceed this threshold, then the system health is considered severe (red). The value should be higher than the one set for the `Diagnostics_RequestCountErrorThreshold` policy, which sets the adverse level of system health.	Integer	10	Scope: Environment Affects: Administration Console
`Diagnostics_RequestCountWarningThreshold`	Sets the threshold of the proportion of failed requests. If the number of failed requests (including unserviceable requests) equals or exceeds this threshold, then the system health is considered adverse (amber).	Integer	0	Scope: Environment Affects: Administration Console
`Logging_Level`	Sets the logging level.	Integer	800	Scope: Mobile Backend Affects: Custom APIs, Storage
`Network_HttpConnectTimeout`	Sets the amount of time spent in milliseconds (ms) connecting to the remote URL. The value should be less than the value of `Network_HttpRequestTimeout`. Set this policy when deploying to another environment by uncommenting the policy.	Integer	There is no default value for this policy. The initial value is set when the connector is created.	Scope: Environment, Mobile Backend, Connector, Fully-Qualified Connector Affects: Connectors

Policy	Description	Type	Default Value	Scope / Affects
Network_HttpPatch	Controls the behavior of PATCH requests. • HEADER sends a POST request with an X-HTTP-Method-Override header set to PATCH. This enables you to send PATCH requests when the target server doesn't support the PATCH method. • LEGACY sends a PATCH request with an X-HTTP-Method-Override header set to PATCH. • METHOD sends a PATCH request without an X-HTTP-Method-Override header set to PATCH.	String	For environments that were provisioned before 18.2.3, the default is LEGACY. For environments that were provisioned on or after 18.2.3, the default is METHOD.	Scope: Environment Affects: Connectors
Network_HttpReadTimeout	Sets the maximum time (in milliseconds) spent waiting to read data. The value should be less than the value of Network_HttpRequestTimeout. Set this policy when deploying to another environment by uncommenting the policy.	Integer	There is no default value for this policy. The initial value is set when the connector is created.	Scope: Environment, Mobile Backend, Connector, Fully-Qualified Connector Affects: Connectors
Network_HttpRequestTimeout	Sets the amount of time in milliseconds (ms) on an HTTP request before it times out. Set this policy when deploying to another environment.	Integer	40,000 ms	Scope: Environment Affects: Custom APIs

Policy	Description	Type	Default Value	Scope / Affects
`Notifications_DeviceCountWarningThreshold`	Defines the threshold level (percentage) of messages sent successfully without returning an error. If the proportion of messages accepted by the service provider is below the threshold, then a warning is displayed. The default value is 70.0 (70%). Set this policy per environment as needed.	Double	70.0 **Note:** For testing purposes only, consider setting this value to 100.0 (100%).	Scope: Environment Affects: Notifications
`Routing_BindAPIToImpl`	Determines which core service to use to resolve the API request. For connectors, set this policy when deploying to another environment by uncommenting the policy.	String	There is no default value for this policy.	Scope: API Affects: Custom APIs, Connectors
`Routing_BindAPIToMock`	Resolves the API request to a mock service instead of the implementation that's bound to the API.	Boolean	false **Note:** Do not modify this policy.	Scope: Fully-Qualified API Affects: Mobile Backends, Custom APIs
`Routing_DefaultImplementation`	Specifies the default implementation for the initially created API (that is, the mock service).	String	MockService/1.0 **Note:** Do not modify this policy.	Scope: Environment Affects: Custom APIs
`Routing_RouteToBackend`	Reroutes mobile API calls made to a mobile backend to the target mobile backend specified. Allows backend fixes (fixes that require a new mobile backend) to be delivered to the mobile app without requiring the mobile app to be recompiled.	String	There is no default value for this policy.	Scope: Mobile Backend Affects: Dispatcher

Policy	Description	Type	Default Value	Scope / Affects
`Security_AllowOrigin`	Enables Cross Origin Resource Sharing (CORS) from HTML5 clients on an external domain. Supported values are: • disallow • url1, url2, url3 By providing URLs as values, specifies a whitelist of URLs from which cross-site requests to MCS APIs can be made. If the origin of the cross-site request matches one of the patterns in the whitelist, the request is allowed. Otherwise, access is restricted. The wildcard character, *, can be used when providing URL values. However, there are rules for its use. See Securing Cross-Site Requests to MCS APIs for detailed information.	String	disallow **Note:** When dealing with browser-based applications, it's highly recommended that cross-site access to MCS APIs either be restricted completely, or be restricted to trusted origins where legitimate applications are known to be hosted to prevent vulnerability to cross-site attacks (e.g., Cross-Site Request Forgery).	Scope: Environment Affects: All cross origin calls to a given environment
`Security_AuthTokenConfiguration`	Provides a configuration to integrate with third-party identify providers through which mobile app users can authenticate. See JWT Tokens and Virtual Users.	JSON object		Scope: Environment Affects: Security

Policy	Description	Type	Default Value	Scope / Affects
`Security_CollectionsAnonymousAccess`	Sets a storage collection to allow anonymous access. For each storage collection listed in the policy, anonymous read and write access will be allowed, provided that the correct anonymous access key is defined in the request headers. Specifying '*' as the version allows anonymous access to all versions of the collection.	A comma-separated list of storage collections following this pattern: `<collection1_name>[(<version>|*)][,<collection2_name>[(<version>|*)]][, ...]`	No default value	Scope: Storage collections Affects: The collections and versions listed in the policy
`Security_ExposeHeaders`	Provides a means for browsers to access the server whitelist headers. By default, Cross Origin Resource Sharing (CORS) disallows accessing returned headers by the browser. Applies to HTML5 clients accessing a given resource from an external domain.	String	"" Indicates that no response headers are to be exposed to the browser.	Scope: Environment Affects: All cross origin calls to a given environment
`Security_IdentityProviders`	Stores identity providers configuration.	String	Facebook identity provider configuration	Scope: Environment Affects: Security
`Security_IgnoreHostnameVerification`	Disables the SSL host name verification. To be applied to connectors (in development environments) that call outbound services using SSL certificates with an invalid or incomplete hostname.	Boolean	false	Scope: Environment Affects: REST, SOAP, ICS, and Fusion Applications Connectors

Policy	Description	Type	Default Value	Scope / Affects
`Security_OwsmPolicy`	Sets the security policy used for outbound security. For connectors, set this policy when deploying to another environment by uncommenting the policy.	Object	There is no default value for this policy. The initial value is set when the connector is created.	Scope: Connector Affects: Connectors
`Security_SsoRedirectWhitelist`	Lists the URL patterns for the SSO `redirct_uri` parameter values that are permitted.	String	disallow	Scope: Environment, Mobile Backend Affects: SSO Token Relay
`Security_TokenExchangeTimeoutPolicy`	Defines the policy that governs the expiration time for MCS-issued tokens generated as a result of token exchange. Valid values are: • `FromTimeoutSecs` - MCS-issued token expiry time is governed by the `Security_TokenExchangeTimeoutSecs` policy. • `FromExternalToken` - MCS-issued token expiry time is set to the same time as the external token expiry time. • `FromExternalTokenLimitedByTimeoutSecs` - MCS-issued token expiry time is set to the value determined from the `Security_TokenExchangeTimeoutSecs` policy or the external token expiry time, whichever comes first.	String	`FromTimeoutSecs`	Scope: Environment Affects: SSO Token Exchange

Policy	Description	Type	Default Value	Scope / Affects
`Security_TokenExchangeTimeoutSecs`	Sets the token expiration time for SSO login.	Integer	216000 s	Scope: Environment Affects: SSO Token Relay
`Security_TransportSecurityProtocols`	Specifies a list of the TLS/SSL protocols that should be used for the outbound connection for the specific connector. By default, only TLSv1.1 and TLSv1.2 protocols are used for outbound connections. This property can be used to override the system defaults so that connections can be established to legacy systems that don't support new versions of TLS/SSL. **Caution:** Use this property carefully as older protocols are more vulnerable to security exploits. Valid value is a comma separated list of the TLS/SSL protocols. Note that extra spaces around the protocol names are ignored. For example, TLSv1, TLSv1.1, TLSv1.2. Supported protocols are: SSLv2Hello, TLSv1, TLSv1.1, TLSv1.2.	String	No default value	Scope: Connectors, Fully-qualified Connectors Affects: All Connectors
`Sync_CollectionTimeToLive`	Sets the default amount of time that data requested by a mobile app from a storage collection remains in the local cache that's used by the Synchronization library.	Integer	86400 s Set this policy per environment as needed.	Scope: Environment Affects: Storage

Policy	Description	Type	Default Value	Scope / Affects
`Url_PercentEncodeQueryParameterSpaces`	Controls how spaces in query parameters of a URL are encoded. If set to true encodes spaces as `%20;` and encodes them as + otherwise. Spaces in other parts of the URL are always encoded as `%20`.	Boolean	false	Scope: Connector Affects: REST Connector
`User_AllowDynamicUserSchema`	Indicates if the user schema can be augmented when unknown properties are part of the user data. This is used when users are imported into a realm or when a user is being updated. The properties defining the user that aren't already defined as user properties are automatically added before importing the users. It isn't possible to augment the user schema when the call is coming from the platform API, regardless of the policy. Set this policy at the environment level.	Boolean	Development environment: **True** Staging and Production environments: **False**	Scope: Environment Affects: Mobile User Management
`User_DefaultUserRealm`	Indicates the default user realm. This is used when creating a new mobile backend. The associated user realm is the one specified by this policy. You can reference only an existing realm. Set this policy at the environment level.	String	1.0	Scope: Environment Affects: Mobile User Management

C

Security Policies for Connector APIs

Connecting to external services usually requires some degree of authentication and authorization. When you configure a connector API, you have the option of specifying the security policies to use when communicating with an external service (except for ICS Connector APIs where the security policy is determined by the WSDL for SOAP-based integrations).

Descriptions of the supported Oracle Web Services Manager (Oracle WSM) security policies for the REST, SOAP, ICS, and Fusion Applications Connector APIs are provided here. Additionally, the policy properties that you can override are also described along with a mapping of policy properties to the policies that contain them.

Note that for connector APIs, only client policies are valid.

Security Policies for REST Connector APIs

The supported Oracle Web Services Manager (Oracle WSM) security policies for REST Connector APIs are described in the following table:

Security Policy	Description
`http_basic_auth_over_ssl_client_policy`	Includes user name and password in an HTTP Basic Authorization header.
`http_jwt_token_client_policy`	Includes a JWT token in the HTTP header. A JSON Web Token represents claims and is generally used in Federated Identity systems where the source and target have mutual trust and a shared identity realm. The JWT token is create automatically. The issuer name and subject name are provided either programmatically or declaratively through the policy. You can specify the audience restriction condition for this policy.
`http_jwt_token_identity_switch_client_policy`	Includes JWT token in the HTTP header. Similar to `http_jwt_token_client_policy` but this policy also performs dynamic identity switching by propagating a different identity than the one based on authenticated Subject (mobile user).

Security Policy	Description
`http_jwt_token_over_ssl_client_policy`	Includes a JWT token in the HTTP header. A JSON Web Token represents claims and is generally used in Federated Identity systems where the source and target have mutual trust and a shared identity realm. The JWT token is created automatically. The issuer name and subject name are provided either programmatically or declaratively through the policy. You can specify the audience restriction condition for this policy. This version of the policy enforces that connections are made over https.
`http_saml20_token_bearer_client_policy`	Includes SAML 2.0 tokens in the HTTP header. SAML provides single sign-on in that multiple services can redirect a user to a single identity provider, which supplies signed assertion tokens. The SAML token with confirmation method `Bearer` is created automatically.
`http_saml20_token_bearer_over_ssl_client_policy`	Includes SAML 2.0 tokens in the HTTP header. SAML provides single sign-on in that multiple services can redirect a user to a single identity provider, which supplies signed assertion tokens. The SAML token with confirmation method `Bearer` is created automatically. This version of the policy enforces that connections are made over https.
`oauth2_config_client_policy`	Provides information about the OAuth2 server, which preforms authorization and issues the access tokens. You must set both this policy and `oracle/http_oauth2_token_client_policy` together.
`http_oauth2_token_client_policy`	Includes OAuth2 access token in the request. OAuth2 allows users to safely grant client applications limited access to protected resources.. You must set both this policy and `oracle/oauth2_config_client_policy` together.
`http_oauth2_token_over_ssl_client_policy`	Includes OAuth2 access token in the request. OAuth2 allows users to safely grant client applications limited access to protected resources. You must set both this policy and `oracle/oauth2_config_client_policy` together. This version of the policy enforces that connections are made over https.

Security Policies for SOAP Connector APIs

The supported Oracle Web Services Manager (Oracle WSM) security polices for SOAP connectors are described in the following table:

Security Policy	Description
`http_basic_auth_over_ssl_client_policy`	Includes credentials in the HTTP header for outbound client requests. This policy also verifies that the transport protocol is HTTPS. Requests over a non-HTTPS transport protocol are refused. This policy can be applied to any HTTP-based endpoint.
`wss_http_token_client_policy`	Includes credentials in the HTTP header for outbound client requests. The credentials can be provided either programmatically or through the current Java Authentication and Authorization Service (JAAS) subject. This policy can be applied to any HTTP-based client. Note: Currently only HTTP Basic Authentication is supported.
`wss_http_token_over_ssl_client_policy`	Includes credentials in the HTTP header for outbound client requests. The credentials are provided either programmatically or through the Java Authentication and Authorization Service (JAAS) subject. It also verifies that the outbound transport protocol is HTTPS. If a non-HTTPS transport protocol is used, then the request is refused. This policy can be applied to any HTTP-based client.
`wss_saml_token_bearer_client_policy`	Includes the SAML `Bearer` token in outbound SOAP request messages. The SAML token is automatically created and is by default signed with an enveloped signature. The issuer name and subject name are provided either programmatically or through the current Java Authentication and Authorization Service (JAAS) subject.
`wss_saml_token_bearer_over_ssl_client_policy`	Includes SAML tokens in outbound SOAP request messages. The SAML token with confirmation method `Bearer` is automatically created. The issuer name and subject name are provided either programmatically or through the current Java Authentication and Authorization Service (JAAS) subject. The policy also verifies that the transport protocol provides SSL message protection. This policy can be attached to any SOAP-based client

Security Policy	Description
`wss_saml20_token_bearer_over_ssl_client_policy`	Includes SAML V2.0 tokens in outbound SOAP request messages. The SAML token with confirmation method `Bearer` is automatically created. The issuer name and subject name are provided either programmatically or through the current Java Authentication and Authorization Service (JAAS) subject. Optionally, attesting entity and audience restriction condition can be specified. The policy also verifies that the transport protocol provides SSL message protection. This policy can be attached to any SOAP-based client.
`wss_saml20_token_bearer_over_ssl_notimestamp_client_policy`	Includes SAML V2.0 tokens in outbound SOAP request messages. The SAML token with confirmation method Bearer is automatically created. The issuer name and subject name are provided either programmatically or through the current Java Authentication and Authorization Service (JAAS) subject. The SOAP header contains no timestamp. Optionally, attesting entity and audience restriction condition can be specified. The policy also verifies that the transport protocol provides SSL message protection. This policy can be attached to any SOAP-based client.
`wss_saml20_token_over_ssl_client_policy`	Includes SAML V2.0 tokens in outbound SOAP request messages. The SAML token is automatically created. The issuer name and subject name are provided either programmatically or through the current Java Authentication and Authorization Service (JAAS) subject. Optionally, attesting entity and audience restriction condition can be specified. The policy also verifies that the transport protocol provides SSL message protection. This policy can be attached to any SOAP-based client.
`wss_username_token_client_policy`	Includes credentials in the `WS-Security UsernameToken` header for all outbound SOAP request messages. Only the plain text mechanism is supported. The credentials can be provided either programmatically, through the Java Authentication and Authorization Service (JAAS), or by a reference in the policy to the configured credential store. This policy can be attached to any SOAP-based client.

Security Policy	Description
`wss_username_token_over_ssl_client_policy`	Includes credentials in the HTTP header for outbound client requests. The credentials are provided either programmatically or through the Java Authentication and Authorization Service (JAAS) subject. It also verifies that the outbound transport protocol is HTTPS. If a non-HTTPS transport protocol is used, then the request is refused. This policy can be applied to any HTTP-based client.
`wss10_message_protection_client_policy`	Provides message integrity and confidentiality for outbound SOAP requests in accordance with the WS-Security v1.0 standard. It uses WS-Security's Basic 128 suite of asymmetric key technologies, specifically RSA key mechanism for message confidentiality, SHA-1 hashing algorithm for message integrity, and AES-128 bit encryption. The keystore on the client side is configured either on a per-request basis or through the security configuration. This policy doesn't authenticate or authorize the requestor.
`wss10_saml_hok_token_with_message_protection_client_policy`	Provides message-level protection and a SAML holder of key based authentication for outbound SOAP messages in accordance with the WS-Security 1.0 standard. It uses WS-Security's Basic 128 suite of asymmetric key technologies, specifically RSA key mechanisms for message confidentiality, SHA-1 hashing algorithm for message integrity, and AES-128 bit encryption. The keystore on the client side is configured either on a per-request basis or through the security configuration. A SAML token, included in the SOAP message, is used in SAML-based authentication with sender vouchers confirmation. These credentials are provided either programmatically or through the security configuration.
`wss10_saml_token_client_policy`	Includes SAML tokens in outbound SOAP request messages. The SAML token is automatically created. The issuer name and subject name are provided either programmatically or through the current Java Authentication and Authorization Service (JAAS) subject.

Security Policy	Description
`wss10_saml_token_with_message_protection_client_policy`	Provides message-level protection and SAML-based authentication for outbound SOAP messages in accordance with the WS-Security 1.0 standard. It uses WS-Security's Basic 128 suite of asymmetric key technologies, specifically RSA key mechanisms for message confidentiality, SHA-1 hashing algorithm for message integrity, and AES-128 bit encryption. The keystore on the client is configured either on a per-request basis or through the security configuration. A SAML token, included in the SOAP message, is used in SAML-based authentication with sender vouchers confirmation. These credentials are provided either programmatically or through the security configuration.
`wss10_saml20_token_client_policy`	Includes SAML V2.0 tokens in outbound SOAP request messages. The SAML token is automatically created. The issuer name and subject name are provided either programmatically or through the current Java Authentication and Authorization Service (JAAS) subject. Optionally, attesting entity and audience restriction can be specified.
`wss10_saml20_token_with_message_protection_client_policy`	Provides message-level protection and SAML V2.0 based authentication for outbound SOAP messages in accordance with the WS-Security 1.0 and SAML Token profile 1.1 standards. It uses WS-Security's Basic 128 suite of asymmetric key technologies, specifically RSA key mechanisms for message confidentiality, SHA-1 hashing algorithm for message integrity, and AES-128 bit encryption. The keystore on the client is configured either on a per-request basis or through the security configuration. A SAML V2.0 token, included in the SOAP message, is used in SAML-based authentication with sender vouches confirmation. These credentials are provided either programmatically or through the security configuration.

Security Policy	Description
`wss10_x509_token_with_message_protection_client_policy`	Provides message-level protection and certificate credential population for outbound SOAP requests in accordance with the WS-Security 1.0 standard. It uses WS-Security's Basic 128 suite of asymmetric key technologies, specifically RSA key mechanisms for message confidentiality, SHA-1 hashing algorithm for message integrity, and AES-128 bit encryption. The keystore on the client side is configured either on a per-request basis or through the security configuration. Authentication credentials are included in the SOAP message through the WS-Security binary security token. These credentials are provided either programmatically or through the security configuration
`wss10_saml_token_with_message_protection_ski_basic256_client_policy`	Provides message-level protection and SAML-based authentication for outbound SOAP messages in accordance with the WS-Security 1.0 standard. It uses WS-Security's Basic 256 suite of asymmetric key technologies, specifically RSA key mechanisms for message confidentiality, SHA-1 hashing algorithm for message integrity, and AES-256 bit encryption. This policy uses the Subject Key Identifier (ski) reference mechanism for an encryption key in the request and for both signature and encryption keys in the response. The keystore on the client is configured either on a per-request basis or through the security configuration. A SAML token, included in the SOAP message, is used in SAML-based authentication with sender vouches confirmation. These credentials are provided either programmatically or through the security configuration.

Security Policy	Description
`wss10_username_id_propagation_with_msg_protection_client_policy`	Enables message-level protection (that is, integrity and confidentiality) and identity propagation for outbound SOAP requests using mechanisms described in WS-Security 1.0. Message protection is provided using WS-Security's Basic 128 suite of asymmetric key technologies, specifically RSA key mechanisms for confidentiality, SHA-1 hashing algorithm for integrity and AES-128 bit encryption. The keystore on the client side is configured either on a per request basis or through the security configuration. Credentials (only user name) are included in outbound SOAP request messages via a `WS-Security UsernameToken` header. No password is included. The user name included can be provided either programmatically, via the current JAAS Subject or by a reference in the policy itself to the configured credential store. This policy can be applied to any SOAP-based client.
`wss10_username_token_with_message_protection_client_policy`	Provides message-level protection (message integrity and confidentiality) and authentication for outbound SOAP requests in accordance with the WS-Security v1.0 standard. It uses WS-Security's Basic 128 suite of asymmetric key technologies, specifically RSA key mechanism for message confidentiality, SHA-1 hashing algorithm for message integrity, and AES-128 bit encryption. The keystore on the client side is configured either on a per-request basis or through the security configuration. Credentials are included in the `WS-Security UsernameToken` header in the outbound SOAP message. Only plain text mechanism is supported. Credentials can be provided either programmatically through the current Java Authentication and Authorization Service (JAAS) subject, or by a reference in the policy to the configured credential store. This policy can be attached to any SOAP-based client.

Security Policy	Description
`wss10_username_token_with_message_protection_ski_basic256_client_policy`	Provides message-level protection and SAML-based authentication for outbound SOAP messages in accordance with the WS-Security 1.0 standard. It uses WS-Security's Basic 256 suite of asymmetric key technologies, specifically RSA key mechanisms for message confidentiality, SHA-1 hashing algorithm for message integrity, and AES-256 bit encryption. This policy uses the Subject Key Identifier (ski) reference mechanism for encryption key in the request and for both signature and encryption keys in the response. The keystore on the client is configured either on a per-request basis or through the security configuration. A SAML token, included in the SOAP message, is used in SAML-based authentication with sender vouches confirmation. These credentials are provided either programmatically or through the security configuration.
`wss11_x509_username_token_with_message_protection_client_policy`	Provides message-level protection and certificate-based authentication for outbound SOAP requests in accordance with the WS-Security 1.1 standard. Messages are protected using WS-Security's Basic 128 suite of symmetric key technologies, specifically RSA key mechanisms for message confidentiality, SHA-1 hashing algorithm for message integrity, and AES-128 bit encryption. The keystore on the client side is configured either on a per-request basis or through the security configuration. Credentials are included in the WS-Security binary security token of the SOAP message. These credentials are provided either programmatically or through the security configuration.
`wss11_saml_token_identity_switch_with_message_protection_client_policy`	Provides message-level protection and SAML-based authentication for outbound SOAP requests in accordance with the WS-Security 1.1 standard. Messages are protected using WS-Security's Basic 128 suite of symmetric key technologies, specifically RSA key mechanisms for message confidentiality, SHA-1 hashing algorithm for message integrity, and AES-128 bit encryption. The keystore on the client is configured either on a per-request basis or through the security configuration. A SAML token, included in the SOAP message, is used in SAML-based authentication with sender vouches confirmation. These credentials are provided either programmatically or through the security configuration. This policy performs dynamic identity switching by propagating a different identity than the one based on an authenticated Subject. This policy can be attached to any SOAP-based client.

Security Policy	Description
`wss11_message_protection_client_policy`	Provides message integrity and confidentiality for outbound SOAP requests in accordance with the WS-Security 1.1 standard. It uses WS-Security's Basic 128 suite of symmetric key technologies, specifically RSA key mechanisms for message confidentiality, SHA-1 hashing algorithm for message integrity, and AES-128 bit encryption. The keystore on the client side is configured either on a per-request basis or through the security configuration. This policy doesn't authenticate or authorize the requestor.
`wss11_saml_token_with_message_protection_client_policy`	Provides message-level protection and SAML-based authentication for outbound SOAP requests in accordance with the WS-Security 1.1 standard. Messages are protected using WS-Security's Basic 128 suite of symmetric key technologies, specifically RSA key mechanisms for message confidentiality, SHA-1 hashing algorithm for message integrity, and AES-128 bit encryption. The keystore on the client is configured either on a per-request basis or through the security configuration. A SAML token, included in the SOAP message, is used in SAML-based authentication with sender vouches confirmation. These credentials are provided either programmatically or through the security configuration. This policy can be attached to any SOAP-based client.
`wss11_username_token_with_message_protection_client_policy`	Provides message-level protection and authentication for outbound SOAP requests in accordance with the WS-Security 1.1 standard. Messages are protected using WS-Security's Basic 128 suite of symmetric key technologies, specifically RSA key mechanisms for message confidentiality, SHA-1 hashing algorithm for message integrity, and AES-128 bit encryption. The keystore on the client side is configured either on a per-request basis or through the security configuration. Credentials are included in the `WS-Security UsernameToken` header of outbound SOAP request messages. Only the plain text mechanism is supported. Credentials are provided either programmatically through the current Java Authentication and Authorization Service (JAAS) subject or by a reference in the policy to the configured credential store. This policy can be attached to any SOAP-based client.

Security Policies for ICS Connector APIs

The supported Oracle Web Services Manager (Oracle WSM) security policies for ICS Connector APIs are described in the following table:

Security Policy	Description
`http_basic_auth_over_ssl_client_policy`	Includes credentials in the HTTP header for outbound client requests. This policy also verifies that the transport protocol is HTTPS. Requests over a non-HTTPS transport protocol are refused. This policy can be applied to any HTTP-based endpoint.
`wss_http_token_over_ssl_client_policy`	Includes credentials in the HTTP header for outbound client requests. The credentials are provided either programmatically or through the Java Authentication and Authorization Service (JAAS) subject. This policy also verifies that the transport protocol is HTTPS. Requests over a non-HTTPS transport protocol are refused. This policy can be applied to any HTTP-based endpoint.
`wss_username_token_over_ssl_client_policy`	Includes credentials in the HTTP header for outbound client requests. The credentials are provided either programmatically or through the Java Authentication and Authorization Service (JAAS) subject. It also verifies that the outbound transport protocol is HTTPS. If a non-HTTPS transport protocol is used, then the request is refused. This policy can be applied to any HTTP-based client.

Security Policies for Fusion Applications Connector APIs

The supported Oracle Web Services Manager (Oracle WSM) security policies for REST Connector APIs are described in the following table:

Security Policy	Description
`wss_http_token_client_policy`	Includes credentials in the HTTP header for outbound client requests. The credentials can be provided either programmatically or through the current Java Authentication and Authorization Service (JAAS) subject. This policy can be applied to any HTTP-based client. Note: Currently only HTTP Basic Authentication is supported.

Security Policy	Description
`wss_saml_token_bearer_over_ssl_client_policy`	Includes SAML tokens in outbound SOAP request messages. The SAML token with confirmation method `Bearer` is automatically created. The issuer name and subject name are provided either programmatically or through the current Java Authentication and Authorization Service (JAAS) subject. The policy also verifies that the transport protocol provides SSL message protection. This policy can be attached to any SOAP-based client
`oauth2_config_client_policy`	Provides information about the OAuth2 server, which preforms authorization and issues the access tokens. You must set both this policy and `oracle/http_oauth2_token_client_policy` together.
`http_oauth2_token_client_policy`	Includes OAuth2 access token in the request. OAuth2 allows users to safely grant client applications limited access to protected resources.. You must set both this policy and `oracle/oauth2_config_client_policy` together.
`http_oauth2_token_over_ssl_client_policy`	Includes OAuth2 access token in the request. OAuth2 allows users to safely grant client applications limited access to protected resources. You must set both this policy and `oracle/oauth2_config_client_policy` together. This version of the policy enforces that connections are made over https.

Security Policy Properties

Every security policy has a set of attributes that defines it. Some of these attributes can be overridden (see Setting Security Policies and Policy Overrides for REST Connector APIs and Setting Security Policies and Policy Overrides for SOAP Connector APIs). The following table lists the attributes that you can modify and their descriptions:

Property	Description
`attesting.mapping.structure`	The mapping attribute used to represent the attesting entity. Only the DN (distinguished name) is currently supported. This attribute is applicable only to sender vouches and then only to message protection use cases. It isn't applicable to SAML over SSL policies.

Property	Description
audience.uri	Audience restriction. The following conditions are supported: • If not set, the service URL is used as the audience URI • If set to NONE (case insensitive), the audience URI is set to null • If set to a value other than NONE, the audience URI is set to this value
authz.code	The previously obtained OAuth2 authorization code.
csf.key	Credential Store key that maps to a user name and password in the Oracle Platform Security Services identity store.
csf.map	Oracle WSM map in the credential store that contains the CSF aliases.
federated.client.token	The federated identity that enables you to consolidate the multiple local identities that you've configured among multiple service providers. Allows you to log on at one service provider site without having to re-authenticate or re-establish your identity.
include.certificate	The signer's certificate.
issuer.name	Name of the JWT issuer. The default value is www.oracle.com
keystore.enc.csf.key	The alias and password used for storing the decryption key password in the keystore. If you set this value, then you can override it. If you do override this value, then the key for the new value must be in the keystore. That is, overriding the value doesn't free you from the requirement of configuring the key in the keystore.
keystore.recipient.alias	Keystore alias associated with the peer certificate. The security runtime uses this alias to extract the peer certificate from the configured keystore and to encrypt messages to the peer. Valid value is orakey.
keystore.sig.csf.key	The alias and password used for storing the signature key password in the keystore. This property allows you to specify the signature key on a per-attachment level instead of at the domain level.
oauth2.client.csf.key	The Credential Store Framework key to the OAuth2 client username and password. The client credentials are the same on every request.
propagate.identity.context	Propagation of the identity context from the web service client to the web service, and then makes it available ("publishes it") to other components for authentication and authorization purposes. This is applicable to both SAML and OAuth, but not to HTTP Basic Authentication.

Property	Description
`redirect.uri`	The redirect URI specified when obtaining the authorization code (set this property if setting `authz.code`).
`role`	SOAP role
`saml.assertion.filename`	Name of the SAML token file.
`saml.audience.uri`	Representation of the relying party, as a comma-separated URI. This field accepts the following wildcards: • * in any location • /* at the end of the URI • .* at the end of the URI
`saml.enveloped.signature.required`	Flag that specifies whether the `Bearer` token is signed using the domain signature key. You can override the domain signature key using the private signature key configured using `keystore.sig.csf.key`. Set this flag to false (in both the client and service policy) to have the `Bearer` token be unsigned.
`saml.issuer.name`	Name identifier for the issuer of the SAML token.
`scope`	Ability for a user to grant the client application access to specific resources rather than a blanket authorization. .Passed to the OAuth2 server token request
`subject.precedence`	Identification of the authenticated principal. If set to false, then allows use of a client-specific user name rather than the authenticated subject. If set to true, then the user name to create the SAML assertion is obtained only from the Subject. Similarly, if set to false, the user name to create the SAML assertion is obtained only from the csf-key user name property.
`token.uri`	The OAuth2 server's token endpoint URI, which issues the access tokens.
`user.attributes`	User attributes related to the principal of the SAML token. Attributes are added as a comma-separated list. The attribute names that you specify must exactly match valid attributes in the configured identity store. The Oracle WSM runtime reads the values for these attributes from the configured identity store, and then includes the attributes and their values in the SAML assertion.
`user.roles.include`	(SOAP) Flag that specifies whether to include SOAP roles. (REST) User roles to be included in the token. If set to true, then the authenticated user roles are included in the token as private claims. The default is false.
`user.tenant.name`	Reserved for use with Oracle Cloud.

The following table shows which security policies have these attributes:

Property	Security Policies Containing the Property
attesting.mapping.structure	SOAP security policies: wss10_saml20_token_with_message_protection_client_policy wss11_saml20_token_with_message_protection_client_policy
audience.uri	REST security policies: http_jwt_token_client_policy http_jwt_token_identity_switch_client_policy http_jwt_token_over_ssl_client_policy http_oauth2_token_client_policy http_oauth2_token_over_ssl_client_policy Fusion Applications security policies: http_oauth2_token_client_policy http_oauth2_token_over_ssl_client_policy
authz.code	REST security policies: http_oauth2_token_client_policy http_oauth2_token_over_ssl_client_policy Fusion Applications security policies: http_oauth2_token_client_policy http_oauth2_token_over_ssl_client_policy

Property	Security Policies Containing the Property
csf.key	REST security policies: http_basic_auth_over_ssl_client_policy http_jwt_token_client_policy http_jwt_token_identity_switch_client_policy http_jwt_token_over_ssl_client_policy http_saml20_token_bearer_client_policy http_saml20_token_bearer_over_ssl_client_policy SOAP security policies: http_basic_auth_over_ssl_client_policy wss_http_token_client_policy wss_http_token_over_ssl_client_policy wss_saml_token_bearer_client_policy wss_saml_token_bearer_over_ssl_client_policy wss_saml20_token_bearer_over_ssl_client_policy wss_saml20_token_over_ssl_client_policy wss_username_token_client_policy wss_username_token_over_ssl_client_policy wss10_saml_token_client_policy wss10_saml_token_with_message_integrity_client_policy wss10_saml_token_with_message_protection_client_policy wss10_saml20_token_client_policy wss10_saml20__token_with_message_protection_client_policy wss10_saml_token_with_message_protection_ski_basic256_client_policy wss10_username_token_with_message_protection_client_policy wss10_username_token_with_message_protection_ski_basic256_client_policy wss11_saml_token_identity_switch_with_message_protection_client_policy wss11_saml_token_with_message_protection_client_policy wss11_saml20_token_with_message_protection_client_policy wss11_username_token_with_message_protection_client_policy Fusion Applications security policies: wss_http_token_client_policy wss_saml_token_bearer_over_ssl_client_policy ICS security policies: http_basic_auth_over_ssl_client_policy

Property	Security Policies Containing the Property

	ICS security policies:
	`http_basic_auth_over_ssl_client_policy`
	Fusion Applications security policies:
	`wss_http_token_client_policy`
	`wss_saml_token_bearer_over_ssl_client_policy`

`csf.map`	REST security policy: `http_jwt_token_identity_switch_client_policy`

`federated.client.token`	REST security policies:
	`http_oauth2_token_client_policy`
	`http_oauth2_token_over_ssl_client_policy`
	Fusion Applications security policies:
	`http_oauth2_token_client_policy`
	`http_oauth2_token_over_ssl_client_policy`

`include.certificate`	REST security policies:
	`http_jwt_token_client_policy`
	`http_jwt_token_identity_switch_client_policy`
	`http_jwt_token_over_ssl_client_policy`
	`http_oauth2_token_client_policy`
	`http_oauth2_token_over_ssl_client_policy`
	Fusion Applications security policies:
	`http_oauth2_token_client_policy`
	`http_oauth2_token_over_ssl_client_policy`

`issuer.name`	REST security policies:
	`http_jwt_token_client_policy`
	`http_jwt_token_identity_switch_client_policy`
	`http_jwt_token_over_ssl_client_policy`
	`http_oauth2_token_client_policy`
	`http_oauth2_token_over_ssl_client_policy`
	Fusion Applications security policies:
	`http_oauth2_token_client_policy`
	`http_oauth2_token_over_ssl_client_policy`

Property	Security Policies Containing the Property
keystore.enc.csf.key	SOAP security policies: wss10_message_protection_client_policy wss10_saml_hok_token_with_message_protection_client_policy wss10_saml_token_with_message_integrity_client_policy wss10_saml_token_with_message_protection_client_policy wss10_saml20_token_with_message_protection_client_policy wss10_x509_token_with_message_protection_client_policy wss10_saml_token_with_message_protection_ski_basic256_client_policy wss10_username_id_propagation_with_msg_protection_client_policy wss10_username_token_with_message_protection_client_policy wss10_username_token_with_message_protection_ski_basic256_client_policy wss11_x509_token_with_message_protection_client_policy wss11_saml_token_identity_switch_with_message_protection_client_policy wss11_message_protection_client_policy wss11_saml_token_with_message_protection_client_policy wss11_saml20_token_with_message_protection_client_policy wss11_username_token_with_message_protection_client_policy

Property	Security Policies Containing the Property
keystore.recipient.alias	SOAP security policies: wss10_message_protection_client_policy wss10_saml_hok_token_with_message_protection_client_policy wss10_saml_token_with_message_protection_client_policy wss10_saml20_token_with_message_protection_client_policy wss10_x509_token_with_message_protection_client_policy wss10_saml_token_with_message_protection_ski_basic256_client_policy wss10_username_id_propagation_with_msg_protection_client_policy wss10_username_token_with_message_protection_client_policy wss10_username_token_with_message_protection_ski_basic256_client_policy wss11_x509_token_with_message_protection_client_policy wss11_saml_token_identity_switch_with_message_protection_client_policy wss11_message_protection_client_policy wss11_saml_token_with_message_protection_client_policy wss11_saml20_token_with_message_protection_client_policy wss11_username_token_with_message_protection_client_policy

Property	Security Policies Containing the Property
keystore.sig.csf.key	REST security policies: http_jwt_token_client_policy http_jwt_token_identity_switch_client_policy http_jwt_token_over_ssl_client_policy http_saml20_token_bearer_client_policy http_saml20_token_bearer_over_ssl_client_policy http_oauth2_token_client_policy http_oauth2_token_over_ssl_client_policy SOAP security policies: wss_saml_token_bearer_client_policy wss_saml_token_bearer_over_ssl_client_policy wss_saml20_token_bearer_over_ssl_client_policy wss10_message_protection_client_policy wss10_saml_hok_token_with_message_protection_client_policy wss10_saml_token_with_message_integrity_client_policy wss10_saml_token_with_message_protection_client_policy wss10_saml20_token_with_message_protection_client_policy wss10_x509_token_with_message_protection_client_policy wss10_saml_token_with_message_protection_ski_basic256_client_policy wss10_username_id_propagation_with_msg_protection_client_policy wss10_username_token_with_message_protection_client_policy wss10_username_token_with_message_protection_ski_basic256_client_policy wss11_x509_token_with_message_protection_client_policy wss11_saml_token_identity_switch_with_message_protection_client_policy wss11_saml_token_with_message_protection_client_policy wss11_saml20_token_with_message_protection_client_policy Fusion Applications security policies: http_oauth2_token_client_policy http_oauth2_token_over_ssl_client_policy wss_saml_bearer_token_over_ssl_client_policy

Property	Security Policies Containing the Property
oauth2.client.csf.key	REST security policies: http_oauth2_token_client_policy http_oauth2_token_over_ssl_client_policy Fusion Applications security policies: http_oauth2_token_client_policy http_oauth2_token_over_ssl_client_policy
propagate.identity.context	REST security policies: http_jwt_token_client_policy http_jwt_token_identity_switch_client_policy http_jwt_token_over_ssl_client_policy http_saml20_token_bearer_client_policy http_saml20_token_bearer_over_ssl_client_policy http_oauth2_token_client_policy http_oauth2_token_over_ssl_client_policy SOAP security policies: wss_saml_token_bearer_client_policy wss_saml_token_bearer_over_ssl_client_policy wss_saml20_token_bearer_over_ssl_client_policy wss_saml20_token_over_ssl_client_policy wss10_saml_token_client_policy wss10_saml_token_with_message_integrity_client_policy wss10_saml_token_with_message_protection_client_policy wss10_saml20_token_client_policy wss10_saml20_token_with_message_protection_client_policy wss10_saml_token_with_message_protection_ski_basic256_client_policy wss11_saml_token_with_message_protection_client_policy wss11_saml20_token_with_message_protection_client_policy Fusion Applications security policies: http_oauth2_token_client_policy http_oauth2_token_over_ssl_client_policy wss_saml_token_bearer_over_ssl_client_policy

Property	Security Policies Containing the Property
redirect.uri	REST security policies: http_oauth2_token_client_policy http_oauth2_token_over_ssl_client_policy Fusion Applications security policies: http_oauth2_token_client_policy http_oauth2_token_over_ssl_client_policy
role	REST security policy: oauth2_config_client_policy SOAP security policies: wss_http_token_client_policy wss_http_token_over_ssl_client_policy wss_username_token_client_policy wss_username_token_over_ssl_client_policy wss10_message_protection_client_policy wss10_x509_token_with_message_protection_client_policy wss10_username_id_propagation_with_msg_protection_client_policy wss10_username_token_with_message_protection_client_policy wss10_username_token_with_message_protection_ski_basic256_client_policy wss11_message_protection_client_policy ICS security policies: wss_username_token_over_ssl_client_policy Fusion Applications security policies: wss_http_token_client_policy http_oauth2_config_client_policy
saml.assertion.filename	SOAP security policy: wss10_saml_hok_token_with_message_protection_client_policy

Property	Security Policies Containing the Property
saml.audience.uri	REST security policies: http_saml20_token_bearer_client_policy http_saml20_token_bearer_over_ssl_client_policy SOAP security policies: wss_saml_token_bearer_client_policy wss_saml_token_bearer_over_ssl_client_policy wss_saml20_token_bearer_over_ssl_client_policy wss_saml20_token_over_ssl_client_policy wss10_saml_token_client_policy wss10_saml_token_with_message_integrity_client_policy wss10_saml_token_with_message_protection_client_policy wss10_saml20_token_client_policy wss10_saml20_token_with_message_protection_client_policy wss10_saml_token_with_message_protection_ski_basic256_client_policy wss11_saml_token_identity_switch_with_message_protection_client_policy wss11_saml_token_with_message_protection_client_policy wss11_saml20_token_with_message_protection_client_policy Fusion Applications security policies: wss_saml_token_bearer_over_ssl_client_policy
saml.enveloped.signature.required	REST security policies: http_saml20_token_bearer_client_policy http_saml20_token_bearer_over_ssl_client_policy SOAP security policies: wss_saml_token_bearer_client_policy wss_saml_token_bearer_over_ssl_client_policy wss_saml20_token_bearer_over_ssl_client_policy Fusion Applications security policies: wss_saml_token_bearer_over_ssl_client_policy

Property	Security Policies Containing the Property
saml.issuer.name	REST security policies: http_saml20_token_bearer_client_policy http_saml20_token_bearer_over_ssl_client_policy SOAP security policies: wss_saml_token_bearer_client_policy wss_saml_token_bearer_over_ssl_client_policy wss_saml20_token_bearer_over_ssl_client_policy wss_saml20_token_over_ssl_client_policy wss10_saml_hok_token_with_message_protection_client_policy wss10_saml_token_client_policy wss10_saml_token_with_message_integrity_client_policy wss10_saml_token_with_message_protection_client_policy wss10_saml20_token_client_policy wss10_saml20_token_with_message_protection_client_policy wss10_saml_token_with_message_protection_ski_basic256_client_policy wss11_saml_token_identity_switch_with_message_protection_client_policy wss11_saml_token_with_message_protection_client_policy wss11_saml20_token_with_message_protection_client_policy Fusion Applications security policies: wss_saml_token_bearer_over_ssl_client_policy
scope	REST security policies: http_oauth2_token_client_policy http_oauth2_token_over_ssl_client_policy Fusion Applications security policies: http_oauth2_token_client_policy http_oauth2_token_over_ssl_client_policy

Property	Security Policies Containing the Property
subject.precedence	REST security policies: http_jwt_token_client_policy http_jwt_token_identity_switch_client_policy http_jwt_token_over_ssl_client_policy http_saml20_token_bearer_client_policy http_saml20_token_bearer_over_ssl_client_policy SOAP security policies: wss_saml_token_bearer_client_policy wss_saml_token_bearer_over_ssl_client_policy wss_saml20_token_bearer_over_ssl_client_policy wss_saml20_token_over_ssl_client_policy wss10_saml_token_client_policy wss10_saml_token_with_message_integrity_client_policy wss10_saml_token_with_message_protection_client_policy wss10_saml20_token_client_policy wss10_saml20_token_with_message_protection_client_policy wss10_saml_token_with_message_protection_ski_basic256_client_policy wss11_saml_token_identity_switch_with_message_protection_client_policy wss11_saml_token_with_message_protection_client_policy wss11_saml20_token_with_message_protection_client_policy Fusion Applications security policies: wss_saml_token_bearer_over_ssl_client_policy
token.uri	REST security policy: oauth2_config_client_policy Fusion Applications security policies: http_oauth2_config_client_policy

Property	Security Policies Containing the Property
user.attributes	REST security policies: http_jwt_token_client_policy http_jwt_token_identity_switch_client_policy http_jwt_token_over_ssl_client_policy http_saml20_token_bearer_client_policy http_saml20_token_bearer_over_ssl_client_policy http_oauth2_token_client_policy http_oauth2_token_over_ssl_client_policy SOAP security policies: wss_saml_token_bearer_client_policy wss_saml_token_bearer_over_ssl_client_policy wss_saml20_token_bearer_over_ssl_client_policy wss_saml20_token_over_ssl_client_policy wss10_saml_hok_token_with_message_protection_client_policy wss10_saml_token_client_policy wss10_saml_token_with_message_integrity_client_policy wss10_saml_token_with_message_protection_client_policy wss10_saml20_token_client_policy wss10_saml20_token_with_message_protection_client_policy wss10_saml_token_with_message_protection_ski_basic256_client_policy wss11_saml_token_with_message_protection_client_policy wss11_saml20_token_with_message_protection_client_policy Fusion Applications security policies: http_oauth2_token_client_policy http_oauth2_token_over_ssl_client_policy wss_saml_token_bearer_over_ssl_client_policy

Property	Security Policies Containing the Property
user.roles.include	REST security policies: http_jwt_token_client_policy http_jwt_token_identity_switch_client_policy http_jwt_token_over_ssl_client_policy http_saml20_token_bearer_client_policy http_saml20_token_bearer_over_ssl_client_policy http_oauth2_token_client_policy http_oauth2_token_over_ssl_client_policy SOAP security policies: wss_saml_token_bearer_client_policy wss_saml_token_bearer_over_ssl_client_policy wss_saml20_token_bearer_over_ssl_client_policy wss_saml20_token_over_ssl_client_policy wss10_saml_hok_token_with_message_protection_client_policy wss10_saml_token_client_policy wss10_saml_token_with_message_integrity_client_policy wss10_saml_token_with_message_protection_client_policy wss10_saml20_token_client_policy wss10_saml20_token_with_message_protection_client_policy wss10_saml_token_with_message_protection_ski_basic256_client_policy wss11_saml_token_identity_switch_with_message_protection_client_policy wss11_saml_token_with_message_protection_client_policy wss11_saml20_token_with_message_protection_client_policy Fusion Applications security policies: http_oauth2_token_client_policy http_oauth2_token_over_ssl_client_policy wss_saml_token_bearer_over_ssl_client_policy

Property	Security Policies Containing the Property

`user.tenant.name`	REST security policies:
	`http_basic_auth_over_ssl_client_policy`
	`http_jwt_token_client_policy`
	`http_jwt_token_identity_switch_client_policy`
	`http_jwt_token_over_ssl_client_policy`
	`http_saml20_token_bearer_client_policy`
	`http_saml20_token_bearer_over_ssl_client_policy`
	`http_oauth2_token_client_policy`
	`http_oauth2_token_over_ssl_client_policy`
	SOAP security policies:
	`http_basic_auth_over_ssl_client_policy`
	`wss_http_token_client_policy`
	`wss_saml_token_bearer_client_policy`
	`wss_saml_token_bearer_over_ssl_client_policy`
	`wss_saml20_token_bearer_over_ssl_client_policy`
	`wss_saml20_token_over_ssl_client_policy`
	`wss_username_token_client_policy`
	`wss_username_token_over_ssl_client_policy`
	`wss10_saml_hok_token_with_message_protection_client_policy`
	`wss10_saml_token_client_policy`
	`wss10_saml_token_with_message_integrity_client_policy`
	`wss10_saml_token_with_message_protection_client_policy`
	`wss10_saml20_token_client_policy`
	`wss10_saml20_token_with_message_protection_client_policy`
	`wss10_saml_token_with_message_protection_ski_basic256_client_policy`
	`wss11_saml_token_identity_switch_with_message_protection_client_policy`
	`wss11_saml_token_with_message_protection_client_policy`
	`wss11_saml20_token_with_message_protection_client_policy`
	`wss11_username_token_with_message_protection_client_policy`
	ICS security policies:
	`http_basic_auth_token_over_ssl_client_policy`
	`http_username_token_over_ssl_client_policy`
	Fusion Applications security policies:
	`wss_http_token_client_policy`

Property	Security Policies Containing the Property
	http_oauth2_token_client_policy http_oauth2_token_over_ssl_client_policy wss_saml_token_bearer_over_ssl_client_policy

D

Identity Domain Relocation

Task	Who Does It? Where?	More Information
1. Submit the identity domain relocation.	Your company's Oracle Cloud **account administrator**. Oracle Cloud Infrastructure Classic Console >  > **Users**	See Identity Domain Overview in *Oracle Cloud Understanding Identity Concepts*.
2. Set up SSO in the new identity domain.	Your company's Oracle Cloud **account administrator**. Oracle Cloud Infrastructure Classic Console >  > **Users**	See Configuring Identity Management (SSO and OAuth).
3. Create (or recreate) team members. You can export the team members from the old domain and import them into the new domain.	A **service administrator** for the MCS environment. Oracle Cloud Infrastructure Classic Console >  > **Users**	See Adding Users and Assigning Roles in *Getting Started with Oracle Cloud*.
4. Assign MCS team member roles to define permissions.	A **service administrator** for the MCS environment. Oracle Cloud Infrastructure Classic Console >  > **Users**	See Assign MCS Team Member Roles.
5. Create (or recreate) mobile users. As with team members, you can export the mobile users from the old domain and import them into the new domain.	A team member with the Oracle Cloud **identity domain administrator** role and the mobile user configuration (`MobileEnvironment_MobileUserConfig`) and mobile user management (`MobileEnvironment_MobileUserMgmt`) MCS team member roles in the MCS environment. Oracle Cloud Infrastructure Classic Console >  > **Users**	See Set Up Mobile Users, Realms and Roles.

Task	Who Does It? Where?	More Information
6. Create (or recreate) mobile user roles. Realm role: {serviceName}_MobileEnvironment_{realmName}_{version using underscores}_Realm Mobile user role: {serviceName}_MobileEnvironment_{roleName}	A team member with the Oracle Cloud **identity domain administrator** role and the mobile user configuration (`MobileEnvironment_MobileUserConfig`) and mobile user management (`MobileEnvironment_MobileUserMgmt`) MCS team member roles in the MCS environment. **MCS** > **Applications** > **Mobile User Management** and Oracle Cloud Infrastructure Classic Console > ☰ > **Users** > **Custom Roles**	See Creating and Managing Mobile User Roles.

Note: Role names are case sensitive and must match

Task	Who Does It? Where?	More Information
 then name is in the old domain, with a new {service_name}.		

Task	Who Does It? Where?	More Information
7. Assign (or reassign) mobile user roles.	A team member with the Oracle Cloud **identity domain administrator** role and the mobile user configuration (`MobileEnvironment_MobileUserConfig`) and mobile user management (`MobileEnvironment_MobileUserMgmt`) MCS team member roles in the MCS environment. Oracle Cloud Infrastructure Classic Console >  > **Users**	See Creating and Managing Mobile User Roles.
8. Reset the credentials for the OAuth Consumer for each mobile backend by performing a "refresh" on the MBE Settings page, and enable SSO if it was previously enabled.	A team member with the Oracle Cloud **identity domain administrator** role and the mobile user configuration (`MobileEnvironment_MobileUserConfig`) and mobile user management (`MobileEnvironment_MobileUserMgmt`) MCS team member roles in the MCS environment. **MCS** > **Mobile Backends** > **Settings**	See Enterprise Single Sign-On in MCS.
9. Update settings for mobile apps.	A mobile app developer with access to the mobile backend and the mobile app. Get the updated settings below from the MCS UI and modify them in the SDK config file and your mobile app code as necessary. **MCS** > **Mobile Backends** > **Settings** ...	See Mobile Backend Authentication and Connection Info and Authentication in MCS. For details on SDK configuration, see the following topics: • Configuring SDK Properties for Android • Configuring SDK Properties for Cordova • Configuring SDK Properties for iOS • Configuring SDK Properties for JavaScript
• API URLs	> **Environment URLs** > **Base URL**	
• Token endpoint URLs (OAuth and SSO)	> **Environment URLs** > **OAuth Token Endpoint**	
• Client ID and secret (OAuth)	> **Access Keys** > **OAuth Consumer**	
• Anonymous key (HTTP Basic authentication)	> **Access Keys** > **HTTP Basic** > **Anonymous Keys**	
10. Register (or reregister) clients for notifications. You can use the UI or the `/mobile/platform/devices/register` endpoint in the REST API.	**MCS** > **Applications** > **Client Management**	See Registering an App as a Client in MCS and REST APIs for Oracle Mobile Cloud Service.

E

Writing Swift Applications Using the iOS Client SDK

You can also use the Oracle Mobile Cloud Service iOS client SDK with Swift applications.

Here are the general steps you take to work with Swift and the client SDK, using Xcode as your IDE:

1. Add the bridging header files.
2. Add the SDK header files and libraries.
3. Add the Objective-C linker flag.
4. Compile and link your app using the iOS client SDK as you would any other iOS project in Xcode.

> **Note:**
>
> Using the SDK with Swift has all the same dependencies as using the SDK with Objective-C. For the list of dependencies, see iOS SDK Dependencies.

For more information on how to work effectively with Swift and Objective-C, see Apple's documentation: https://developer.apple.com/library/content/documentation/Swift/Conceptual/BuildingCocoaApps/InteractingWithObjective-CAPIs.html.

Adding the Bridging Header File

You need to use a bridging header file to import the header files of the Objective-C public classes that your Swift app calls. All of the available public classes in the MCS client SDK can be found in the SDK's `include` folder.

To create a bridging header file in Xcode:

1. Select **File > New... > File...** and then from **iOS/Source** choose **Header file** using the **.h** icon. You can give the bridging header file any name you choose.

   Depending on the SDK classes that your app uses, the contents should look something like the following:

```
#ifndef GettingStartedSwift_Bridging_Header_h
#define GettingStartedSwift_Bridging_Header_h

#import "OMCCore.h"
#import "OMCAuthorization.h"
#import "OMCMobileBackend.h"
#import "OMCMobileBackendManager.h"
#import "OMCServiceProxy.h"
```

```
#import "OMCUser.h"

#import "OMCStorage.h"
#import "OMCMobileBackend+OMC_Storage.h"
#import "OMCStorageCollection.h"
#import "OMCStorageObject.h"

#import "OMCSynchronization.h"
#import "OMCMobileBackend+OMC_Synchronization.h"
#import "OMCFetchObjectCollectionBuilder.h"
#import "OMCMobileResource.h"
#import "OMCSyncGlobals.h"

#import "OMCAnalytics.h"
#import "OMCMobileBackend+OMC_Analytics.h"

#import "OMCNotifications.h"
#import "OMCMobileBackend+OMC_Notifications.h"

#import "OMCLocation.h"
#import "OMCMobileBackend+OMC_Location.h"


#endif /* GettingStartedSwift_Bridging_Header_h */
```

2. After you have created the header file, note the location of the file in the Build Settings for the `Objective-C Bridging Header` setting.

   It's best to keep the header location specified relative to the project, rather than as an absolute path, in case the project is shared.

Adding the SDK Headers and Libraries to a Swift App

The set of headers and libraries you add depends upon which of the client SDK's static libraries you include in your app. At a minimum, you need the `libOMCCore.a` and `libIDMMobileSDK.a` libraries.

To add the SDK headers and libraries:

1. Download and unzip the SDK, as described in iOS Applications.
2. From the location where you've unzipped the SDK files, drag the libraries and header files you want into your Swift project in Xcode.

> **Note:**
>
> The contents of the SDK libraries are hierarchically arranged by category, so you'll need to drag over entire folders to preserve the includes of other headers.

3. Under the **Build Phases** settings, add the static libraries plus the iOS frameworks required by the IDM library to the **Link with Binary Libraries** phase.
4. Add the header files to your search path. Under the project settings, configure the `Header Search Paths` to include the location of the parent directory of the SDK

folders, that is, the parent directory of `libOMCCore.a`, `libIDMMobileSDK.a`, and so on. Be sure to use a relative path to the project.

5. Edit the bridging header file to include the header files you'll actually need for your code.

   This means that you'll also need to add headers that are used by the class you wish to use.

   For example, to make sure that all the methods of `OMCAuthorization.h` are accessible, you'd also need to add `OMAuthView.h`, `OMCUser.h` and `OMDefinitions.h`. Without these files in the bridging header file, some methods and properties of `OMCAuthorization` won't be visible, and the compiler won't warn you with errors.

Using SDK Objects in Swift Apps

The rules for converting from Objective-C to Swift are well described in the Apple documentation. For general information on the relationship and usage of these two languages together, be sure you look there.

Watch out for the following:

- The auto-complete feature of the Code Editor in Xcode generally works well enough to get you the mappings. However, sometimes it puts the a label in the first parameter that isn't supposed to be there. Watch for it if you're using auto-complete.
- When Objective-C `init` methods come over to Swift, they take on native Swift initializer syntax. This means the `init` prefix is sliced off and becomes a keyword to indicate that the method is an initializer. See the Apple documentation for complete details.
- Pay special attention to the `!` and `?` optional parameter specifications, as well as any parametrized types in the declarations. The optional types are auto-determined by the compiler when mapping Objective-C to Swift.

You should be able to compile and run your mobile app using Swift and the MCS client SDK on both the Xcode Simulator and an actual device.

Here's an example of Objective-C and the comparable Swift code that uses the MCS client SDK.

The following Objective-C code to register a device token for Push notifications:

```
// Get notifications sdk object
OMCNotifications* notifications = [[appDelegate myMobileBackend]
notifications];

// Register device token with MCS server using notifications sdk
[notifications registerForNotifications:[appDelegate getDeviceTokenData]

            onSuccess:^(NSHTTPURLResponse *response) {

                NSLog(@"Device token registered successfully on MCS
server");

                dispatch_async(dispatch_get_main_queue(), ^{
```

```
                    // Update UI here
                }) ;
            }

        onError:^(NSError *error) {

            NSLog(@"Error: %@", error.localizedDescription);

            dispatch_async(dispatch_get_main_queue(), ^{
                // Update UI here
            }) ;
        }];
```

might be written in the following way in Swift:

```
@IBAction func registerForPushNotifications() {

    // Get notifications sdk object
    let notifications = appDelegate.myMobileBackend().notifications();

    // Get device token first, and assign it here
    let deviceTokenData:NSData! = nil;

    // Register device token with MCS server using notifications sdk
    notifications.registerForNotifications(deviceTokenData, onSuccess:
{ (response:NSHTTPURLResponse!) in

        NSLog("Device token registered successfully on MCS server");

        dispatch_async(dispatch_get_main_queue()) {
            // Update UI here
        }

    }) { (error) in

        print("Error: %@", error.localizedDescription);
    };
}
```

F

Supported Browsers and Languages

Supported Browsers

This table describes the minimum requirements for web browsers that Oracle Mobile Cloud Service supports.

Web Browser	Version
Microsoft Internet Explorer	11
Google Chrome	43
Mozilla Firefox	37, 38
Apple Safari	8.0

Supported Languages

Oracle Mobile Cloud Service supports the following languages in its web interface:

- German (de)
- English (en)
- Spanish (es)
- French (fr)
- Italian (it)
- Japanese (ja)
- Korean (ko)
- Portuguese (pt)
- Chinese - Simplified (zh_CN)
- Chinese - Traditional (zh_TW)

G

Identity Provider Integration

Here are the steps you need to follow to integrate various third-party identity providers with MCS.

Use Case: Configuring OKTA to Obtain a SAML Token

Here are the required fields that you must fill in if you're configuring a SAML 2.0 app from OKTA.

Assuming that you have a user role with administrator privileges in OKTA:

1. Log in to OKTA.
2. Click **Admin**.
3. Go to the Directory tab and specify the users to have access privileges to the application:
   - Select **People** to specify individual users
   - Select **Group** to specify a group of users

     By setting a group, you can later map a group of individuals to specific MCS roles by setting Role Attribute rules in the Keys and Certificates dialog. See Rule Types.
   - Select **Directory Integration**, then **Add Active Directory** to include all the users in the directory server

     or, alternatively, select LDAP to include all the users in an LDAP directory server
4. Go to the Applications tab and click **Add Application** to create a new SAML 2.0 application.
5. On the General Settings page, configure the SAML application.

   You'll see several fields to fill in. For the token to be viable with MCS, you must fill-in the following fields:
   - **Single Sign-On URL**. This is the redirect URL where the response from the third-party IdP is sent. For example:

     `https://hostname:####/saml`
   - **Audience URI**. This is the intended audience of the SAML assertion. Set this value to the MCS SSO token endpoint URL.

     To exchange an externally-issued SAML token for an MCS-issued token that can be used with subsequent MCS API calls, ensure that the audience value specified in the token includes the MCS token exchange URL. The URL must include a port number, even when a default port is being used. For example:

     `https://hostname:443/mobile/platform/sso/token`

     You construct this endpoint by appending `/mobile/platform/sso/exchange-token` to your instance's base URL. You can determine the base URL by

opening any mobile backend in MCS, clicking its **Settings** tab, and looking in the Environment URLs section. For example:

```
https://hostname:443/mobile/platform/sso/exchange-token
```

- **Group Statement**. This is where you can add additional group attributes to the token. In this field, you can filter which groups to add. There are different types of filtering options that you can choose from. For instance, if you used a naming convention for your group names, you can set an option (**Regex** or **Start with**) to filter groups that begin with a specific prefix.

  For example, say you defined several group of users, two groups for FixItFast employees, `FIF-group1` and `FIF-group2`, and a group for RepairItFast employees, `RIF-group1`. If you enter `FIF*` as a value, only the users in the FixItFast group are added to the token.

6. Once you've configured the app, go to the Single Sign-On page.

   This is where you'll get the token issuer name that you'll enter into the Token Issuer panel of token issuer. See Adding a Token Issuer.
   You'll also want to get token certificate contents from this page. Paste the certificate contents in the Web Service and Token Certificates panel of the Keys and Certificates dialog when you add a token certificate. See Configuring a Web Service or Token Certificate.

Use Case: Configuring AD FS to Obtain a SAML Token

Configuring Active Directory Federation Services (AD FS) to obtain a SAML token involves providing similar information as you would for configuring another identity provider to obtain the token. You'll configure an audience, provide a redirect URL to obtain the token, and configure some rules.

In addition to having access to the AD FS server, you'll need the following items:

- A defined set of users and groups.
- A Certificate Authority (CA) root certificate and a Signing Certificate from a valid certificate authority. You'll import these certificates into your AD FS instance.

  These are the token certificates and corresponding private key that are imported into AD FS so that it can generate and sign SAML tokens. These certificates must also be added to the Token Certificates panel of the CSF Keys and Certificate dialog in MCS so that MCS can validate the token. These are the token certificates that will be associated with the token issuer in MCS.

For testing purposes, you can create a root certificate and a self-signing certificate as shown in the following examples but don't use them in a production environment.

Here's an example of how to create a root certificate:

```
$ openssl req -x509 -nodes -days 3650 -subj "/C=US/ST=CA/L=Local/
O=SampleCA/OU=Self-Signed/CN=ca.test.local" -newkey rsa:2048 -keyout
testCARootPrivateKey.key -out testCARootCertificate.crt
```

Here's an example of how to create a new key pair and the corresponding certificate:

```
$ openssl req -nodes -days 3650 -subj "/C=US/ST=CA/L=Local/O=SampleCA/
OU=Self-Signed/CN=sts-signing.test.local" -newkey rsa:2048 -keyout
testSigningPrivateKey.key -out testSigningCertificate.csr

$ openssl x509 -req -days 3650 -in testSigningCertificate.csr -CA ../ca/
testCARootCertificate.crt -CAkey ../ca/testCARootPrivateKey.key -
CAcreateserial -out testSigningCertificate.crt

$ openssl pkcs12 -export -out testSigningCertificate.pfx -inkey
testSigningPrivateKey.key -in testSigningCertificate.crt
```

Creating Users and Groups in AD FS

You need to create users and assign them to groups in AD FS. In MCS, these user groups are mapped to existing MCS roles. This assumes that you have the AD FS server installed.

Start AD and add users:

1. Select **Tools > Active Directory Users and Computers**.
2. Open the `Active Directory and Users and Computers` folder.

   This is the directory where you'll add users and groups.
3. Right-click the `Users` folder and select **New > User**.
4. In the **New Object - User** dialog, provide a first and last name for each user you add and the user logon name. The logon name must match the user email address for that user in MCS.

   For example, if the user is John Smith, and his address is `jsmith@local.domain`, the address must match the email address for user John Smith in MCS.
5. Click **Next** and then **OK** to add the user.

   Repeat these steps for each user you want to add.

To add a group and assign a user to it:

1. Right-click the **Users** folder in the `Active Directory and Users and Computers` directory and select **New > Group**.
2. In the **New Object - Group** dialog, enter a name for the group.
3. Leave the default settings of Global and Security, for **Group Scope** and **Group Type** and click **OK**.
4. Right-click on the user name in the `Active Directory and Users and Computers` directory and select **Add to a group...**.
5. In the **Select Group** dialog, click **Advanced**.
6. In the advanced version of the **Select Groups** dialog, click **Find Now**.
7. Locate the group name from the Search results list, select it, and click **OK**.
8. Click **OK** in the **Select Group** dialog to complete the group assignment.

To verify that you've added the user to the correct group:

1. Click on the group name in the `Active Directory and Users and Computers` directory to open the group's properties dialog.
2. In the properties dialog, click **Members** and look to see if the user you added is listed.

A group should have a corresponding role in MCS. The user assigned to the group would then be assigned to the corresponding MCS role.

Configuring the SAML App in AD FS

After you've added your users and groups and have a valid root certificate and signing certificate, you can configure the SAML token. You'll begin by adding and configuring a relying party trust. The relying party defines the way in which AD FS recognizes the relying party application and issues claims to it.

1. From the Server Manager, select **Tools > AD FS Management**.
2. In the **AD FS** window, select **Action > Add Relying Party Trust...**.
3. Click **Start** in the Add Relying Trust wizard.
4. On the **Select Data Source** panel, select **Enter data about the relying party trust manually** option.
5. Click **Next** to go to the **Specify Display Name** panel.
6. Enter the name of your SAML app in the **Display Name** field.

   This app name will be listed in the `Trust Relationships > Relying Party Trust` directory after you add it.
7. Click **Next** to go to the **Choose Profile** panel.
8. Select **AD FS profile** (the default value).

   This is the profile type that supports the SAML 2.0 protocol.
9. Click **Next** and **Next** again to go to the **Configure URL** panel.

   You can upload the signing certificate on the **Configure Certificate** panel now or upload it later. You don't need to upload an encryption certificate unless you want the SAML assertion encrypted as well as signed. Having an encrypted SAML assertion can be useful in cases where sensitive data is added to the SAML assertion claims.
10. Select **Enable support for the SAML 2.0 Web SSO protocol** and enter the redirect URL in the **Relying party SAML 2.0 SSO service URL** field.

    The redirect URL is the address where you want the request to post back to so you can intercept the token.
11. Click **Next** to go to the **Configure Identifiers** panel.
12. Enter the SSO token endpoint in the **Relying party trust identifier** field and click **Add**.

    You construct this endpoint by appending `/mobile/platform/sso/exchange-token` to your instance's base URL. You can determine the base URL by opening any mobile backend in MCS, clicking its **Settings** tab, and looking in the Environment URLs section. For example:

    `https://`*`hostname`*`:443/mobile/platform/sso/exchange-token`

This is how you specify the audience for the SAML assertion.

13. Click **Next** to go to the **Configure Multi-factor Authentication Now** panel.

    Use the default setting, I do not want to configure multi-factor authentication settings for this relying party trust.

14. Click **Next** to go to the **Choose Issuance Authorization Rules** panel.

    Use the default setting, Permit all users to access this relying party.

15. Click **Next** to go to the **Ready to Add Trust** panel, click **Next** again.

16. Click **Finish**.

    Leave the default setting, **Open the Edit Claim Rules dialog for this relying party trust** to continue configuring your SAML app.

17. Click **Close** to exit the wizard.

    The **Edit Claim Rules** dialog opens when you exit the wizard.

Configuring Claim Rules in AD FS

The next step to configure your SAML app is setting the claim rules. The claim rule specifies how the values for LDAP attributes are mapped to the outgoing claim type. You'll use the Add Transform Claim Rule wizard available from the **Edit Claim Rules** dialog to add AD claims and transform NameID transform rule which specify the claims that are sent to the relying party.

1. Open the `Relying Party Trust` folder under the `Trust Relationships` directory and right-click your app name. Then select **Edit Claim Rules**.

   If you're continuing on from the previous section, the **Edit Claim Rules** dialog opens automatically when you exit the Add Relying Trust wizard.

2. Make sure the **Issuance Transform Rules** tab is open and click **Add Rule** to open the Add Transform Claim Rule wizard.

3. In the **Choose Rule Type** tab, select the **Send LDAP Attributes as Claims** template from the drop-down list.

4. Click **Next** to go to the **Configure Claim Rule** tab.

5. Enter a claim rule name. For example, `AD Claims`.

6. Select **Active Directory** as the Attribute store.

   In the next set of steps, you'll map the LDAP attributes to the outgoing claim types:

LDAP Attributes	Outgoing Claim Type
E-Mail Addresses	E-Mail Address
Token-Groups-Unqualified Name	Group
User-Principal-Name	Common Name

7. Open the **LDAP Attributes** list and select `E-Mail Addresses`.

8. Open the **Outgoing Claim Type** list and select `E-Mail Address`.

9. Repeat steps 7 and 8 to map `Token-Groups-Unqualified Name` to `Group` and to map `User-Principal-Name` to `Common Name`.

10. Click **Finish**.

Configuring Transform Rules in AD FS

You set transform rules to map incoming claim types to outgoing claim types and specify the action that determines what output should occur based on the values from the incoming claim.

1. Open the **Edit Claim Rules** dialog and open the **Issuance Transform Rules** tab.
2. Click **Add Rule** to open the Add Transform Claim Rule wizard.
3. In the **Choose Rule Type** tab, select **Transform an Incoming Claim**.
4. Click **Next** to go to the **Configure Claim Rule** tab.
5. Perform the following actions on this tab:
   - Enter `Transform NameID` for the transform claim rule.
   - Select **EMAIL ADDRESS** for the incoming claim type.
   - Select **Name ID** for the outgoing claim type.
   - Leave as unspecified the ingoing and outgoing nameID formats.
   - Select the **Pass through all claim values** option.
6. Click **Finish**.
7. Click **Apply** and **OK** in the **Edit Claim Rules** dialog.

Specifying the Signature Verification Certificate in AD FS

You must specify the signature verification certificates for requests from the relying party trust.

1. Open the `Relying Party Trusts` folder, right-click your app name, and select **Properties**.
2. In the properties dialog for your app, select **Signature** and click **Add**.
3. In the **Select a Request Signature Verification Certificate** dialog, navigate to the directory where you stored (or created) the signing certificate and select the certificate.
4. Click **Open**.
5. (Optional) Click the **Endpoints** tab in the app properties dialog and review the SAML assertion endpoints.

   Click the endpoint URL to view its details in the **Edit Endpoint** dialog. The endpoint type should be `SAML Assertion Consume`. Set the **Binding** field for the type of SAML response to receive:

   - If the client expects a POST, set **Binding** to `POST`.
   - If the client expects to receive the SAML Response as a `GET` parameter, set **Binding** to `Redirect`.

> **Note:**
>
> There can be issues using a redirect in the case of long assertions because some browsers have limits to the length of the URL.

Integrating Microsoft Azure Active Directory with Oracle Cloud

As an example of adding a remote identity provider, here is what you do to enable use of Microsoft Azure Active Directory as the remote identity store for apps that use MCS mobile backends and which have users that have Oracle Cloud accounts.

The general sequence of steps is:

1. In Azure, create an application and configure it to use single sign-on.

   This application will provide the context for configuring the SSO relationship and identify the set of users to whom that relationship is applicable.
2. In Oracle Cloud, configure Azure Active Directory as the identity provider.
3. In your Azure app, add the Oracle Cloud service provider information.
4. In your Azure app, assign users to access the app.
5. In Oracle Cloud, import the Azure users.
6. In Oracle Cloud, enable the SSO configuration.
7. In MCS, enable SSO in a mobile backend.
8. Test the SSO with a mobile backend.

This procedure assumes that you have a Windows Azure account with Azure Active Directory Premium enabled.

Creating and Configuring the App in Azure that Will Serve as the Identity Store

The first step is to create an application in Azure and then configure that app to use single sign-on. This app doesn't have any end-user functionality.

1. Sign in to the Azure portal, browse to the directory you want to use, select **Applications**, and click **Add**.
2. Select **Add an application from the gallery**.
3. Select **Custom**, select **Add an unlisted application my organization is using**, provide a name, and save.
4. On the application page, click **Configure single sign-on**.
5. Select **Microsoft Azure AD Single Sign-On** and click **Next**.
6. On the **Configure App Settings** page, add values for **Issuer** and **Reply URL**.

   These values are just temporary placeholders, so just enter any syntactically correct URLs, such as `https://www.example.com`.

   You will add the real values later once you have set up your Oracle Cloud account to use Azure Active Directory as a remote identity provider.
7. On the **Configure single sign-on at ...** page, click **Download Metadata (XML)** and save the file as `IdP-Metadata.xml`.

   You will need this file to configure your Oracle Cloud account.
8. Check **Confirm that you have configured single sign-on as described above**.
9. In the next screen, confirm the notification email (optional) and save.

Configuring Azure Active Directory as the Identity Provider in Oracle Cloud

Now that you have set up the app in Azure to hold the identity store, you can configure your Oracle Cloud account to use it.

The configuration you do here will determine how an Oracle Cloud user record is identified from the information that Azure AD provides (via the SAML token).

1. Log in to Oracle Cloud, go to **Users** and then **SSO Configuration** and click **Configure SSO**.
2. In the popup window, select **Import identity provider metadata** and load the Azure metadata file (`IdP-Metadata.xml`) that you just downloaded from Azure.
3. From the **SSO Protocol** dropdown, select `HTTP POST`.
4. From the **User Identifier** dropdown, select one of the following to specify which field in the Oracle Cloud user record you will use to match with the Azure AD record.
   - **User's Email Address**
   - **User ID**
5. From the **Contained in** dropdown, select the attribute from Azure AD (such as user name or email address) that you want to be matched again the User Identifier value above.
6. Click **Save**.
7. Under **Configure your Identity Provider Information**, make a note of the **Provider ID** and **Assertion Consumer Service URL** values.

   You will use these values when configuring the Azure App to work with Oracle Cloud.
8. Click **Export Metadata** , select **Provider Metadata**, and save the file.

   This metadata may come in handy later if configuration problems arise.

Adding the Oracle Cloud Service Provider Information to the Azure App

In this step, you go back to Azure and fill in the Oracle Cloud service provider information that you just generated.

1. Go back to the Azure portal, and select your directory, then click **Applications** and then on the application created before.
2. Click **Configure single sign-on**.
3. Select **Microsoft Active Directory** again, and then click **Next**.
4. In the **Issuer** field, enter the value of the **Provider ID** that you copied after configuring Azure AD as an identity provider in Oracle Cloud.
5. In the **Reply URL** field, enter the value of the **Assertion Consumer Service URL** that you copied above.
6. For the next steps, continue with the defaults and then save at the end.

> **Note:**
>
> If you have problems with the **Issuer** and **Reply URL** values, you can double-check them in the metadata you exported after configuring Azure AD as the identity provider in Oracle Cloud. The **Provider ID** (and thus the **Issuer**) value should correspond with value of the `entityID` attribute of the `EntityDescriptor` element. The **Assertion Consumer Service URL** (and thus the **Reply URL**) value should correspond with the value of the `Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST"` attribute of the `AssertionConsumerService` element.

Assigning Azure Users to Access Your App

Next you populate your Azure app with the users that you want to be able to log in via the SSO feature.

1. In the Azure portal, navigate to your directory, click **Applications** and choose the container application you created.
2. Go to the **Users And Groups** tab, search for the groups you would like to be able to access MCS apps, and assign them by clicking the **Assign** button at the bottom of the page.

Importing Users Into Your MCS Realm

And now you import those users into MCS, via Oracle Cloud.

1. Export the users from Azure, using the recommended method, depending on the source of the users.
   - If the users originate from an on-premises Active Directory installation, use the standard Active Directory tools to export them.
   - If the users originate from Azure directly, use Azure Windows Power Tools.
2. Insert those users into a CSV file, with the following structure: First Name, Last Name, Email, User Login.

   The User Login must match the same username used to log-in to Azure.
3. Import the users into Oracle Cloud and assign them the realm that you want to use as described in Importing Groups of Mobile Users Into MCS Using Oracle Cloud.

Enabling the SSO Configuration

Once you have assigned users for your application in Azure AD, and have imported those users into Oracle Cloud, enable the SSO Configuration by following these steps:

1. On the **Single Sign-On (SSO) Configuration** page in Oracle Cloud, navigate to the **Test your SSO** section and click **Test**.
2. If that test is successful, navigate to the**Enable SSO** section of the page and click **Enable SSO**.

Testing the SSO with an MCS Mobile Backend

Once SSO has been fully configured and enabled and you have enabled SSO in a mobile backend, you can test it with that mobile backend.

If you haven't yet enabled SSO in a mobile backend, see Enabling Single Sign-On for a Mobile Backend.

To test SSO access to a mobile backend:

- Open a web browser and navigate to the following URL:

  `<environment URI>>/mobile/platform/sso/token?clientID=<OAuth client ID>`

  where `<environment URI>` is the URI used to access platform APIs for the given MCS instance, and `<OAuth client ID>` can be obtained from the **Settings** page for the given mobile backend.

H

Migrate to Oracle Mobile Hub

New features are being introduced into Mobile Hub and if you want to use those features you will need to migrate your Oracle Mobile Cloud configurations from MCS to Mobile Hub.

For more information, see Migrating to Oracle Mobile Hub.

Glossary

analytics

Enables mobile cloud administrators and mobile application developers to gauge the success of a mobile application, respond to user preferences, and explore business opportunities by observing patterns in performance and usage.

Analytics API

A platform API that allows mobile applications to collect custom events. Combined with system analytics gathered by MCS, this information can be used to provide insight into user engagement, usage patterns, and adoption rates of your mobile application.

anonymous access

Access to custom API endpoints without requiring users to log in. When allowing anonymous access, an app authenticates itself by passing a mobile backend ID and an anonymous access key instead of a user name and password.

anonymous key

A MCS-generated string used by a mobile application to get access (through HTTP Basic authentication) to APIs that don't require login.

anonymous user

A mobile user that can access custom APIs in MCS without logging in.

API

Application Program Interface. An interface description that provides the methods, return values, and other parameters that make it possible to be implemented in written code. In MCS, all APIs are RESTful APIs and their interface is described with a RAML document.

API key

A Google identifier you can use to access Google public APIs. MCS uses the Google Cloud Messaging system and the API key to send notifications to mobile applications running on Android devices.

APNS

Apple Push Notifications Service. The Apple network service used to deliver notifications from MCS to mobile applications running on iOS devices.

Apple Feedback Service

An Apple service that monitors failures in the notifications sending process. The Apple Feedback Service helps MCS keep its device registry up to date and ensures a higher rate of successful notifications deliveries on the Apple network.

application ID

A unique string approved by a platform vendor to identify a given mobile application. For Google, the application ID is the package name for the mobile application assigned in the manifest file. For Apple, the application ID is the bundle identifier assigned in the mobile application's Xcode project. For Microsoft, the application ID is the name you gave your app when you registered it in the Windows Dashboard.

application key

A unique ID issued by MCS to each mobile application registered in a mobile backend, used for tracking purposes in Diagnostics and Analytics.

artifact

An artifact in MCS can be a mobile backend, a custom API, a connector API, a collection, or a realm. An artifact exists in a specific environment and can be deployed and versioned.

collection

An MCS user-defined container that is used to store mobile application data on an MCS server. Data can be an image file, a text file, or a JSON payload. Using collections allows for efficient sharing of data between mobile users sharing the same mobile application, or by a single user sharing between several devices.

connector

A connector simplifies the task of connecting securely to a backend system, such as a database, or a system based on SOAP or REST. Connectors produce a class of APIs; you create an instance (using a wizard) and the result is a connector API.

connector API

An API produced as the result of using a connector. The API can then be invoked from custom code to simplify the task of exchanging data with a backend system, such as a database, or a system based on SOAP or REST.

correlation

A means of associating a given message with other messages logged for the same request. MCStags this message with a Correlation ID. This ID includes an ECID. By querying all of the log messages generated during that request, you can analyze what happened to the request, as well as any problems that it encountered.

correlation ID

An ID that associates a request received by the server with other logging data. The correlation ID appears in the Message Details view for a request and enables you to filter the logging data by the ECID.

custom API

Any API that a developer creates in the API Designer and implements in MCS using custom code they write themselves using those APIs.

custom event

A customer-defined analytic event generated by a mobile application to track a significant event occurrence that is of value to a business, such as login, checkout, or knowledge-based search.

Database API

A platform API that you can call from custom code to add, update, view, and delete rows in a database table.

Database Management API

A platform API that you can use to add, replace, and drop database tables as well as view the tables' metadata.

Data Offline and Sync API

A platform API that allows data to be locally available on a mobile device, by managing the efficient caching and network synchronization of data with MCS.

deploy

Once an artifact is in a published state, it can be deployed from one MCS environment to another. For example, a mobile backend that is in the published state can be deployed from a Development environment into a Production environment.

device handshake

The start-up registration process between a mobile app, the platform’s network service (such as GCM, FCM, or APNS), and MCS. The device handshake provides authentication to MCS that the mobile application is trusted by the network and helps facilitate the sending of notifications from MCS.

device registry

A mechanism instead of MCS that matches and keeps track of applications, users, and devices. The device registry does this automatically, which frees MCS developers from having to do it manually.

diagnostics

A feature that gathers and displays data relating to all traffic flowing through MCS, such as incoming mobile API requests, outgoing connector requests, and other data transactions. It also provides a way to view and control custom code log messages. Diagnostics is the primary mechanism for mobile cloud administrators to diagnose issues that may occur with the service.

Draft state

When an artifact is initially created in MCS, it is placed in a draft state. Only artifacts in a draft state can be modified. As part of the MCS lifecycle, when artifacts in a draft state are completed, they can be moved to a published state.

ECID

Execution Context ID. A globally unique identifier to correlate events or requests associated with the same transaction in the Oracle technology stack.

endpoint

One end of a communication channel, for example, a URI. An API endpoint is a noun (resource) and verb (HTTP method).

enterprise systems

On-premises or cloud-based processing systems that are integrated and extended by a mobile backend through the use of MCS connectors.

environment

The runtime containers where artifacts and metadata are developed and deployed in MCS. The default environments are Development, Staging, and Production.

ETag

The entity tag. A field value in an HTTP header that identifies a resource from a specific web service endpoint. The ETag's value represents the version of a resource. For example, to avoid overriding an object in the mobile application's cache, the Storage API uses the ETag value in the `If-Match` and `If-None-Match` fields for `GET` and `HEAD` operations.

funnel

Used in analytics, funnels provide a way to analyze sequences of events, to gain insight into user behavior and usage of a mobile application.

GCM

Google Cloud Messaging. A network delivery system provided by Google, which is being replaced by Google Firebase Messaging (FCM). It is used by MCS to deliver notifications to targeted users running mobile applications developed with MCS on Android devices. Both XMPP and HTTP are transport mechanisms supported by the Google Cloud Messaging system.

GUID

Globally Unique Identifier. The unique identifier that gets generated for every stored object in MCS. Any mobile user with the correct permissions can use this identifier to access the associated data object in a collection.

JSON

Javascript Object Notation. An open standard format that uses human-readable text to transmit data objects between a server and web application. Commonly used to attach a body of content to a REST—based API transaction.

lifecycle

The overall process of maintaining MCS artifacts and managing versions of those artifacts.

mobile app

An application resident on a mobile device. The mobile application can be an in-house application developed using MCS for distribution to internal company employees or an application released through a public store, such as the Google Play Store or the Apple App Store.

mobile backend

A secure container of APIs and other resources for a defined set of mobile apps. Mobile backend capabilities include platform APIs like storage, notifications, my profile management, and analytics, as well functionality described by custom APIs.

mobile backend ID

An identifier issued by MCS that allows a mobile application to access APIs associated with a mobile backend when authenticating with HTTP Basic authentication.

client SDK

A bundle of libraries, utilities, and wrapper classes that MCS provides for multiple mobile platforms to simplify use of MCS features in your apps.

mobile user

The user of a mobile application built using MCS. A mobile user is granted access to the mobile backend and platform APIs associated with the mobile application when it gets installed on a device and the user is authenticated with MCS.

mobile user management

APIs and other functionality that allows you to manage mobile users, roles, realms, and team members.

notification

A short, highly-tailored message sent to a specified set of recipients running a mobile application on a mobile device.

Notifications API

A platform API that provides the ability to send short, tailored messages to different groupings of recipients, immediately or on a schedule. Recipients can include everyone in a specific mobile backend, a user or set of users, a specific device or collection of devices, or a unique role.

platform APIs

APIs provided by MCS that simplify mobile app development. Features such as Analytics, Mobile Object Storage, Notifications, and Data Offline are platform APIs. Some platform APIs can be called by mobile applications, whereas others are called exclusively from custom code.

Published state

When an artifact is in a published state in MCS it can no longer be modified. As part of the MCS lifecycle, published artifacts can be deployed to a target environment.

RAML

RESTful API Modeling Language. A non-proprietary specification built on broadly used standards, which is used to describe the structure of practically RESTful APIs. It is capable of describing APIs that do not obey all the constraints of REST.

realm

A set of mobile users associated with a mobile backend. A realm exists within an environment and helps manage roles, properties, devices, and mobile users access to a mobile backend and associated platform APIs.

REST

Representational State Transfer. An internet software methodology used by MCS, which defines a set of APIs and a transport mechanism (HTTP) for interacting with those APIs.

REST connector

A type of connector API used to connect a mobile backend and its artifacts to a backend system based on REST.

RESTful

A description of a software implementation that uses REST APIs and conforms to the network transport, content body types, and other expectations of REST design.

RID

Relationship ID. An identifier that distinguishes the work done in one thread (and on one process) from work done by any other threads or processes, on behalf of the same request. The RID enables you to use log file entries to correlate messages from one application or across components.

role

A set of permissions granted to a team member or mobile user. The permissions may allow a mobile user access to a specific backend, or a team member access to certain functionality within MCS. Both team members and mobile users have roles, but the roles are different and not related to each other.

sender ID

A Google identifier used to identify a mobile application to the Google network. The sender ID is assigned by Google and used only when sending notifications from MCS to an Android application over the Google Cloud Messaging network, using XMPP.

SOAP

Simple Object Access Protocol. A protocol specification for exchanging structured information in the implementation of web services.

SOAP connector

A type of connector API used to connect a mobile backend and its artifacts to a backend system based on SOAP.

Storage API

A platform API that provides an easy-to-use, non-SQL, cloud-based mechanism for mobile applications to store and share text and binary data.

Synchronization API

The API for data offline and sync functionality in mobile applications. It is exposed as part of the MCS client SDKs and provides functionality to sync REST resources such as JSON payloads from custom code, and view and edit these resources while the mobile device is offline, as well as sync them back to the cloud when the device comes online.

system analytic data

Data gathered by MCS about runtime events received from mobile applications connected to mobile backends and from custom code.

team members

People who are authorized to log into MCS. These people include developers, administrators, enterprise architects, and others working with MCS.

user isolation

The ability to isolate objects within a storage collection by user. This enables a mobile application developer to create generic mobile applications without worrying about one user's storage conflicting with another's.

Mobile User Management API

A platform API that provides the ability to manage mobile users, roles, and realms.

versioning

A new copy of an artifact, that is unique and distinct from the original artifact. MCS has major versions of artifacts and minor versions. Minor versions are backward-compatible, major versions are not.

virtual user

A user that does not have a corresponding record in Oracle Cloud's Shared Identity Management (SIM). Virtual users are associated with a configured set of default roles based either on the content in the token or by the application of a configured set of roles.

XMPP

Extensible Messaging and Presence Protocol. An open source network messaging framework used by Google and other cloud network vendors that provides custom capabilities for messaging transmissions.

Zero Footprint SSO

A single sign-on (SSO) method that allows any mobile user that is able to authenticate with a trusted external identity provider to be able to access MCS services, subject to authorization configuration within the external identity provider, without requiring corresponding accounts for such users to have been provisioned within Oracle Cloud's Shared Identity Management (SIM).