

Oracle® Cloud

Developing Applications with Oracle Autonomous Mobile Cloud Enterprise



Release 18.2.5
E95337-06
September 2019



Copyright © 2018, 2019, Oracle and/or its affiliates. All rights reserved.

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

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, delivered to U.S. Government end users are "commercial computer software" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, shall be subject to license terms and license restrictions applicable to the programs. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

Contents

Preface

Audience	xx
Documentation Accessibility	xx
Conventions	xx

1 A Developer's View of Oracle Autonomous Mobile Cloud Enterprise

Client SDKs	1-1
Features for Your Apps	1-2
Custom APIs	1-2
Backends	1-2
Security	1-3

2 Android Apps

Getting the SDK	2-1
Creating a Backend	2-1
Adding the SDK	2-1
Configuring SDK Properties	2-2
Authentication Properties	2-4
OAuth	2-4
HTTP Basic	2-5
Token Exchange	2-5
Facebook Login	2-5
Configuring Your Android Manifest File	2-6
Calling Mobile APIs	2-7
Loading the Backend's Configuration	2-7
Authenticating and Logging In	2-7
Calling Platform APIs	2-12
Calling Custom APIs	2-13
Libraries and Dependencies	2-13
Next Steps	2-14

3 iOS Applications

Getting the SDK	3-1
Creating a Backend	3-1
Adding the SDK	3-1
Configuring SDK Properties	3-2
Authentication Properties	3-4
OAuth	3-4
HTTP Basic	3-5
Token Exchange	3-5
Calling Mobile APIs	3-6
Loading the Backend's Configuration	3-6
Authenticating and Logging In	3-6
Calling Platform APIs	3-8
Calling Custom APIs	3-9
Libraries and Dependencies	3-10
Next Steps	3-11

4 Cordova Applications

Getting the SDK	4-1
Creating a Backend	4-1
Adding the SDK	4-1
Adding Support for Push Notifications	4-2
Configuring SDK Properties	4-3
Authentication Properties	4-5
OAuth	4-5
HTTP Basic	4-6
Token Exchange	4-6
Facebook Login	4-7
Calling Mobile APIs	4-7
Loading the Backend's Configuration	4-8
Authenticating and Logging In	4-8
Securing Browser-Based Apps Against Cross-Site Request Forgery Attacks	4-9
Calling Platform APIs	4-9
Calling Custom APIs	4-9
Using TypeScript	4-10
Libraries	4-13
Next Steps	4-13

5 JavaScript Applications

Getting the SDK	5-1
Creating a Backend	5-1
Adding the SDK	5-1
Configuring SDK Properties	5-2
Authentication Properties	5-3
OAuth	5-4
HTTP Basic	5-4
Token Exchange	5-5
Facebook Login	5-5
Calling Mobile APIs	5-6
Loading the Backend's Configuration	5-6
Authenticating and Logging In	5-6
Securing Browser-Based Apps Against Cross-Site Request Forgery Attacks	5-7
Calling Platform APIs	5-7
Calling Custom APIs	5-8
Using TypeScript	5-8
Libraries	5-11
Next Steps	5-12

6 Notifications

What Can I Do with Notifications?	6-1
Setting Up a Mobile App for Notifications	6-1
Setting Up the Device Handshake for Notifications	6-3
Setting Up a Device Handshake for Android (FCM)	6-3
Setting Up a Device Handshake for iOS	6-5
Setting Up a Device Handshake for Windows	6-7
Sending Notifications to and from Your App	6-7
Testing Notifications from the UI	6-7
Cancelling a Scheduled Notification from the UI	6-8
Sending Notifications Using the Notifications API	6-8
REST	6-10
iOS	6-15
Cordova/JavaScript/TypeScript	6-18
How Are Notifications Sent and Received?	6-19
What is the Device ID or Notification Token?	6-20
Troubleshooting Notifications	6-20
Checking Notification Status in the UI	6-21

7 My Profile

User Types	7-1
Getting User Profile Information	7-1

8 Storage

What Can I Do with Storage?	8-1
Android	8-1
Adding an Object to a Collection	8-1
Fetching an Object	8-1
Getting Multiple Objects from a Collection	8-2
Getting a Shared Collection	8-2
Retrieving an Object	8-2
Updating an Object	8-3
Uploading a New Object to a Collection	8-3
iOS	8-3
Adding an Object to a Collection	8-3
Deleting an Object	8-4
Downloading Data to a Collection	8-4
Getting a User Isolated Collection	8-5
Getting Multiple Objects from a Collection	8-5
Getting Object Data as a Stream	8-5
Retrieving a Storage Object	8-6
Updating an Object	8-6
Uploading Data to a Collection	8-6
Cordova, JavaScript, and TypeScript	8-7
Adding an Object to a Collection	8-7
Deleting an Object	8-8
Fetching an Object	8-8
Getting a Collection	8-8
Getting an Object from a User Isolated Collection	8-9
Getting Multiple Objects from a Collection	8-9
Updating an Object	8-9
Custom Code	8-10
Retrieving and Storing Collections and Objects	8-10
REST API	8-10
Storage API Endpoints	8-10
Getting a Single Collection	8-10

Getting All Collections Associated with a Mobile Backend	8-11
Storing an Object	8-11
Specifying the Object Identifier	8-11
Creating an Object (If One Doesn't Already Exist)	8-11
Generating an Object Identifier	8-12
What Happens When an Object is Created?	8-12
Updating an Object	8-12
What Happens When an Object Is Updated?	8-12
Optimistic Locking	8-12
Retrieving a List of Objects	8-13
Paging Through a List of Objects	8-13
Ordering	8-14
Querying	8-14
Retrieving an Object	8-14
Deleting an Object	8-15
Optimizing Performance	8-15
Check If Exists	8-15
Get If Newer	8-16
Reading Part of an Object (Chunking Data)	8-16
Testing Runtime Operations Using the Endpoints Page	8-17
Managing Collections	8-17
Shared and User Isolated Collections	8-17
Storage Configuration	8-20
Defining a Collection	8-21
Collection Metadata	8-22
Adding Access Permissions to a Collection	8-22
Adding Objects to a Collection	8-24
Object Metadata	8-24
Updating the Collection	8-25
Offline Data Storage	8-25
Associating a Collection with a Backend	8-26
Removing a Collection from a Backend	8-27

9 Data Offline and Sync

Building Apps that Work Offline Using Sync Express	9-2
Building Apps that Work Offline Using the Synchronization Library	9-6
What Can I Do with the Synchronization Library?	9-6
Synchronization Library Process Flow	9-9
Video: Overview of the Data Offline & Synchronization API	9-9
Android Synchronization Library	9-9

Setting Up Your Mobile App for the Android Synchronization Library	9-10
Fetching Resources	9-10
Fetching Filtered Resources	9-13
Specifying Which Resources to Synchronize First	9-17
Setting a Resource's Synchronization Policies Programmatically	9-17
Detecting and Handling Conflicts	9-18
Reviewing and Discarding Offline Edits	9-21
iOS Synchronization Library	9-25
Setting Up Your Mobile App for the iOS Synchronization Library	9-25
Fetching Resources	9-25
Fetching Filtered Resources	9-27
Specifying Which Resources To Synchronize First	9-29
Setting a Resource's Synchronization Policies Programmatically	9-30
Detecting and Handling Conflicts	9-31
Reviewing and Discarding Offline Edits	9-34
Making Custom APIs Synchronizable	9-37
Synchronization Policies	9-38
Video: Introduction to the Data Offline & Sync Policies	9-40
Synchronization Policy Options	9-40
Video: Deep-Dive into the Data Offline & Sync Policies	9-43
Synchronization Policy Levels and Precedence	9-43
Defining Synchronization Policies Using a Configuration File	9-44
Defining Synchronization Policies and Cache Settings in a Response Header	9-50
Tracking Cache Hits with the Synchronization Library	9-51
How Synchronization Works with the Storage APIs	9-51

10 Location

What Can I Do With Location?	10-1
A Few Important Location Terms	10-1
Android	10-2
Querying for Location Objects	10-2
Retrieving a Location Object	10-3
iOS	10-5
Querying for Location Objects	10-5
Retrieving a Location Object	10-6
Retrieving iBeacon Identifiers	10-7
Defining a Geofence	10-8
Retrieving Custom Attributes	10-9
REST API - Location	10-10
Querying for Location Devices, Places and Assets	10-10

Querying for Location Devices	10-10
Querying for Places	10-14
Querying for Assets	10-18
Retrieving Location Objects and Properties	10-20
Setting Up Location Devices, Places and Assets	10-21
Defining Places	10-21
Uploading Places Using a CSV File	10-21
Defining Location Assets	10-22
Uploading Assets Using a CSV File	10-23
Registering Location Devices	10-24
Uploading Location Devices Using a CSV File	10-25

11 Database

What Can I Do with Database APIs?	11-1
Database Access API	11-1
Calling the Database Access API from Custom Code	11-1
Creating and Restructuring Database Tables	11-3
Adding and Updating Table Rows	11-5
Retrieving Table Rows	11-6
Deleting Table Rows	11-8
Executing SQL on a Table	11-9
Passing Parameters to the SQL Statement	11-10
Labeling Calculated Columns in Select Statements	11-12
Preserving Case in SQL Statements	11-12
Preventing SQL Injection	11-13
Database Management API	11-14
Creating a Table Explicitly	11-14

12 App Policies

What Are App Policies and What Can I Do With Them?	12-1
Setting an App Policy	12-1
Android	12-2
Retrieving App Policies	12-2
iOS	12-3
Retrieving App Policies	12-3
REST	12-4
Retrieving App Policies	12-4
Cordova	12-4
Retrieving App Policies	12-4

JavaScript	12-5
Retrieving App Policies	12-5
Updating an App Policy Value in a Published Mobile Backend	12-5

13 Backends

What Are Backends and How Can I Use Them?	13-1
What's the Backend Development Process?	13-1
Creating and Populating Backends	13-1
Creating a Backend	13-2
Backend Authentication and Connection Info	13-2
Role-Based Backends	13-3
Associating APIs with a Backend	13-3
Associating Storage Collections with a Backend	13-4
Notification Profiles and Client Apps	13-4
How Notification Profiles Work	13-4
How App Clients Work	13-4
Getting Network Credentials for Notifications	13-5
Android: Google API Key	13-5
iOS: Apple Secure Certificates	13-6
Windows: WNS Credentials	13-7
Syniverse: SMS Credentials	13-7
Creating a Notifications Profile	13-9
Registering an App Client	13-10
What Can I Change in a Backend?	13-11
Connecting Your App to a Backend	13-11

14 Mobile Users and Roles

Navigate to Your Oracle Identity Cloud Service Application	14-1
Adding Users and Groups in Oracle Identity Cloud Service	14-1
Creating and Managing Mobile Roles	14-2
Permissions Required for Platform APIs	14-3

15 Authentication in AMCe

OAuth Consumer Authentication in AMCe	15-1
HTTP Basic Authentication in AMCe	15-2
Enterprise Single Sign-On in AMCe	15-2
Third-Party SAML and JWT Tokens	15-3
SAML Tokens and Virtual Users	15-3
JWT Tokens and Virtual Users	15-7

Mapping Users from a Third-Party IdP to IDCS Users	15-24
Browser-Based SSO through AMCe	15-25
Testing APIs in a Backend with SSO Login	15-25
Token Expiration for SSO Login	15-26
Facebook Login in AMCe	15-26
Registering an App for Login Through Facebook	15-27
Enabling Facebook Login in a Mobile Backend	15-27
Configuring an App to Use Facebook Login	15-27
Adding APIs to a Mobile Backend with Facebook Login	15-28
Getting a Facebook User Access Token Manually	15-28
Headers Needed for API Calls with Facebook Authentication	15-29
Authenticating in Direct REST Calls	15-29
Authenticating with OAuth in Direct REST Calls	15-29
Authenticating with HTTP Basic in Direct REST Calls	15-31
Securing Cross-Site Requests to AMCe APIs	15-31

16 Creating APIs Fast with the Express API Designer

What are Resources?	16-1
Creating An API	16-1
Completing Your Resources	16-3
Adding Additional Fields	16-4
Shaping the Payload for Your Resource	16-4
Adding More Sample Data	16-5
Referenced Resources	16-6
Fields	16-8
Methods	16-9
Shaping Payloads	16-10
Read-Only Fields	16-12
Sample Data	16-12
Using the Express API Designer with MAX	16-15
How Do I Surface My API in MAX?	16-15
Who Uses MAX?	16-15
Enabling Uploadable Images	16-16
Tips for User-Friendly Business Objects in MAX	16-17
Video: An Introduction to Mobile Application Accelerator (MAX)	16-27
Creating Resources with JSON Schemas	16-27
Defining Fields in a Schema	16-28
Defining Field Types, Formats, and Enums	16-29
Defining Child Objects	16-31
Defining Fields for List, Details, Create, and Update Screens	16-32

Collection Actions	16-34
Create Actions	16-37
Update Actions	16-39
Delete Actions	16-40
Custom Actions	16-40
Creating Mock Data	16-40
Which API Designer Should I Use?	16-41

17 Custom API Design

API Design Process	17-1
Generating Custom APIs for Connectors	17-3
How Do I Generate a Custom API from a Connector	17-4
Completing the Custom API	17-6
Working with the Implementation	17-6
The API Designer	17-9
Spec Out a Custom API	17-9
Creating a Complete Custom API	17-14
Setting Up Your API	17-14
Defining Endpoints	17-15
Adding Methods to Your Resources	17-17
Defining a Request for the Method	17-18
Defining a Response for the Method	17-19
Testing API Endpoints Using Mock Data	17-21
Providing a Schema	17-22
Security in Custom APIs	17-24
Setting Access to the API	17-24
Testing Your Custom API	17-26
Creating Resource Types	17-28
Creating Resource Traits	17-30
Providing API Documentation	17-31
How Do I Write in Markdown?	17-33
Getting Diagnostic Information	17-34
API Design Considerations	17-34
Valid URLs	17-34
API Timeouts	17-36
API Resources	17-36
URI Parameters	17-37
Endpoint Requirements for Sync Compatibility	17-38
Schemas	17-39
RAML	17-40

Editing a Custom API	17-43
Video: End-to-End Custom API Demo	17-44
Troubleshooting Custom APIs	17-44

18 Implementing Custom APIs

What Can I Do with Custom Code?	18-1
How Does Custom Code Work?	18-2
What's the Foundation for the Custom Code Service?	18-2
Video: Node.js Technology Primer	18-4
Setting Up Tooling for Custom Code	18-4
Steps to Implement a Custom API	18-4
Downloading a JavaScript Scaffold for a Custom API	18-5
Writing Custom Code	18-5
Key JavaScript Constructs in Custom Code	18-5
Accessing the Body of the Request	18-9
Inserting Logging Into Custom Code	18-9
Storing Data Locally	18-11
Video: Working with Node - Common Code	18-11
Implementing Synchronization-Compatible APIs	18-11
Video: Working with Custom APIs via Data Offline & Sync	18-12
Requirements for a Synchronization-Compatible Custom API	18-12
Returning Cacheable Data	18-17
Specifying Synchronization and Cache Policies	18-19
Calling Web Services and APIs from Custom Code	18-20
Packaging Custom Code into a Module	18-21
Required Artifacts for an API Implementation	18-21
package.json Contents	18-22
Declaring the API Implementation Version	18-23
Declaring the Node Version	18-24
Packaging Additional Libraries with Your Implementation	18-24
Uploading the Custom Code Module	18-25
Testing and Debugging Custom Code	18-25
Testing with Mock Data	18-25
Testing Custom Code from the UI	18-26
Offline Debugging with the AMCe Custom Code Test Tools	18-26
Tools for Testing Custom Code Outside the UI	18-27
Accessing Logging Messages for Custom Code	18-27
Troubleshooting Custom API Implementations	18-30
Diagnosing Syntax Errors	18-31
Common Custom Code Errors	18-32

19 Calling APIs from Custom Code

How to Send Requests to Platform, Connector, and Custom APIs	19-1
API Request Pattern	19-1
Common options Argument Properties	19-2
API Response Patterns	19-5
Handling a Stream	19-5
Handling a Promise	19-6
Accessing Mobile Backend Information from Custom Code	19-13
mbe.getMBE()	19-14
Calling Platform APIs from Custom Code	19-14
Accessing the Analytics Collector API from Custom Code	19-15
analytics.postEvent(events, options, httpOptions)	19-15
Accessing the App Policies API from Custom Code	19-18
appConfig.getProperties(httpOptions)	19-19
Accessing the Database Access API from Custom Code	19-19
database.delete(table, keys, options, httpOptions)	19-20
database.get(table, keys, options, httpOptions)	19-21
database.getAll(table, options, httpOptions)	19-23
database.insert(table, object, options, httpOptions)	19-25
database.merge(table, object, options, httpOptions)	19-30
database.sql(sql, args, options, httpOptions)	19-35
Accessing the Location API from Custom Code	19-37
location.assets.getAsset(id, httpOptions)	19-38
location.assets.query(queryObject, httpOptions)	19-40
location.devices.getDevice(id, httpOptions)	19-43
location.devices.query(queryObject, httpOptions)	19-45
location.places.getPlace(id, httpOptions)	19-47
location.places.query(queryObject, httpOptions)	19-48
Accessing the Location Management API from Custom Code	19-52
location.assets.register(assets, context, httpOptions)	19-53
location.assets.remove(id, context, httpOptions)	19-54
location.assets.update(id, asset, context, httpOptions)	19-56
location.devices.register(devices, context, httpOptions)	19-58
location.devices.remove(id, context, httpOptions)	19-60
location.devices.update(id, device, context, httpOptions)	19-61
location.places.register(places, context, httpOptions)	19-63
location.places.remove(id, context, httpOptions)	19-65
location.places.removeCascade(id, context, httpOptions)	19-66

location.places.update(id, place, context, httpOptions)	19-67
Accessing the Mobile Devices API from Custom Code	19-69
devices.deregister(device, httpOptions)	19-69
devices.register(device, httpOptions)	19-70
Accessing the My Profile API from Custom Code	19-71
ums.getMe(httpOptions)	19-71
ums.getUser(options, httpOptions)	19-74
ums.getUserExtended(options, httpOptions)	19-76
ums.updateUser(fields, options, httpOptions)	19-78
Accessing the Notifications API from Custom Code	19-79
notification.getAll(context, options, httpOptions)	19-80
notification.getById(id, context, options, httpOptions)	19-83
notification.post(notification, context, options, httpOptions)	19-84
notification.remove(id, context, options, httpOptions)	19-86
Accessing the Storage API from Custom Code	19-87
storage.doesCollectionExist(collectionId, options, httpOptions)	19-87
storage.doesExist(collectionId, objectId, options, httpOptions)	19-88
storage.getAll(collectionId, options, httpOptions)	19-91
storage.getById(collectionId, objectId, options, httpOptions)	19-95
storage.getCollection(collectionId, options, httpOptions)	19-99
storage.getCollections(options, httpOptions)	19-101
storage.remove(collectionId, objectId, options, httpOptions)	19-104
storage.store(collectionId, object, options, httpOptions)	19-106
storage.storeById(collectionId, objectId, object, options, httpOptions)	19-109
Calling Connector APIs from Custom Code	19-112
Calling a Connector to a REST Web Service	19-116
Calling a Connector to a SOAP Service	19-117
Calling Connectors that Require Form Data	19-118
Passing Headers to the Target Service	19-120
Overriding SSL Settings for Connectors	19-121
Calling Custom APIs from Custom Code	19-122
Specifying the API Version in Calls to Custom and Connector APIs	19-124
Using Generic REST Methods to Access APIs	19-125
optionsList Argument	19-127
Learning About Platform, Custom, and Connector APIs	19-128

20 Connectors

What Is a Connector API?	20-1
REST Connector APIs	20-1
How REST Connector APIs Work	20-1

Why Use Connectors Instead of Direct Calls to External Resources?	20-1
Creating a REST Connector API	20-2
Basic Connector Setup	20-2
Rules	20-4
Security Policies and Overriding Properties	20-5
Testing in Advanced Mode	20-7
Security and REST Connector APIs	20-10
Security Policy Types for REST Connector APIs	20-11
CSF Keys and Web Service Certificates	20-12
Query and Header Parameters	20-13
Setting Query Parameters in Remote URLs	20-14
Editing a REST Connector API	20-15
Using Your REST Connector API in an App	20-15
Troubleshooting REST Connector APIs	20-16
SOAP Connector APIs	20-16
How SOAP Connector APIs Work	20-16
Why Use SOAP Connectors Instead of Direct Calls to External Resources?	20-17
Creating a SOAP Connector API	20-17
Setting the Basic Information for Your SOAP Connector API	20-18
Selecting a Port	20-21
Setting Security Policies and Overriding Properties for SOAP Connector APIs	20-22
Testing a SOAP Connector API	20-24
SOAP Connector API Design Tips	20-27
How Does XML Get Translated into JSON?	20-28
Using XML Instead of JSON	20-29
Security Policy Types for SOAP Connector APIs	20-30
CSF Keys and Web Service Certificates	20-31
Editing a SOAP Connector API	20-32
Using Your Connector API in an App	20-32
Troubleshooting SOAP Connector APIs	20-33
ICS Connector APIs	20-34
How ICS Connector APIs Work	20-34
ICS Connector API Flow	20-35
How Do I Create an ICS Connector API?	20-36
Setting the Basic Information for Your ICS Connector API	20-37
Connecting to an Integration Cloud Service Instance	20-39
Selecting or Creating an ICS Instance Connection	20-39
Selecting an Active Integration	20-40
Editing the ICS Connector API	20-41
Setting Runtime Security for the ICS Connector API	20-42

Creating a New CSF Key	20-43
Testing the ICS Connector API	20-43
Security and ICS Connector APIs	20-47
CSF Keys	20-47
Using Your Connector API in an App	20-48
Troubleshooting ICS Connector APIs	20-48
Fusion Applications Connector APIs	20-49
How Fusion Applications Connector APIs Work	20-49
Fusion Applications Connector API Flow	20-50
How Do I Create a Fusion Applications Connector API?	20-51
Setting the Basic Information for Your Fusion Applications Connector API	20-52
Connecting to a Fusion Applications Instance	20-54
Creating a Fusion Applications Instance Connection	20-54
Selecting Fusion Applications Resources	20-55
Setting Resource Attributes	20-57
Editing the Fusion Applications Connector API	20-58
Setting Runtime Security for the Fusion Applications Connector API	20-59
Testing the Fusion Applications Connector API	20-61
Security Policy Types for Fusion Applications Connector APIs	20-63
CSF Keys and Web Service Certificates	20-64
Using Your Fusion Application Connector API in an App	20-65
Troubleshooting Fusion Applications Connector APIs	20-66

21 Diagnostics

What Can I Do with Diagnostics?	21-1
Viewing Environment Health	21-1
Viewing Server Load	21-2
Viewing Errors	21-2
Viewing Underperforming Requests	21-2
Viewing Log Messages Related to a Request	21-2
Viewing Storage Usage	21-3
Monitoring a Selected Backend	21-3
Viewing API Performance	21-4
Adjusting the Performance Threshold Configurations	21-4
Viewing Status Codes for API Calls and Outbound Connector Calls	21-5
Relating Log Messages	21-7
How Client SDK Headers Enable Device and Session Diagnostics	21-8
Viewing Log Messages	21-8
Viewing Message Details	21-10
Taking a Look at Exported Messages	21-11

Configuring the Logging Level for Custom Code	21-18
Diagnosing Custom Code	21-18
Use Case: Using Correlation to Diagnose Custom Code	21-19
Use Case: Using Correlation to Diagnose Connector Issues	21-21

22 Packages

What's a Package?	22-1
Why Do I Want a Package?	22-1
Exporting a Package	22-2
Adding Artifacts to the Package	22-2
Reviewing Dependencies During Export	22-3
Setting Environment Policies During Export	22-4
Completing the Export	22-6
Re-exporting a Package	22-7
Importing a Package	22-7
Uploading the Package	22-7
Examining the Contents of the Import Package	22-8
Setting Environment Policies During Import	22-9
What Happens When You Import a Package?	22-11
Import Results	22-11
Exporting Updated Artifacts	22-12
Examining a Package	22-12
Moving a Package to the Trash	22-13
Environment Policy Settings for Packaged Artifacts	22-14

A HTTP Headers

API Headers	A-1
SDK Headers	A-2

B Oracle Autonomous Mobile Cloud Enterprise Policies

AMCe Policies and Values	B-1
--------------------------	-----

C Security Policies for Connector APIs

Security Policies for REST Connector APIs	C-1
Security Policies for SOAP Connector APIs	C-3
Security Policies for ICS Connector APIs	C-11
Security Policies for Fusion Applications Connector APIs	C-11

D Writing Swift Applications Using the iOS Client SDK

Adding the Bridging Header File	D-1
Adding the SDK Headers and Libraries to a Swift App	D-2
Using SDK Objects in Swift Apps	D-3

E Supported Browsers and Languages

Supported Browsers	E-1
Supported Languages	E-1

F Identity Provider Integration

Use Case: Configuring OKTA to Obtain a SAML Token	F-1
Use Case: Configuring AD FS to Obtain a SAML Token	F-2

Preface

Welcome to Oracle Autonomous Mobile Cloud Enterprise.

Audience

This guide is intended for developers who use Oracle Autonomous Mobile Cloud Enterprise (AMCe) to develop mobile applications, mobile APIs, and intelligent chatbots.

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.

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.
<i>italic</i>	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
<code>monospace</code>	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1

A Developer's View of Oracle Autonomous Mobile Cloud Enterprise

Welcome to Oracle Autonomous Mobile Cloud Enterprise! AMCe is a cloud-based service that provides a unified hub for developing, deploying, maintaining, monitoring, and analyzing your mobile apps, intelligent bots, and the resources that they rely on. As a developer, you can use this guide to help you incorporate AMCe's features into your apps.

This guide covers what you need to know to:

- Add support for authentication, notifications, geo-location, sync, storage, and other features to your apps.
- Create custom APIs that your apps and bots can use to connect to your enterprise systems.

This guide doesn't cover development of bots themselves. For that, see *Creating Intelligent Bots with Oracle Autonomous Mobile Cloud Enterprise*.

There is also a separate guide for using AMCe's Mobile Application Accelerator (MAX) to rapidly develop cross-platform mobile apps. See *Building No-Code Applications with Oracle Autonomous Mobile Cloud Enterprise*.

Client SDKs

As an app developer, the first thing you'll want to do is get the AMCe client SDK for your mobile platform. The client SDKs help you use AMCe features and custom APIs that you develop through AMCe in your apps.

You can get the SDKs from the Oracle Technology Network's [Oracle Autonomous Mobile Cloud Enterprise download page](#).

For specific info on each SDK, see the following topics:

- [Android Apps](#)
- [iOS Applications](#)
- [Cordova Applications](#)
- [JavaScript Applications](#)

Note:

For information on using the REST APIs directly, see the [platform's REST API reference docs](#).

Features for Your Apps

AMCe comes with a set of platform APIs that you can use in your apps. You can call these APIs directly from your app code (via client SDK or REST call) and/or from the implementation code of custom APIs.

The available platform APIs include the following:

- **Notifications** for writing code to send notifications to your mobile apps.
- **My Profile** to retrieve the current app user's profile.
- **Storage** to work with collections and objects (such as images and documents) that you associate with your backend.
- **Data Offline and Sync** 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.
- **Location** to define location devices and places and query for them from your mobile apps.
- **Database Access** and **Database Management** to access the database associated with your AMCe instance.
- **App Policies** to retrieve application configuration properties that you have set in the backend.

Custom APIs

You can create your own custom APIs in AMCe to serve the needs of your apps and bots.

You **design** the API using one or both of the following tools:

- **Express API Designer**: Enables you to quickly create APIs based on CRUD resources.
- **API Designer**: Enables you to create or modify an API using the full set of RAML capabilities.

You **implement** the API with Node.js-based custom code. From your implementation code, you can:

- Call AMCe platform APIs for features such as notifications, object storage, database access, location, and sync.
- Access enterprise services (whether cloud-based or on premises) via connectors APIs. You can create connector APIs based on connector types for REST, SOAP, Oracle Integration Cloud Service, and Fusion Applications.

Backends

You'll use *backends* to group the APIs and other resources that your apps and bots need.

A backend is a logical grouping of custom APIs, storage collections, and other resources that serves as a cloud-based companion to one or more related mobile

apps or bots. The backend provides the security context, meaning that the user has to authenticate through the backend to access those services.

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

- Add APIs and test their endpoints with mock data.
- Add object storage collections and enable offline data caching.
- Specify roles that users must have to access the applications associated with the backend.
- Set up notifications for your apps using the services provided by the platform vendors (such as Apple Push Notifications Service (APNS) for iOS and Firebase Cloud Messaging (FCM) for Android). 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 backend serves as a deployment unit with dependency management for all of the artifacts you need to support the set of mobile apps and bots that use it.

Security

For each backend that you create, you set up how to *authenticate* with that backend. You can choose from these mechanisms:

- **OAuth**, where Oracle Identity Cloud Service (IDCS) is the identity provider, and you use credentials generated by the backend.
- **HTTP Basic**, where IDCS is the identity provider, and you use credentials generated by the backend.
- **SAML and JWT tokens** from third-party identity providers.
- **Browser-based SSO**, where IDCS is the identity provider and the app uses OAuth 2.0's authorization code grant to get an authorization token.
- **Facebook Login**.

Further, you can restrict access to resources by defining *roles* in AMCe and applying them to backends, APIs, and/or storage collections.

2

Android Apps

Oracle Autonomous Mobile Cloud Enterprise (AMCe) provides an SDK for Android that makes it easy to use AMCe's features.

Getting the SDK

To get the AMCe client SDK for Android, go to the [Oracle Autonomous Mobile Cloud Enterprise Downloads page](#) on OTN.

Creating a Backend

You create a backend to serve as a secure gateway between your app and AMCe features, such as platform and custom APIs. For your app to access these resources, it authenticates with a backend.

1. Click  to open the side menu and select **Development > Backends**.
2. Click **New Backend**.
3. Once you complete the dialog and the backend is created, keep the Settings page open.

You'll need to configure your app with some of this information.

Adding the SDK

Assuming a basic app setup, without intervening frameworks, here's what you would do to add the Android client SDK to an app

1. If you haven't already done so, unzip the Android client 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. Import the `IDMMobileSDK.jar` into the project. (In Android Studio, select **File > New > New Module**, click **Import .JAR/.AAR Package**, click **Next**, select `IDMMobileSDK.jar` and click **Next**.)
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 'com.squareup.okhttp3:okhttp:3.9.0'  
    compile 'org.slf4j:slf4j-jdk14:1.7.13'  
    //to enable the app to receive notifications, include the following:  
    compile 'com.google.firebaseio:messaging:11.0.2'  
}
```

7. Open assets/oracle_mobile_cloud_config.xml and fill in the environment details for the mobile backend that the app will be using.

Configuring SDK Properties

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

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 sample GettingStarted app, it's in /GettingStarted/src/main/assets.

The file is essentially divided into the following parts:

- The mobileBackend element and its contents.
You include this part if you are using a backend with the app. The SDK classes use the environment and authentication details you specify there to access the backend and construct HTTP headers for REST calls made to APIs.
- Elements that apply to the configuration as a whole, such as logLevel and oAuthTokenEndpoint. These elements generally, but don't have to, appear at the top of the file.

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

```
<config>  
  
    <!--This value is required if you are using OAuth to authenticate  
    against the mobile backend-->  
    <oAuthTokenEndPoint>YOUR_OAUTH_TOKEN_END_POINT</oAuthTokenEndPoint>  
    <!--Set to true if you want to get logging information-->  
    <enableLogger>true</enableLogger>  
    <logLevel>DEBUG</logLevel>  
    <!--Whether to log HTTP call request and response bodies and headers-->  
    <logHTTP>true</logHTTP>  
  
    <!-- Include the mobileBackend element and its sub-elements if you are  
    going  
        to be using a backend to access custom and platform APIs.-->  
    <mobileBackend>  
        <name>MBE_NAME</name>
```

```
<baseUrl>BASE_URL</baseUrl>
<enableAnalytics>true</enableAnalytics>
<authentication>
    <!--possible values for type are [oauth, basic, facebook,
tokenExchange ]-->
    <type>AUTH_TYPE</type>
    <oauth>
        <clientId>CLIENT_ID</clientId>
        <clientSecret>CLIENT_SECRET</clientSecret>
        <enableOffline>true</enableOffline>
    </oauth>
    <basic>
        <mobileBackendId>MOBILE_BACKEND_ID</mobileBackendID>
        <anonymousKey>ANONYMOUS_KEY</anonymousKey>
        <enableOffline></enableOffline>
    </basic>
    <facebook>
        <appId>FACEBOOK_APP_ID</appId>

<scopes>public_profile,user_friends,email,user_location,user_birthday</
scopes>
    <basic>
        <mobileBackendId>MOBILE_BACKEND_ID</mobileBackendID>
        <anonymousKey>ANONYMOUS_KEY</anonymousKey>
    </basic>
    </facebook>
    <tokenExchange>
        <! tokenExchange can contain an 'oauth' sub-element or a 'basic'
sub-element.
        <oauth>
            <clientId>CLIENT_ID</clientId>
            <clientSecret>CLIENT_SECRET</clientSecret>
        </oauth>
        <basic>
            <mobileBackendId>MOBILE_BACKEND_ID</mobileBackendID>
            <anonymousKey>ANONYMOUS_KEY</anonymousKey>
        </basic>
        <tokenExchange>
        </authentication>
        <!-- additional properties go here -->
    </mobileBackend>

</config>
```

The values that you need to fill in for a given backend can be found on the Settings and App Profile pages for that mobile backend.

Here are some more notes on the file's elements.

- `oAuthEndPoint` — The URL of the OAuth server from where your application gets its authentication token. This key needs to be provided for all apps that rely on OAuth to authenticate. You get this from the backend's **Settings** page.
- `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 specify `NONE`.

- `enableLogger` — When set to `true`, logging is included in your app.
- `logHTTP` — When set to `true`, the SDK logs the HTTP and HTTPS headers in requests and responses.
- `mobileBackend` — An element containing authentication details for your backend and other optional details, such as synchronization properties.

You get the authentication details, such as the OAuth and HTTP credentials, from the backend's **Settings** page.

- `mobileBackend/baseUrl` — The base URL for all APIs that you call through the backend. You get this from the backend's **Settings** page.
- `mobileBackend/authentication` — Contains the following sub-elements:
 - The `type` sub-element, with possible values of `oauth`, `basic`, `facebook`, and `tokenExchange`.
 - One or more sub-elements for authentication types, each containing authentication credentials.

You can also add the `offlineEnabled` key and set its value to `true`.

See [Authentication Properties](#) for examples of each authentication type.

- `enableOffline` — If set to `true`, offline login will be allowed. This applies only to the Basic and OAuth login types. 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>
```

Authentication Properties

The contents and sub-elements of `authentication` depend on what kind of authentication the app will be using.

OAuth

- Set the value of the `<type>` element to `oauth`.
- Fill in the `clientID` and `clientSecret` credentials provided by the backend.
- Optionally, if you want to disable offline authentication, add the `enableOffline` element and set it to `false`.
- At the top level of the file, supply the `oAuthAccessTokenEndPoint` value.

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

```
<oAuthAccessTokenEndPoint>http://oam-server.oracle.com/oam/oauth2/tokens</
oAuthAccessTokenEndPoint>

<authentication>
    <type>oauth</type>
    <oauth>
```

```
<clientId>f2d3ca5c-7e6f-4d1c-aabc-a2f3caf7ec4e</clientId>
<clientSecret>vZMRkgniIbhNUiPnSRT2</clientSecret>
<enableOffline>false</enableOffline>
</oauth>
</authentication>
```

HTTP Basic

- Set the value of the `type` element to `basic`.
- Fill in the `mobileBackendID` and `anonymousKey` that are provided by the backend.
- Optionally, if you want to disable offline authentication, add the `enableOffline` sub-element and set it to `false`.

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

```
<authentication>
  <type>basic</type>
  <basic>
    <mobileBackendID>6d3744b8-cab2-479c-998b-ebba2c31560f</mobileBackendID>
    <anonymousKey>UFJJTUVfREVDRVBUSUNPTl9NT0JJTEVfQU5PTll</anonymousKey>
    <enableOffline>false</enableOffline>
  </basic>
</authentication>
```

Token Exchange

If you are authenticating using a third-party token, do the following:

- Set the value of the `<type>` element to `tokenExchange`.
- Create a `<basic>` sub-element and fill in the OAuth Consumer credentials provided by the backend.

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

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

Facebook Login

For Facebook login:

- Set the value of the `<type>` property to `facebook`.
- Create a `<facebook>` sub-element.
- Fill in the `<appID>` for the Facebook app.

- Fill in <scopes> with any relevant Facebook permissions (optional).
- Within <facebook>, created a <basic> element and fill in the HTTP Basic credentials provided by the backend.

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

```
<authentication>
    <type>facebook</type>
    <facebook>
        <basic>
            <mobileBackendId>MOBILE_BACKEND_ID</mobileBackendId>
            <anonymousKey>ANONYMOUS_KEY</anonymousKey>
        </basic>
        <appId>123456789012345</appId>
        <scopes>public_profile,user_friends,email,user_location,user_birthday</scopes>
    </facebook>
<authentication>
```

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, there are a number of permissions necessary for the app to be able to receive notifications. For a rundown on the available permissions, 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.sample" >
    <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>
        <receiver android:name="oracle.cloud.mobile.network.NetworkHelper" >
```

```
<intent-filter>
    <action android:name="android.net.conn.CONNECTIVITY_CHANGE" />
</intent-filter>
</receiver>

(.....)
</application>
</manifest>
```

Adding the client 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](#).

Calling Mobile APIs

In AMCe, a backend is a logical grouping of custom APIs, storage collections, and other resources that you can use in your apps. The backend also provides the security context for accessing those resources.

Here are the general steps for using a backend in your Android app:

1. Add the client SDK to your app.
2. Fill in the `oracle_mobile_cloud_config.xml` with environment and authentication details for the backend.
3. Add an SDK call to your app to load the configuration info.
4. Add an SDK call to your app to handle authentication.
5. Add any other SDK calls that you want to use.

Loading the Backend's Configuration

For any calls to AMCe APIs using the Android client SDK to successfully complete, you need to have the backend's configuration loaded from the app's `oracle_mobile_cloud_config.xml` file. You do this using the `MobileManager` class:

```
MobileManager.getManager().getMobileBackend(this)
```

Authenticating and Logging In

Here is some sample code that you can use for authentication through AMCe 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;

try {
```

```
    mobileBackend = MobileManager.getManager().getMobileBackend(this);
} catch (ServiceProxyException e) {
    e.printStackTrace();
}

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");
        }
    }
}
```

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 AMCE, 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();

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

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

Then you use the `authenticateUsingTokenExchange` method to attempt authentication.

```
mAuthorization.authenticateUsingTokenExchange(mCtx, token, false,  
mLoginCallback);
```

Here's the callback:

```
AuthorizationCallback mLoginCallback = new AuthorizationCallback() {  
    @Override  
    public void onCompletion(ServiceProxyException exception) {  
        if (exception == null) {  
            //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();  
        }  
    }  
};
```

Note:

The default expiration time for storing a third-party token in AMCe is 6 hours. You can adjust this time by changing the `Security_TokenExchangeTimeoutSecs` policy. See Modifying Policies in *Managing Oracle Autonomous Mobile Cloud Enterprise*.

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 =
```

```
MobileManager.getManager().getMobileBackend(this).getAuthorization();
    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);
```

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;

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

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:

```
SocialAuthorizationAgent mAuthorization;
SocialMobileBackend socialMobileBackend;
try {
    socialMobileBackend =
SocialMobileBackendManager.getManager().getMobileBackend(mCtx);
} catch(ServiceProxyException e){
    e.printStackTrace();
}
mAuthorization = socialMobileBackend.getSocialAuthorization();
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

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

The root object in the Android SDK is `MobileManager`. The `MobileManager` object manages `MobileBackend` objects.

The `MobileBackend` object manages connectivity, authentication, and other transactions between your application and its associated backend, including calls to platform APIs and any custom APIs you have defined. It manages calls to platform APIs through instances of `ServiceProxy` such as `Storage` and `Location`.

Here's an example of how you would use these classes to upload an image using the `Storage` API:

```
try {  
    Storage storage =  
    MobileManager.getManager().getMobileBackend(this).getServiceProxy(Storage.c  
lass);  
    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();  
    ...  
}
```

The `ServiceProxy` instance created there manages calls to the `Storage` platform API, including the constructing of the HTTP headers with the mobile backend credentials necessary to access the API.

And here's how you could retrieve an image using the `Storage` API:

```
try {  
    Storage storage =  
    MobileManager.getManager().getMobileBackend(this).getServiceProxy(Storage.c  
lass);  
    StorageCollection imagesCollection =  
    storage.getStorageCollection("FIF_Images");  
    StorageObject image = imagesCollection.get("3x4mple-st0r4g3-0bj3ct-  
    k3y");byte[] imageBytes = image.getPayloadBytes();  
} catch(ServiceProxyException e) {int errorCode = e.getErrorCode();  
    ...  
}
```

Calling Custom APIs

The client SDK provides the `CustomHttpResponse` class, the `GenericCustomCodeClientCallBack` interface, and the `invokeCustomCodeJSONRequest` method in the authorization classes to simplify the calling of custom APIs in AMCe. 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.MobileManager;  
.....  
  
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 =  
MobileManager.getManager().getMobileBackend(this).getAuthorization();  
  
authorization.authenticate(mActivity, "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);
```

Libraries and Dependencies

Libraries

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

- `omce-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.
- `IDMMobileSDK.jar` - The identity management library used by all applications.
- `omce-android-sdk-location-<version-number>` - The Location library, which lets you access details about location devices that have been registered in AMCe and the places and assets they are associated with.
- `omce-android-sdk-notifications-<version-number>.jar` - The Notifications library, which lets you set up your application to receive notifications sent from your mobile backend.
- `omce-android-sdk-social-<version-number>` - The Social Login library, which allows you to set up your app to use Facebook login.
- `omce-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.
- `omce-android-sdk-sync-<version-number>` - 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.

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 AMCe 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.

Next Steps

Once you have the Android SDK set up, you can start using it to add AMCe features to your app.

- [Authentication in AMCe](#)
- [Notifications](#)
- [My Profile](#)
- [Storage](#)
- [Data Offline and Sync](#)
- [Location](#)
- [Database](#)
- [App Policies](#)

3

iOS Applications

If you are an iOS app developer, you can use the client SDK that Oracle Autonomous Mobile Cloud Enterprise (AMCe) provides for iOS. This SDK simplifies authentication with AMCe and provides Objective-C wrapper classes for AMCe platform APIs.

Getting the SDK

To get the AMCe client SDK for iOS, go to the [Oracle Autonomous Mobile Cloud Enterprise Downloads page](#) on OTN.

Creating a Backend

You create a backend to serve as a secure gateway between your app and AMCe features, such as platform and custom APIs. For your app to access these resources, it authenticates with a backend.

1. Click  to open the side menu and select **Development > Backends**.
2. Click **New Backend**.
3. Once you complete the dialog and the backend is created, keep the Settings page open.

You'll need to configure your app with some of this information.

Adding the SDK

Assuming a basic app setup, without intervening frameworks, here's what you would do to add the iOS client SDK to an app.

1. Unzip the download file, `omce-ios-sdk-{n}.zip` (where {n} is the version number of the SDK) into some directory on your machine.
2. From the extracted contents of the zip, drag and drop the `oracle_mobile_ios_sdk` directory 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.

3. Select the target for your project, select the **Build Phases** tab, expand **Link Binary with Libraries**, click the **+** button, and add the following libraries:

- CoreData.framework
 - CoreLocation.framework
 - libsqlite3.0.tbd
 - Security.framework
 - SystemConfiguration.framework
4. In the **Build Settings** section for the project, double-click **Other Linker Flags** (under **Linking**) and add -ObjC.
5. Also in **Build Settings**, expand **Search Paths** and:
- a. Add oracle_mobile_ios_sdk/release-iphoneos to **Library Search Paths**.
 - b. Add oracle_mobile_ios_sdk/release-iphoneos/include to **User Header Search Paths**.
6. 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.
7. Edit the just-copied OMC.plist file. See [Configuring SDK Properties](#).
8. Starting with Xcode 7, 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.

Configuring SDK Properties

To use the client 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 backend in AMCe, as well as other configuration information. In turn, the SDK classes use this information to help manage authorization, logging, event tracking, data synchronization, and other features.

You package the configuration file in the root of your app's main bundle.

The file is essentially divided into the following parts:

- The mobileBackend key and its contents.

You include this part if you are using a backend with the app. The SDK classes use the environment and authentication details you specify there to access the backend and construct HTTP headers for REST calls made to APIs.

- Keys that apply to the configuration as a whole, such as `logLevel` and `oAuthTokenEndpoint`. These keys generally, but don't have to, appear at the top of the file.

Here's the same file in text form:

```
<?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>logLevel</key>
    <string>debug</string>
    <key>logHTTP</key>
    <true/>
    <key>oAuthTokenEndPoint</key>
    <string>https://MY_TOKEN_ENDPOINT</string>
    <key>mobileBackend</key>
    <dict>
        <key>name</key>
        <string>EasyShoppingMBE</string>
        <key>baseURL</key>
        <string>https://MY_CLOUD_DOMAIN.oracle.com</string>
        <key>authentication</key>
        <dict>
            <key>type</key>
            <string>oauth</string>
            <key>oauth</key>
            <dict>
                <key>clientID</key>
                <string>11dac238ffaa4b029e78e982114642ab</string>
                <key>clientSecret</key>
                <string>5624cbdd-a7c5-4c10-a758-6019a5ab8da8</string>
                <key>enableOffline</key>
                <true/>
            </dict>
        </dict>
    </dict>
</dict>
</plist>
```

And here is a description of some of the more important entries in the `OMC.plist` file.

- `oAuthTokenEndPoint` — The URL of the OAuth server from where your application gets its authentication token. This key needs to be provided for all apps that rely on OAuth to authenticate. You get this from the backend's **Settings** page. The endpoint should be only the base URL (in the form `https://host.domain:port`).
- `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`.

- `logHTTP` — When set to `true`, the SDK logs the headers and bodies of all HTTP requests and responses.
- `mobileBackend` — A dictionary entry containing authentication details for your backend and other optional details, such as synchronization properties.

You get the authentication details, such as the OAuth and HTTP credentials, from the backend's **Settings** page.
- `mobileBackend/baseUrl` — The base URL for all APIs that you call through the backend. You get this from the backend's **Settings** page.
- `mobileBackend/authentication` — Contains a dictionary with the following elements:
 - The `type` sub-key, with possible `(string)` values of `oauth`, `basic`, `facebook`, and `tokenExchange`.
 - One or more sub-keys for authentication types, containing a dictionary with the authentication credentials.

Within sub-keys for `basic` and `oauth`, you can also add the `enableOffline` key. By default, this property is set to `true`.

See [Authentication Properties](#) for examples of each authentication type.

Authentication Properties

The contents and sub-elements of the `mobileBackend/authentication` key depend on what kind of authentication the app will be using.

OAuth

- Set the value of the `type` key to `oauth`.
- Create an `oauth` sub-key and fill in the `clientID` and `clientSecret` credentials provided by the backend.
- At the top level of the file, supply the `oAuthTokenEndPoint` value that is supplied but *without* the `oauth2/v1/token` that is appended on the backend's **Settings** page.
- Optionally, if you want to disable offline authentication, add the `enableOffline` sub-key and set it to `false`.

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

```
<key>authentication</key>
<dict>
    <key>type</key>
    <string>oauth</string>
    <key>oauth</key>
    <dict>
        <key>clientID</key>
        <string>11dac238ffaa4b029e78e982114642ab</string>
        <key>clientSecret</key>
        <string>5624cbdd-a7c5-4c10-a758-6019a5ab8da8</string>
    </dict>
</dict>
```

```
<key>basic</key>
</dict>
```

HTTP Basic

- Set the value of the `type` key to `basic`.
- Create a `basic` sub-key and fill in the HTTP Basic credentials (`mobileBackendID` and `anonymousKey`) provided by the backend.
- Optionally, if you want to disable offline authentication, add the `enableOffline` sub-key and set it to `false`.

The resulting authentication entry might look something like this:

```
<key>authentication</key>
<dict>
  <key>type</key>
  <string>basic</string>
  <key>basic</key>
  <dict>
    <key>mobileBackendID</key>
    <string>a8c6a34f-61bb-4bee-948c-d43dd2c077d7</string>
    <key>anonymousKey</key>
    <string>dXNlcmlkOnBhc3N3b3Jk</string>
  </dict>
</dict>
```

Token Exchange

If you are authenticating using a third-party token, do the following:

- Set the value of the `type` key to `tokenExchange`.
- Create a `tokenExchange` sub-key and fill in the OAuth Consumer credentials provided by the backend.

The resulting authentication section might look something like this:

```
<key>authentication</key>
<dict>
  <key>type</key>
  <string>tokenExchange</string>
  <key>tokenExchange</key>
  <dict>
    <key>oauth</key>
    <dict>
      <key>clientID</key>
      <string>b39ba08d30d54e24970332fcdfec3a7</string>
      <key>clientSecret</key>
      <string>23953fe8-76ed-4c89-a5cb-6042db10cfaf</string>
    </dict>
    <key>basic</key>
    <dict>
      <key>mobileBackendID</key>
      <string>8d3744b8-cab2-479c-998b-ebba2c31560f</string>
    </dict>
  </dict>
</dict>
```

```
<key>anonymousKey</key>
<string>ZFJJTUVfREVDRVBUSUNPTl9NT0JJTEVfQU5PTll</string>
</dict>
</dict>
</dict>
```

Calling Mobile APIs

In AMCe, a backend is a logical grouping of custom APIs, storage collections, and other resources that you can use in your apps. The backend also provides the security context for accessing those resources.

Here are the general steps for using a backend in your iOS app:

1. Add the client SDK to your app.
2. Fill in the `OMC.plist` with environment and authentication details for the backend.
3. Add an SDK call to your app to load the configuration info.
4. Add an SDK call to your app to handle authentication.
5. Add any other SDK calls that you want to use.

Loading the Backend's Configuration

For any calls to AMCe APIs using the iOS client 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 that is configured in OMC.plist file
 */
OMCMobileBackend* mbe = [[OMCMobileManager sharedManager] mobileBackend];
```

Authenticating and Logging In

Here is some sample code that you can use for authentication through AMCe in your iOS apps. Each sample is based on the `OMCAuthorization.h` class and relies on the following imports:

```
#import "OMCCore/OMCAuthorization.h"
#import "OMCCore/OMCMobileBackend.h"
#import "OMCCore/OMCMobileManager.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: (nullable OMCErrorCompletionBlock) completionBlock;
```

This method terminates the connection to AMCe and clears the user name and password from the iOS keychain:

```
- (void) logout: (nullable OMCErrorCompletionBlock) 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 AMCe, see [Third-Party SAML and JWT Tokens](#).

Once you have the token, use it to authenticate. The code in this example checks to see if the token is already stored in AMCe before logging in again:

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

Note:

The default expiration time for storing a third-party token in AMCe is 6 hours. You can adjust this time by changing the `Security_TokenExchangeTimeoutSecs` policy. See [Modifying Policies in Managing Oracle Autonomous Mobile Cloud Enterprise](#).

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 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.

Calling Platform APIs

Once the backend's configuration info is loaded into the app, you can make calls to client SDK classes based on the iOS Core library classes.

The iOS Core library (`libOMCCore.a`) provides the following key interfaces:

- `OMCMobileManager`
- `OMCMobileBackend` (a sub-class of `OMCMobileComponent`)
- `OMCServiceProxy`

The root object in the SDK is the `OMCMobileManager`. The `OMCMobileManager` manages the `OMCMobileBackend` objects.

The `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. It manages calls to platform APIs via subclasses of `OMCServiceProxy` such as `OMCLocation` and `OMCStorage`.

Here's an example of using SDK classes to call the Storage API:

```
#import "OMCMobileBackend.h"
#import "OMCMobileManager.h"
#import "OMCAuthorization.h"
#import "OMCStorage.h"
#import "OMCMobileBackend+OMC_Storage.h"
#import "OMCSynchronization.h"

- (NSData*)dataFromStorageObjectWithID:(NSString*)objectID collectionID:
(NSString*)collectionID {

    // Get mobile backend
    OMCMobileBackend* mbe = [[OMCMobileManager sharedManager] mobileBackend];

    // Get storage object
    OMCStorage* storage = [mbe storage];

    // Get your collection
    OMCStorageCollection* collection = [storage getCollection:collectionID];
```

```
// Get your object from your collection
OMCStorageObject* object = [collection get:objectID];

// Get the data from payload of your object
NSData* data = [object getPayloadData];

return data; }

}
```

 **Note:**

Methods written in Objective-C that are used in the AMCe SDK for iOS can also be mapped to Swift. For more information, see [Writing Swift Applications Using the iOS Client SDK](#).

Calling Custom APIs

The client SDK provides the `OMCCustomCodeClient` class to simplify the calling of custom APIs in AMCe.

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 request 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 AMCe 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 = [[OMCMobileManager sharedManager]
mobileBackend];
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
}];
```

Libraries and Dependencies

Libraries

The iOS client 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.Oracle-Mobile-Cloud-Enterprise-iOS-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'll need to add to your app and populate with the configuration details for your mobile backend.
- `oracle_mobile_ios_sdk/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.
 - `libOMCLocation.a` - The Location library, which lets you access details about location devices that have been registered in AMCe 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.

Dependencies

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 `libbOMCSynchronization.a`.
- If your app uses `libbOMCSynchronization.a`, you must also include the `SyncStore.momd` folder, which contains initialization data.

Next Steps

Once you have the iOS SDK set up, you can start using it to add AMCe features to your app.

- [Authentication in AMCe](#)
- [Notifications](#)
- [My Profile](#)
- [Storage](#)
- [Data Offline and Sync](#)
- [Location](#)
- [Database](#)
- [App Policies](#)

4

Cordova Applications

If you develop hybrid apps based on the Apache Cordova framework, you can use the client SDK that Oracle Autonomous Mobile Cloud Enterprise (AMCe) provides for Cordova. This SDK simplifies authentication with AMCe and provides Cordova wrapper classes for AMCe 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

To get the AMCe client SDK for Cordova, go to the [Oracle Autonomous Mobile Cloud Enterprise Downloads page](#) on OTN.

Creating a Backend

You create a backend to serve as a secure gateway between your app and AMCe features, such as platform and custom APIs. For your app to access these resources, it authenticates with a backend.

1. Click  to open the side menu and select **Development > Backends**.
2. Click **New Backend**.
3. Once you complete the dialog and the backend is created, keep the Settings page open.

You'll need to configure your app with some of this information.

Adding the SDK

Assuming a basic app setup, without intervening frameworks, here's what you would do to add the Cordova client SDK to an app:

1. If you haven't already done so, unzip the Cordova SDK zip.
2. Copy `mcs.js` (and/or `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`.
4. Add script tags for the SDK and the configuration file in your app's `index.html` file:

```
<script src="lib/mcs/mcs.js"></script>
<script src="app/oracle_mobile_cloud_config.js"></script>
```

5. If you will be using notifications in your apps, install the `oracle-mcs-notifications-cordova-plugin` plugin:

```
cordova plugin add PATH_TO_UNZIPPED_SDK/oracle-mcs-notifications-cordova-plugin
```

6. (Optional) For RequireJS environments, load `mcs.js` in your app using RequireJS.

 **Note:**

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. For detailed instructions on adding Sync Express to your app, see [Building Apps that Work Offline Using Sync Express](#).

Adding Support for Push Notifications

If you want to use push notifications in an app, these additional steps are required.

1. (For Android) Register your app for notifications on the Firebase Cloud Messaging (FCM) console. See [Set Up a Firebase Cloud Messaging Client App on Android](#) on Google's developer site.

When you generate the configuration file for your app, make sure you choose to enable the Cloud Messaging service.

When generation is complete, the **Project Number** (aka **Sender ID**) and **API Key** are displayed. These credentials 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 AMCe.

2. (For Android) Download the generated Firebase configuration file and put it in the root of your project.
3. (For Android) If you haven't already done so, install the notifications plugin that is supplied with the SDK:

```
cordova plugin add PATH_TO_UNZIPPED_SDK/oracle-mcs-notifications-cordova-plugin
```

4. (For iOS) Set up the app for notifications with APNS. See [iOS: Apple Secure Certificates](#)
5. Create the app in AMCe and notifications profiles for Android and iOS. See [Creating a Notifications Profile](#).

6. In your app code, register for notifications:

```
...
document.addEventListener("deviceready", handleDeviceReady, false);
...
function handleDeviceReady(){
    MCSNotificationsCordovaPlugin.onTokenRefresh(handleTokenRefresh,
handleError);
}
...
function handleTokenRefresh(token){
    console.log('NotificationsService Token refreshed', token);
    mcs.mobileBackend.notifications.registerForNotifications(token,
packageName, appVersion, 'FCM')
        .then(handleRegisterForNotifications)
        .catch(handleError);
}

function handleRegisterForNotifications(response){
    console.log('NotificationsService, device registered for
notifications');
}
function handleError(error){
    console.error('NotificationsService Error', error);
}
```

7. In your app code, subscribe to notifications events:

```
...
function handleDeviceReady(){

    MCSNotificationsCordovaPlugin.onMessageReceived(handleMessageReceived,
handleError);
}
...
function handleMessageReceived(data){
    console.log('NotificationsService Message received', data);
}

function handleError(error){
    console.error('NotificationsService Error', error);
}
```

Configuring SDK Properties

To use the client 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 backend in AMCe. In turn, the SDK classes use this information to construct HTTP headers for REST calls made to AMCe.

 **Note:**

If any of your apps will be browser-based, you need to manage cross-origin resource sharing (CORS) for access to AMCe 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 file is essentially divided into the following parts:

- Properties that apply to the configuration as a whole, such as `logLevel` and `oAuthTokenEndpoint`. These keys generally, but don't have to, appear at the top of the file.
- The `mobileBackend` property and its contents.

You include this part if you are using a backend with the app. The SDK classes use the environment and authentication details you specify there to access the backend and construct HTTP headers for REST calls made to APIs.

The following example shows the structure of a generic `oracle_mobile_cloud_config.js` file:

```
var mcs_config = {
    "logLevel": mcs.LOG_LEVEL.NONE,
    "logHTTP": true,
    "oAuthTokenEndPoint": "OAUTH_BASE_URL",
    "mobileBackend": {
        "name": "NAME",
        "baseUrl": "BASE_URL",
        "authentication": {
            "type": mcs.AUTHENTICATION_TYPES.oauth,
            "oauth": {
                "clientId": "CLIENT_ID",
                "clientSecret": "CLIENT_SECRET"
            }
        },
        "syncExpress": {
            "handler": "OracleRestHandler",
            "policies": [
                {
                    "path": '/mobile/custom/firstApi/tasks/:id(\d+)?',
                },
                {
                    "path": '/mobile/custom/secondApi/tasks/:id(\d+)?',
                }
            ]
        }
    };
};
```

Here are some notes on the file's elements.

- `oAuthTokenEndpoint` — The URL of the OAuth server from where your application gets its authentication token. This key needs to be provided for all apps that rely

on OAuth to authenticate. You get this from the backend's **Settings** page. The endpoint should be only the base URL (in the form `https://host.domain:port`).

- `logLevel` — Determines how much SDK logging is displayed in the app's console. The default value is `mcs.LOG_LEVEL.INFO` (only important events are logged). Other possible values are `mcs.LOG_LEVEL.NONE`, `mcs.LOG_LEVEL.ERROR` (only errors are logged) or `mcs.LOG_LEVEL.VERBOSE`.
- `enableLogger` — When set to `true`, logging is included in your app.
- `logHTTP` — When set to `true`, the SDK logs the HTTP and HTTPS headers in requests and responses.
- `mobileBackend` — An element containing authentication details for your backend and other optional details, such as synchronization properties.

You get the authentication details, such as the OAuth and HTTP credentials, from the backend's **Settings** page.

- `mobileBackend/baseUrl` — The base URL for all APIs that you call through the backend. You get this from the backend's **Settings** page.
- `mobileBackend/authentication` — Contains the following sub-elements:
 - The `type` sub-element, with possible values of `mcs.AUTHENTICATION_TYPES.oauth`, `basic`, `facebook`, or `token`.
 - One or more sub-elements containing authentication credentials.
 - (Optional) You can add the `offlineEnabled` key and set its value to `true`.

See [Authentication Properties](#) for details and examples of each authentication type.

For details on sync elements, see [Building Apps that Work Offline Using Sync Express](#).

Authentication Properties

The contents and sub-elements of `authentication` depend on what kind of authentication the app will be using.

OAuth

- Set the value of the `type` property to `mcs.AUTHENTICATION_TYPES.oauth`.
- At the same level as the `type` property, create a property called `oauth` and fill in the `clientID` and `clientSecret` credentials provided by the backend.
- At the top level of the file, supply the `oAuthTokenEndPoint` value that is supplied but *without* the `oauth2/v1/token` that is appended on the backend's **Settings** page.

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

```
var mcs_config = {
  ...
  "oAuthTokenEndPoint": "BASE_OAUTH_URL_WITH_oauth2/v1/token_REMOVED",
  "mobileBackend": {
    "name": "NAME",
    "baseUrl": "BASE_URL",
```

```
        "authentication": {
            "type": mcs.AUTHENTICATION_TYPES.oauth,
            "oauth": {
                "clientId": "CLIENT_ID",
                "clientSecret": "CLIENT_SECRET"
            }
        }
    };
}
```

HTTP Basic

- Set the value of the `type` property to `mcs.AUTHENTICATION_TYPES.basic`.
- At the same level as the `type` property, create a property called `basic` and fill in the `mobileBackendID` and `anonymousKey` that are provided by the backend.

The resulting entries might look something like this:

```
var mcs_config = {
    ...
    "mobileBackend": {
        "name": "NAME",
        "baseUrl": "BASE_URL",
        "authentication": {
            "type": mcs.AUTHENTICATION_TYPES.basic,
            "basic": {
                "mobileBackendId": "MOBILE_BACKEND_ID",
                "anonymousKey": "ANONYMOUS_KEY"
            }
        }
    }
};
```

Token Exchange

If you are authenticating using a third-party token, do the following:

- Set the value of the `type` property to `mcs.AUTHENTICATION_TYPES.token`.
- Fill in the `mobileBackendId` and `anonymousKey` that are provided by the backend.

The resulting properties might look something like this:

```
var mcs_config = {
    ...
    "mobileBackend": {
        "name": "NAME",
        "baseUrl": "BASE_URL",
        "authentication": {
            "type": mcs.AUTHENTICATION_TYPES.token,
            "token": {
                "mobileBackendId": "YOUR_BACKEND_ID",
                "anonymousKey": "ANONYMOUS_KEY"
            }
        }
    }
};
```

```
        }  
    }  
};
```

Facebook Login

- Set the value of the `type` property to `mcs.AUTHENTICATION_TYPES.facebook`.
- Fill in the HTTP Basic auth credentials and/or the OAuth credentials provided by the backend.
- Fill in the `appId` for the Facebook app.
- Fill in the relevant `scopes`.

The resulting authentication entry might look something like this:

```
var mcs_config = {  
    ...  
    "mobileBackend": {  
        "name": "NAME",  
        "baseUrl": "BASE_URL",  
        "authentication": {  
            "type": mcs.AUTHENTICATION_TYPES.facebook,  
            "facebook": {  
                "appId": "YOUR_FACEBOOK_APP_ID",  
                "mobileBackendId": "YOUR_BACKEND_ID",  
                "anonymousKey": "YOUR_ANONYMOUS_KEY",  
                "scopes":  
                    "public_profile,user_friends,email,user_location,user_birthday"  
                }  
            }  
        }  
    };
```

Calling Mobile APIs

In AMCe, a backend is a logical grouping of custom APIs, storage collections, and other resources that you can use in your apps. The backend also provides the security context for accessing those resources.

Here are the general steps for using a backend in your Cordova app:

1. Add the client SDK to your app.
2. Fill in the `oracle_mobile_cloud_config.js` with environment and authentication details for the backend.
3. Add an SDK call to your app to load the configuration info.
4. Add an SDK call to your app to handle authentication.
5. Add any other SDK calls that you want to use.

Loading the Backend's Configuration

Before you can make calls to AMCe APIs using the Cordova client SDK, you need to load the configuration for the backend you are going to use. In the following snippet, `mcs_config` is the name of the configuration that is defined in the `oracle_mobile_cloud_config.js` file that you have added to your app.

```
mcs.init(mcs_config);
```

Authenticating and Logging In

Here are some examples of using the Cordova client SDK's `Authorization` class. These examples assume you already configured the SDK config file for the type of authentication you're using, as described in [Configuring SDK Properties](#).

OAuth and HTTP Basic

Set the authentication type for the backend to `oauth` (or `basic`):

```
mcs.mobileBackend.setAuthenticationType(mcs.AUTHENTICATION_TYPES.oauth);
```

Then add a function that calls `Authorization.authenticate` on the backend, passes it a user name and specifies callbacks for success and failure:

```
mcs.mobileBackend.authorization.authenticate(username,  
password).then(callback).catch(errorCallback);
```

If you want to use anonymous authentication, the method to call is `authenticateAnonymous`:

```
mcs.mobileBackend.authorization.authenticateAnonymous().then(callback).catc  
h(errorCallback);
```

SSO with a Third-Party Token

To use SSO with a third-party token, your app first 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 AMCe, see [Third-Party SAML and JWT Tokens](#).

Set the authentication type for the backend to `token` and then pass the token in the authorization call:

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

Facebook

Set the authentication type for the backend to facebook and then call authenticate():

```
mcs.mobileBackend.setAuthenticationType(mcs.AUTHENTICATION_TYPES.facebook);
mcs.mobileBackend.authorization.authenticate().then(callback).catch(errorCallback);
```

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 AMCe 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 AMCe 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

Once you include the Cordova client SDK libraries in your application, and adjust configuration settings, you're ready to use the SDK classes in your apps.

Here's an example of how you could use these classes to get an object from a Storage collection in the mobile backend:

```
mcs.mobileBackend.storage.getCollection(<collection id>)
  .then(function(collection){
    return collection.getObject(<object id>, 'blob');
  })
  .then(function(object){
    console.log(object);
  })
  .catch(function(response){
    console.error(response);
  })
```

Calling Custom APIs

The Cordova client SDK provides the `CustomCode` class to simplify the calling of custom APIs. 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.mobileBackend.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.  
});
```

Using TypeScript

It is also possible to use TypeScript objects with the Cordova and JavaScript client SDKs.

Here are some basic steps and examples for using TypeScript with the SDK. The examples assume your app is using the Ionic framework (though you can also use TypeScript without it).

Setting Up the SDK

1. Install the SDK in your project by running this command in your project folder:

```
npm install {path to unzipped SDK location}
```

2. Add import statements to your service to import SDK types:

```
import {IMCS} from 'mcs'
```

3. Create the configuration file for the app:

```
import {IMCS,
        IOracleMobileCloudConfig,
        IMobileBackendConfig,
        IAuthenticationConfig,
        IBasicAuthConfig,
        IOAuthConfig,
        import * as mcssdk from 'mcs'
const mcs: IMCS = mcssdk;

export const mcsConfig: IOracleMobileCloudConfig = {
    logLevel: mcs.LOG_LEVEL.NONE,
    logHTTP: true,
    oAuthTokenEndPoint: 'OAUTH_URL',
    mobileBackend: <IMobileBackendConfig>{
        name: 'NAME',
        baseUrl: 'BASE_URL',
        authentication: <IAuthenticationConfig>{
            type: mcs.AUTHENTICATION_TYPES.basic,
            basic: <IBasicAuthConfig>{
                mobileBackendId: 'MOBILE_BACKEND_ID',
                anonymousKey: 'ANONYMOUS_KEY'
            }
        }
    }
};
```

4. Import the configuration into the app. If the above file is called `mcs-config.ts`, the import would look like :

```
import { mcsConfig } from "../mcs-config";
```

Calling Mobile APIs

1. Add these import statements to your service or component:

```
import {IMCS} from 'mcs';
import * as mcssdk from 'mcs'; And in your class add declaration
statement:
```

2. Add the declaration statement in your class:

```
export class ComponentClass{
    mcs: IMCS = mcssdk;
}
```

3. Initialize the SDK library with a configuration:

```
this.mcs.init(mcsConfig);
```

4. Call backend functionality:

```
this.mcs.mobileBackend.setAuthenticationType(this.mcs.AUTHENTICATION_TYPE.basic);
this.mcs.mobileBackend.authorization.authenticate(username, password);
```

Adding Support for Location Services (Ionic Only)

```
ionic cordova plugin add cordova-plugin-geolocation
```

Adding Support for Push Notifications (Ionic only)

1. (For Android) Register your app for notifications on the Firebase Cloud Messaging (FCM) console. See [Set Up a Firebase Cloud Messaging Client App on Android](#) on Google's developer site.

When you generate the configuration file for your app, make sure you choose to enable the Cloud Messaging service.

When generation is complete, the **Project Number** (aka **Sender ID**) and **API Key** are displayed. These credentials 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 AMCe.

2. (For Android) Download the generated Firebase configuration file and put it in the root of your project.
3. (For Android) If you haven't already done so, install the notifications plugin that is supplied with the SDK:

```
cordova plugin add PATH_TO_UNZIPPED_SDK/oracle-mcs-notifications-
cordova-plugin
```

4. (For iOS) Set up the app for notifications with APNS. See [iOS: Apple Secure Certificates](#)
5. Create the app in AMCe and notifications profiles for Android and iOS. See [Creating a Notifications Profile](#).
6. In your app code, register for notifications:

```
...
MCSNotificationsCordovaPlugin.onTokenRefresh(this.handleTokenRefresh.bind(this), this.handleError.bind(this));
...
handleTokenRefresh(token: string){
    console.log('NotificationsService Token refreshed', token);

    this.mcs.mobileBackend.notifications.registerForNotifications(token,
    packageName, appVersion, 'FCM')
        .then(this.handleRegisterForNotifications.bind(this))
        .catch(this.handleError.bind(this));
}

handleRegisterForNotifications(response: INetworkResponse){
    console.log('NotificationsService, device registered for
    notifications');
}
```

```
        }
        handleError(error: any){
            console.error('NotificationsService Error', error);
        }
    }
```

7. In your app code, subscribe to notifications events:

```
...
MCSNotificationsCordovaPlugin.onMessageReceived(this.handleMessageReceived.bind(this), this.handleError.bind(this));
...
handleMessageReceived(data: any){
    console.log('NotificationsService Message received', data);
}
handleError(error: any){
    console.error('NotificationsService Error', error);
}
```

Libraries

The Cordova client SDK includes the following items:

- `jsdocs.zip` — The compiled documentation for the library.
- `loki-cordova-fs-adapter` — A plugin used for Sync Express feature for Cordova to extend amount of available storage.
- `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-mcs-notifications-cordova-plugin` — A Cordova plugin that enables iOS and Android notifications.
- `oracle_mobile_cloud_config.js` — An AMCe configuration file, in which you can insert environment and authentication details for the mobile backends that your app will access.
- `types` — Contains TypeScript definitions for the SDK's modules and plugins.

Next Steps

Once you have the Cordova SDK set up, you can start using it to add AMCe features to your app.

- [Authentication in AMCe](#)
- [Notifications](#)
- [My Profile](#)

- Storage
- Data Offline and Sync
- Location
- Database
- App Policies

5

JavaScript Applications

If you develop JavaScript-based mobile apps, you can use the client SDK that Oracle Autonomous Mobile Cloud Enterprise (AMCe) provides for JavaScript. This SDK simplifies authentication with AMCe and provides JavaScript wrapper classes for AMCe 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

To get the AMCe client SDK for JavaScript, go to the [Oracle Autonomous Mobile Cloud Enterprise Downloads page](#) on OTN.

Creating a Backend

You create a backend to serve as a secure gateway between your app and AMCe features, such as platform and custom APIs. For your app to access these resources, it authenticates with a backend.

1. Click  to open the side menu and select **Development > Backends**.
2. Click **New Backend**.
3. Once you complete the dialog and the backend is created, keep the Settings page open.

You'll need to configure your app with some of this information.

Adding the SDK

Assuming a basic app setup, without intervening frameworks, here's what you would do to add the JavaScript client SDK to an app:

1. If you haven't already done so, unzip the SDK zip.
2. Copy `mcs.min.js` (and/or `mcs.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`.
4. Add script tags for the SDK and the configuration file in your app's `index.html` file:

```
<script src="lib/mcs/mcs.js"></script>
<script src="app/oracle_mobile_cloud_config.js"></script>
```

5. (Optional) For RequireJS environments, load `mcs.js` in your app using RequireJS.

 **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

To use the client SDK in a JavaScript app, add the `oracle_mobile_cloud_config.js` configuration file to the app and fill it in with environment details for your backend in AMCe. In turn, the SDK classes use this information to construct HTTP headers for REST calls made to AMCe.

 **Note:**

For browser-based apps, you need to manage cross-origin resource sharing (CORS) for access to AMCe 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 file is essentially divided into the following parts:

- The `mobileBackend` property and its contents.

You include this part if you are using a backend with the app. The SDK classes use the environment and authentication details you specify there to access the backend and construct HTTP headers for REST calls made to APIs.

- Properties that apply to the configuration as a whole, such as `logLevel` and `oAuthTokenEndpoint`. These keys generally, but don't have to, appear at the top of the file.

The following example shows the structure of a generic `oracle_mobile_cloud_config.js` file:

```
var mcs_config = {
  "logLevel": mcs.LOG_LEVEL.NONE,
  "logHTTP": true,
  "oAuthTokenEndPoint": "OAUTH_URL",
  "mobileBackend": {
    "name": "NAME",
    "baseUrl": "BASE_URL",
    "authentication": {
      "type": mcs.AUTHENTICATION_TYPES.oauth,
      "oauth": {
        "clientId": "CLIENT_ID",
        "clientSecret": "CLIENT_SECRET"
      }
    }
}
```

```
};

"syncExpress": {
  "handler": "OracleRestHandler",
  "policies": [
    {
      "path": '/mobile/custom/firstApi/tasks/:id(\d+)?',
    },
    {
      "path": '/mobile/custom/secondApi/tasks/:id(\d+)?',
    }
  ]
}
};
```

Here are some notes on the file's elements.

- `oAuthTokenEndPoint` — The URL of the OAuth server from where your application gets its authentication token. This key needs to be provided for all apps that rely on OAuth to authenticate. You get this from the backend's **Settings** page. The endpoint should be only the base URL (in the form `https://host.domain:port`).
- `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`.
- `enableLogger` — When set to `true`, logging is included in your app.
- `logHTTP` — When set to `true`, the SDK logs the HTTP and HTTPS headers in requests and responses.
- `mobileBackend` — An element containing authentication details for your backend and other optional details, such as synchronization properties.

You get the authentication details, such as the OAuth and HTTP credentials, from the backend's **Settings** page.

- `mobileBackend/baseUrl` — The base URL for all APIs that you call through the backend. You get this from the backend's **Settings** page.
- `mobileBackend/authentication` — Contains the following sub-elements:
 - The `type` sub-element, with possible values of `oauth`, `basic`, `facebook`, and `token`.
 - One or more sub-elements for authentication types, each containing authentication credentials.

You can also add the `offlineEnabled` key and set its value to `true`.

See [Authentication Properties](#) for examples of each authentication type.

For details on sync elements, see [Building Apps that Work Offline Using Sync Express](#).

Authentication Properties

The contents and sub-elements of `authentication` depend on what kind of authentication the app will be using.

OAuth

- Set the value of the `type` property to `mcs.AUTHENTICATION_TYPES.oauth`.
- At the same level as the `type` property, create a property called `oauth` and fill in the `clientId` and `clientSecret` credentials provided by the backend.
- At the top level of the file, supply the `oAuthTokenEndPoint` value that is supplied but *without* the `oauth2/v1/token` that is appended on the backend's **Settings** page.

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

```
var mcs_config = {  
    ...  
    "oAuthTokenEndPoint": "BASE_OAUTH_URL_WITH_oauth2/v1/token_REMOVED",  
    "mobileBackend": {  
        "name": "NAME",  
        "baseUrl": "BASE_URL",  
        "authentication": {  
            "type": mcs.AUTHENTICATION_TYPES.oauth,  
            "oauth": {  
                "clientId": "CLIENT_ID",  
                "clientSecret": "CLIENT_SECRET"  
            }  
        }  
    }  
};
```

HTTP Basic

- Set the value of the `type` property to `mcs.AUTHENTICATION_TYPES.basic`.
- At the same level as the `type` property, create a property called `basic` and fill in the `mobileBackendID` and `anonymousKey` that are provided by the backend.

The resulting entries might look something like this:

```
var mcs_config = {  
    ...  
    "mobileBackend": {  
        "name": "NAME",  
        "baseUrl": "BASE_URL",  
        "authentication": {  
            "type": mcs.AUTHENTICATION_TYPES.basic,  
            "basic": {  
                "mobileBackendID": "MOBILE_BACKEND_ID",  
                "anonymousKey": "ANONYMOUS_KEY"  
            }  
        }  
    }  
};
```

Token Exchange

If you are authenticating using a third-party token, do the following:

- Set the value of the `type` property to `mcs.AUTHENTICATION_TYPES.token`.
- Fill in the `mobileBackendId` and `anonymousKey` that are provided by the backend.

The resulting properties might look something like this:

```
var mcs_config = {  
    ...  
    "mobileBackend": {  
        "name": "NAME",  
        "baseUrl": "BASE_URL",  
        "authentication": {  
            "type": mcs.AUTHENTICATION_TYPES.token,  
            "token": {  
                "mobileBackendId": "YOUR_BACKEND_ID",  
                "anonymousKey": "ANONYMOUS_KEY"  
            }  
        }  
    }  
};
```

Facebook Login

- Set the value of the `type` property to `mcs.AUTHENTICATION_TYPES.facebook`.
- Fill in the HTTP Basic auth credentials and/or the OAuth credentials provided by the backend.
- Fill in the `appId` for the Facebook app.
- Fill in the relevant `scopes`.

The resulting `authentication` entry might look something like this:

```
var mcs_config = {  
    ...  
    "mobileBackend": {  
        "name": "NAME",  
        "baseUrl": "BASE_URL",  
        "authentication": {  
            "type": mcs.AUTHENTICATION_TYPES.facebook,  
            "facebook": {  
                "appId": "YOUR_FACEBOOK_APP_ID",  
                "mobileBackendId": "YOUR_BACKEND_ID",  
                "anonymousKey": "YOUR_ANONYMOUS_KEY",  
                "scopes":  
                    "public_profile,user_friends,email,user_location,user_birthday"  
            }  
        }  
    }  
};
```

Calling Mobile APIs

In AMCe, a backend is a logical grouping of custom APIs, storage collections, and other resources that you can use in your apps. The backend also provides the security context for accessing those resources.

Here are the general steps for using a backend in your JavaScript app:

1. Add the client SDK to your app.
2. Fill in the `oracle_mobile_cloud_config.js` with environment and authentication details for the backend.
3. Add an SDK call to your app to load the configuration info.
4. Add an SDK call to your app to handle authentication.
5. Add any other SDK calls that you want to use.

Loading the Backend's Configuration

Before you can make calls to AMCe APIs using the JavaScript client SDK, you need to load the configuration for the backend you are going to use. In the following snippet, `mcs_config` is the name of the configuration that is defined in the `oracle_mobile_cloud_config.js` file that you have added to your app.

```
mcs.init(mcs_config);
```

Authenticating and Logging In

Here are some examples of how to use the `Authorization` class of the JavaScript client SDK in your code. These examples assume you already configured the SDK config file for the type of authentication you're using, as described in [Configuring SDK Properties](#).

OAuth and HTTP Basic

Set the authentication type for the backend to `oauth` (or `basic`):

```
mcs.mobileBackend.setAuthenticationType(mcs.AUTHENTICATION_TYPES.oauth);
```

Then add a function that calls `Authorization.authenticate` on the backend, passes it a user name and password, and specifies callbacks for success and failure:

```
mcs.mobileBackend.authorization.authenticate(username,  
password).then(callback).catch(errorCallback);
```

If you want to use anonymous authentication, the method to call is `authenticateAnonymous`:

```
mcs.mobileBackend.authorization.authenticateAnonymous().then(callback).catc  
h(errorCallback);
```

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 AMCe, see [Third-Party SAML and JWT Tokens](#).

Set the authentication type for the backend to `token` and then pass the token in the authorization call:

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

Facebook

Set the authentication type for the backend to `facebook` and then call `authenticate()`:

```
mcs.mobileBackend.setAuthenticationType(mcs.AUTHENTICATION_TYPES.facebook);  
mcs.mobileBackend.authorization.authenticate().then(callback).catch(errorCallback);
```

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 AMCe 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 AMCe 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

Once you include the client SDK libraries in your application, and adjust configuration settings, you're ready to use the SDK classes in your apps.

Here's an example of how you could use these classes to get an object from a Storage collection in the mobile backend:

```
mcs.mobileBackend.storage.getCollection(<collection id>)  
.then(function(collection){  
    return collection.getObject(<object id>, 'blob');
```

```
})
.then(function(object){
  console.log(object);
})
.catch(function(response){
  console.error(response);
})
```

Calling Custom APIs

The JavaScript client SDK provides the `CustomCode` class to simplify the calling of custom APIs. 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.mobileBackend.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-section10.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:errorMessage": "/mobile/custom/cordovaJSApi2/tasks" }
}
*/
//Depends on the response format defined in the API.
});
```

Using TypeScript

It is also possible to use TypeScript objects with the Cordova and JavaScript client SDKs.

Here are some basic steps and examples for using TypeScript with the SDK. The examples assume your app is using the Ionic framework (though you can also use TypeScript without it).

Setting Up the SDK

1. Install the SDK in your project by running this command in your project folder:

```
npm install {path to unzipped SDK location}
```

2. Add import statements to your service to import SDK types:

```
import {IMCS} from 'mcs'
```

3. Create the configuration file for the app:

```
import {IMCS,
        IOraclMobileCloudConfig,
        IMobileBackendConfig,
        IAuthenticationConfig,
        IBasicAuthConfig,
        IOAuthConfig,
        import * as mcssdk from 'mcs'
const mcs: IMCS = mcssdk;

export const mcsConfig: IOraclMobileCloudConfig = {
    logLevel: mcs.LOG_LEVEL.NONE,
    logHTTP: true,
    oAuthTokenEndPoint: 'OAUTH_URL',
    mobileBackend: <IMobileBackendConfig>{
        name: 'NAME',
        baseUrl: 'BASE_URL',
        authentication: <IAuthenticationConfig>{
            type: mcs.AUTHENTICATION_TYPES.basic,
            basic: <IBasicAuthConfig>{
                mobileBackendId: 'MOBILE_BACKEND_ID',
                anonymousKey: 'ANONYMOUS_KEY'
            }
        }
    }
};
```

4. Import the configuration into the app. If the above file is called `mcs-config.ts`, the import would look like :

```
import { mcsConfig } from "../mcs-config";
```

Calling Mobile APIs

1. Add these import statements to your service or component:

```
import {IMCS} from 'mcs';
import * as mcssdk from 'mcs'; And in your class add declaration
statement:
```

2. Add the declaration statement in your class:

```
export class ComponentClass{  
    mcs: IMCS = mcssdk;  
}
```

3. Initialize the SDK library with a configuration:

```
this.mcs.init(mcsConfig);
```

4. Call backend functionality:

```
this.mcs.mobileBackend.setAuthenticationType(this.mcs.AUTHENTICATION_TYPE.basic);  
this.mcs.mobileBackend.authorization.authenticate(username, password);
```

Adding Support for Location Services (Ionic Only)

```
ionic cordova plugin add cordova-plugin-geolocation
```

Adding Support for Push Notifications (Ionic only)

1. (For Android) Register your app for notifications on the Firebase Cloud Messaging (FCM) console. See [Set Up a Firebase Cloud Messaging Client App on Android](#) on Google's developer site.

When you generate the configuration file for your app, make sure you choose to enable the Cloud Messaging service.

When generation is complete, the **Project Number** (aka **Sender ID**) and **API Key** are displayed. These credentials 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 AMCe.

2. (For Android) Download the generated Firebase configuration file and put it in the root of your project.
3. (For Android) If you haven't already done so, install the notifications plugin that is supplied with the SDK:

```
cordova plugin add PATH_TO_UNZIPPED_SDK/oracle-mcs-notifications-cordova-plugin
```

4. (For iOS) Set up the app for notifications with APNS. See [iOS: Apple Secure Certificates](#)
5. Create the app in AMCe and notifications profiles for Android and iOS. See [Creating a Notifications Profile](#).
6. In your app code, register for notifications:

```
...  
MCSNotificationsCordovaPlugin.onTokenRefresh(this.handleTokenRefresh.bind(this), this.handleError.bind(this));  
...  
handleTokenRefresh(token: string){
```

```
        console.log('NotificationsService Token refreshed', token);

        this.mcs.mobileBackend.notifications.registerForNotifications(token,
packageName, appVersion, 'FCM')
            .then(this.handleRegisterForNotifications.bind(this))
            .catch(this.handleError.bind(this));
    }

handleRegisterForNotifications(response: INetworkResponse){
    console.log('NotificationsService, device registered for
notifications');
}
handleError(error: any){
    console.error('NotificationsService Error', error);
}
```

7. In your app code, subscribe to notifications events:

```
...
MCSNotificationsCordovaPlugin.onMessageReceived(this.handleMessageReceived.bind(this), this.handleError.bind(this));
...
handleMessageReceived(data: any){
    console.log('NotificationsService Message received', data);
}
handleError(error: any){
    console.error('NotificationsService Error', error);
}
```

Libraries

The JavaScript client 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 AMCe configuration file, in which you can insert environment and authentication details for the mobile backends that your app will access.
- types — Contains TypeScript definitions for the SDK's modules and plugins.

Next Steps

Once you have the JavaScript SDK set up, you can start using it to add AMCe features to your app.

- [Authentication in AMCe](#)
- [Notifications](#)
- [My Profile](#)
- [Storage](#)
- [Data Offline and Sync](#)
- [Location](#)
- [Database](#)
- [App Policies](#)

6

Notifications

Oracle Autonomous Mobile Cloud Enterprise (AMCe) 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.

Setting Up a Mobile App for Notifications

To make notifications work in your mobile apps, there are several key steps.

1. Install the client SDK for your platform.
2. Get credentials from notification providers to establish your mobile app as a known item on the network. For detailed instructions, see [Getting Network Credentials for Notifications](#).
3. Create notifications profiles to hold the credentials, described in [Creating a Notifications Profile](#).

Next, you need to register an app client and add the notifications profile to it:

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.

2. Click  to open the side menu and select **Development > Notification Profiles**.

3. Click **Clients**.

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. 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 get 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.

9. Set up the app to connect to the notification provider from the mobile device and establish rules for communication, described below.

Now that you have registered the app client in AMCe, you have a few options for sending notifications to your app, as shown in [Sending Notifications to and from Your App](#).

Setting Up the Device Handshake for Notifications

To allow notifications to be delivered to your mobile app through the network, every platform requires some form of “device handshake” to register and establish the protocol for communication.

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.firebaseio.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.mobilebackend.MobileManager;

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 =
MobileManager.getManager().getMobileBackend(this).getServiceProxy(Notificat
ions.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 AMCe, 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
        BroadcastReceiver mRegistrationBroadcastReceiver = new
BroadcastReceiver() {
            @Override
            public void onReceive(Context context, Intent intent) {

                SharedPreferences sharedpreferences =
                    PreferenceManager.getDefaultSharedPreferences(context);
                boolean sentToken = sharedpreferences
                    .getBoolean(NotificationConfig.SENT_TOKEN_TO_SERVER,
false);
                if (sentToken){
```

```
        Logger.debug(TAG, "Token retrieved and sent to server! App can use
FCM");
    }else{
        Logger.debug(TAG, "An error occurred while registering the
device");
    }
}
};

//Call on successful registration
LocalBroadcastManager.getInstance(getApplicationContext()).registerReceive
r(
    mRegistrationBroadcastReceiver,
    new IntentFilter(NotificationConfig.REGISTRATION_COMPLETE));

//Initialization of notifications service
Notifications notifications =
MobileBackendManager.getManager().getDefauleMobileBackend(getApplicationContext())
.getServiceProxy(Notification.class);
boolean result = notifications.initialize(view.getContext());
```

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 notifications service
Notifications notifications =
MobileManager.getManager().getDefauleMobileBackend(getApplicationContext())
.getServiceProxy(Notification.class);
boolean result = notifications.deregisterDevice(view.getContext());

Logger.debug(TAG, "unregister " + result);
```

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 **didFailToRegisterForRemoteNotificationsWithError:** method is called.

2. Get the Notifications SDK object.

```

(OMCNotifications *) getOMCNotifications{
    OMCAuthorization *auth = [[[OMCMobileManager sharedManager]
mobileBackend] mobileBackendForName:
    <Name_of_Mobile_Backend_from_OMC.Plist>].authorization;
    OMCNotifications* omcNotifications=nil;
    NSError* err = [auth authenticate:<Username> password:<Password>];
    if (!err){
        omcNotifications = [[[OMCMobileManager sharedManager]
mobileBackendForName:
            <Name_of_Mobile_Backend_from_OMC.Plist>] notifications];
    }
    return omcNotifications;
}

```

3. Register for notifications using the Notifications SDK object.

```

-(void) registerForMCSNotifications:(id) sender {
    // Get notifications object from your mobile backend object.
    OMCNotifications* notifications = [mbe notifications];

    // Call the register api and pass your iOS device's device token
    // data.
    [notifications registerForNotifications:[self getDeviceTokenData]
    onSuccess:^(NSHTTPURLResponse *response) {
        NSLog(@"Device registered successfully.");
        dispatch_async(dispatch_get_main_queue(), ^{
            // Update UI if-needed.
        });
    } onError:^(NSError *error) {
        NSLog(@"Error registering your device.");
        dispatch_async(dispatch_get_main_queue(), ^{
            // Update UI if-needed.
        });
    }];
}

```

Next, register your mobile app with the associated backend, and enable notifications. See [Backends](#).

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 a Device Handshake for Windows

This section assumes you have already registered your mobile app with WNS, described in [Windows: WNS Credentials](#).

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 backend, and enable notifications. For detailed instructions, see the [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 UI

AMCe provides a notifications testing UI that allows you to send scheduled notifications to a defined set of recipients.

1. Click  to open the side menu and select **Development > Backends**.
2. On the Backends page, select the 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 notifications profile as described in [Creating a Notifications Profile](#). 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 backend**. A single 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, 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`: This header should include the name and password for a team member with either the Administrator or Developer 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, 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 Autonomous Mobile Cloud Enterprise.

REST

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 either the Administrator or Developer 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
VGVzdElvYmlsZVVzZXIyYzE4YWRiZjMyMDg0ZWZkOWQyODM0NjA1OGNmExampleAuthString="

-H "Oracle-Mobile-Backend-ID: 7cf06198-053e-4311-8186-cae145900d59"
-H "Content-Type:application/json"
-d '{"mobileClient": {"id":
"MyClientac3d8baflaa348b48d80e9b7fd026067", "version": "1.0", "platform": "IOS"}, "notificationProvider": "APNS", "notificationToken": "03767dea-29ac-4440-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
VGVzdElvYmlsZVVzZXIyYzE4YWRiZjMyMDg0ZWZkOWQyODM0NjA1OGNmExampleAuthString=
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":  
    "MyClientac3d8baflaa348b48d80e9b7fd026067", "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 `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": [ "xxxxxx"], "payload": { "services": { "fcm": {"notification": { "title": "Sale On Now!", "body": "50% off until Saturday" } } } } }
```

Data object:

```
"notificationTokens": [ "xxxxxxxx"], "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
<code>title</code>	The alert title. If a title is specified, the body parameter is also required.	string	"Sale On Now! "

Parameter	Description	Data Type	Example
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
sound	The sound file to play with the notification. Only .wav format is supported by APNS , WNS, and FCM. <ul style="list-style-type: none"> 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
```

iOS

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 either the Administrator or Developer 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 code shows how to register an iOS device for push notifications:

```
-(void) registerForMCSNotifications:(id) sender {  
  
    // Get notifications object from your mobile backend object.  
  
    OMCNotifications* notifications = [mbe notifications];  
  
    // Call register api and pass the iOS device token data.  
  
    [notifications registerForNotifications:[self getDeviceTokenData]  
        onSuccess:^(NSHTTPURLResponse *response) {  
            NSLog(@"Device registered successfully.");  
        } onFailure:^(NSError *error) {  
            NSLog(@"Error registering device: %@", error);  
        }];  
}
```

```
        dispatch_async(dispatch_get_main_queue(), ^{
            // Update UI if needed.
        }) ;
    } onError:^(NSError *error) {
        NSLog(@"Error registering your device.");
        dispatch_async(dispatch_get_main_queue(), ^{
            //Update UI if needed.
        }) ;
    }];
}
```

Deregistering a Device

```
// Call register api and pass your iOS device's device token data.

[notifications deregisterForNotifications:[self getDeviceTokenData]
onSuccess:^(NSHTTPURLResponse *response) {
    NSLog(@"Device deregistered successfully.");
    dispatch_async(dispatch_get_main_queue(), ^{
        // Update UI if-needed.
    }) ;
} onError:^(NSError *error) {
    NSLog(@"Error deregistering your device.");
    dispatch_async(dispatch_get_main_queue(), ^{
        // Update UI if-needed.
    }) ;
}];
```

Cordova/JavaScript/TypeScript

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 either the Administrator or Developer 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 code shows how to register a device for push notifications using Cordova/JavaScript/TypeScript:

```
...  
document.addEventListener("deviceready", handleDeviceReady, false);
```

```
...
function handleDeviceReady(){
    MCSNotificationsCordovaPlugin.onTokenRefresh(handleTokenRefresh,
handleError);
}
...
function handleTokenRefresh(token){
    console.log('NotificationsService Token refreshed', token);
    mcs.mobileBackend.notifications.registerForNotifications(token,
packageName, appVersion, 'FCM')
        .then(handleRegisterForNotifications)
        .catch(handleError);
}

function handleRegisterForNotifications(response){
    console.log('NotificationsService, device registered for
notifications');
}
function handleError(error){
    console.error('NotificationsService Error', error);
}
```

Deregistering a Device

```
this.mcs.mobileBackend.notifications
.deregisterForNotifications(token, packageName, appVersion, 'FCM')
.then(deregisterSuccess)
.catch(deregisterError);

function deregisterSuccess(response)
{ console.log(response); }
function deregisterError(response)
{ console.error(response); }
```

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
- 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](#).

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, the wrong API key or Project Number/Sender ID means the user might have disabled notifications on their device.

AMCe 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 AMCe and the notifications provider to refresh it.
- The user deleted the app from their device
- The API key or certificate in AMCe 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. 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 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, 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"
        }
    ]
}
```

My Profile

As a mobile app developer, you use the My Profile API to access details about the currently authorized user.

 **Note:**

This API only retrieves user information. To add or update users programmatically, see [REST API for Oracle Identity Cloud Service](#).

User Types

The information that the API returns depends on what type of user you are inquiring about. Here are the types of users:

- **IDCS users:** These users have accounts that are managed by the domain's Oracle Identity Cloud Service (IDCS) as described in [Mobile Users and Roles](#).
- **Virtual users:** These users pass a third-party token for authorization as described in [Enterprise Single Sign-On in AMCe](#).
- **Social users:** These users have logged into the app from Facebook, as described in [Facebook Login in AMCe](#).

Getting User Profile Information

If your app needs user information, such as full names and roles, you can call the User Profile API to get that information from their profile.

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 [Oracle Autonomous Mobile Cloud Enterprise REST API Reference](#).
- You can call the `ums.getMe(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/me`. 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/me
```

The contents of the response body depends on the user type:

- When the user is an IDCS user, the response contains the IDCS user information, including roles.
- When the user is a virtual user, the response contains the user name and roles.
- When the user is a social user, the response contains the user's mobile ID.

Here's an example of a response for an IDCS user:

```
{
  "idcsCreatedBy": {
    "type": "User",
    "display": "admin opc",
    "value": "ABCD6996a13b1641eca66fbf4c75af42",
    "$ref": "https://my.idcs.com:443/admin/v1/Apps/
ABCD6996a13b1641eca66fbf4c75af42"
  },
  "id": "ABCD5f2a5eb5e1664cbc94fc651a8562",
  "active": true,
  "displayName": "Doe Joe",
  "idcsLastModifiedBy": {
    "value": "ABCD5316fb6146d1bb3b60b75363",
    "display": "idcssso",
    "type": "App",
    "$ref": "https://my.idcs.com:443/admin/v1/Apps/
ABCD5316fb6146d1bb3b60b75363"
  },
  "userName": "jdoe",
  "urn:ietf:params:scim:schemas:oracle:idcs:extension:user:User": {
    "isFederatedUser": false,
    "grants": [
      {
        "appId": "c6a3a4174e8a4975b688f43e8aaace18",
        "grantMechanism": "ADMINISTRATOR_TO_USER",
        "value": "273979318ee2482bba40a44142a6c90a",
        "$ref": "https://my.idcs.com:443/admin/v1/Grants/
273979318ee2482bba40a44142a6c90a"
      ],
      "appRoles": [
        {
          "value": "ABCD278fbf50244c0ab13f16a117b7da7",
          "adminRole": false,
        }
      ]
    ],
    "legacyGroupName": "ABCD29F71D52E41B49EBD6F895FDA.ServiceBusinessUser",
    "appId": "ABCD220aeebc14937b4b82df71671",
    "appName": "ABCD2F71D52E41B49EBD6F895FDA_APPID",
    "display": "MyMobileAppUser",
    "$ref": "https://my.idcs.com:443/admin/v1/AppRoles/
ABCD278fbf50244c0ab13f16a117b"
  ],
  "accounts": [
    {
      "value": "125daaab1e7e47b995232f51516544c6",
      "appId": "c6a3a4174e8a4975b688f43e8aaace18",
      "$ref": "https://my.idcs.com:443/admin/v1/AccountMgmtInfos/
125daaab1e7e47b995232f51516544c6"
    }
  ],
  "emails": [
}
```

```
{  
    "type": "recovery",  
    "primary": false,  
    "value": "jdoe@example.com",  
    "verified": false},  
    {  
        "type": "work",  
        "primary": true,  
        "value": "jdoe@example.com",  
        "verified": false}  
],  
"name": {  
    "givenName": "Joe",  
    "familyName": "Doe",  
    "formatted": "Joe Doe"},  
"schemas": [ "urn:ietf:params:scim:schemas:core:2.0:User"  
]}
```

Here's an example of a response for a virtual user:

```
{  
    "userName": "jdoe",  
    "urn:ietf:params:scim:schemas:oracle:idcs:extension:user:User":{  
        "appRoles": [  
            {  
                "display": "FIF_TECHNICIAN"  
            }  
        ]  
    }  
}
```

Here's an example of a response for a social (Facebook) user:

```
{  
    "userName": "1 :623:165"  
}
```

8

Storage

Oracle Autonomous Mobile Cloud Enterprise 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 Can I Do with Storage?

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 AMCe instance. The media are stored as *opaque* objects, which means that each object is stored and retrieved from the collection by a user- or system-generated GUID (globally unique ID). You use mobile user roles to control who can read and write the objects in the collection.

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, or to host HTML 5 applications like a content management system (CMS).

Android

Adding an Object to a Collection

```
String text = "This is sample text file";
String name = "sampleText.txt";
StorageObject storageObject = new StorageObject(null, text.getBytes(),
"text/plain");
storageObject.setDisplayName(name);
```

Fetching an Object

This fetches the storage object from a collection and reads its contents in a stream:

```
int i=0;
for (StorageObject storageObject: storageObjects) {
    i++;
    InputStream payload = storageObject.getPayloadStream();
    int n;
    char[] buffer = new char[1024 * 4];
    InputStreamReader reader = null;
    try {
        reader = new InputStreamReader(payload, "UTF8");
    } catch (UnsupportedEncodingException e) {
        e.printStackTrace();
    }
    StringWriter writer = new StringWriter();
```

```
assert reader != null;

try {
    while (-1 != (n = reader.read(buffer))) {
        writer.write(buffer, 0, n);
    }
} catch (IOException e) {
    e.printStackTrace();
}
Logger.debug(TAG, "Storage Object "+i+" "+writer.toString());
}
```

Getting Multiple Objects from a Collection

```
List<storageObject> storageObjects = null;
try {
    Storage storage = mobileBackend.getServiceProxy(Storage.class);
    storageObjects = storageCollection.get(0,10,true);
} catch (ServiceProxyException e) {
    e.printStackTrace();
}
```

Getting a Shared Collection

This gets a specific shared collection called sharedCollection:

```
StorageCollection storageCollection= null;
try {
    Storage storage = mobileBackend.getServiceProxy(Storage.class);
    storageCollection = storage.getStorageCollection("sharedCollection");
} catch (ServiceProxyException e) {
    e.printStackTrace();
}
```

Retrieving an Object

```
private Storage mStorage;
private String collectionID = "YOUR_COLLECTION_ID";
private String objectID = "YOUR_OBJECT_ID";

...

try {
    //Initialize and obtain the storage client
    mStorage =
MobileManager.getManager().getDefaultValueBackend(this).getServiceProxy(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();
}
```

Updating an Object

```
StorageObject storageObject = null;
try {
    Storage storage = mobileBackend.getServiceProxy(Storage.class);
    storageObject = storageCollection.get("26651715-9259-4676-a035-
df47ef3e7e79");
} catch (ServiceProxyException e) {
    e.printStackTrace();
}

String text = "This is modified text in a text file";

storageObject.setPayload(text.getBytes(), "text/plain");

try {
    Storage storage = mobileBackend.getServiceProxy(Storage.class);
    storageCollection.put(storageObject);
} catch (ServiceProxyException e) {
    e.printStackTrace();
}
```

Uploading a New Object to a Collection

```
try {
    Storage storage = mobileBackend.getServiceProxy(Storage.class);
    storageCollection.post(storageObject);
} catch (ServiceProxyException e) {
    e.printStackTrace();
}
```

iOS

Adding an Object to a Collection

```
- (void) uploadData{
    NSString* collection_Id = @"myCollection";
    NSString* payload = @"This is a simple text object";
```

```
NSString* contentType = @"text/plain";

if ( payload == nil || [payload isEqualToString:@""])
{
    NSLog(@"There is nothing to upload");
}
else{

    // Get storage object.
    OMCStorage* storage = [mbe storage];

    // Get collection where you want to upload new data.
    OMCStorageCollection* aCollection = [storage
getCollection:collection_Id];

    // Create new data from payload (in case your payload is not
already in NSData format )
    NSData* payloadData = [payload
dataUsingEncoding:NSUTF8StringEncoding];
    OMCStorageObject* aObject = [[OMCStorageObject alloc]
setPayloadFromData:payloadData

        withContentType:contentType];

    // Post data.
    [aCollection post:aObject];

    NSLog(@"Upload finished");
}
}
```

Deleting an Object

```
NSString* collection_Id = @"";
// Get your collection
OMCStorageCollection* aCollection = [storage
getCollection:collection_Id];

// Create/Update an object with the same objectID.
NSString* objectID = @"object2";
BOOL isDeleteSuccessful = [aCollection deleteWithKey:objectID];
```

Downloading Data to a Collection

This downloads data from any storage collection where:
collectionID is the id for the target collection.
objectID is the id for the target object.

```
-(void) downloadData{

    NSString* collection_Id = @"";
```

```
NSString* object_Id = @"";

// Get storage object.
OMCStorage* storage = [mbe storage];

// Get your collection
OMCStorageCollection* aCollection = [storage
getCollection:collection_Id];

// Get your object from your collection.
OMCStorageObject* anObject = [aCollection get:object_Id];

// Get the data from payload of your object.
NSData* data = [anObject getPayloadData];
NSLog(@"Download finished");
}
```

Getting a User Isolated Collection

```
NSString* collection_Id = @"";

NSString* user_Id = @"";

// Get user isolated collection.
OMCStorageCollection* aCollection = [storage getCollection:collection_Id
forUserId:user_Id];
```

Getting Multiple Objects from a Collection

```
NSString* collection_Id = @"";

// Get your collection.
OMCStorageCollection* aCollection = [storage
getCollection:collection_Id];

NSUInteger offset = 0; NSUInteger limit = 10;
NSArray<OMCStorageObject*>* objects = [collection get:offset
withLimit:limit getAllObjects:NO];
```

Getting Object Data as a Stream

```
NSString* collection_Id = @"";

OMCStorageCollection* aCollection = [storage
getCollection:collection_Id];

NSString* object_Id = @"";

OMCStorageObject* anObject = [aCollection get:object_Id];
```

```
NSInputStream* inStream = [anObject getPayloadStream];
```

Retrieving a Storage Object

```
- (void) downloadData{  
  
    //Fill in IDs for collection and object.  
    NSString* collection_Id = @"";  
    NSString* object_Id = @"";  
  
    // Get storage object.  
    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 your object's payload.  
    NSData* data = [aObject getPayloadData];  
    NSLog(@"Download finished");  
}
```

Updating an Object

```
NSString* collection_Id = @"";  
  
// Get your collection.  
OMCStorageCollection* aCollection = [storage  
getCollection:collection_Id];  
  
// Create/Update object with the same objectID.  
NSString* objectID = @"";  
NSData* payload = @{@"This is updated object" dataUsingEncoding:  
NSUTF8StringEncoding};  
OMCStorageObject* object = [[OMCStorageObject alloc]  
initPayload:objectID  
  
withData:payload  
  
andContentType:@"plain/text"];  
OMCStorageObject* returnedObject = [aCollection put:object];
```

Uploading Data to a Collection

```
-(void) uploadData{  
    NSString* collection_Id = @"";  
    NSString* payload = @"";
```

```
NSString* contentType = @"";

if ( payload == nil || [payload isEqualToString:@""] )
{
    NSLog(@"There is nothing to upload");
}
else{

    // Get the storage object from your MobileBackend object.
    OMCStorage* storage = [mbe storage];

    // Get the collection where you want to upload new data.
    OMCStorageCollection* aCollection = [storage
getCollection:collection_Id];

    // Create new data from payload (in case your payload is not
already in NSData format).
    NSData* payloadData = [payload
dataUsingEncoding:NSUTF8StringEncoding];
    OMCStorageObject* anObject = [[OMCStorageObject alloc]
setPayloadFromData:payloadData

withContentType:contentType];

    // Post data to the collection.
    [aCollection post:anObject];

    NSLog(@"Upload finished");
}
}
```

Cordova, JavaScript, and TypeScript

Adding an Object to a Collection

```
var obj = new mcs.StorageObject(collection);
obj.setDisplayName("XYZ.pdf");
obj.loadPayload("Hello World from Oracle Autonomous Mobile Cloud
Enterprise Cordova SDK", "text/plain");

collection.postObject(obj).then(onSuccess, onFailure);
function onSuccess(collection) {
    console.log(collection);
    return collection;
}

function onFailure(error) {
    console.error(error);
    return Promise.reject(error);
}
```

Deleting an Object

```
collection.deleteObject(objectId)
.thenonDeleteObjectSuccess)
.catch(onDeleteObjectFailure);
function onDeleteObjectSuccess(response) {
    console.log(response);
    return response;
}

function onDeleteObjectFailure(error) {
    console.error(error);
    return Promise.reject(error);
}
```

Fetching an Object

This fetches the storage object from a collection and reads its contents in a stream:

```
collection.getObject(objectId, 'json')
.then(onGetObjectSuccess)
.catch(onGetObjectFailed);

function onGetObjectSuccess(object)
{ console.log(object);
    return object;
}

function onGetObjectFailed(error)
{ console.error(error);
    return Promise.reject(error);
}
```

Getting a Collection

```
var backend = mcs.mobileBackend;
backend.storage.getCollection(collectionName)
.then(onGetCollectionSuccess)
.catch(onGetCollectionFailed);

function onGetCollectionSuccess(collection){
    console.log(collection);
    return collection;
}

function onGetCollectionFailed(error) {
    console.error(error);
    return Promise.reject(error);
}
```

Getting an Object from a User Isolated Collection

This gets an object from a user isolated collection belonging to another user:

```
let backend = mcs.mobileBackend;
backend.storage.getCollection(collectionName, userId)
.then(onGetCollectionSuccess)
.catch(onGetCollectionFailed);

function onGetCollectionSuccess(collection) {
    console.log(collection);
    return collection;
}

function onGetCollectionFailed(error){
    console.error(error);
    return Promise.reject(error);
}
```

Getting Multiple Objects from a Collection

Gets a collection, then uses that collection to get multiple objects:

```
collection.getObjects(2, 3, false)
.then(onSuccess)
.catch(onFailure);

function onSuccess(collection) {
    console.log(collection);
    return collection;
}

function onFailure(error) {
    console.error(error);
    return Promise.reject(error);
}
```

Updating an Object

```
collection.getObject(objectId)
.then(onGetObjectSuccess)
.then(onSaveObjectSuccess)
.catch(onGetObjectFail);

function onGetObjectSuccess(response){
    response.name = 'NewName';
    return collection.putObject(response);
}

function onSaveObjectSuccess(response){
    console.log(response);
```

```
        return response;
    }

    function onGetObjectFail(error){
        console.error(error);
        return Promise.reject(error);
    }
}
```

Custom Code

Retrieving and Storing Collections and Objects

For information on how custom code can retrieve collection information and store and retrieve objects, see [Accessing the Storage API from Custom Code](#).

REST API

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 [Oracle Autonomous Mobile Cloud Enterprise REST API Reference](#).

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 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
```

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 **Development** and then **APIs**).

Managing Collections

Mobile apps can only use collections that are associated with a backend. You can make this association by adding existing collections to the backend when you create it. You can also create new collections as part of this process.

You can also use the Storage configuration pages in the AMCe UI to associate a collection with a backend, as well as create and configure a collection, and define whether the collection is shared or user isolated.

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.



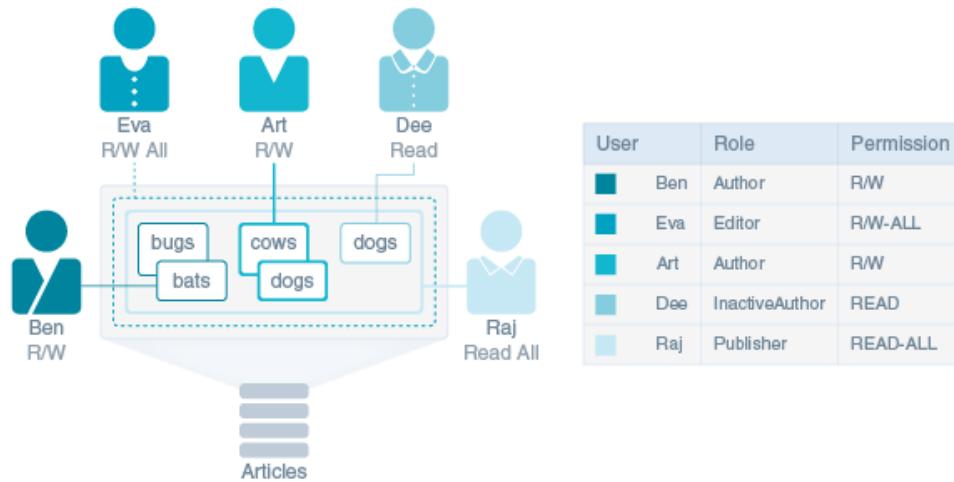
User Isolated Collection

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.



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, 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.

Storage Configuration

The Storage configuration pages in the UI can help you with a variety of tasks, such as creating and editing collections, and associating backends with collections.

Using the Storage Configuration Pages

There are two Storage configuration pages you can use: Manage all collections in your instance from the  > **Development>Storage** page. Manage collections for a specific backend from the **Storage** tab on the backend page.

Storage Configuration for All Collections

To open the Storage page for all collections, click  > **Development>Storage**.

Using this page, you can create collections, edit existing ones, associate them with mobile backends, and publish them.

You can find out when the collections listed were created or updated and which backends are using them by first selecting a collection and then expanding **Used By** and **History**.

Used By			
	Mobile Backend	FIF_Technician 1.0	
	Mobile Backend	FIF_Customer 1.0	
History			
	Updated by Chris	3 minutes ago	
	Created version 1.0 by Chris	9 minutes ago	

Storage Configuration for a Specific Backend

To manage collections for a specific backend, click  > **Development>Backends > Storage**. This page shows which collections are associated with the backend and allows you to create and update associated collections.

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**.

New Collection

Collections provide a way to group and manage related data objects. You can control who has permission to upload files, or to use the files in a collection.

* Collection Name 1.0

Collection Type Shared User Isolated [?](#)

The collection type can't be changed after the collection is created.

Short Description *Displayed in the collections list. 100 character limit.*

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.
id	string	<p>The collection name, which is used in the uniform resource identifier (URI). For example: <code>{baseURI}/mobile/platform/storage/collections/{collection}</code></p> <p>The collection name is case-sensitive, meaning that <code>mycollection</code> and <code>Mycollection</code> are two different collections.</p>

Adding Access Permissions to a Collection

Collection access is granted through anonymous user settings in the 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 My Profile UI and click **Roles**. To learn more about roles and mobile users, see [Mobile Users and Roles](#).

Anonymous Access to Collections

Anonymous access is often given to users who just want to check information on an app without having to log in or needing a defined role. Weather apps, where a user can check their local weather, are a good example of this.

Likewise, you can grant anonymous access to a shared collection. Once a shared collection is created, the administrator adds its name to the `Security_CollectionsAnonymousAccess` policy. You can then access the shared collection via the REST API or the client SDK for your mobile platform. Also, if you want to access this anonymous shared collection from the UI, a workaround is to grant Read-Write permission to any role on the properties page.

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.

Role-based Access to Collections

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.

Permission	Shared	User Isolated
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.

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 <code>image/jpeg</code> for a JPEG image, or <code>application/json</code> 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 <code>If-Match</code> and <code>If-None-Match</code> 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.

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).

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](#).

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 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.

The screenshot shows the Oracle Mobile Cloud Service interface. On the left, there is a sidebar with 'DEVELOPMENT' selected. Under 'Applications', 'Storage' is highlighted. The main area is titled 'APPLICATIONS > STORAGE'. It shows a 'Storage API' section with a 'New Collection' button and a list of collections: 'FIF_Deals 1.0' (DRAFT) and 'FIF_UserData 1.0' (DRAFT). To the right, a detailed view for 'FIF_Deals 1.0' is shown. It includes tabs for 'Open', 'Publish', 'Deploy', and 'More'. Below these are sections for 'Deployments' (empty), 'Used By' (listing 'Mobile Backend' and 'Mobile Backend' under 'FIF_Technician 1.0' and 'FIF_Customer 1.0' respectively, each with a delete 'X' icon), and 'History' (listing 'Updated by chris 3 minutes ago' and 'Created version 1.0 by chris 9 minutes ago').

- 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**.

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	<ul style="list-style-type: none">CordovaJavaScript	<ul style="list-style-type: none">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.

API	Platforms	Features
Synchronization	<ul style="list-style-type: none"> Android iOS 	<ul style="list-style-type: none"> 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,
  "mobileBackend": {
    "name": "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,
  "mobileBackend": {
    "name": "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,
    "mobileBackend": {
        "name": "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, Enterprise Android SDK Reference](#).

 **Tip:**

The client SDK download page contains an `examples` zip, 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 `MobileManager`, 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 =  
        MobileManager.getManager().  
        getMobileBackend(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_ONLINE);
policy.setExpirationPolicy(SyncPolicy.EXPIRATION_POLICY_EXPIRE_ON_RESTART);
policy.setEvictionPolicy(SyncPolicy.EVICTION_POLICY_EVICT_ON_EXPIRY_AT_STARTUP);
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 =
        MobileManager.getManager().getMobileBackend(this).getServiceProxy(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);  
    }  
}  
}  
}  
}  
}
```

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);
```

- 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, Enterprise iOS SDK Reference](#).

 **Tip:**

The client SDK download page contains an `examples.zip`, 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 `OMCMobileManager`, 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 effect 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(@"%@", 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(@"%@", 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 ];
        UIAlertView* okBtn = [UIAlertViewAction
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];

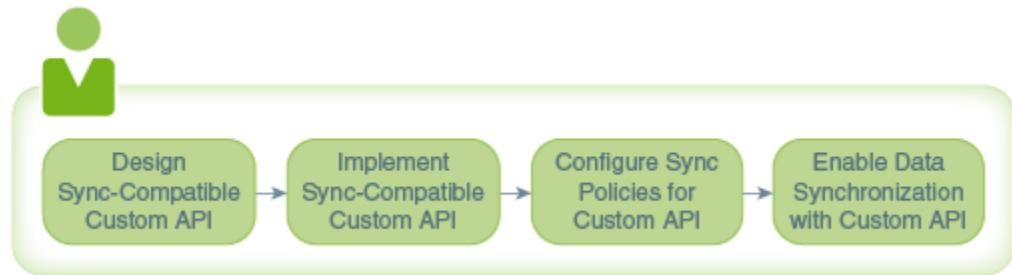
    [aResource reloadResource:YES
                      reloadFromService:NO
                        onSuccess:^(id mobileResource) {

        // Offline edits successfully 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:



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	<p>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.</p> <p>If a resource is not in the local cache, then the Synchronization library returns null.</p>
FETCH_FROM_SERVICE	<p>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.</p> <p>If the app is offline, the Synchronization library returns null.</p>
FETCH_FROM_SERVICE_IF_ONLINE	<p>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.</p>
FETCH_FROM_SERVICE_ON_CACHE_MISS	<p>Instructs the Synchronization library to fetch resources from the local cache if it is present.</p> <p>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.</p>
FETCH_FROM_SERVICE_ON_CACHE_MISS_OR_EXPIRY	<p>Instructs the Synchronization library to fetch resources from the local cache if they are present and not expired. Make sure to set <code>expireAfter</code> parameter to a suitable time period.</p> <p>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.</p>
FETCH_FROM_CACHE_SCHEDULE_REFRESH	<p>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.</p> <p>If a resource is not in the local cache, then the Synchronization library returns null.</p>

Policy	Description
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:



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
	<ul style="list-style-type: none"> • PERIODIC_REFRESH_POLICY_REFRESH_NONE • PERIODIC_REFRESH_POLICY_REFRESH_EXPIRED_ITEM_ON_STARTUP • PERIODIC_REFRESH_POLICY_PERIODICALLY_REFRESH_EXPIRED_ITEMS 	
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 `synchronization` 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_EVICTON</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_EVICTON</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>
            </periodicRefreshPolicy>
        </synchronization>
    </mobileBackend>
</mobileBackends>

```

```

<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	<p>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 yyyy HH:mm:ss z for SimpleDateFormat.</p> <p>The following synchronization policies are set for the resource object that is created from the response:</p> <ul style="list-style-type: none"> • 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

Header	Description
Oracle-Mobile-Sync-Expires	Specifies when the returned resource will be marked as expired. Uses RFC 1123 format, for example EEE, dd MMM yyyy 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](#).

10

Location

Use the Location API to access details about location devices, places, and assets that have been registered in AMCe.

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.

A Few Important Location Terms

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. Location devices transmit an ID within a defined space, so mobile apps can use these signals to trigger notifications and other actions. 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.

Android

Querying for Location Objects

The `LocationQuery` class in the Android SDK allows you to construct queries for location devices, places and assets.

Below is an example of using the Android SDK to query for a place by name:

```
Location location =
MobileManager.getManager().getDefaultValueBackend(mActivity).getServiceProxy(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.LocationDeviceContainerQueryOrderByAttributeType
    .LocationDeviceContainerQueryOrderByName);
query.setFormat(LocationObjectQuery.LocationObjectQueryFormatType.LocationObjectQueryFormatTypeShort);

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 a Location Object

Here's how to use a place ID to retrieve the properties for the place:

```
Location location =
MobileManager.getManager().getDefaultValueBackend(mActivity).getServiceProxy(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.LocationDeviceContainerQueryOrderByAttributeType
    .LocationDeviceContainerQueryOrderByAttributeName);
query.setFormat(LocationObjectQuery.LocationObjectQueryFormatType.LocationObjectQueryFormatTypeShort);

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.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());
```

iOS

Querying for Location Objects

The OMCLocationQuery class in the iOS SDK allows you to construct queries for location devices, places and assets.

Below is an example of using the iOS SDK to query for a place by name.

```
OMCLocation* location = [[OMCMobileManager sharedManager] 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](#).

Retrieving a Location Object

This example uses the place ID to retrieve the properties for the place:

```
OMCLocation* location = [[OMCMobileManager sharedManager] location];

// query for all places
// sort results by name, in ascending order
// results will be in "short" format
OMCLocationPlaceQuery* query = [location buildPlaceQuery];
query.orderByAttribute =
OMCLocationDeviceContainerQueryOrderByAttributeName;
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);
```

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);
```

Retrieving iBeacon Identifiers

The first step to monitoring a place that uses beacons is to retrieve the beacon identifiers, as shown in this example:

```
CLLocationManager *locationManager = [[CLLocationManager alloc] init]; // iOS CoreLocation object

OMCLocation* location = [[OMCMobileManager sharedManager] 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 =
OMCLocationDeviceContainerQueryOrderByAttributeName;
queryPlace.orderByOrder = OMClocationObjectQueryOrderByOrderTypeAscending;

[queryPlace
executeWithCompletionHandler:^(OMCLocationObjectQueryResult<OMCLocationPlaceQueryResult<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
}];
}];
```

Defining a Geofence

You can use a geofence to define a monitoring area as a place, as shown here:

```
CLLocationManager *locationManager = [[CLLocationManager alloc] init]; // iOS CoreLocation object

OMCLocation* location = [[OMCMobileManager sharedManager]
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 =
OMCLocationDeviceContainerQueryOrderByAttributeName;
queryPlace.orderByOrder = OMCLocationObjectQueryOrderByOrderTypeAscending;

[queryPlace
executeWithCompletionHandler:^(OMCLocationObjectQueryResult<OMCLocationPlaceQuery *, 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
];
];
};
```

Retrieving Custom Attributes

Many location objects use custom attributes. The iOS SDK makes it easy to access these properties, as shown in the examples below.

Retrieving a Custom Attribute for a Place

The iOS SDK example below retrieves a custom attribute for a place:

```
CLLocationManager *locationManager = [[CLLocationManager alloc] init]; // iOS CoreLocation object

OMCLocation* location = [[OMCMobileManager sharedManager]
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 =
OMCLocationDeviceContainerQueryOrderByAttributeName;
queryPlace.orderByOrder = OMCLocationObjectQueryOrderByOrderTypeAscending;

[queryPlace
executeWithCompletionHandler:^(OMCLocationObjectQueryResult<OMCLocationPlaceQueryResult *> * queryResult, NSError * queryError) {
    OMCLocationPlace *place = queryResult.items.firstObject;

    NSString *myCustomProperty = [place
attributeForKey:@"MyCustomProperty"];
    NSLog(@"%@", myCustomProperty);
}];
```

Retrieving a Custom Attribute for a Location Device

The iOS SDK example below is very similar to the one above, but uses `OMCLocationDevice` to retrieve a custom attribute for a beacon:

```
OMCLocation* location = [[OMCMobileManager sharedManager] 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<OMCLocationDeviceQuery *, OMCLocationDevice *>* queryResult, NSError * queryError) {
    OMCLocationDevice *device = queryResult.items.firstObject;

    // Retrieve device/beacon custom property
    NSString *customProperty = (NSString *) [device
attributeForKey:@"MyCustomProperty"];
}];
```

REST API - Location

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 one of the SDKs 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
<code>name</code>	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): <ul style="list-style-type: none"> • 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: <ul style="list-style-type: none"> • :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 100. 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*itFast 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.
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: <ul style="list-style-type: none"> • :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 100. 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": 37.98560003
            }
          ]
        }
      }
    }
  ]
}
```

```
        "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.
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.

Parameter	Description
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 <code>gpsPoint</code> (latitude, longitude) to return the closest asset. Can't be combined with other parameters.
inGeoFence	Specify a <code>gpsCircle</code> (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: <ul style="list-style-type: none"> • <code>:asc</code> for ascending • <code>:desc</code> for descending For example, <code>name:asc</code> .
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 100. Generally meant to be used with <code>offset</code> for pagination.
format	By default, the response is in <code>long</code> 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 <code>short</code> 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",
      "label": "bed",
      "attributes": {
        "EquipmentManufacturer": "Example Company"
      }
    }
  ]
}
```

```
        "name": "hospital bed #233",
        "label": "hospital bed",
        "description": "model 1225 hospital bed",
        "lastKnownLocation": {
            "placeId": 244
        },
        "devices": [
            3409
        ],
        "attributes": {
            "EquipmentManufacturer": "Example Company",
            "SJIId": "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",
            "SJIId": "6754843090"
        }
    }
],
"totalResults": 2,
"offset": 0,
"limit": 100,
"count": 2,
"hasMore": false
}
```

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.

Setting Up Location Devices, Places and Assets

To set up a location in AMCe, define the related places and/or assets and register the associated location devices in the UI under **Development>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](#).

1. Click  to open the side menu and select **Development > 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;lat3:1
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](#).

1. Click  to open the side menu and select **Development > 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 **Development >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,#instance,#url,#list of Attributes
Name,Description,uuid,major,minor,id1,id2,id3,namespace,instance,url,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 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,#instance,#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.

11

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 **Development > APIs**. In the Platform APIs section at the bottom of the APIs page, click **Database Management**.

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.

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": "lynnnf|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.

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 <code>id</code> , 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.

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}]}
```

Executing SQL on a Table

If neither `delete()`, `get()`, `getAll()`, `insert()`, nor `merge()` let you perform the database operation that you need to do, then use the `sql` method.

The `sql` method lets you execute SQL statements such as `insert()`, `update()`, `merge()`, `delete()`, or `select()`. You can use this method for complex actions, such as when you need to join tables, use aggregate functions like `count()` and `sum()`, or use a `where` clause to delete a set of rows.

Note that you can't use the `sql` method to create a table or add columns to it. You must either use the `insert()` or `merge()` methods to create and restructure the table implicitly, or use the Database Management API to create and re-create it explicitly, as described in [Database Management API](#). In addition, the predefined fields are not populated automatically when you use `sql`.

To use the the `sql` method:

- Set the required `sql` argument to the SQL statement that you want to execute. For example:

```
SELECT COUNT("incidentReport") "reportCount"  
FROM "FIF_Incidents" WHERE "status" = :status
```

See [Preventing SQL Injection](#) to learn about precautions that you should take when you write the SQL statement.

- If your SQL statement takes parameters, then you need to pass them in the `bindings` argument, which is a JSON object. For example, if you use the SQL statement shown for the `sql` argument, then you would set `bindings` to `{status: 'Open'}`. If the SQL statement doesn't use parameters, then use `null` or `[]`.

Here's an example of executing a SQL statement. In this example, the `sql` argument is set to a SQL statement that counts the number of rows in the `FIF_Incidents` table with a status of `Open`.

```
/**  
 * Get Count of Open Incidents  
 */  
service.get('/mobile/custom/incidentreport/openReportCount',  
    function (req, res) {  
        req.oracleMobile.database.sql(  
            'SELECT COUNT("incidentReport") "reportCount" ' +  
            'FROM "FIF_Incidents" WHERE "status" = :status',  
            {status: 'Open'}).then(  
                function (result) {  
                    res.status(statusCode).send(result.result);  
                },  
                function (error) {  
                    res.status(statusCode).send(error.error);  
                }  
            );  
    });
```

Passing Parameters to the SQL Statement

You might want to let users specify some of the values in the SQL statement. For example, you might want your custom API to have a `GET /incidents/count` operation, which counts the number of incidents for a given status, and let the user specify which status to count by passing it as a request parameter. You use the `bindings` argument to pass the parameter to the `executeSQL()` method.

There are two ways to reference parameters in the SQL statement:

- Use the `:name` syntax to reference parameters by name. This is the preferred method. For example:

```
Select SELECT COUNT("incidentReport") "reportCount"  
      FROM "FIF_Incidents" WHERE "status" = :status
```

To pass the named parameter, you use a JSON object like this:

```
bindings = {status:'Open'}
```

- Use the ? syntax to reference parameters by the generic names arg1, arg2, arg3, and so on. This is called an anonymous parameter. For example:

```
Select SELECT COUNT("incidentReport") "reportCount"
      FROM "FIF_Incidents" WHERE "status" = ?
```

To pass the anonymous parameter, you use a JSON object like this:

```
bindings = {arg1:'Open'}
```

Note that unlike anonymous parameters, named parameters can be bound at multiple places in a SQL statement. In the following example, the named parameters :TITLE and :TOTAL_GROSS are bound twice: once if there's an UPDATE and once if there's an INSERT.

```
MERGE INTO "Movies" t0
  USING
    (SELECT :TITLE "TITLE" FROM DUAL) t1
  ON
    (t0."TITLE" = t1."TITLE")
  WHEN MATCHED THEN
    UPDATE SET t0."TOTAL_GROSS" = :TOTAL_GROSS
  WHEN NOT MATCHED THEN
    INSERT (t0."TITLE", t0."TOTAL_GROSS") VALUES (:TITLE, :TOTAL_GROSS)
```

If you use anonymous parameters, then you must use a different generic parameter for each occurrence. For example, with the following SQL statement, you must pass in 4 parameters: arg1 and arg3 provide the title, and arg2 and arg4 provide the total gross:

```
MERGE INTO "Movies" t0
  USING
    (SELECT ? "TITLE" FROM DUAL) t1
  ON
    (t0."TITLE" = t1."TITLE")
  WHEN MATCHED THEN
    UPDATE SET t0."TOTAL_GROSS" = ?
  WHEN NOT MATCHED THEN
    INSERT (t0."TITLE", t0."TOTAL_GROSS") VALUES (?, ?)
```

Here's an example of how to execute a SQL statement that has a parameter.

```
/**
 * Get Count of Incidents for a Given Status
 */
service.get('/mobile/custom/incidentreport/openReportCount',
  function (req, res) {
    req.oracleMobile.database.sql(
```

```
'SELECT COUNT("incidentReport") "reportCount" ' +
'FROM "FIF_Incidents" WHERE "status" = :status',
{status: 'Open'}).then(
  function (result) {
    res.status(statusCode).send(result.result);
  },
  function (error) {
    res.status(statusCode).send(error.error);
  }
);
});
```

Labeling Calculated Columns in Select Statements

As with all response bodies for this endpoint, the response body for a SELECT statement is in JSON format. To make it easier to extract a calculated value from the JSON object, always label the functions. Take, for example, the following SQL statement:

```
SELECT SUM("incidentReport") FROM "FIF_Incidents"
```

The JSON response looks like this:

```
{ "SUM(\"incidentReport\")" : 678 }
```

In this example, the function is labeled reportCount:

```
SELECT SUM("incidentReport") "reportCount" FROM "FIF_Incidents"
```

The JSON response looks like this:

```
{ "reportCount" : 678 }
```

Preserving Case in SQL Statements

By default, Oracle Database is case-insensitive. However, the tables and columns that you create using the Database Access API are case-sensitive. Therefore, you must enclose the table name, columns, and labels in SQL statements in double quotation marks ("..."). Otherwise, the call might not return any rows.

Say, for example, that your SQL statement is:

```
Select incidentReport from FIF_Incidents
```

Because the table and column names are not protected by double quotation marks, the SQL statement will not work as expected. You might get a status of 400 with a message that the table or view does not exist or that there is an invalid identifier.

Instead, use:

```
Select "incidentReport" from "FIF_Incidents"
```

Preventing SQL Injection

SQL injection is an attack technique that allows hackers access to databases by co-opting user input with a SQL block that can be interpreted by a backend database. To prevent this type of attack, you must ensure that SQL statements are never passed to the custom code from a mobile app. The SQL statements allowed by the Database Access API must reside in the custom code.

Here are some common SQL injection considerations:

- [Passing SQL as User Input](#)
- [Preventing SQL Injection with Bind Parameters](#)

Preventing Passing SQL to the Execute SQL Operation

Don't let users pass SQL into your custom code for use in the `executeSQL()` method or the `POST /mobile/platform/database/sql` operation.

For example, don't write code like the following example, which lets users put a SQL statement in the SQL header, and then pass that SQL statement to be executed. Instead, hard-code the SQL statement, and use parameters when necessary.

```
/**  
 * Example of Code that Lets  
 * Users Inject SQL  
 */  
service.get('/mobile/custom/incidentreport/openReportCount',  
    function (req, res) {  
        req.oracleMobile.database.sql(  
            req.headers.sql  
        ).then(  
            function (result) {  
                res.status(statusCode).send(result.result);  
            },  
            function (error) {  
                res.status(statusCode).send(error.error);  
            }  
        );  
    });
```

Preventing SQL Injection with Bind Parameters

In this example, a hacker can pass an escaped SQL block in the `status` field of an input entry form:

```
/**  
 * Example of Code that Lets  
 * Users Inject SQL  
 */  
service.get('/mobile/custom/incidentreport/openReportCount',  
    function (req, res) {  
        req.oracleMobile.database.sql(  
            'SELECT COUNT("incidentReport") "reportCount" ' +
```

```
'FROM "FIF_Incidents" WHERE "status" = ' +
req.body.status
).then(
  function (result) {
    res.status(statusCode).send (result.result);
  },
  function (error) {
    res.status(statusCode).send(error.error);
  }
);
});
```

To prevent this type of attack, use parameters as shown in this example:

```
service.get('/mobile/custom/incidentreport/openReportCount',
  function (req, res) {
    req.oracleMobile.database.executeSQL(
      'SELECT COUNT("incidentReport") "reportCount" ' +
      'FROM "FIF_Incidents" WHERE "status" = ' +
      ':status,
      {status: 'Open'}
    ).then(
      function (result) {
        res.status(statusCode).send (result.result);
      },
      function (error) {
        res.status(statusCode).send(error.error);
      }
    );
  });
});
```

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, and re-create tables.

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 **Development** 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 [*Oracle Autonomous Mobile Cloud Enterprise REST API Reference..*](#)

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.	<ul style="list-style-type: none"> • 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.

12

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. Click  to open the side menu and select **Development > Backends**.
2. Open the backend. (Select it and click **Open**.)
3. Click the **App Policies** tab.
4. 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 AMCE user interface. You can't do this programmatically from app code.

Android

Retrieving App Policies

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.

To fetch app policies for your Android apps 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.MobileManager
    .getManager().getMobileBackend().getAppConfig();
```

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.MobileManager.getManager()
    .getMobileBackend().getAppConfig().getString(myPolicyName, "No policy configured");
```

To load a new app policy asynchronously and make a network call:

```
mobileBackend.loadAppConfig(new AppConfigCallback() {
    @Override
    public void onResult(McsError error, AppConfig config) {
```

To return the value of a specific app policy of type `String` where `test_string` is the name of the policy if exists and returns the value of `null` if `test_string` doesn't exist:

```
String testString = config.getString("test_string", null);
```

To return the value of a specific app policy of type `Integer`, where `test_int` is the name of the policy and `0` is the value returned if `Test_int` doesn't exist:

```
int testInt = config.getInt("test_int", 0);
```

To return the value of a specific app policy of type `Boolean`, where `test_bool` is the name of the policy and `false` is the value returned if `test_bool` doesn't exist:

```
Boolean testBool = config.getBoolean("test_bool", false);
```

To return the value of a specific app policy of type Double, where test_double is the name of the policy and 0.0 is the value returned if test_double doesn't exist:

```
double testDouble = config.getDouble("test_double", 0.0);
```

To return the value of a specific app policy of type Number, where test_number is the name of the policy and 0.0 is the double value returned if test_number doesn't exist. Works for both double and single integer values. Returns exact value with which it is initialized:

```
Number testNumber = config.getNumber("test_number", 0.0);
```

To return a local copy of the app policy, and returns an empty app policy object if the app policy doesn't exist:

```
AppConfig config = mobileBackend.getAppConfig();
```

iOS

Retrieving App Policies

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.

To fetch app policies for your iOS apps for the first time, you use an asynchronous 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 = [[OMCMobileManager sharedManager] mobileBackend];

__block OMCA AppConfig* appConfig = nil;
__block NSError* error = nil;
__block BOOL executing = YES;
[_mbe appConfigWithCompletionHandler:^(OMCA AppConfig* 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];
```

REST

Retrieving App Policies

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.

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.

Cordova

Retrieving App Policies

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.

To fetch app policies for your Cordova apps, call `loadAppConfig()` on your mobile backend object, e.g.

```
mcs.mobileBackend.loadAppConfig(success, error);
```

JavaScript

Retrieving App Policies

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.

To fetch app policies , call `loadAppConfig()` on your mobile backend object:

```
mcs.mobileBackend.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. Click  to open the side menu and select **Development > Backends**.
2. Open the mobile backend. (Select it and click **Open**.)
3. Click the **App Policies** tab.
4. In the table of app policies, select the policy and click **Edit**.
5. Edit the value and click **Save**.

13

Backends

Oracle Autonomous Mobile Cloud Enterprise (AMCe) is built around the concept of mobile backends, which enable you to develop and deploy groupings of APIs that are designed to support a specific set of mobile apps or bots. You can then associate one or more apps or bots with the mobile backend to access those APIs.

What Are Backends and How Can I Use Them?

In AMCe, a backend is a secure grouping of APIs and other resources for a set of apps. Within a 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 or FCM) in the 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 AMCe, it is always in the context of a backend. The app authenticates with credentials (OAuth Consumer or HTTP Basic Authentication) specific to the backend or through an identity provider (or social login provider) that is mediated by your backend.

What's the Backend Development Process?

Generally speaking, using AMCe to develop apps involves these steps:

- Getting the AMCe [client SDK](#) for your target platform to simplify the use of platform features such notifications, storage, sync, and location.
- Developing any custom APIs that your app or bot may need.
- Creating a backend and populating it with any APIs and other resources that the app or bot will need.
- Configuring apps and/or bots to connect to the backend.

The development model is flexible, allowing you to largely work in parallel on APIs, backends, apps, and bots.

Creating and Populating Backends

You create and populate backends directly in AMCe. Once you have created a backend, you can:

- Assign roles that can be used to access the backend.
- Associate APIs and Storage collections with the backend.
- Set up profiles for the apps that will use the backend.

Creating a Backend

You create a backend to serve as a secure gateway between your app and AMCe features, such as platform and custom APIs. For your app to access these resources, it authenticates with a backend.

1. Click  to open the side menu and select **Development > Backends**.
2. Click **New Backend**.
3. Once you complete the dialog and the backend is created, keep the Settings page open.

You'll need to configure your app with some of this information.

Backend Authentication and Connection Info

The following authentication and connection details are generated when you create a backend and are displayed on the backend's **Settings** page:

- **Access Keys**

You can use these to control access to the backend. They are unique for each backend.

- **OAuth Consumer** keys are generated in the form of a client ID and a client secret.
- **HTTP Basic Authentication** keys are generated for you in the form of a backend ID and an anonymous key.

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 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.

- **Environment URLs**

- The **Base URL** is needed for all API calls. This URL is unique for each instance that you have provisioned.
- The **OAuth Token Endpoint** is the URL that your app needs to make OAuth token requests.
- The **OAuth Authorize Endpoint** is the URL that your app can use to get an authorization code to exchange for an OAuth access token.

 **Note:**

You can also find the environment URLs and some other useful data by clicking  to open the side menu, selecting **Development**, and clicking **Instance Details**.

For details on using these and other authentication methods, see [Authentication in AMCE](#).

To make it easier to incorporate these details in your apps, use the client SDKs for your app platforms. See [Client SDKs](#).

Role-Based Backends

You provide an additional layer of security for a backend (and, by extension, the resources it represents) by making it role-based and then designating user roles that enable access.

1. Click  to open the side menu and select **Development > Backends**.
2. Open the backend that you want to make role-based.
3. In the left navigation for the backend, click **Security**.
4. Set the **Role-based Access** switch to the ON position.
5. In the **Roles** field, select any roles that you want to associate with the backend.

See [Mobile Users and Roles](#) for info on how roles work.

 **Note:**

If your app uses Facebook login, **Role-based Access** needs to be turned OFF.

Associating APIs with a Backend

Once you have a backend, you can use the API Catalog to select the custom APIs you want to access through that 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. Click  to open the side menu and select **Development > Backends**.
2. Select your backend and click **Open**.
3. In the left navbar, click **APIs**.
4. Click **Select APIs**.
5. 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.

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.

6. Click the + (Add) icon for each API that you want to include.

 **Note:**

Platform APIs (for Storage, Notifications, Location, etc.) are automatically available in your backends.

Associating Storage Collections with a Backend

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

To associate your backend with an existing collection:

1. Click  to open the side menu and select **Development > Backends**.
2. Select your backend and click **Open**.
3. In the left navbar of the backend, click **Storage**.
4. Click **Select Collections**.
5. 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](#).

Notification Profiles and Client Apps

If you are using notifications in an app, you can create a notifications profile to manage the network credentials. You can then register the client app in AMCe and associate it with the notifications profile and a given backend.

Setting up a notifications profile and registering the client app accomplishes the following things:

- Enables you to store the ID that is needed for the app store.
- Enables the app to receive notifications via AMCe.
- Simplifies lifecycle management of the app and its associated backend and related artifacts.

How Notification Profiles Work

You use notifications profiles to store credentials for notification services that you use in your apps. After you create a notification profile, you can associate it with a client that you have registered for an app and associated with a backend.

How App Clients Work

Here are the principles behind app clients:

- A client 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 register 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.
- 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.

- A client can be published and deployed in a way similar to other artifacts. When a client is deployed, its backend and other dependencies are deployed with it.

For a rundown on publishing, deploying, and versioning app profiles, see App Profile Lifecycle in *Managing Oracle Autonomous Mobile Cloud Enterprise*.

Getting Network Credentials for Notifications

To enable your app to send and receive notifications via AMCe, you create an app profile and configure an associated notifications profile with the appropriate network credentials. Here's how to get the network credentials for the different platforms.

Android: Google API Key

Configuring an Android mobile app for notifications requires getting a Google API Key through Firebase Cloud Messaging (FCM).

1. Set up your Android mobile application in FCM according to the instructions at [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.

2. In the Android app's `AndroidManifest.xml` file, within the `<application>` node, add the following entries:

```
<service
    android:name="oracle.cloud.mobile.notifications.McsRegistrationIntentService"
    android:exported="false" />
<service
    android:name="oracle.cloud.mobile.notifications.GcmTokenRefreshListenerService"
    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 **API Key** are displayed. You need these credentials to register the mobile app for notifications in AMCe. 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 AMCe.

 **Note:**

It is still possible to use Google Cloud Messaging (GCM), but you should configure new apps with FCM. See [Migrate a GCM Client App for Android to Firebase Cloud Messaging](#) on Google Developers.

iOS: Apple Secure Certificates

Notifications make special use of Apple's network, so Apple wants extra security protections. In addition to the certificate used to set up your account, 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.

You need your certificate to register the mobile app for notifications in AMCe. 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 AMCe, described in [Setting Up a Device Handshake for iOS](#).

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. For further details on using notifications in iOS, see the *Local and Remote Notification Programming Guide* on <http://developer.apple.com>.

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 AMCe. Do not password protect the .p12 secure certificate. (Leave the password field blank when you save the .p12 file.)

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 AMCe. 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:
 - a. From the **Applications** page, click the gear icon next to your app name and select **Edit**.
 - b. Click **Application permissions**.
 - c. Make sure **Require user token validation** is unchecked.

- d. Click the first check box next to your app name to give your application full entitlements.
- e. Click **Auth Keys** to generate the credentials.
- f. Copy and store the keys to a safe place on your computer.
- g. Click **Save**.

Creating a Notifications Profile

You create notifications profiles to hold notification credentials that your apps need.

To create a notifications profile:

1. Click  to open the side menu and select **Development > Notification Profiles**.
2. Click **Profiles**.
3. 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), 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 UI, deselect **Consent Management Enabled**.

4. Click **Create**.

Once a notifications 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 Client

- 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.

- 2.** Click  to open the side menu and select **Development > Notification Profiles**.

- 3.** Click **Clients**.

- 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 get 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.

After you have registered the app client in AMCe, you have a few options for sending notifications to your app, as shown in [Sending Notifications to and from Your App](#).

What Can I Change in a Backend?

If you haven't yet published your backend, you can change the following parts of the backend at any time:

- Notification profiles
- Custom APIs (and their implementations)
- Any connector APIs that are called from custom API implementations
- Storage collections
- App policies

Once you have published a backend, its content is frozen. At that point, you would need to create a new version of the backend to make any changes.

 **Note:**

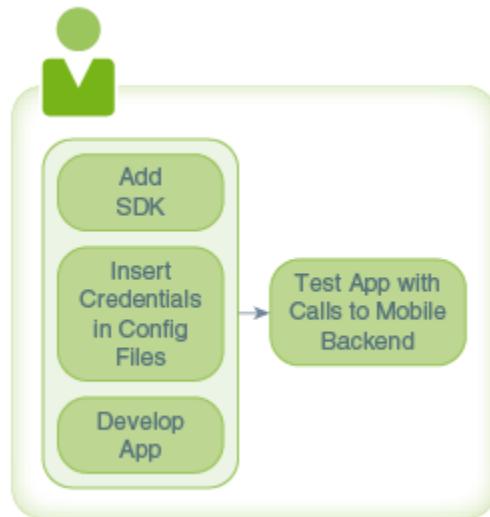
Though you can't change the list of app policies in a published backend, you can change their *values*.

Connecting Your App to a Backend

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

Connecting your app to a backend involves these basic steps:

- Adding the client 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 backend. The SDK classes that you use to make calls to the 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 AMCe APIs in your app.
The APIs available include AMCe platform APIs and any custom APIs that you or other members of your team have developed in AMCe.
- Testing your app.



Mobile Users and Roles

You can set up users for your apps in one of these ways:

- In Oracle Identity Cloud Service (IDCS).
- Through a third-party identity provider (IdP).
- By using Facebook Login.

For users provisioned in IDCS or a third-party identity provider (IdP), you can set up role-based security by doing the following:

1. Creating user roles in AMCe.
2. Applying roles to backends and APIs.
3. Assigning the roles to the users.

For details on integrating with a third-party IdP or Facebook, see [Authentication in AMCe](#).

Navigate to Your Oracle Identity Cloud Service Application

Oracle Identity Cloud Service is provided as part of your mobile service stack, and you use it to add and edit users, groups, and roles. For each mobile instance, you have an Oracle Identity Cloud Service application.

To navigate to the Oracle Identity Cloud Service application for an instance:

1. Sign in to your Oracle Cloud account.
2. From the Infrastructure Console , click the navigation menu  in the top left corner, expand **Identity**, then click **Federation**.
3. In the Instance Overview that appears, click the **Oracle Identity Cloud Service Console** link.

Adding Users and Groups in Oracle Identity Cloud Service

Unless you are using a 3rd-party IdP or Facebook as your identity store, you add users by creating user accounts with Oracle Identity Cloud Service. You can create groups to organize users and assign roles.

Note:

You must have an identity domain administrator role in Oracle Identity Cloud Service to add mobile users. If you don't have this role, ask your service administrator for help.

To add a single user, follow the steps below. Oracle Identity Cloud Service also provides a REST API for creating and managing users and groups, described in *REST API for Oracle Identity Cloud Service*.

1. From Oracle Identity Cloud Service, click  and select **Users**.
2. Click **Add**.
3. Enter the first name and last name of the user in the corresponding fields.
 - If the user is going to log in with a user name, enter the user name in the **User Name** field and enter the user's email address in the **Email** field.
Be sure to clear the **Use the email address as the user name** option, which makes the user name the same as the user's email address.
 - If the user is going to log in using an email address, make sure the **Use the email address as the user name** option is checked and enter the email address for the user account in the **User Name/Email** field.
4. Click **Next** if you want to assign the user to a group or click **Finish**.
To assign a group, just select the groups that you want to assign to this user account and click **Finish**.
5. From the Details page displayed for the new user, click the **Access** tab.
6. Search for your mobile core application and click **Assign**.
Repeat this step for each application the user should have access to.

Creating and Managing Mobile Roles

Mobile user roles allow you to define permissions for your backends and APIs. You can define as many roles as you need, and you can assign multiple roles to the same user.

To create mobile user roles:

1. In AMCe, click  to open the side menu and select **Development > Roles**.
2. Click **+ New Role** to add a role.

Once you've defined roles, use them to:

- Restrict access to a backend as explained in [Role-Based Backends](#).
- Restrict access to custom APIs as described in [Setting Access to the API](#).

Roles for Users That Are Set Up in IDCS

For mobile users that are set up in IDCS, you assign roles (to individual users or groups of users) through IDCS:

1. From the **Users** tab, click **Applications**.
2. Select your AMCe mobile core application, then select the **Application Roles** tab.
3. For each role, click **Action > Assign Users**. Select one or more users from the Role window and click **Assign**.

Roles for Users That Are Set Up in a 3rd-Party IdP

There are several ways to assign roles to users who provisioned in 3rd-party IdP. See [Associating Roles with a SAML Token](#) and [Associating Roles with a JWT Token](#).

Permissions Required for Platform APIs

The types of users that can access a platform API, the way they can access it, and the roles they need to access it vary by API. Here's a quick rundown:

API	Access and Required Permissions
App Policies	<ul style="list-style-type: none"> Accessible to IDCS, virtual, and social users from both client app code (either via REST or client SDK) and custom API implementation code. For IDCS and virtual users, must have a role associated with the mobile backend if the backend is role based.
Database Access	<ul style="list-style-type: none"> Accessible to IDCS, virtual, and social users. For security reasons, you can call these operations only from custom API implementations by using the custom code SDK. You can't make direct requests from client applications. For IDCS and virtual users, must have a role associated with the mobile backend if the backend is role based.
Database Management	<ul style="list-style-type: none"> Accessible to team members with either the Administrator or Developer role.
Location	<ul style="list-style-type: none"> Accessible to IDCS, virtual, and social users from both client app code (either via REST or client SDK) and custom API implementation code. For IDCS and virtual users, must have a role associated with the mobile backend if the backend is role based.
Location Management	<ul style="list-style-type: none"> Accessible to team members with the Administrator role.
My Profile	<ul style="list-style-type: none"> Accessible to IDCS, virtual, and social users from both client app code (either via REST or client SDK) and custom API implementation code. For IDCS and virtual users, must have a role associated with the mobile backend if the backend is role based.
Notifications (device registration)	<ul style="list-style-type: none"> Accessible to IDCS, virtual, and social users from both client app code (either via REST or client SDK) and custom API implementation code. For IDCS and virtual users, must have a role associated with the mobile backend if the backend is role based.
Notifications (create, delete, and return)	<ul style="list-style-type: none"> Accessible to team members with either the Administrator or Developer role.

API	Access and Required Permissions
Storage	<ul style="list-style-type: none">• Accessible to IDCS, virtual, and social users from both client app code (either via REST or client SDK) and custom API implementation code.• For IDCS and virtual users, must have a role associated with the mobile backend if the backend is role based.• Access depends on whether the given collection is shared or isolated, whether it's listed in the <code>Security_CollectionsAnonymousAccess</code> environment policy, and whether you need READ or READ_WRITE access.

15

Authentication in AMCe

In Oracle Autonomous Mobile Cloud Enterprise (AMCe), all resources are secured and can only be accessed via API calls made by authenticated users that are authorized to access those resources. As a mobile app developer, you enable one or more authentication methods for a mobile backend, associate the APIs that you want to use with that backend, and then write app code using one of those authentication methods.

The authentication methods available are:

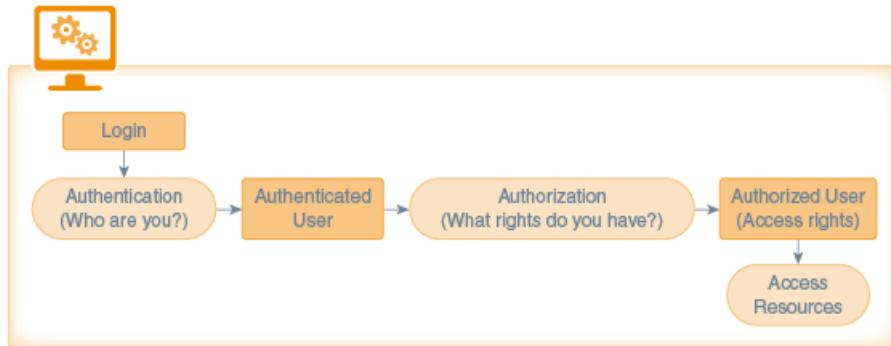
- OAuth Consumer
- HTTP Basic
- Enterprise Single Sign-On (SSO)

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 ensuring a user is who he or she claims to be, usually based on a user name and password, and often in combination with other credentials.
- **Authorization** is the process of determining whether a user has access to given backends and APIs, based on permissions granted to the user via roles.



OAuth Consumer Authentication in AMCe

The ability to use OAuth 2.0 as your authentication mechanism is built in to all backends and enabled by default. Whenever you create a backend, the OAuth Consumer keys are generated for you.

For details on the access keys and backend details provided, see [Backend Authentication and Connection Info](#).

Once you have these keys, you can use them in your apps. When using [Client SDKs](#) 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 backend. If you are coding the REST calls manually, see [Authenticating with OAuth in Direct REST Calls](#).

OAuth authentication in AMCe is handled by Oracle Identity Cloud Service (IDCS), which supports the standard OAuth grant types: authorization code, implicit, resource owner password credentials, and client credentials.

HTTP Basic Authentication in AMCe

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

To enable or disable HTTP Basic as an authentication method:

1. Open the 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.

Once you have these keys, you can use them in your apps. When using [Client SDKs](#) 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](#).

For details on the access keys and environment details provided, see [Backend Authentication and Connection Info](#).

Enterprise Single Sign-On in AMCe

If you want to use your own identity provider (IdP) for users of your apps, you can use AMCe's single sign-on (SSO) support to create a trust relationship with that IdP in AMCe so that those users from that IdP can log in to those apps. This is particularly useful if you're 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 through the following mechanisms:

- **SAML or JWT tokens from a 3rd-party IdP.** The app obtains a token from a 3rd-party IdP that is registered in AMCe as a trusted token issuer, makes an API call to the AMCe token exchange endpoint, and receives back an AMCe-issued token, which you include as a bearer token on each subsequent AMCe API call. The client SDKs support this token exchange.

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

- **Browser-based SSO using IDCS as the IdP.** Oracle Identity Cloud Service is the IdP. The app uses OAuth 2.0's authorization code grant to get an authorization token. The client SDKs currently do not provide support for this grant type.

Third-Party SAML and JWT Tokens

AMCe supports the use of SAML and JWT tokens in the following ways:

- 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 Identity Cloud Service (IDCS) and the third-party IdP. Roles are associated for the user based on information provided in IDCS.

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 AMCe 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 IDCS):

1. You configure your 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 AMCe, selecting the backend's **Settings** page, and setting the switch for HTTP Basic Authentication to ON.

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 AMCe.

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

- By designating AMCe roles to be associated with all tokens based on a given certificate.
 - By deriving role names (that match existing AMCe roles) from given token attributes.
 - By mapping given token attribute values to existing AMCe roles (where the attribute values don't already match the AMCe 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 AMCe token exchange endpoint to get an AMCe-issued token in return.
 - c. Use the AMCe token for all subsequent API calls to AMCe.

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.

2. 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 AMCe 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 AMCe, clicking its **Settings** tab, and looking in the Environment URLs section.

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

For concrete examples, see [Identity Provider Integration](#).

Registering the Token Issuer in AMCe

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 AMCe. Here are the steps:

1. In AMCe, click  and select **Settings > Credentials** from the side menu.
2. Click **Certificates**.
3. Click **New Certificate** to clear the Certificate Details and provide the following information:
 - In the **Alias** field, enter a unique identifiable name for the certificate.
 - In the **Certificate** field, paste the definition of the token certificate that was provided by the identity provider.
4. Click **Save**.
5. Wait for the token certificate to be propagated in the system. This should take no longer than 10 minutes.
6. Click the **Token Issuers** tab.
7. Click **New Token Issuer**.
8. Enter the name of the token issuer in the **Name** field under **Issuer Details**.
9. Click **Add Certificate Subject Name (+)**.
10. 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.

11. Back on the Token Issuers tab, select **Enable Virtual User**.
12. In the **Username Attribute** field, type the name of the token's attribute that identifies the user.
13. Optionally, designate user roles and mappings. The next topic has more information on how this works.
14. Click **Save**.

Associating Roles with a SAML Token

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

- Use roles already defined in the token that match the names of AMCe roles.
You do this by typing a comma-separated list of token attribute names in the **Role Attribute** field. 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 AMCe, provide a mapping between the two.

You do this by:

1. In the **Role Attribute** field, typing a comma-separated list of token attributes that contain the role names.
2. Clicking **Add Role Mapping Setting** (+) to create a mapping between a role derived from the token with one or more AMCe user roles.

You can create multiple mappings.

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

You do this by selecting one or more AMCe user roles in the **Default Roles** field.

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 AMCe token exchange to receive an AMCe 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 AMCe token.

Using a SAML Token to Authenticate with AMCe

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

AMCe'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 = MobileManager.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 AMCe token, with which you then authenticate.

1. In the app's login sequence, call the AMCe token exchange endpoint to exchange the third-party token for an AMCe-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 AMCe 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 AMCe 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 IDCS):

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

You can do this by selecting the backend in AMCe, selecting the backend's **Settings** page, and setting the switch for HTTP Basic 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 AMCe.

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

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

- By deriving role names (that match existing AMCe roles) from given token attributes.
 - By mapping given token attribute values to existing AMCe roles (where the attribute values don't already match the AMCe 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 AMCe token exchange endpoint to get an AMCe-issued token in return.
 - c. Use the AMCe token for all subsequent API calls to AMCe.

 **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 AMCe

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 AMCe. 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 Policies in Managing Oracle Autonomous Mobile Cloud Enterprise](#).

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 AMCe's virtual user (zero footprint) capability so that you don't need to have corresponding records for the user in Oracle Identity Cloud Service (IDCS).
- 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 AMCe instance (`AMCE-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 AMCe's virtual user (zero footprint) capability so that you don't need to have corresponding records for the user in Oracle Identity Cloud Service (IDCS).
- 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 AMCe'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 AMCe's virtual user (zero footprint) capability so that you don't need to have corresponding records for the user in Oracle Identity Cloud Service (IDCS).
- 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 AMCe'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 AMCe via the `Security_AuthTokenConfiguration` policy. You have the following possibilities:

- Use roles already defined in the token that match the names of AMCe 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 AMCe, 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 AMCe user roles.

You can create multiple mappings.

- Apply one or more AMCe 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 AMCe 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 AMCe 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 AMCe 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 AMCe 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 AMCe-issued token will expire at the same time the external token being exchanged will expire (i.e. `tokenTimeoutSeconds` is ignored).
 - `FromExternalTokenLimitedByTimeoutSecs` — The AMCe-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 AMCe through the Settings tab's Credentials page. (See [Registering the Token Issuer in AMCe](#).)

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 contain 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 *primary* 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 AMCe 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 AMCe 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 AMCe roles. Use this field when the values derived from role attributes do not match AMCe 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 AMCe 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 AMCe does not require this to be the case.

If a `discoveryUri` is configured for a token issuer, the AMCe 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, AMCe 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 AMCe 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, AMCe 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 `WebViewClient` 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+'

```

```
        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(mContext, 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 strings 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 AMCe-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 AMCe

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

AMCe'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 = MobileManager.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(errorCallback);
```

Coding the JWT Token Exchange Manually

Once your mobile administrator has registered an IdP as a token issuer in your instance and you have code in your app to acquire a 3rd-party token, you can use the AMCe 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 AMCe token, with which you then authenticate.

In the app's login sequence, you call the AMCe token exchange endpoint to exchange the third-party token for an AMCe-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 AMCe 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 AMCe 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 AMCe 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.abcdef0iJqb2UiLA0
  KICJleHAI0jEzMmA4MTkzODAsDQogImh0dHA6Ly9leGFtcGx1LmNvbS9pc19yb290Ijp0cnVlfQ
  .dBjftJeZ4CVP-mB92K27uhbUJU1plr_wWlgFWFOejXk",
  "token_type": "Bearer",
  "id_token": null,
  "expires_in": 28800 }
```

Mapping Users from a Third-Party IdP to IDCS Users

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

For this matching to work, the following conditions apply:

- When registering the token issuer in AMCe, 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 IDCS.
- In JWT tokens, the `sub` or `prn` attributes must identify either the user's username or email address as defined in IDCS.

User roles can be applied in any of these ways:

- Within IDCS, your administrator can assign such users the roles that are needed to access the backend and/or APIs.
- In the process of registering the IdP as a token issuer in AMCe, your administrator can specify one or more mobile roles to give to users authenticated with this IdP (via the default role rule).
- In the process of registering the IdP as a token issuer in AMCe, your administrator can create rules to map information extracted from the token (such as role names) to AMCe mobile roles (via role attribute rules).

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

See [Mobile Users and Roles](#) if you need info on setting up user accounts in IDCS.

Browser-Based SSO through AMCe

To enable browser-based single sign-on (SSO) with a third-party identity provider in AMCe, you need to first set up that identity provider in Oracle Identity Cloud Service (IDCS). To do so, you need to have an identity domain administrator role in IDCS.

See Managing Oracle Identity Cloud Service Identity Providers in *Administering Oracle Identity Cloud Service*.

To get your apps to work with browser-based SSO:

1. In AMCe, click  to open the side menu and select **Development > Backends**.
2. Open the backend you want the app to use and click its **Settings** tab.
3. Copy the **OAuth Authorize Endpoint** and **OAuth Token Endpoint** values.
4. Code your app to:
 - a. Open the OAuth authorize endpoint in a browser, redirect to the login screen of the third-party identity provider.
 - b. Upon user login, return the authorization code to the redirect URL you have specified in the identity provider.
 - c. Post the returned authorization code to the OAuth token endpoint.

The app should then receive an OAuth token in return.

Testing APIs in a Backend with SSO Login

Once you add an API to a mobile backend with SSO login enabled, you can use the API's Test page with SSO as the authentication method. This helps you ensure that the API call works end to end. You can test with the AMCe-issued SSO token or a token from a third-party provider.

To test a custom API with SSO login:

1. Click  and select **Development > 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 or Bearer Token**.
8. Obtain a valid SSO token for the mobile backend.
9. In the **Single Sign-On Token**, text field, paste the SSO token.
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 *Modifying Policies in Managing Oracle Autonomous Mobile Cloud Enterprise* for information on changing the policy.

Facebook Login in AMCe

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 AMCe platform APIs (with the exception of any APIs that are role-based).
- Register for notifications.

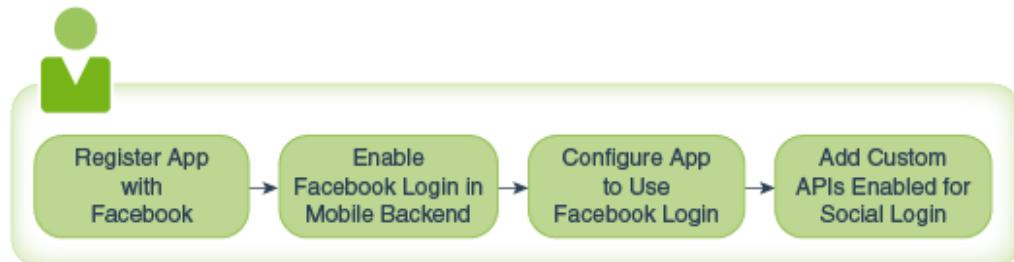
The main steps for setting up an app to use Facebook for login are:

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 AMCe.

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 AMCe, 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.)
5. Click the backend's **Security** tab and make sure that role-based access is *not* enabled. (Facebook login only works with anonymous access.)

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 AMCe 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 client SDK, which enables you to specify all of the credentials in a single configuration file. See [Client 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 AMCe platform APIs endpoints that allow anonymous access. The Analytics Collector, App Policies, Devices, AMCe, 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 **Development > 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 **Development > 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 AMCe 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 [Client 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 AMCe 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
- Base URL

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 and include scope:
 - For access without a logged-in user, use:
`grant_type=client_credentials&scope=baseURLurn:opc:resource:consumer::all`
 - For access with a logged-in user, use:
`grant_type=password&username=username&password=password&scope=baseURLurn:opc:resource:consumer::all`. 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&scope=baseURLurn:opc:resource:consumer::all"
--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...F1qFiA"}
```

- 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 AMCe client SDK, the authentication credentials are stored in one place so you don't need to manually insert them.

To authenticate with AMCe 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 AMCe 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](#).

Securing Cross-Site Requests to AMCe APIs

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

See [Oracle Autonomous Mobile Cloud Enterprise Policies](#) for a rundown of environment policies and how to use them.

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.

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?

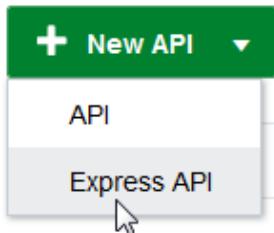
With your methods sketched in, you can then start using the API as part of your development effort by testing its endpoints and taking a look at mock data that it returns. Your service developers can implement a service for this API using JavaScript and Node. For more design and customization options, 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”.

Creating An API

1. Click the side menu (≡), choose **Development** 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 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:**

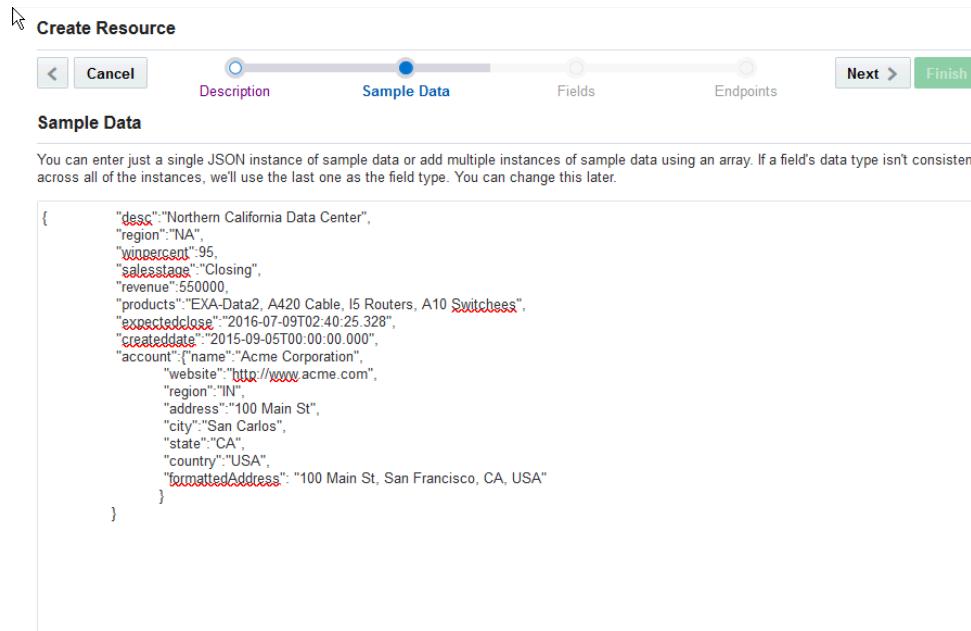
The name and description that you enter here display in the Data Palette in MAX.

When you add a resource to your API, AMCe 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 AMCe creates for you, click **Export RAML** () to download 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**, AMCe 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 AMCe 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.



- 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.

- 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.
- Click **Finish** when you're done.

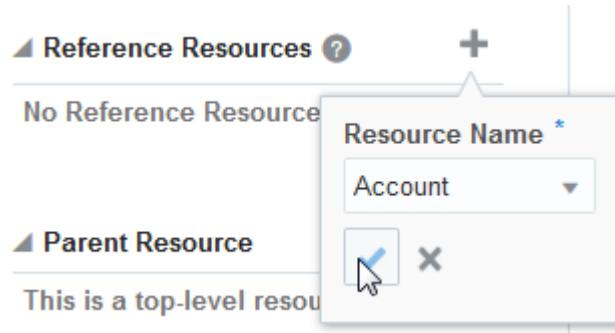
After you've created your resource, the Express API Designer opens so you can select the fields and methods you want to use to complete your resource. You can also shape request and response payloads for your methods. See [Completing Your Resources](#).

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).

Collection Methods	Single Item Methods
<input checked="" type="checkbox"/> Include resource collection Display Name (plural) * Accounts Resource Paths Single /account/{id} Collection /account/	<input type="checkbox"/> Include resource collection Display Name (plural) * Accounts Resource Paths Single /account/{id}
Methods Find Account List Accounts Create Account Edit Account Delete Account	Methods Find Account Edit Account Delete Account

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, AMCe 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](#).

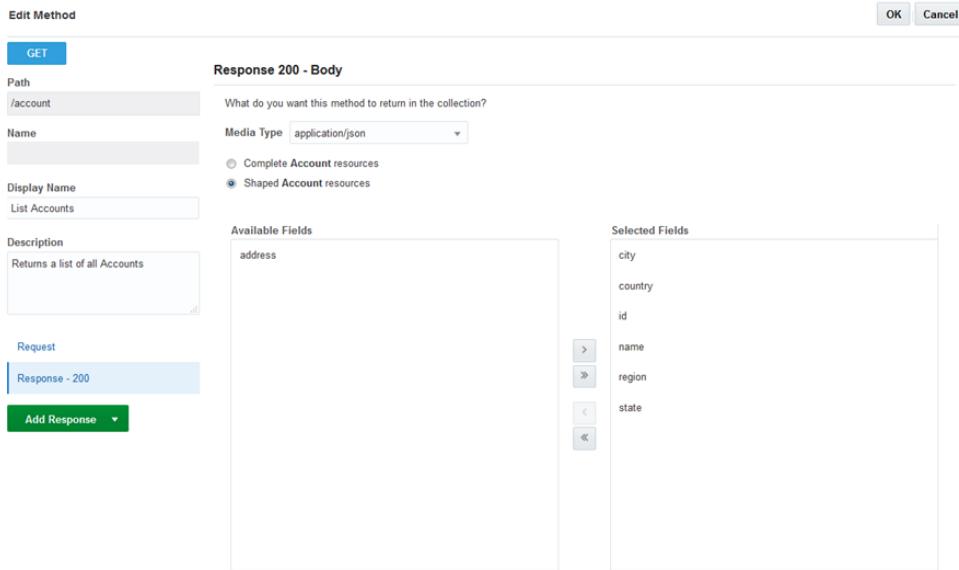
Shaping the Payload for Your Resource

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.

- 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.



- 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 AMCe 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 AMCe-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"  
  }]
```

```
        "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

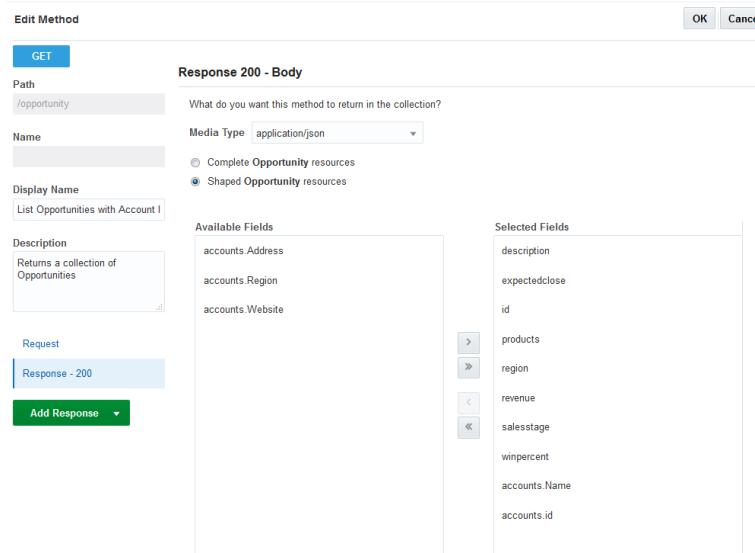
Your resources can reference each other as peers; that is, they occupy the same level. Suppose your API includes two resources that complement each other but are distinct. For example, an API that returns CRM (Customer Relationship Management) data might have two such resources: Accounts and Opportunities. 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.

The screenshot shows a user interface for managing resources. At the top, there are two tabs: 'Resources' (which is selected) and 'Endpoints'. Below the tabs is a green button labeled '+ New Resource'. The main area displays a list of resources with their descriptions:

- Opportunity: Opportunities
- Statistics: Description for statistics
- Accounts: Description for accounts
- Account Notes: Description for accountnotes

The 'Accounts' resource is currently selected, as indicated by a blue background and a dropdown arrow icon.

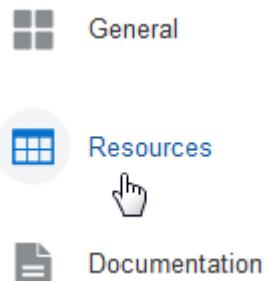
You can include the fields from a referenced resource in the payloads. When the Opportunities resource references the Accounts resource, for example, its payload for the Find Opportunities' 200 response includes `account.id` and other fields defined for the Accounts resource.



Referencing Resources

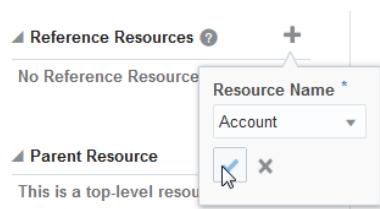
To reference a resource:

- 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. AMCe 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. AMCe 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.**

- 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.

The screenshot shows the MAX Designer interface for defining an API endpoint. On the left, there's a 'Request' section with a 'Path' field set to '/opportunity'. Below it is a 'Description' field containing 'Returns a collection of Opportunities'. To the right, under 'Response - 200', there's a 'Selected Fields' panel listing 'desc', 'id', 'region', 'revenue', 'salesstage', and 'winpercent'. A large central area is titled 'Response 200 - Body' with the sub-section 'Available Fields'. This list includes various fields like 'expectedclose', 'products', and several 'account.' prefixed fields such as 'account.address', 'account.city', and 'account.website'. There are also buttons for moving fields between the 'Available' and 'Selected' lists.

- 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.

This screenshot shows the 'Related Objects' tab for the 'Opportunity' object in the Service Catalog. On the left is a sidebar with icons for Fields, Related Objects, Actions, and Help. The main area has tabs for 'Fields' (which is active), 'Related Objects' (highlighted in blue), and 'Actions'. Under 'Child Objects', there's a list with 'Opportunity Notes' and a right-pointing arrow. Under 'Reference Objects', there's a list with 'Account' and a right-pointing arrow. Each list item has a question mark icon next to it.

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.

AMCe 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.

Type	Display Label	Name
String	nid	id

Methods

AMCe 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.

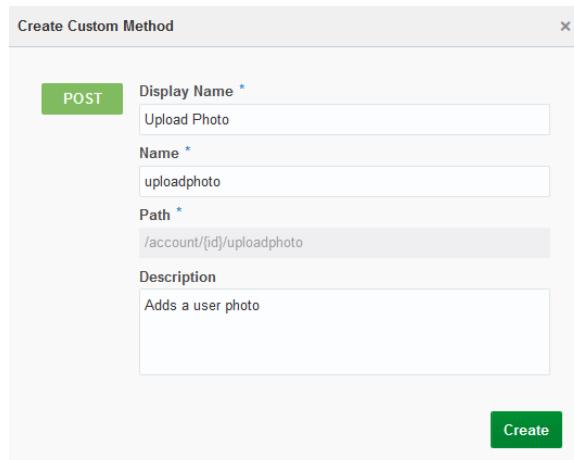
<input checked="" type="checkbox"/>	GET	/opportunity/{id}	Returns a single Opportunities
<input checked="" type="checkbox"/>	GET	/opportunity	Returns a collection of Opportunities
<input checked="" type="checkbox"/>	POST	/opportunity	Creates a new Opportunities via POST
<input checked="" type="checkbox"/>	PATCH	/opportunity/{id}	Updates an existing instance of Opportunities
<input checked="" type="checkbox"/>	DELETE	/opportunity/{id}	Delete an instance of Opportunities

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 AMCe 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 AMCe creates for you.

<input checked="" type="checkbox"/> DELETE	Delete Opportunities /opportunity/{id}	Delete an instance of Opportunities
 POST	Create Opportunity Note /opportunity/{id}/createopportunitynote	Creates a note for the specified opportunity.

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.

Path	Method	Method Name	Description
/CRM/account	GET	List Accounts	Returns a collection of Accounts
/CRM/account	POST	Create Account	Creates a new Account via POST

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, AMCe 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**.

Edit Method

GET

Path: /account

Name: List Accounts

Display Name: List Accounts

Description: Returns a collection of Accounts

Request

Response - 200

Add Response ▾

OK Cancel

Response 200 - Body

What do you want this method to return in the collection?

Media Type: application/json

Complete Account resources

Shaped Account resources

Available Fields

- address
- website

Selected Fields

- city
- country
- id
- name
- region
- state

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, AMCe 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 AMCe uses to generate the both the resource's fields and the resource itself.

The screenshot shows two side-by-side interfaces. On the left is a 'JSON Instance of Sample Data' window containing a JSON object with fields like 'desc', 'region', 'winpercent', 'salesstage', 'revenue', 'products', 'expectedclose', and 'createddate'. On the right is a 'Sample Data for the Resource' interface for an 'Opportunity' type. It has tabs for 'Overview', 'Fields', 'Methods', and 'Sample Data'. The 'Sample Data' tab is selected, showing a table with columns 'Id', 'Desc', 'Region', 'Winpercent', 'Salesstage', 'Revenue', and 'P'. A single row is present with the ID 147812149630, Desc 'Northern California Data Center', Region 'NA', Winpercent 95, Salesstage 'Closing', Revenue 550000, and P 'E'.

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 AMCe creates the `id` field for each resource, you don't need to include it your JSON.

AMCe 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, AMCe 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.

How Do I Surface My API in MAX?

After you've defined the resources, you can make your API available to MAX by publishing it. Any API that you publish for use in MAX must include resources, not endpoint definitions. Because MAX has no concept of endpoints, it can't discover them and therefore can't surface your API for MAX users. To allow mobile apps running on smartphones (via the MAX App) to access the API, you need to publish both the API and its implementation. If there are multiple environments, you must deploy the API and its implementation prior to publishing them.

 **Note:**

Take a look at [Exploring Services](#) to find out about more about the role of business objects in the MAX Designer.

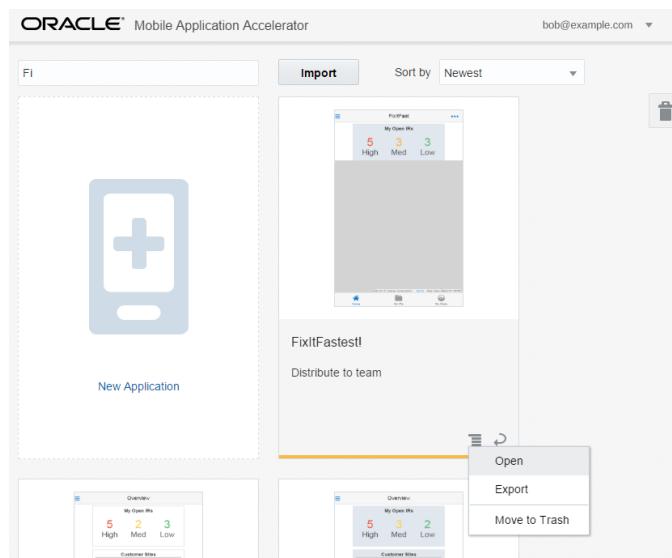
Who Uses MAX?

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 AMCe 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 AMCe 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 AMCe. Unlike Mobile Cloud developers, business users are MAX-only users: they're granted the `BusinessUser` role, so they never see AMCe (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.

The screenshot shows the Oracle MAX Express API Designer interface. On the left, there is a mobile phone-shaped preview window displaying a user interface titled "My Photos". Inside the preview, there are four cards: "From Service" (with a thumbnail of a smartphone), "Upload" (with a thumbnail of a dog), "Customer" (with a thumbnail of a woman's face), and "My Picture" (empty). Each card has an "Edit" button. On the right, the "IMAGE" tab of the data configuration panel is selected. The "Properties" tab is active. The "Data" tab is selected, showing options to either "Select image" or "Map to business object". A dropdown menu under "Source" is set to "Incident". Below this, there are sections for "Data Mapper" (with a "Map values to icons" link) and "Image Update Action" (set to "Upload Incident Picture").

Tips for User-Friendly Business Objects in MAX

You can help business users pick services and map data by adding metadata in AMCe.

The metadata that you enter in AMCe...

The API metadata entered in the General Page of the API Designer:

- API Display Name
- Short Description
- icon

The screenshot shows the Oracle Mobile Cloud Service API Designer interface. On the left, there's a sidebar with icons for General, Endpoints, Security, Schema, Types, and Traits. The General tab is selected. In the main area, there's a form for the API Catalog Properties. The fields filled in are:

- * API Display Name: FixItFast
- * API Name: ifixitfast
- Default Media Type: application/json
- * Short Description: The iFixItFast technician service.
- Icon: A wrench and gear icon.

...Is surfaced here in MAX

The service name and description in the Service Catalog:

The screenshot shows the Service Catalog in MAX. It lists various services with their names, descriptions, and versions. The FixItFast service is highlighted in blue. The other services listed are Acme4Max, CRM, FastRepairs, HCM_Employees_AP, and INGInnovation.

Service	Description	Version
Acme4Max	Custom API for Auraplayer Acme Orders System	v1.4
CRM	CRM Sample API	v1.7
FastRepairs	test api	v1.0
FixItFast	The iFixItFast technician service.	v1.2
HCM_Employees_AP	The HCM test case	v1.8
INGInnovation	ING Innovation day beacon visits	v1.5

The metadata that you enter in AMCe...

The endpoint description and display name for top-level resources entered in the Endpoints page of the API Designer:

...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:

The metadata that you enter in AMCe...

The property members of the schema definition (including the description key) that are defined for the GET response:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "id": "incident",
  "title": "Incident Detail",
  "type": "object",
  "properties": {
    "id": {
      "type": "string",
      "description": "Unique incident ticket number"
    },
    "problem": {
      "type": "string",
      "description": "Short description of the incident"
    }
  }
}
```

...Is surfaced here in MAX

- The field names and descriptions in the Fields tab of the Data Palette

The screenshot shows the MAX interface with the title 'THATTITLE' and a breadcrumb navigation bar. The 'Fields' tab is selected in the Data Palette. A search bar labeled 'Filter' is present. Below it, four fields are listed with their types and descriptions:

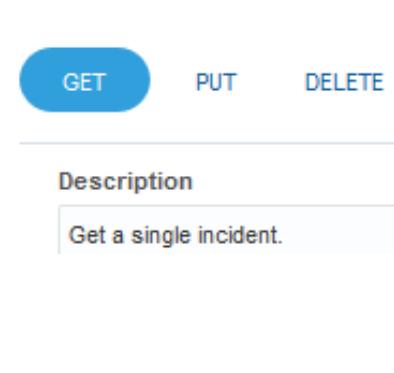
- A **id**: Unique incident ticket number
- A **problem**: Short description of the incident
- A **description**: Detailed description of the incident
- A **createdOn**: When the incident was logged

- The Data page of the Add Data QuickStart and the Data Mapper:

The screenshot shows the MAX Data Mapper interface. On the left, there are two tabs: 'DATA' and 'QUERY'. The 'DATA' tab has a sub-section titled 'Business Object' with a 'Map your UI to data' button. The 'QUERY' tab has a sub-section titled 'List Component Fields' with a 'Filter the data queried for your selected business object' button. In the center, there is a 'Cancel' button and a 'Business Object' card. To the right, a 'List Component Fields' panel is open, showing a table with columns for 'Title' and 'Field'. The table contains rows for 'id', 'lastUpdatedOn', 'priority', 'problem', 'status', and 'customer'.

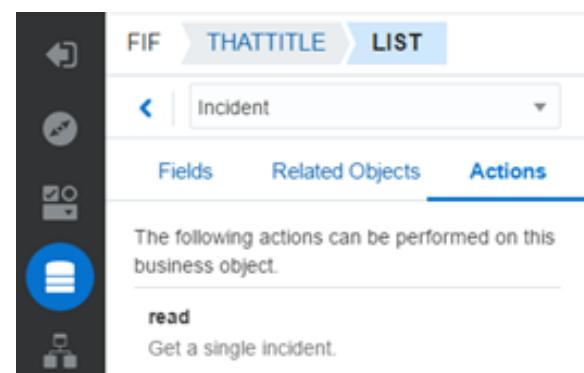
The metadata that you enter in AMCe...

The display name and description for endpoint methods entered in the Endpoints page of the API Designer:



...Is surfaced here in MAX

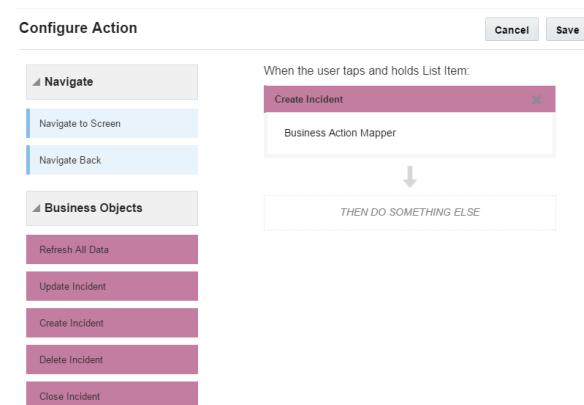
The Actions tab of the Data Palette:



The title key in the schema:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "id": "patch-incident",
  "title": "Update Incident",
  "type": "object",
  ...
}
```

The Configure Action page of the Properties Inspector:



The metadata that you enter in AMCe...

GET method Query definitions for the request parameters entered in the Endpoints page of the API Designer:

Request | Responses

Parameters (2)

Query	Technician	Technician User Name	String
<input checked="" type="checkbox"/> Required	The username of the technician or authenticated user whose incidents are retrieved. The special value "<~>" can be used to indicate the authenticated user.		
technician<~>			

...Is surfaced here in MAX

The Query page of the Add Data QuickStart and the Data Mapper:

ORACLE Mobile Application Accelerator

DATA QUERY

Map your UI to data Filter the data queried for your selected business object

Cancel <

1 Tap a category to show data values you can use.

Application User

Environment

Fixed Value

Query Parameters

Technician * User Name

FilterBy

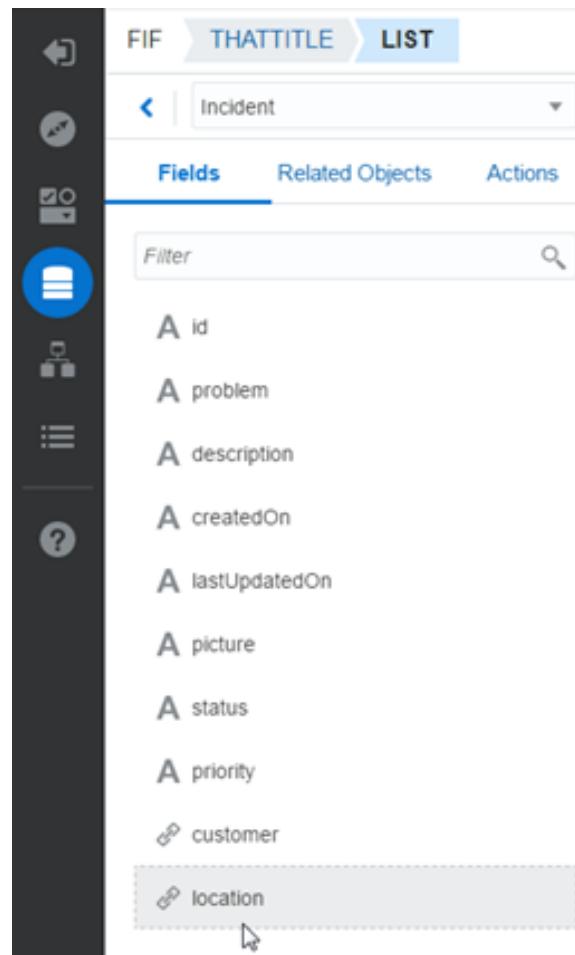
The metadata that you enter in AMCe...

The \$ref definition that point to other top-level resources in the GET response schema definition.

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "id": "incident",
  "title": "Incident Detail",
  "type": "object",
  "properties": {
    ...
    "customer": {
      "$ref": "incident-customer",
      "title": "Details of customer who logged the incident."
    },
    "location": {
      "$ref": "IncidentLocation",
      "title": "Location where the incident occurred"
    }
}
```

...is surfaced here in MAX

- 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

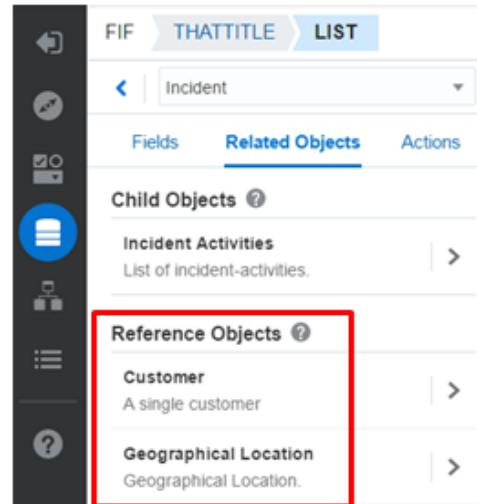
The metadata that you enter in AMCe...

The display name and description for endpoint methods entered in the Endpoints page of the API Designer:



...Is surfaced here in MAX

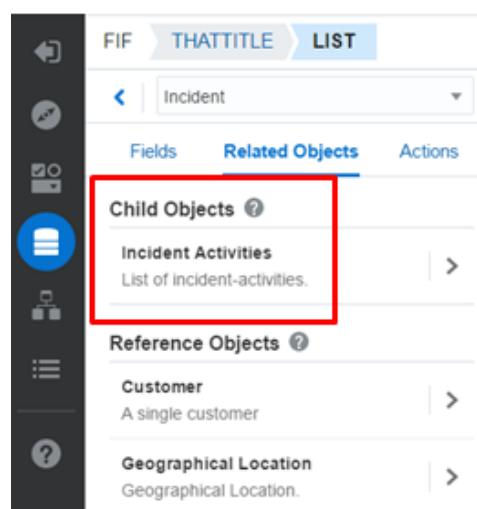
The Related Objects tab of the Data Palette (under Reference Objects):



The display name and description for nested resources that are entered in the Endpoints page of the API Designer:



The Related Objects tab of the Data Palette (under Child Objects):



The metadata that you enter in AMCe...	...Is surfaced here in MAX
<p>The property members of the schema definition (including the title key) that are defined for the GET response for a nested object.</p> <pre>{ "\$schema": "http://json-schema.org/draft-04/schema#", "id": "incident-activity", "title": "Incident Activity", "type": "object", "description": "A single activity reported on an incident.", "properties": { "incidentId": { "type": "string", "description": "Incident Identifier that this activity record belongs to" }, "firstName": { "type": "string", "description": "The first name of the person who created the activity" }, "lastName": { "type": "string", "description": "The person's last name" } } }</pre>	<p>The Data Source page Data pages of the Data Mapper and the Add Data QuickStart for a detail screen.</p>

The metadata that you enter in AMCe...

Mock data defined for requests and responses in the API Designer:

```
{
  "id": "inc-201",
  "problem": "Incident New",
  "description": "I learned that beneath my goody two shoes lie some very dark socks.",
  "createdOn": "2015-08-18",
  "lastUpdatedOn": "2015-08-20",
  "picture": "/builtin/images/broken-water-heater.png",
  "status": "open",
  "priority": "medium",
  "customer": {
    "id": "cus-101",
    "username": "julie.simpson",
    "firstName": "Julie",
    "lastName": "Simpson",
    "mobile": "6505067000",
    "home": "5105552121",
    "email": "julie.simpson@springtime.com"
  }
}
```

...Is surfaced here in MAX

- The Live Data view for both the Data Mapper and the Add Data QuickStart:

The screenshot shows the Oracle MAX Data Mapper interface. On the left, under 'List Component Fields', there are four input fields: 'Icon' (priority), 'Title' (problem), 'Subtitle' (customer.firstName), and 'Subtitle' (customer.lastName). Below these is a 'Separate With' dropdown set to 'Space'. Underneath are two more input fields: 'Value 1' (status) and 'Value 2' (lastUpdatedOn). On the right, a 'Live Data Preview' table displays three rows of incident data:

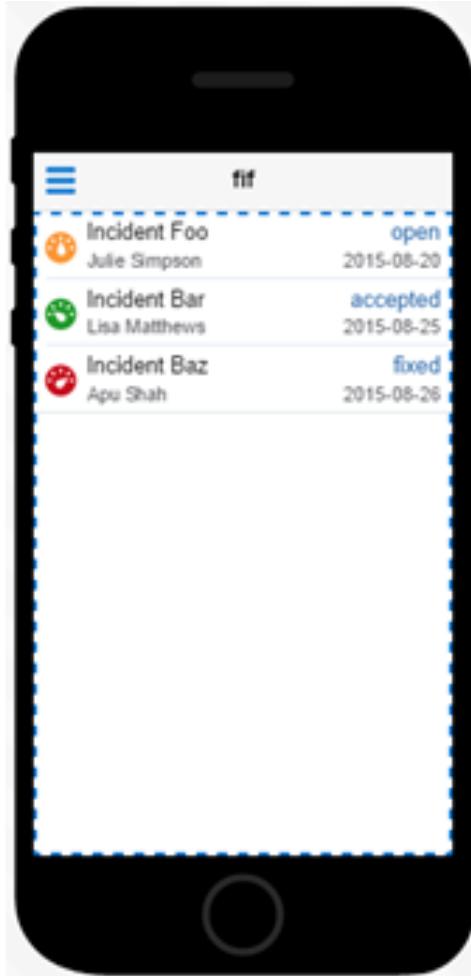
id	problem	status	lastUpdatedOn
Incident Foo	Jule Simpson	open	2015-08-20
Incident Bar	Lia Matthews	accepted	2015-08-25
Incident Baz	Apu Shan	fixed	2015-08-26

A blue button at the bottom right of the preview area is labeled 'Live Data Preview'.

- The Preview:

The metadata that you enter in AMCe...

...is surfaced here in MAX



Video: An Introduction to Mobile Application Accelerator (MAX)

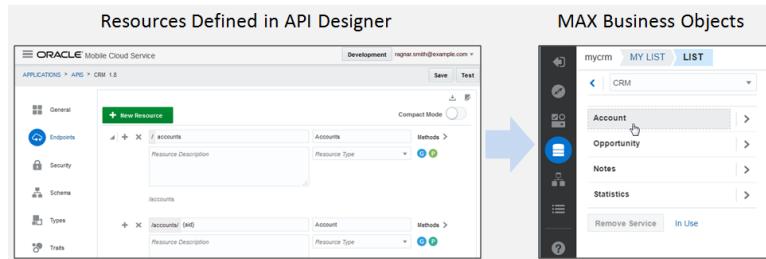
To see how you can build, test, and publish mobile apps using MAX, take a look at this video:



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.

The screenshot shows the Oracle MAX Designer interface for defining API resources. On the left, a sidebar lists categories: General, Endpoints, Security, Schema (selected), Types, Traits, Documentation, and Implementation. Under the Schema category, there are several tabs: New Schema, account, getAccount, getAccounts, getAccountOpportunities, postAccount, and patchAccount. The 'account' tab is active, displaying a JSON schema. The schema defines an 'account' type with properties like name, website, region, address, city, state, and country, each with specific types and constraints. The JSON code is numbered from 1 to 37.

```

1  {
2      "$schema": "http://json-schema.org/draft-04/schema#",
3      "id": "account",
4      "type": "object",
5      "properties": {
6          "name": {
7              "id": "name",
8              "type": "string"
9          },
10         "website": {
11             "id": "website",
12             "type": "string"
13         },
14         "region": {
15             "id": "region",
16             "type": "string",
17             "enum": ["IN", "NA", "SA", "AP", "EU"]
18         },
19         "address": {
20             "id": "address",
21             "type": "string"
22         },
23         "city": {
24             "id": "city",
25             "type": "string"
26         },
27         "state": {
28             "id": "state",
29             "type": "string"
30         },
31         "country": {
32             "id": "country",
33             "type": "string"
34         }
35     }
36 }
37

```

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`, `minlength`, `maxLength`, `minimum`, and `maximum` to the property:

```

{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "id": "postOpportunity",
    "type": "object",
    "allOf": [
        {"$ref": "opportunity"}
    ],
    "required": [
        "desc",
        "name"
    ]
}

```

```
        "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 16-1 Taking a Look at Properties in the JSON Schema

In the following example, a schema called `account` that defines 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.

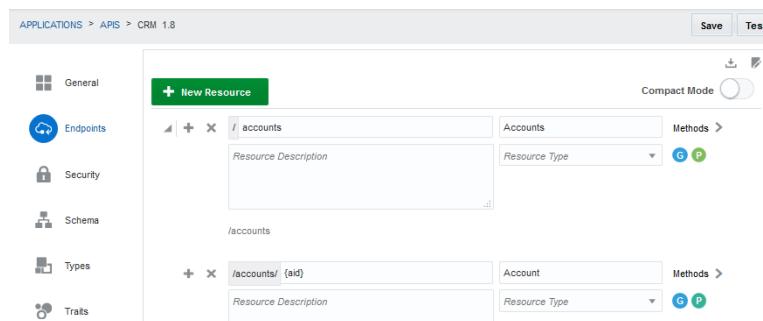
Field Behavior	Description	Used in These MAX Components	Method	Tips
Collection	Returns multiple items (or records) of the object. Calls GET on the collection resource (/items) to return all fields. See Collection Actions .	<ul style="list-style-type: none"> List Components DVT Components 	GET	Specify 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, technician 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": {
      "type": "string",
      "format": "name"
    },
    "region": {
      "type": "string",
      "format": "region"
    },
    "description": {
      "type": "string",
      "format": "description"
    }
  },
  "required": [
    "name",
    "region"
  ]
}
```

```
"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:

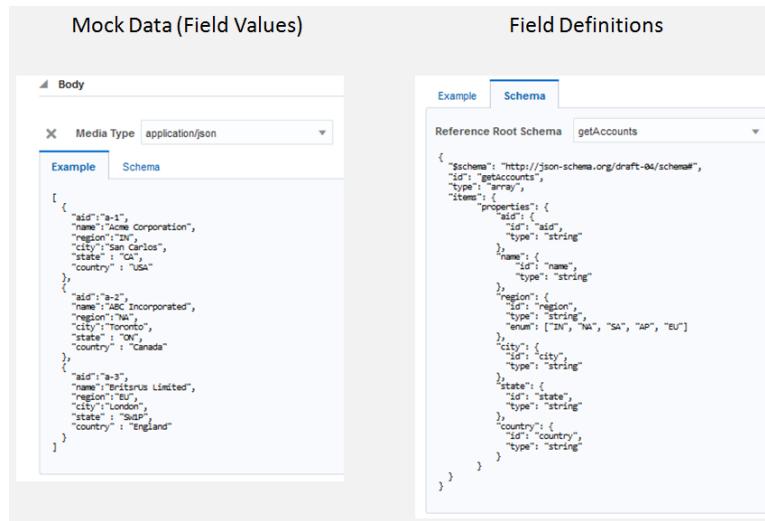
```
{  
    "$schema": "http://json-schema.org/draft-04/schema#",  
    "id": "incident",  
    "title": "Incident Detail",  
    "type": "object",  
    "properties": {...},  
    "links": [  
        {  
            "rel": "self",  
            "title": "Incident",  
            "href": "/incidents/{id}",  
            "method": "GET",  
            "targetSchema": {"$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 AMCe doesn't verify mock data against a schema.



The screenshot shows two panels of the Oracle API Designer. On the left, the 'Mock Data (Field Values)' panel displays a JSON array of three objects representing accounts. Each object has fields like 'id', 'name', 'region', 'city', 'state', and 'country'. On the right, the 'Field Definitions' panel shows the corresponding JSON schema. The schema defines a root object with an array type, where each item is an account object. This schema maps fields to types such as string, integer, and enum for region and state.

```

Mock Data (Field Values)
Field Definitions

{
  "Body": {
    "Media Type": "application/json",
    "Example": [
      {
        "id": "a-1",
        "name": "Acme Corporation",
        "region": "IN",
        "city": "San Carlos",
        "state": "CA",
        "country": "USA"
      },
      {
        "id": "a-2",
        "name": "ABC Incorporated",
        "region": "NA",
        "city": "New York",
        "state": "NY",
        "country": "Canada"
      },
      {
        "id": "a-3",
        "name": "Oracles Limited",
        "region": "EU",
        "city": "London",
        "state": "UKP",
        "country": "England"
      }
    ]
  },
  "Schema": {
    "Reference Root Schema": "getAccounts",
    "Schema": {
      "$schema": "http://json-schema.org/draft-04/schema",
      "id": "getAccounts",
      "type": "array",
      "items": [
        {
          "properties": {
            "id": {
              "type": "string"
            },
            "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"
            }
          }
        }
      ]
    }
  }
}

```

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?	<ul style="list-style-type: none"> 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 <p>The focus is on flexibility and control of the development process.</p>	<ul style="list-style-type: none"> 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). <p>The focus is on speed, creating a spec to export to the API Designer for further development, and creating APIs to use with MAX.</p>
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.

Custom API Design

In Oracle Autonomous Mobile Cloud Enterprise (AMCe), 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.

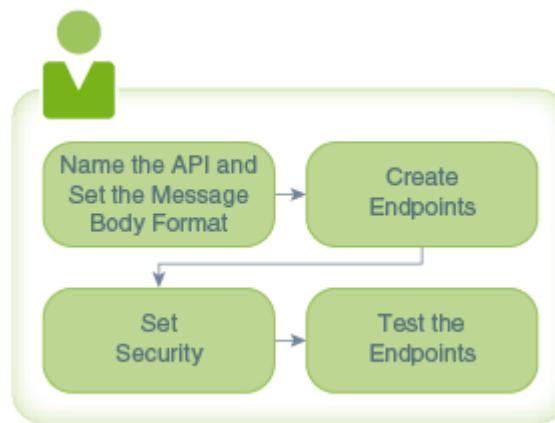
Unlike the AMCe platform APIs, which provide a core set of known services, custom APIs let you use Node.js to code any service your mobile app needs, published through a REST interface. You can relay data by using an AMCe connector to a backend service, which transforms complex data into mobile-friendly 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 AMCe 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

The screenshot shows the Oracle API Designer interface. On the left, there's a sidebar with icons for General, Endpoints, Security, Schema, Types, Traits, Documentation, Implementation, and API Catalog. The 'General' icon is selected. The main area displays a confirmation message: "FIFIncidentReports version 1.0 has been created and is ready for you to start designing it." Below this, a "Suggested next steps:" section lists: "Take a tour of the API Designer," "Learn more about recommended best practices when designing RESTful APIs for mobile," and "Learn more about RAML, the definition language we're using for your API." The main configuration area shows the following details for the API:

- API Display Name:** FIFIncidentReports
- API Name:** incidentreport
- Base URL:** <http://cloud.oracle.com/mobile/custom/incidentreport>
- Default Media Type:** application/json

 A tooltip for the "API Catalog Properties" section explains: "To help familiarize application developers with the published APIs in the catalog, briefly describe the purpose of the API and associate an icon with it. The description and icon image are not saved in the API's generated RAML document, and are not displayed in the Source view." In the "Short Description" field, the text "Customer reports" is entered, with 84 characters remaining. Below the text input is a "Select" button next to a wrench and gear icon, used for selecting an icon.

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.

Generating Custom APIs for Connectors

Oracle Autonomous Mobile Cloud Enterprise (AMCe) 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. Click  and select **Development > 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.

2. Click **More** and from the drop-down list, select **Generate Custom API**.

The Generate Custom API dialog appears.

Generate Custom API

Generate a custom code API, including a default implementation, exposing the methods of your connector.

* Title <API Title>

* Version <Version string>

* Name <APIName>

/mobile/custom/<APIName>/

Description Generated from connector RESTConnCP 1.0.

100 characters left

Generate

3. 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 **Development > 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 passes through all requests. You can edit the implementation to shape the data returned, which is useful if there is a lot of data.

1. Click  and select **Development > 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.

2. Click the **Implementations** navigation link, select the implementation which will have the same name as the custom API, and click **Download**.
3. 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 AMCe command-line development tools.
4. 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
                body += ' customized request';
            } else if (typeof body == 'object'){
                if (Buffer.isBuffer(body)){
                    // to enable binary parsing uncomment
                    body = Buffer.concat([Buffer.alloc(8, '00000000'),
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
                        body = Buffer.concat([Buffer.alloc(8, '00000000'),
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;
        }
    }
}
```

```
 }*/  
,
```

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](#) in *Managing Oracle Autonomous Mobile Cloud Enterprise*).

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.

AMCe APIs are based on the RESTful API Modeling Language (RAML) standard. Once you've begun to configure your API, AMCe generates a RAML document of the configuration. See [RAML](#) to learn more about it.

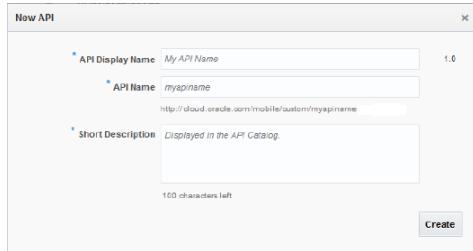
If you want to work on the RAML document outside of AMCe, you can export it by clicking **Export RAML document**  at the top of the page.

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. Click  and select **Development > Mobile Backends** from the side menu.
2. Select the mobile backend that you want to associate the API with from the list of backends and click **Open**.
3. Click the **APIs** navigation link.
4. Select **New API > API**.

The New API dialog opens. Here's where you enter the basic information for your API:



Select **API** to craft a custom API with the API Designer. Or choose **Express API** to open the Express API Designer to quickly create a no-code API using sample data you provide. See [Which API Designer Should I Use?](#) to learn about the difference between the two designers.

- 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.

5. Click **Create**.

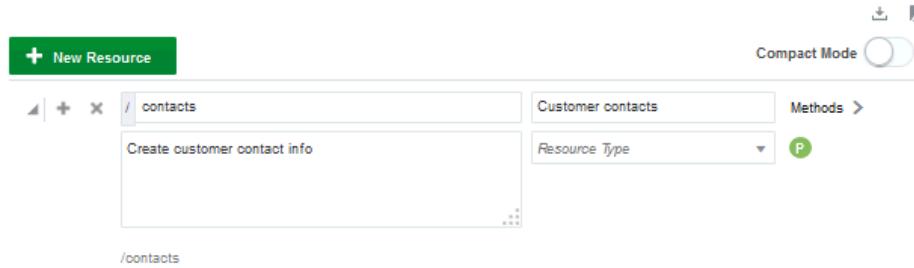
The General page of the API Designer is displayed. If you want to change the name of your API or its description, then you can do it here.

6. 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.

7. 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.

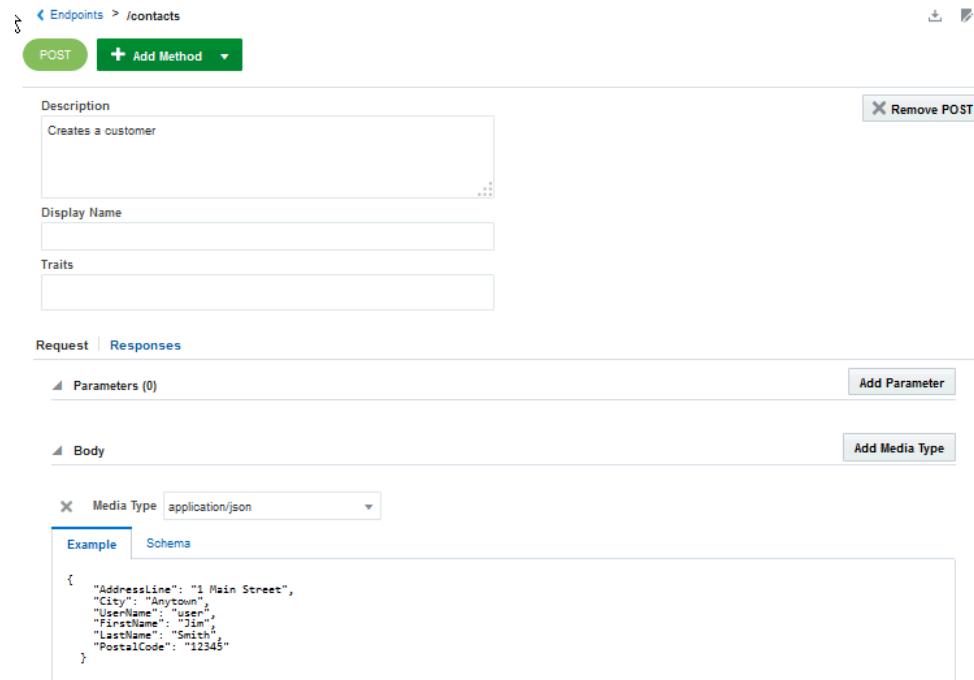
- (Optional) If you want to add a nested resource (a child resource of `contacts`), click **Add (+)** next to the **Resource** name field. Enter a name, a display name, and a description of the nested resource. Click **Add (+)** again to add more nested resources if you need them.

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](#).

- 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 FixItFast example:

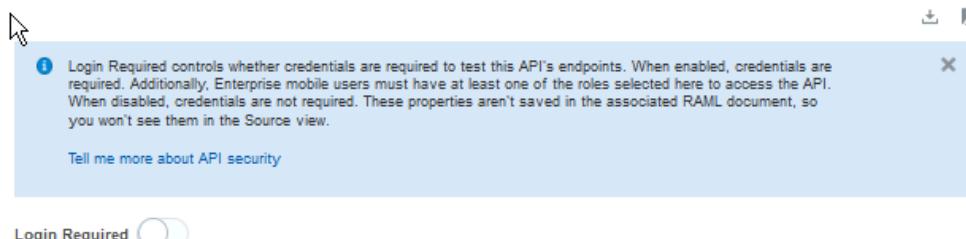
```
{
    "AddressLine": "1 Main Street",
    "City": "Anytown",
    "UserName": "user",
    "FirstName": "Jim",
    "LastName": "Smith",
    "PostalCode": "12345"
}
```

- 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**.
9. Set security access for your API by clicking **Endpoints** to get back to the Endpoints page. From there, click the **Security** navigation link.
10. 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.

- 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.

Endpoints (1) Documentation

Filter endpoints

Customer contacts

POST /contacts

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.

FIFIncidentReports 1.0

Customer reports

Base URI | <http://cloud.oracle.com/mobile/custom/incidentreport>

Endpoints (6) Documentation

Filter endpoints

Customer contacts

POST /contacts

Creates a customer

Request **Response**

Parameters	Description	Test Console
BODY	string	
body	string	<input type="button" value="Use Example"/>

```
{
  "AddressLine": "1 Main Street",
  "City": "Anytown",
  "UserName": "user",
  "FirstName": "Jim",
  "LastName": "Smith",
  "PostalCode": "12345"
}
```

Authentication

Provide the mobile backend, version, and mobile user credentials to test this API. When applicable, you must also specify an authentication method.

* Mobile Backend

* Version

* Authentication Method

- 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.

13. 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 **Development > 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. Click  and select **Development > APIs** from the side menu.
2. 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.

3. 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.

4. 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.

5. 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.

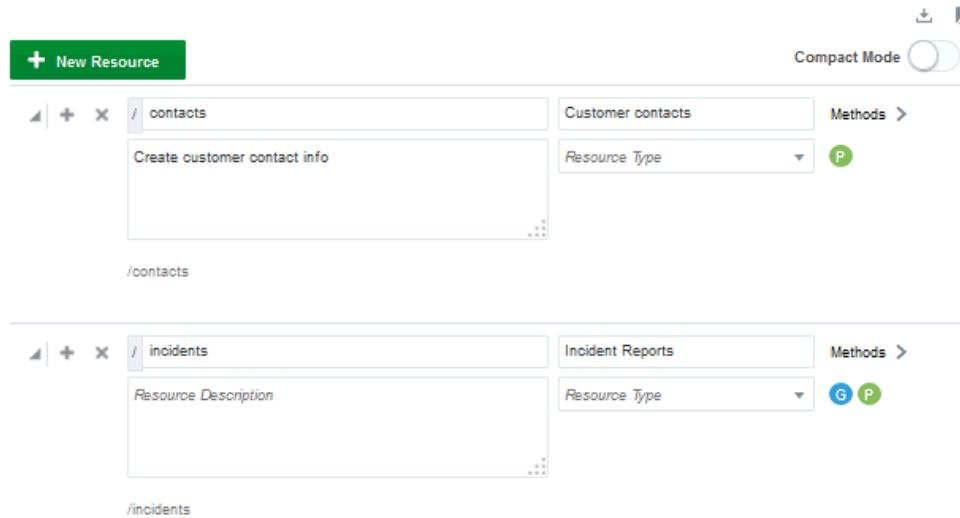
Now that you've provided the basic information, it's time to define endpoints for your API.

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 defined 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.

- 2xx indicates a successful connection
- 3xx indicates a redirection occurred
- 4xx indicates a user error occurred
- 5xx 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": "alalal"
        "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": { "description": "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" : "Postalcode
                                         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 AMCe, 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 a remote identity provider. For information on how to enable single sign-on for a mobile backend, see [Authentication in AMCe](#).

- 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 AMCe 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 AMCe by specifying the social identity provider and providing the access token.

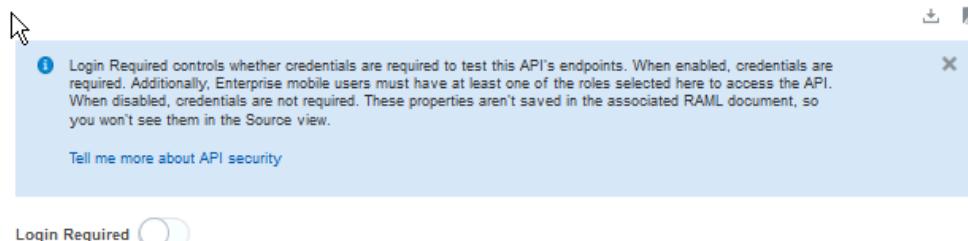
To find out how to get an access token, see [Getting a Facebook User Access Token Manually](#).

To learn about authentication in AMCe, see [Enterprise Single Sign-On in AMCe](#).

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:

Login Required **Enterprise** **Social Identity**

API Access

Select the roles that can access this API.

Roles

Endpoint Access

Optional: grant selected role access to individual endpoints. Endpoint-specific settings override the API-level settings.

Filter endpoints

	POST Customer contacts /contacts	Click to assign one or more roles at the endpoint level.
--	----------------------------------	--

- Select **Enterprise** to set access for mobile users who login with their AMCE 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.

See [Mobile Users and Roles](#) for information on how roles can be defined.

- 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 via the /users/me REST call or the `ums.getMe()` method in the custom code for the API. See [Accessing the My Profile 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: **Current User**, **HTTP Basic**, **OAuth Consumer**, **Social**, or **Single-Sign On or Bearer Token**.

- 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 AMCe.
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 or Bearer Token** as the authentication method and enter either the AMCe-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 [Enterprise Single Sign-On in AMCe](#).
 - If **Social Identity** is selected, select a social authentication provider and enter the access token that you got from your provider.

 **Note:**

AMCe 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 in *Managing Oracle Autonomous Mobile Cloud Enterprise*.

To learn how to get a Single Sign-On OAuth token, see [Enterprise Single Sign-On in AMCe](#).

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:

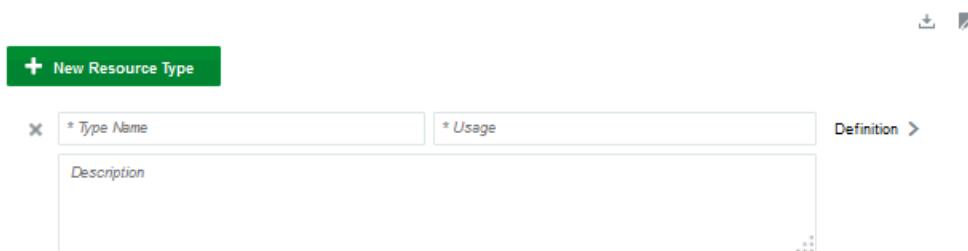
Resource types can't be used with nested resources.

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:

The screenshot shows a 'New Trait' form. At the top left is a green button labeled '+ New Trait'. Below it are three input fields: 'Trait Name' (with placeholder '* Trait Name'), 'Usage' (with placeholder '* Usage'), and 'Description' (with placeholder 'Description'). To the right of the 'Description' field is a 'Definition' button with a right-pointing arrow. The entire form is set against a light blue background.

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:	#First-Level Heading	First-Level Heading
Use hash marks (#) to denote headers	## Second-Level Heading	Second-Level Heading
	### Third-Level Heading	Third-Level Heading
Paragraph:	This is a paragraph.	This is a paragraph
Separate paragraphs with one or more blank lines.	This is a second paragraph.	This is a second paragraph.
Simple List:	- list item 1	- list item 1
Use +, -, or * followed by a space to denote list items.	+ list item 2	- list item 2
	* list item 3	- list item 3
List markers are interchangeable.		
Nested List:	-list item 1	- list item1
Use +, -, or * followed by a space to denote list items and indent nested list item by exactly four spaces.	+ list item 1a	- list item 1a
	+ list item 1b	- list item 1b
	-list item 2	- list item 2
Ordered List:	1. list item 1	1. list item 1
Precede each item with a number in a consecutive sequence followed by a space.	2. list item 2	2. list item 2
	* list item 2a	2a. list item 2a
	* list item 2b	2b. list item 2b
	3. list item 3	3. list item 3
Emphasis Italics:	*text*	text
Wrap text with an asterisk (*) or single underscore.	_more text_	more text
Emphasis Bold:	**text**	text
Wrap text with two asterisks (*) or double underscores.	__more text__	more text
Inline code:	This is an `inline code` example.	This is an inline code example.
Use back quotes (`) around the text.		
Code Block:	Format a block of preformatted code:	Format a block of preformatted code:
Indent each line by four spaces	This is a code line.	This is a code line.
Links:	This is an [example link](http://example.com).	This is an example link.
Put the link text in brackets, followed immediately by the URL in parentheses.		

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 2xx doesn't necessarily mean that the test failed. If the operation was supposed to return a null response, then the response should show a 4xx code.

For every message you send, AMCe 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 AMCe 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 [AMCE Policies and Values](#). For information about environment policies in general, see Policies in *Managing Oracle Autonomous Mobile Cloud Enterprise*.

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: FIFIIncidentReports
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-incidentreport` 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": { "description": "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" : "Postalcode  
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 AMCe 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/api_name), 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:**

AMCe 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, FIFIIncident 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, AMCe 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. Click  and select **Development > APIs** from the side menu.

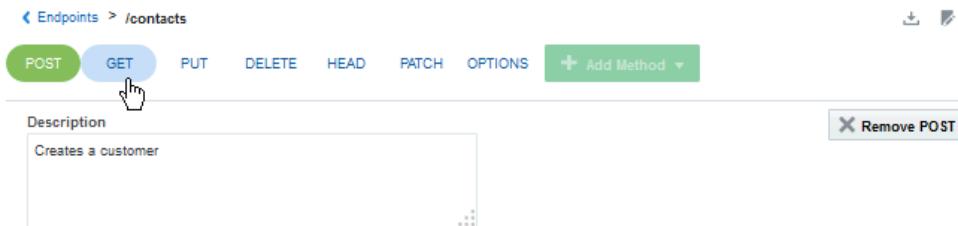
Now that at least one custom API exists, the APIs page is displayed.

2. Select the draft API that you want to edit and click **Open**.

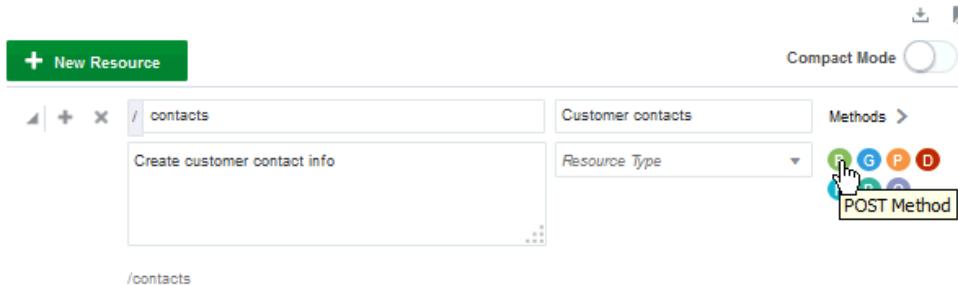
You can filter the list by version number or status. You can also sort the list alphabetically by name or by last modified date.

3. 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.

4. Save your changes if you didn't select the option to always save the configuration before testing when you created the API.

5. 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 in *Managing Oracle Autonomous Mobile Cloud Enterprise*. If you need to make a change to a published API, you'll have to create a new version of it.

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](#).

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.

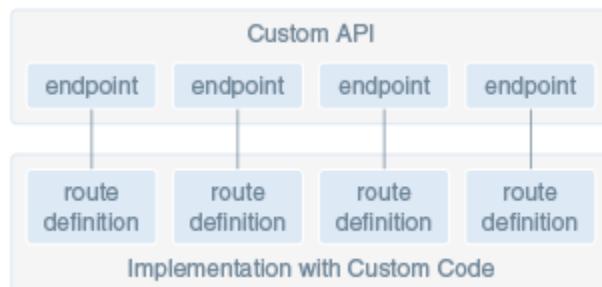
Your custom API implementation can call platform APIs (such as Storage and Notifications), other custom APIs, and external REST and SOAP web services. And it can access the external web services either directly or through connectors.

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	<p>Node provides the backbone for the custom code service. When you implement a custom API, you create a Node.js module.</p> <p>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.</p>
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 following table, the default library versions depend on whether your environment was provisioned from the current release or upgraded from an earlier 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 a custom API implementation isn't compatible with the default library versions for your environment, use one of the following processes to change the Node version for that implementation. See [ccc_DefaultNodeConfiguration](#) in [AMCe Policies and Values](#) for the available Node versions.

- Add a `node` property to the configuration section in the custom API implementation's `package.json` file as described in [Declaring the Node Version](#).
- Ask your mobile cloud administrator to change the node version that is specified by the appropriate `CCC_DefaultNode` environment policy. You can set this policy at different scopes, such as environment scope, mobile backend scope, and API scope. Whenever you change a `CCC_DefaultNode` 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 [Common Custom Code Errors](#).

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:



Setting Up Tooling for Custom Code

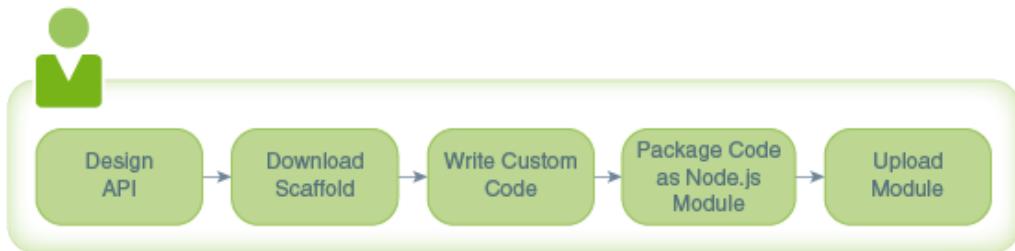
The custom code service is based on Node. You don't need to install Node on your system to create custom API implementations, but you'll need the tooling that it provides, such as the Node package manager (npm). Having Node on your system also makes it easier for you to write the code.

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.

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 **Development**, 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.
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
<i>req.body</i>	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 .
<i>req.headers</i>	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.

Property	Description
<code>req.params</code>	An object that contains properties that map to parameters in the endpoint's URI. For example, if the endpoint is <code>attachments/:collection/objects/:objectid</code> , then you use <code>req.params.collection</code> and <code>req.params.objectid</code> to get the parameter values. When you use the <code>req.params</code> 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 <code>{id}</code> , then you must use <code>req.params.id</code> to get the value, and not <code>req.params.Id</code> .
<code>req.query</code>	The query string parameters that are passed in the URI. For example, if the request is <code>GET /incidents?q=joe&order=desc</code> then you use <code>req.query.q</code> and <code>req.query.order</code> to get the query parameters.

Here are some methods that you typically use to inquire about the request:

Method	Description
<code>req.get(field)</code> and <code>req.header(field)</code>	Both these methods return the value for the header named by <code>field</code> . For example, <code>req.header('content-type')</code> . The match is case-insensitive. Note that <code>req.header</code> is an alias for <code>req.get</code> .
<code>req.is(mimeType)</code>	Boolean method that you can use to find out if the request's Content-Type header matches the <code>mimeType</code> . For example, <code>req.is('json')</code> .

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 **Diagnostics** main menu, click **Logs**, and then click **Server Settings**.

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:



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:



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	<ul style="list-style-type: none"> To return a single item, use <code>setItem()</code> 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 <code>item</code>. To return a collection, use <code>addItem()</code> 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 <code>items</code>, <code>uris</code>, and <code>etags</code> arrays. For example: <pre>{ items:[], uris:[], etags:[] }</pre>	<ul style="list-style-type: none"> Oracle-Mobile-Sync-Resource-Type: Must be set to <code>item</code> for a single item, or <code>collection</code> for an array of items. The <code>setItem()</code> and <code>addItem()</code> methods set this header automatically for items and collections. If the response body is a file, you optionally can set this header to <code>file</code>. ETag: If the Oracle-Mobile-Sync-Resource-Type header is set to <code>item</code> or <code>file</code>, 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.	<ul style="list-style-type: none"> 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. 	<p>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 *.</p> <ul style="list-style-type: none"> 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: <ul style="list-style-type: none"> 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.	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> If there's an <code>If-Match</code> request header and its value isn't <code>*</code>, 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 <code>*</code> in the <code>If-Match</code> header when the conflict resolution policy is <code>CLIENT_WINS</code>. 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
<code>setItem(response, item)</code>	Set the response body to the item.
<code>addItem(response, item, uri, etag)</code>	Add the item to a collection, which will be returned in the response body in a cacheable format.
<code>clear(response)</code>	Undoes all calls to <code>setItem</code> and <code>addItem</code> .

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": [
    "\\"26-5vTpRVI09SakJoLYEQrQ0Q\\",
    "\\"27-+lkt0Y9aA46ySRE00/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 <code>item</code> if the JSON object includes a single item. The value is <code>collection</code> if the JSON object contains an array of items. Note that when the response is a file, you optionally can set the value to <code>file</code> . When this header isn't included in the response, the Synchronization library assumes that the type is <code>file</code> . That is, when this header is not set, then the MobileResource that the Synchronization library <code>fetchObjectBuilder</code> and <code>fetchCollectionBuilder</code> methods return is of type <code>MobileFile</code> .	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	<p>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 YYYYY HH:mm:ss z for SimpleDateFormat.</p> <p>The following synchronization policies are set for the resource object that is created from the response:</p> <ul style="list-style-type: none"> • 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 <p>.</p>	Number
Oracle-Mobile-Sync-Expires	Specifies when to mark the returned resource as expired. Uses RFC 1123 format, for example EEE, dd MMM YYYYY 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.

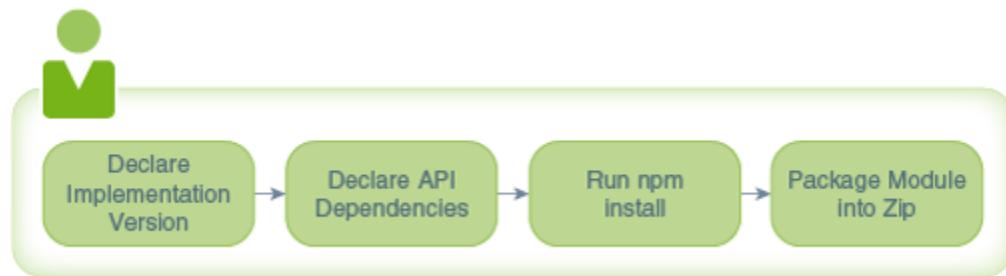
 **Note:**

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.

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 AMCe Policies and 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": {  
  "ModuleName": "VersionNumber",  
  "ModuleName": "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 AMCe Custom Code Test Tools](#).

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.

1. Click **Test** in the API Designer.
The test page displays all the operations.
2. From the endpoints list, click the operation that you want to test.
3. If the endpoint has parameters, then enter the required parameters and any optional parameters that you want to test.
4. If the endpoint accepts a request body, then provide the body or click **Use Example**.
5. Select a backend.
6. Optionally, select an API version.
7. Select the authentication method. If you select **Current User**, then the authentication method is OAuth Consumer.
8. If you aren't using anonymous access, then you must provide a user name and the password. This 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. In addition, the user must have one of the roles that is associated with the backend that you use to test the endpoint. You can see these roles in the backend's Security page.
9. Click **Test Endpoint**.

The Response Status section displays the status and the response. Click **Request** to see the request URI and headers.

You also can test an API from the API Catalog and from a backend.

Offline Debugging with the AMCe Custom Code Test Tools

AMCe offers a set of custom code test tools that you can use to iteratively debug your custom code. You can download the tools from the [Oracle Autonomous Mobile Cloud Enterprise Downloads](#) page on OTN.

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 AMCe.

Detailed instructions on using the tools are located in the `README.MD` file that is packaged within the `omce-tools.zip`.

Tools for Testing Custom Code Outside the UI

You can use tools that were designed for testing web services to test custom code, such as cURL.

The way you remotely access an 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 Facebook User Access Token Manually](#)
- [Headers Needed for API Calls with Facebook Authentication](#)

When you create a custom API, a mock implementation is created automatically. You can use this mock implementation 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](#).

 **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 the **Diagnostics** menu, and then click **Request History**. Next, to 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 **Diagnostics** menu 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 **Diagnostics** menu.

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. From the **Logs** page, click **Server Settings**, 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 `database.insert` method makes a `PUT` request to `/mobile/platform/database/objects/FIF_Incidents`.

```
service.post('/mobile/custom/incidentreport/incidents', function (req, res) {
    req.oracleMobile.database.insert('FIF_Incidents', req.body).then(
        function (result) {
            res.status(result.statusCode).send(result.result);
        },
        function (error) {
            res.status(result.statusCode).send(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 `POST` request to `/platform/database/objects/{table}` ended. The top (later) message was logged when the `service.post` call to `/mobile/custom/incidentreport/incidents` ended.

Timestamp	Message Type	Call	Message
6/9/18 9:11:33.226 am -10:00	INFO	cktest 1.0 > custom/incidentreport 2.1 > POST /incidents	The API invocation ended.
6/9/18 9:11:33.219 am -10:00	INFO	cktest 1.0 > platform/database 1.0 > POST /objects/{table}	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 as shown here:

Message Type	Call	Message
INFO	cktest 1.0 > custom/incidentreport 2.1 > POST /incidents	The API invocation ended.
INFO	cktest 1.0 > custom/incidentreport 2.1	CCC response body: { "id": 5 }
INFO	cktest 1.0 > platform/database 1.0 > POST /objects/{table}	The API invocation ended.
INFO	cktest 1.0 > custom/incidentreport 2.1	OracleMobile.rest request rid:0:1:2:1 body: {"title":"Water heat...
INFO	cktest 1.0 > custom/incidentreport 2.1	CCC request body: {"title":"Water heater is leaking","technician" ...

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`.

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 Type	Call	Message
INFO	cktest 1.0 > custom/incidentreport 2.1 > POST /incidents	The API invocation ended.
FINEST	cktest 1.0 > custom/incidentreport 2.1	CCC response post /mobile/custom/incidentreport/incidents st...
FINEST	cktest 1.0 > custom/incidentreport 2.1	OracleMobile Promise response post /mobile/platform/database...
INFO	cktest 1.0 > platform/database 1.0 > POST /objects/{table}	The API invocation ended.
FINEST	cktest 1.0 > custom/incidentreport 2.1	OracleMobile Promise request post /mobile/platform/database...
FINEST	cktest 1.0 > custom/incidentreport 2.1	CCC request body body: {"title": "Water heater is leaking", "tech...
FINEST	cktest 1.0 > custom/incidentreport 2.1	CCC request post /mobile/custom/incidentreport/incidents

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.

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

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.

Status Code	Description
500 INTERNAL SERVER ERROR	General error when an exception is thrown on the server side.

You can use the request's log entries to pinpoint where the error occurred as described in [Accessing Logging Messages for Custom Code](#). To learn how to identify custom code syntax errors from the log, see [Diagnosing Syntax Errors](#).

If you don't see any messages that identify the source of the problem, then you can change to a finer level for logging messages. From the **Logs** page, click **Server Settings**, 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 to help identify the code that's causing the problem. See [Inserting Logging Into Custom Code](#).

 **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](#).

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:

Custom Code: incidentreport 2.0 - SEVERE
incidentreports 3.0

Message Custom Code Problem:
Text request
get /mobile/custom/incidentreport/incidents
caused unhandled error.
ReferenceError: request is not defined
[at /scratch/aime/mobile/mobile_ccc/custom_code_modules/ccc8761682297981923964/incidentreports/incidentreports.js:430:19](#)

Request	Context
Mobile test 1.0 Backend API incidentreport 2.0 Service incidentreports 3.0	Request 0059bqTLicf9lZs_Sh02yc0003Hq0000IM Correlation Id

Message

Message SEVERE
Level

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\\n at /scratch/aime/mobile/mobile_ccc/custom_code_modules/\nccc2455344468806884059/incidentreports/incidentreports.js:354:17\n\\n at callbacks (/scratch/aime/mobile/mobile_ccc/mcs-node-router/node_modules/\nexpress/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

When you use a generic REST method from the custom code SDK, 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 `optionsList.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

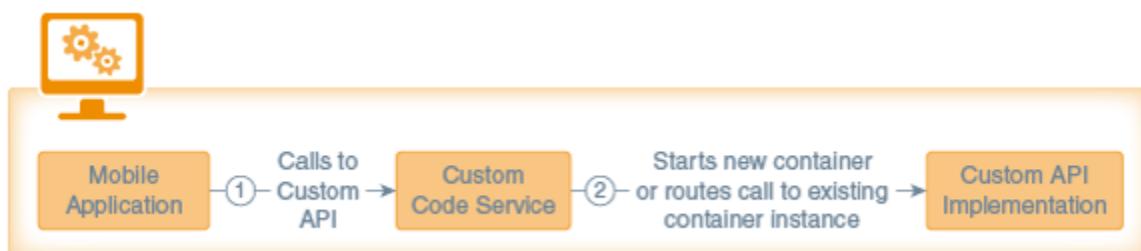
In the UI, open the API and click **Security**. If **Login Required** is turned on and **Enterprise** is selected, then look at the roles that have been configured. If no roles are configured, then no one has permission to log in to the mobile backend. If one or more roles are configured, ensure that 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.

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 Platform, Connector, and Custom APIs

You use custom code SDK methods to send requests to platform, custom, and connector APIs. When you call one of these methods, that method makes a RESTful HTTP call to the 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	<p>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.</p> <p>If json, then the method sets the Content-Type header to application/json automatically.</p> <p>You typically set this property when the custom code builds the request body that you are sending to the API.</p>	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	<p>The response body type. The value can be one of the following:</p> <ul style="list-style-type: none">• 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. <p>This property is supported only by the Storage API and the connector and custom APIs. All other APIs use the default response behavior.</p>	String	Undefined. The response body is converted to a string using the UTF8 encoding.

Property	Description	Type	Default Value
encodeURI	<p>When true, encodes the URI and the following arguments and properties:</p> <ul style="list-style-type: none"> • 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. <p>This option is useful for multibyte characters.</p>	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 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 Collector API from Custom Code](#)
- [Accessing the App Policies API from Custom Code](#)
- [Accessing the Database Access API from Custom Code](#)
- [Accessing the Location API from Custom Code](#)
- [Accessing the Location Management API from Custom Code](#)
- [Accessing the Mobile Devices API from Custom Code](#)
- [Accessing the My Profile API from Custom Code](#)
- [Accessing the Notifications API from Custom Code](#)
- [Accessing the Storage 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 [Oracle Autonomous Mobile Cloud Enterprise REST API Reference](#).

Accessing the Analytics Collector 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.

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`.
- Events are JSON objects. All events must have a `name` and a `timestamp`. The `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'}  
}
```

- You can group events by session. For example, a session can mark the beginning and ending of a function within a mobile app or when an app 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'
    }
}

```

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 [Oracle Autonomous Mobile Cloud Enterprise REST API Reference](#).

`options`: Optional. JSON object. This object can have these properties in addition to those listed in [Common options Argument Properties](#):

Property	Description	Type	Default
<code>applicationKey</code>	Deprecated.	String	None
<code>deviceId</code>	Identifies the device. This is the ID that is returned when you register the device with the Mobile Devices API.	String	None
<code>sessionId</code>	Specifies a default session ID. Use a session ID to group all events by a user-defined session. When present, the <code>sessionId</code> 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/log',  
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')});
    );
}
);

```

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, as well as execute SQL statements.

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.
- `database.sql(sql, args, options, httpOptions)`: Executes an SQL statement.

`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
<code>encodeURI</code>	Set to <code>true</code> to URI-encode the table and keys values. This option can be useful for multibyte values.	Boolean	<code>false</code>

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 URL:

```
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
<code>encodeURI</code>	Set to true to URI-encode the table and fields values. This option can be useful for multibyte values.	Boolean	false
<code>fields</code>	A comma separated list of the fields to return. For example, <code>customer, status</code> .	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

Property	Description	Type	Default
extraFields	<p>For an implicit table creation, optionally provide a comma-separated list that specifies which predefined columns to include in the table from amongst <code>id</code>, <code>createdBy</code>, <code>createdOn</code>, <code>modifiedBy</code>, and <code>modifiedOn</code>. For example, <code>createdOn,createdBy</code>.</p> <p>To not include any predefined columns, specify <code>none</code>.</p>	String	To include all the predefined columns, do not include this property. Note that the <code>id</code> column is added to the table automatically if both the <code>primaryKeys</code> and <code>extraFields</code> 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.

N
o
t
e
:
B
e
c
a
u
s
e
y
o
u
c
a
n
'
t
r
e
t
r
i
e
v
e
t
h
e
p
r
i
m
a
r
y

Property	Description	Type	Default
key	the key or identifier for the document that you would like to make sure the platform can find.	String	the key or identifier for the document that you would like to make sure the platform can find.

Property	Description	Type	Default
	order of the primary keys.	o r d e r o f t h e p r i m a r y f i e l d s . .	

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 <code>id</code> column to the table, and generates the column's values automatically, as long as you don't also include <code>extraFields</code> without <code>id</code> in the list.

N
o
t
e
:
B
e
c
a
u
s
e
y
o
u
c
a
n
't
r
e
t
r
i
e
v
e
t
h
e
p
r
i
m
a
r
y

Property	Description	Type	Default
	keyorderforfromtheatablelementaddedatamakesurethatyoudocumentthe	String	keyorderforfromtheatablelementaddedatamakesurethatyoudocumentthe

Property	Description	Type	Default
	order of the primary keys.	Object	{}

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 (err) {
      res.status(500).send(err.message);
    }
  );
});
```

```

        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.sql(sql, args, options, httpOptions)

This method lets you execute an SQL statement, such as INSERT, UPDATE, MERGE, DELETE, or SELECT. You can use this method for complex actions, such as when you need to join tables, use aggregate functions like COUNT() and SUM(), or use a WHERE clause to delete a set of rows.

Unlike the other `database` methods, this operation doesn't alter your schema automatically. For example:

- If the table doesn't exist, then it isn't created.
- If a column doesn't exist, then the table isn't altered to include it.
- If the value is larger than the column size, then the column isn't resized.

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

`sql`: Required. String. The SQL statement to execute. Examples:

```
INSERT into Statuses (status, code) values (:status, :code)
```

```
SELECT SUM("totalGross") "salesByGenre", "genre" FROM "Movies" GROUP BY "genre"
```

```
SELECT COUNT("incidentReport") "reportCount" FROM "FIF_Incidents" WHERE "status" = :status
```

`args`: Optional. If your SQL statement takes parameters, then pass them in this JSON object.

To execute the SQL statement once with a single set of parameters, use this format for the `args` argument:

```
{
  status : 'Open',
  code : '3'
}
```

To execute the SQL statement multiple times, use this format for the `args` argument:

```
[
  {
    {
      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
<code>encodeURI</code>	Set to <code>true</code> to URI-encode the SQL statement. This option can be useful for multibyte values.	Boolean	<code>false</code>

Response

The response body is a JSON object. The structure of the JSON object depends on the SQL verb and whether the table has a primary key. For example, here's a response body for a SELECT statement:

```
{"items": [ { "code" : 3} ] }
```

Here's an example of a response body for an INSERT, DELETE, UPDATE, OR MERGE where the table has a primary key:

```
{ "rowCount" : 2 }
```

Here's an example of a response body for an INSERT, DELETE, UPDATE, OR MERGE where the row key is the `id` column:

```
{"items": [ { "id":42}, { "id":43} ] }
```

Examples

Here's an example of calling the method to insert rows:

```
service.post('/mobile/custom/incidentreport/status', function (req, res) {
  req.oracleMobile.database.sql(
    'insert into Statuses (status, code) values (:status, :code)',
    [
      {
        status: 'Closed',
        code: '0'
      },
      {
        status: 'Completed',
        code: '9'
      }
    ]).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 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 [Oracle Autonomous Mobile Cloud Enterprise REST API Reference](#)

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 [Oracle Autonomous Mobile Cloud Enterprise REST API Reference](#). 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 [Oracle Autonomous Mobile Cloud Enterprise REST API Reference](#). 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",
                "beacon": {
                    "iBeacon": {
                        "uuid": "B9407F30-F5F8-466E-AFF9-25556B57FE6D",
                        "major": "1.0",
                        "minor": "1.1"
                    }
                }
            }
        }
    ]
}
```

```

    "modifiedBy": "jdoe",
    "links": [
        {
            "rel": "canonical",
            "href": "/mobile/platform/location/assets/333"
        },
        {
            "rel": "self",
            "href": "/mobile/platform/location/assets/333"
        }
    ],
    "attributes": {
        "EquipmentManufacturer": "Example Company",
        "SJIId": "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",
        "SJIId": "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 [Oracle Autonomous Mobile Cloud Enterprise REST API Reference](#).

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/custom/incidentreport/devices/12345"
      }
    ]
  }
}
```

```

        "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 [Oracle Autonomous Mobile Cloud Enterprise REST API Reference](#). 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 [Oracle Autonomous Mobile Cloud Enterprise REST API Reference](#). 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 [Oracle Autonomous Mobile Cloud Enterprise REST API Reference](#).

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 [Oracle Autonomous Mobile Cloud Enterprise REST API Reference](#). 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 [Oracle Autonomous Mobile Cloud Enterprise REST API Reference](#). 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](#).

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.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 [Oracle Autonomous Mobile Cloud Enterprise REST API Reference](#). 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",  
        "SJIId": "6754843090"  
      }  
    }  
  ]  
}
```

context: This argument is ignored.

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 [Oracle Autonomous Mobile Cloud Enterprise REST API Reference](#).

Examples

Here's an example of calling this method.

```
service.post('/mobile/custom/incidentreport/assets', function (req, res) {  
  req.oracleMobile.location.assets.register(req.body).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": 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",  
        "SJIId": "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: This argument is ignored.

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 [Oracle Autonomous Mobile Cloud Enterprise REST API Reference](#).

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.

```
service.delete('/mobile/custom/location/assets', function(req,res) {  
    var contextObject = {};  
    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 [Oracle Autonomous Mobile Cloud Enterprise REST API Reference](#). Here's an example:

```
{  
    "lastKnownLocation": {  
        "gpsPoint": {  
            "latitude": 37.5548,  
            "longitude": -121.1566  
        }  
    },  
    "devices": [  
        11  
    ]  
}
```

context: This argument is ignored.

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 [Oracle Autonomous Mobile Cloud Enterprise REST API Reference](#).

Examples

Here's an example of calling this method.

```
service.put('/mobile/custom/incidentreport/assets/:id', function (req,  
res) {  
    req.oracleMobile.location.assets.update(  
        req.params.id, req.body).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 [Oracle Autonomous Mobile Cloud Enterprise REST API Reference](#). 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: This argument is ignored.

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 [Oracle Autonomous Mobile Cloud Enterprise REST API Reference](#).

Examples

Here's an example of calling this method.

```
service.post('/mobile/custom/incidentreport/devices', function (req, res) {
    req.oracleMobile.location.devices.register(
        req.body).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: This argument is ignored.

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 [Oracle Autonomous Mobile Cloud Enterprise REST API Reference](#).

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.

```
service.delete('/mobile/custom/location/devices', function(req,res) {
  var contextObject = {};
  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 [Oracle Autonomous Mobile Cloud Enterprise REST API Reference](#). Here's an example:

```
{
  "attributes":{
    "status":"Inactive",
    "visibility":"Private"
  }
}
```

`context`: This argument is ignored.

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 [Oracle Autonomous Mobile Cloud Enterprise REST API Reference](#).

Examples

Here's an example of using this method.

```
service.put('/mobile/custom/incidentreport/device/:id', function (req, res) {
    req.oracleMobile.location.device.update(req.params.id, req.body).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",
            "SJIId": "6754843090"
        },
        "links": [
            {
                "rel": "canonical",
                "href": "/mobile/platform/location/assets/11"
            },
            {
                "rel": "self",
                "href": "/mobile/custom/incidentreport/device/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 [Oracle Autonomous Mobile Cloud Enterprise REST API Reference](#). 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: This argument is ignored.

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 [Oracle Autonomous Mobile Cloud Enterprise REST API Reference](#).

Examples

Here's an example of calling this method.

```

service.post('/mobile/custom/incidentreport/places', function (req, res) {
    req.oracleMobile.location.places.register(req.body).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: This argument is ignored.

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 [Oracle Autonomous Mobile Cloud Enterprise REST API Reference](#).

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
            }
        }
    ]
}
```

```

        "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.

```

service.delete('/mobile/custom/location/places', function(req,res) {
    var contextObject = {};
    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`: This argument is ignored.

Examples

In this example, if the `cascade` query parameter is `true`, then the method calls `removeCascade()` instead of `remove()`.

```

service.delete('/mobile/custom/location/places/:id', function(req,res) {
    var contextObject = {};
    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 [Oracle Autonomous Mobile Cloud Enterprise REST API Reference](#). Here's an example:

```
{
    "address": {
        "gpsPoint": {
            "latitude": -121.1566,
            "longitude": 37.5548
        }
    },
    "devices": [
        1111
    ]
}
```

context: This argument is ignored.

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 [Oracle Autonomous Mobile Cloud Enterprise REST API Reference](#).

Examples

Here's an example of calling this method.

```

service.put('/mobile/custom/incidentreport/place/:id', function (req, res)
{
    req.oracleMobile.location.place.update(req.params.id, req.body).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 Mobile 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 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 [Oracle Autonomous Mobile Cloud Enterprise REST API Reference](#).

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 [Oracle Autonomous Mobile Cloud Enterprise REST API Reference](#).

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 [Oracle Autonomous Mobile Cloud Enterprise REST API Reference](#).

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 My Profile API from Custom Code

The My Profile API let you get information about the current mobile, virtual, or social user.

This API has the following methods:

- [ums.getMe\(httpOptions\)](#): Retrieves information about the current user. For mobile and virtual users, it retrieves the user's roles.
- [ums.getUser\(options, httpOptions\)](#): (Deprecated) Retrieves information about the current user. This operation doesn't retrieve roles. For mobile users, it retrieves only the ID, user name, last name, and email address.
- [ums.getUserExtended\(options, httpOptions\)](#): (Deprecated) Retrieves the same information as [ums.getUser\(options, httpOptions\)](#). In addition, for mobile and virtual users, it retrieves the user's roles.
- [ums.updateUser\(fields, options, httpOptions\)](#): (Deprecated) Updates the current mobile user's last name, first name, or email address, or a combination thereof.

ums.getMe(httpOptions)

This method lets you retrieve the information about the currently authorized user.

- When the user is a mobile user, this operation retrieves the user's information from Oracle Identity Cloud Service.
- 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. To learn about social users and social identity, see [Facebook Login in AMCe](#).

Response

If the current user is a social user, then the response body includes the generated `userName`, as shown in this example. To learn more about social identity see [Facebook Login in AMCe](#).

```
{  
    "userName": "1 :623:165"  
}
```

If the current user is a virtual user, then the response body includes the `username` and `appRoles`, as shown in this example.

```
{  
    "userName": "jdoe",  
    "urn:ietf:params:scim:schemas:oracle:idcs:extension:user:User":{  
        "appRoles": [  
            {  
                "display": "FIF_TECHNICIAN"  
            }  
        ]  
    }  
}
```

In all other cases, the response body contains the same response as provided by Oracle Identity Cloud Service. For example:

```
{  
    "idcsCreatedBy": {  
        "type": "App",  
        "display": "instance1",  
        "value": "346373e8a",  
        "$ref": "https://myIdentity.example.com/admin/v1/Apps/  
3463731bd0cc43c7ba1b79a9c6e25e8a"  
    },  
    "id": "7e56fd80",  
    "active": true,  
    "displayName": "Joe Doe",  
    "idcsLastModifiedBy": {  
        "value": "346373e8aa",  
        "display": "instance1",  
        "type": "App",  
        "$ref": "https://myIdentity.example.com/admin/v1/Apps/  
3463731bd0cc43c7ba1b79a9c6e25e8a"  
    },  
    "userName": "jdoe",  
    "emails": [  
        {  
            "primary": true,  
            "value": "jdoe@example.invalid",  
            "type": "work"  
        }  
    ],  
}
```

```

    "name": {
        "familyName": "Doe",
        "givenName": "Joe",
        "formatted": "Joe Doe"
    },
    "urn:ietf:params:scim:schemas:oracle:idcs:extension:user:User": {
        "grants": [
            {
                "value": "89d8b111",
                "grantMechanism": "ADMINISTRATOR_TO_USER",
                "appId": "346373e8a",
                "$ref": "https://myIdentity.example.com/admin/v1/Grants/89d80073ae7f48838798cc864031b111"
            }
        ],
        "appRoles": [
            {
                "value": "a31245f1dd",
                "adminRole": false,
                "legacyGroupName": "instance1.ReadWriteRole",
                "appId": "346373e8a",
                "appName": "instance1_app_name",
                "display": "ReadWriteRole",
                "$ref": "https://myIdentity.example.com/admin/v1/AppRoles/a31245ce4ed94d2a8563d39cd888f1dd"
            }
        ],
        "accounts": [
            {
                "appId": "346373e8a",
                "value": "3819e1be",
                "active": true,
                "$ref": "https://myIdentity.example.com/admin/v1/AccountMgmtInfos/3819dd966cf34aa593df61809d62e1be"
            }
        ]
    },
    "schemas": [
        "urn:ietf:params:scim:schemas:core:2.0:User"
    ]
}

```

Examples

Here's an example of calling this method:

```

req.oracleMobile.ums.getMe().then(
    function (result) {
        res.status(result.statusCode).send(result.result);
    },
    function (error) {
        res.status(error.statusCode).send(error.error);
    }
);

```

ums.getUser(options, httpOptions)

Deprecated. This method lets you retrieve the information about the currently authorized user.

- When the user is a mobile user, this operation retrieves the user's ID, user name, first name, last name, and email address.
- 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 AMCe](#).

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
<code>fields</code>	Specifies which user properties to get. For example, you can set <code>options.fields</code> to <code>firstName,lastName</code> 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 AMCe](#).

```
"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

The response body also contains links to the API endpoint for the resource.

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.status(result.statusCode).send(result.result);  
  },  
  function (error) {  
    res.status(error.statusCode).send(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.status(result.statusCode).send(result.result);  
    },  
    function (error) {  
        res.status(error.statusCode).send(error.error);  
    }  
)  
};
```

Here's an example of a response body for this request:

```

    "lastName": "Doe",
    "links": [
        {
            "rel": "canonical",
            "href": "/mobile/platform/users/joe"
        },
        {
            "rel": "self",
            "href": "/mobile/platform/users/joe"
        }
    ]
}

```

ums.getUserExtended(options, httpOptions)

Deprecated. This method lets you retrieve the information about the currently authorized user, including the user's roles.

- When the user is a mobile user, this operation retrieves the user's ID, user name, first name, last name, roles, and email address.
- 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 AMCE](#).

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
<code>fields</code>	Specifies which user properties to get. For example, you can set <code>options.fields</code> to <code>firstName,lastName</code> 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)

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.status(result.statusCode).send(result.result);
  },
  function (error) {
    res.status(error.statusCode).send(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.status(result.statusCode).send(result.result);
    },
    function (error) {
        res.status(error.statusCode).send(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 operation is deprecated. To update user information, see [REST API for Oracle Identity Cloud Service](#).

Arguments

fields: Required. A JSON object that contains the fields to update. For example:
{lastName: 'Doe'}.

options: Optional. A JSON object as described in [Common options Argument Properties](#).

Response

The response body is a JSON object that follows the root response schema that's shown for the `PUT /mobile/platform/users/{username}` operation in [Oracle Autonomous Mobile Cloud Enterprise REST API Reference](#).

Examples

Here's an example of calling this method to update the user's last name. In this example, the request body would look like this: {lastName: 'Doe'}.

```
service.put(
  '/mobile/custom/incidentreport/customer',
  function (req, res) {
    req.oracleMobile.ums.updateUser({lastName: req.body.lastName}).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": "295e450a-63f0-41fa-be43-cd2dbc21598",
  "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"
    }
  ]
}
```

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.

 **Tip:**

When you access the Notifications API endpoints directly through REST requests, the user that you specify in the `Authorization` header must be a team member (not a mobile user) who has the `Administrator` or `Developer` role. However, this restriction doesn't apply when you use these `notification` methods in your custom implementation.

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`: This argument is ignored.

`options`: Optional. JSON object. This object can have these properties in addition to those listed in [Common options Argument Properties](#):

Property	Description	Type	Default
<code>createdOnOrAfter</code>	Criteria: Filter by <code>createdOn</code> on or after the given UTC date/time (in <code>YYYY-DD-MM[Thh:mm]Z</code> format).	String	None
<code>createdOnOrBefore</code>	Criteria: Filter by <code>createdOn</code> on or before the given UTC date/time (in <code>YYYY-DD-MM[Thh:mm]Z</code> format).	String	None
<code>limit</code>	The maximum number of items to be returned. If the requested limit is too large, then a lower limit is substituted.	Integer	None
<code>offset</code>	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 [Oracle Autonomous Mobile Cloud Enterprise REST API Reference](#).

Examples

Here's an example of calling this method:

```
service.get('/mobile/custom/incidentreport/notifications',
  function (req, res) {
    req.oracleMobile.notification.getAll()
    .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": []
    }
  ]
};
```

```
        "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: This argument is ignored.

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 [Oracle Autonomous Mobile Cloud Enterprise REST API Reference](#).

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.getId(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 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 [Oracle Autonomous Mobile Cloud Enterprise REST API Reference](#). Here's an example:

```
{
    message:'This is the alert message.',
    tag:'Marketing',
    notificationTokens: [ 'APNSdeviceToken' ]
}
```

context: This argument is ignored.

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 [Oracle Autonomous Mobile Cloud Enterprise REST API Reference](#).

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)
    .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: This argument is ignored.

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)
      .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
<code>encodeURI</code>	Set to true to URI-encode the <code>collectionId</code> value. This option can be useful for multibyte values.	Boolean	false

Property	Description	Type	Default
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
<code>contentDisposition</code>	This property lets you specify the value of the Content-Disposition response header.	String	None
<code>encodeURI</code>	Set to true to URI-encode the <code>collectionId</code> , <code>objectId</code> , and <code>user</code> values. This option can be useful for multibyte values.	Boolean	false
<code>ifMatch</code>	The call completes successfully only if the ETag of the corresponding object matches one of the values specified in this property.	String	None
<code>ifModifiedSince</code>	Date and time in HTTP-date format. For example, <code>Mon, 30 Jun 2014 19:43:31 GMT</code> . 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

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. You can use this property to reduce the amount of data that is transported by not 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
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
<code>encodeURI</code>	Set to true to URI-encode the <code>collectionId</code> , <code>orderBy</code> , and <code>user</code> values. This option can be useful for multibyte values.	Boolean	false
<code>limit</code>	The maximum number of items to be returned. If the requested limit is greater than 100, then 100 is used instead.	Integer	None
<code>offset</code>	The zero-based index of the first item to return.	Integer	None

Property	Description	Type	Default
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
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

Property	Description	Type	Default
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 [Oracle Autonomous Mobile Cloud Enterprise REST API Reference](#).

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
<code>contentDisposition</code>	This property lets you specify the value of the Content-Disposition response header.	String	None
<code>encodeURI</code>	Set to true to URI-encode the <code>collectionId</code> , <code>objectId</code> , and <code>user</code> values. This option can be useful for multibyte values.	Boolean	false
<code>ifMatch</code>	The call completes successfully only if the ETag of the corresponding object matches one of the values specified in this property.	String	None
<code>ifModifiedSince</code>	Date and time in HTTP-date format. For example, <code>Mon, 30 Jun 2014 19:43:31 GMT</code> . 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

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. You can use this property to reduce the amount of data that is transported by not 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
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

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 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
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

Header	Description	Type
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
<code>encodeURI</code>	Set to true to URI-encode the <code>collectionId</code> value. This option can be useful for multibyte values.	Boolean	false
<code>ifMatch</code>	The call completes successfully only if the ETag of the corresponding object matches one of the values specified in this property.	String	None
<code>ifNoneMatch</code>	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
<code>sync</code>	When this property is present and has a value of <code>true</code> , 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
<code>Cache-Control</code>	Describes how the result may be cached.	String

Header	Description	Type
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 [Oracle Autonomous Mobile Cloud Enterprise REST API Reference](#).

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 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 [Oracle Autonomous Mobile Cloud Enterprise REST API Reference](#).

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
<code>encodeURI</code>	Set to true to URI-encode the <code>collectionId</code> , <code>objectId</code> , and <code>user</code> values. This option can be useful for multibyte values.	Boolean	false
<code>ifMatch</code>	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
<code>ifModifiedSince</code>	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 media type to return 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
<code>contentLength</code>	The size of the object in bytes.	Number	If the object is a string or a buffer, then the default is <code>object.length</code> . Otherwise, the default is the sum of its members' lengths.
<code>contentType</code>	The media type of object being stored. This property also specifies the media type to return when the object is requested.	String	If the <code>inType</code> is <code>json</code> , then the <code>Content-Type</code> header is set to <code>application/json</code> automatically. Otherwise, the default is <code>application/octet-stream</code> .
<code>encodeURI</code>	Set to <code>true</code> to URI-encode the <code>collectionId</code> , <code>mobileName</code> , and <code>user</code> values. This option can be useful for multibyte values.	Boolean	<code>false</code>
<code>mobileName</code>	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 [Oracle Autonomous Mobile Cloud Enterprise REST API Reference](#).

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
<code>contentLength</code>	The size of the object in bytes.	Number	If the object is a string or a buffer, then the default is <code>object.length</code> . Otherwise, the default is the sum of its members' lengths.
<code>contentType</code>	The media type of object being stored. This property also specifies the media type to return when the object is requested.	String	If the <code>inType</code> is <code>json</code> , then the <code>Content-Type</code> header is set to <code>application/json</code> automatically. Otherwise, the default is <code>application/octet-stream</code> .

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 are 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 [Oracle Autonomous Mobile Cloud Enterprise REST API Reference](#).

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
    ]
}
```

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
<code>externalAuthorization</code>	If you haven't configured a security policy for the connector, then put the <code>Authorization</code> value for the external service in the <code>options.externalAuthorization</code> property. When this property is present, the connector sets the outgoing <code>Authorization</code> header with the value in <code>options.externalAuthorization</code> property before it sends the request to the external service.	String	None

Property	Description	Type	Default
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)
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.
- METHOD sends a PATCH request without an `X-HTTP-Method-Override` header set to PATCH. This is the default.

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 [AMCe Policies and Values](#).

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(
      {headers: {Authorization: 'Basic ' + b64Auth},
       params: {path: 'blob/123'}});
```

```
'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://schemas.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>.

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
<code>versionToInvoke</code>	The version of the custom API. When you use the <code>oracleMobile.cust</code> namespace, you must include this option if the API version is not declared in <code>package.json</code> . When you use the <code>oracleMobile.cust.om.<apiName></code> namespace, the API version must be declared in <code>package.json</code> , and you optionally can use this property to override that version.	String	The version that is declared in the <code>package.json</code> file.

Both namespaces provide methods for each HTTP operation, as shown in this table:

HTTP Operation	<code>oracleMobile.custom</code> Method	<code>oracleMobile.custom.<apiNa me></code> Method
GET	<code>get(apiName, resourceName, options, httpOptions)</code>	<code>get(resourceName, options, httpOptions)</code>
PUT	<code>put(apiName, resourceName, object, options, httpOptions)</code>	<code>put(resourceName, object, options, httpOptions)</code>
POST	<code>post(apiName, resourceName, object, options, httpOptions)</code>	<code>post(resourceName, object, options, httpOptions)</code>
DELETE	<code>del(apiName, resourceName, options, httpOptions)</code>	<code>del(resourceName, options, httpOptions)</code>
HEAD	<code>head(apiName, resourceName, options, httpOptions)</code>	<code>head(resourceName, options, httpOptions)</code>
OPTIONS	<code>options(apiName, resourceName, options, httpOptions)</code>	<code>options(resourceName, options, httpOptions)</code>
PATCH	<code>patch(apiName, resourceName, object, options, httpOptions)</code>	<code>patch(resourceName, object, options, httpOptions)</code>

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);
    }
);

```

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	<code>oracleMobile.rest</code> Methods
GET	<code>get(optionsList, handler)</code> <code>getAsync(optionsList)</code>
PUT	<code>put(optionsList, handler)</code> <code>putAsync(optionsList)</code>
POST	<code>post(optionsList, handler)</code> <code>postAsync(optionsList)</code>
DELETE	<code>del(optionsList, handler)</code> <code>delAsync(optionsList)</code>
HEAD	<code>head(optionsList, handler)</code> <code>headAsync(optionsList)</code>
OPTIONS	<code>options(optionsList, handler)</code> <code>optionsAsync(optionsList)</code>
PATCH	<code>patch(optionsList, handler)</code> <code>patchAsync(optionsList)</code>

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, [Oracle Autonomous Mobile Cloud Enterprise REST API Reference](#) provides information about the platform APIs. For example, it provides cURL examples as well as details about request and response bodies and headers.

20

Connectors

In Oracle Autonomous Mobile Cloud Enterprise (AMCe), you use connectors to simplify calls to external services, such as enterprise systems and third-party APIs.

AMCe provides connector types for REST, SOAP, Oracle Integration Cloud Service (ICS), and Oracle Fusion Applications.

What Is a Connector API?

A connector API is an interface for connecting to an external service. Connectors APIs give you a standard way to connect to external services and at the same time benefit from AMCe's built-in security, diagnostics, and analytics features.

Each connector is based on a configuration where you define any connection details, security policies, and rules for things such as default parameter values and proxy path.

You can call a connector API with a simple REST call from the implementation code of a custom API.

REST Connector APIs

You can 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 AMCe.

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.

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 AMCe 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 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. Click  and select **Development > APIs** from the side menu.
2. Click **REST** (if this is the first connector API to be created) or **New Connector** and from the drop-down list, select **REST**.
3. 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.
4. Click **Create**.
5. 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 policy, which has a default value of 40,000 ms. To learn more about policies, see [Oracle Autonomous Mobile Cloud Enterprise 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 **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.

7. Click **Save**.
8. 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](#) in *Managing Oracle Autonomous Mobile Cloud Enterprise*.

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 **Development > 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`

Security Policies and Overriding Properties

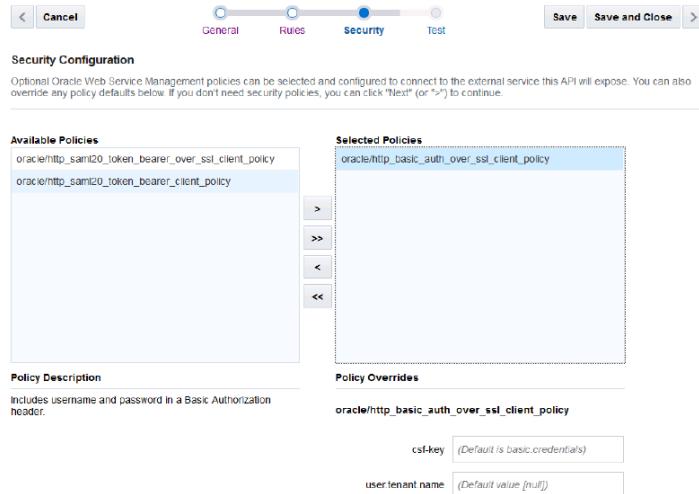
Before you finalize your connector API, you should consider how to handle its security. You can use either security policies or authorization headers. Selecting a security policy that describes the authentication scheme of the service to which you're connecting to is the recommended approach.

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 **Development > 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 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 AMCe 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:

- 2xx: indicates a successful connection
- 3xx: indicates a redirection occurred
- 4xx: indicates a user error occurred
- 5xx: 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 <code>PUT</code> or <code>POST</code> operation.
204 NO CONTENT	Successful connection but no response body (used for <code>DELETE</code> and <code>UPDATE</code> 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 and published, 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 in *Managing Oracle Autonomous Mobile Cloud Enterprise*.

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.

By examining multiple messages, you can more easily determine where issues occur. For every message that you send, AMCe 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 AMCe 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

AMCe 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 AMCe-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 AMCe 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 AMCe is reused to authenticate with the other service.
- An access token generated by a service is passed to an AMCe 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 AMCe 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	AMCe 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 AMCe to secure REST connections. This flow is used when the client owns the resources (that is, the client is the resource owner).

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 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 AMCe, 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 do so 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. Click  and select **Development > APIs** from the side menu.
2. Select the draft connector API that you want to edit and click **Open**.
3. Click **Refresh** () if you're using the same descriptor and just want to get the latest resources.
4. 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.
5. Test your changes.

Using Your REST 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.

You have two options for creating such a custom API for a REST connector API:

- Generate a custom API for the connector, as described in [Generating Custom APIs for Connectors](#).
This only works for connector APIs that are based on a descriptor URL.
- Design a custom API and add calls to the connector in the custom API's implementation code as described in [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 (4xx) and server (5xx) 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 [AMCe Policies and 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 APIs

You can 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 AMCe.

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.

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 AMCe 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 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.

For more information on policies, see [Oracle Autonomous Mobile Cloud Enterprise Policies](#).

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, or move it to the trash.

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. Click  and select **Development > 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.

2. 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.

The screenshot shows a dialog box titled 'New SOAP Connector API'. It contains fields for 'API Display Name' (set to 'My SOAP Connector API'), 'API Name' (set to 'mysoapconnectorapi'), 'WSDL URL' (set to 'http://example.com/soapconnector'), and a 'Short Description' field (set to 'Displayed in Custom Code API Catalog'). A note at the bottom says '100 characters left'. At the bottom right is a 'Create' button.

3. 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 AMCE, 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.

4. Set the timeout values:



The screenshot shows the 'Remote Service Connection Settings' dialog. It contains two input fields for timeouts. The first field is 'HTTP Read Timeout' with a value of '25,000' and a unit of 'Milliseconds'. The second field is 'HTTP Connection Timeout' with a value of '25,000' and a unit of 'Milliseconds'. Both fields have up and down arrows for adjusting the value.

- **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 0ms means an infinite timeout is permitted.

The HTTP timeout values must be less than the `Network_HttpRequestTimeout` policy, which has a default value of 40,000 ms. To learn about policies, see [Oracle Autonomous Mobile Cloud Enterprise 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.

5. 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.

6. 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.

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.

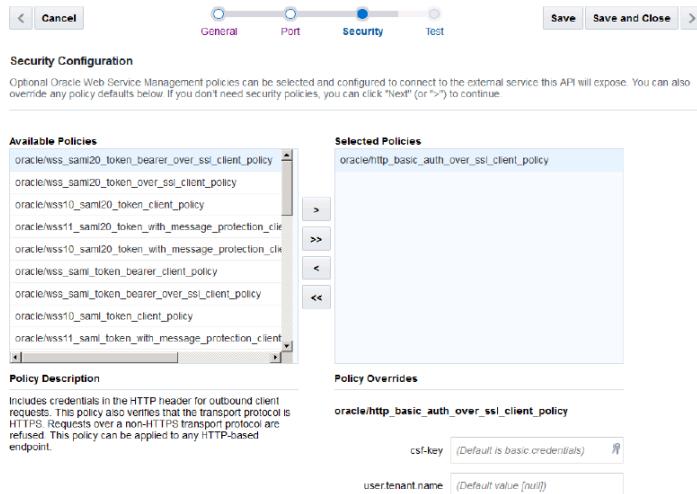
Click **Next >** to go to the next step in configuring your connector API.

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.

- Click the **Security** navigation link at the top of the SOAP Connector API wizard.



- 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](#).

- Select a policy to read its description.
- 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](#).

- Click **Save** to save your work or **Save and Close** to save your work and exit the SOAP Connector API wizard.

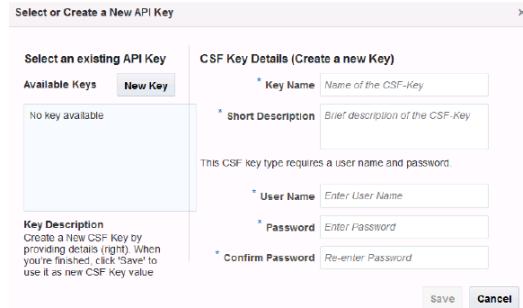
Before you can test your connection, you must save your configuration. If you proceed to the testing page without saving the API configuration, you'll see a dialog asking you to save it. You can check the **Always save before testing**

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 AMCe 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:

- 2xx - indicates a successful connection
- 3xx - indicates a redirection occurred
- 4xx - 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 in *Managing Oracle Autonomous Mobile Cloud Enterprise*.

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.

By examining multiple messages, you can more easily determine where issues occur. For every message that you send, AMCe 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 AMCe 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, AMCe 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.

AMCe 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.

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: 'appplcation/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://schemas.xmlsoap.org/soap/  
envelope">  
  
    <SOAP-ENV:Header>  
        <!-- Add any SOAP headers here -->  
    </SOAP-ENV:Header>  
  
    <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 AMCe, 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. Click  and select **Development > Connectors** from the side menu.

Since at least one connector API exists, the Connectors page is displayed.

2. 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.

3. 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.

4. Save your changes if you didn't select the option to always save the configuration before testing when you created the API.

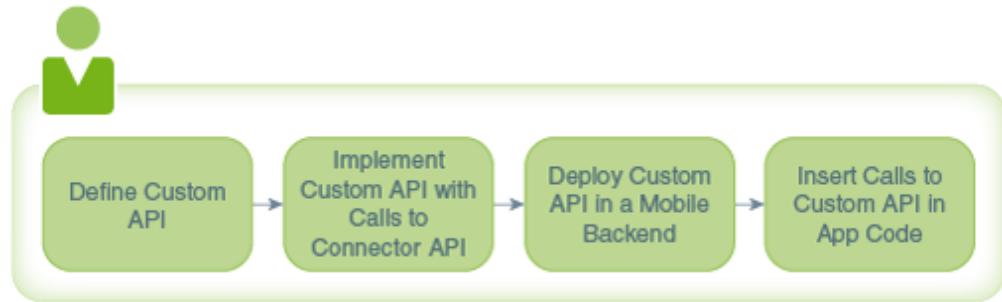
5. Test your changes.

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.

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 (4xx) and server (5xx) 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 [AMCe Policies and Values](#).

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 [AMCe Policies and 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 AMCe, 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.

ICS Connector APIs

Oracle Autonomous Mobile Cloud Enterprise (AMCe) 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.

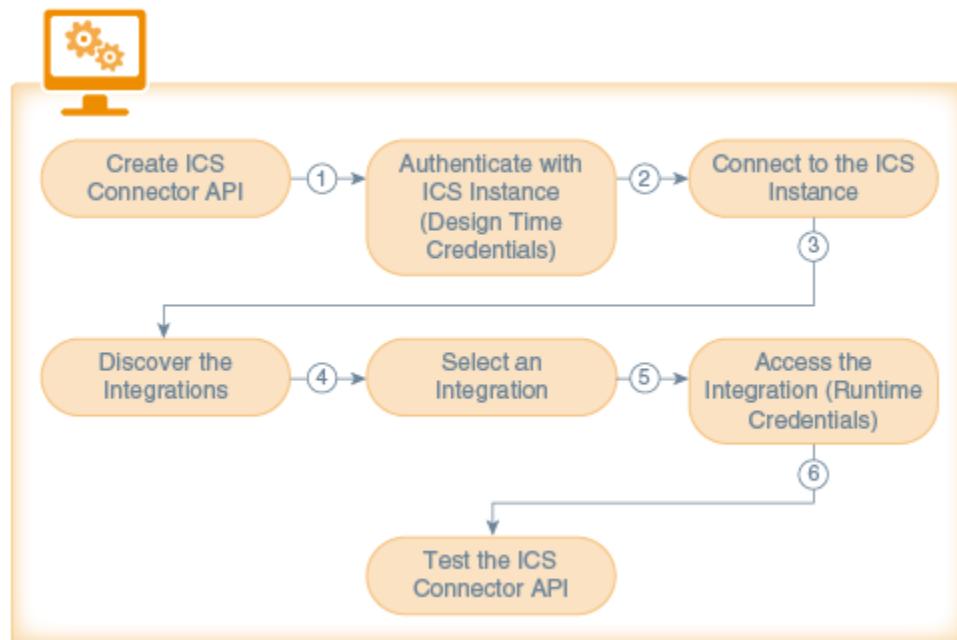
Once you have created an ICS connector API, you can call it from the implementation of a custom API.

Note:

Only SOAP-based integrations are supported.

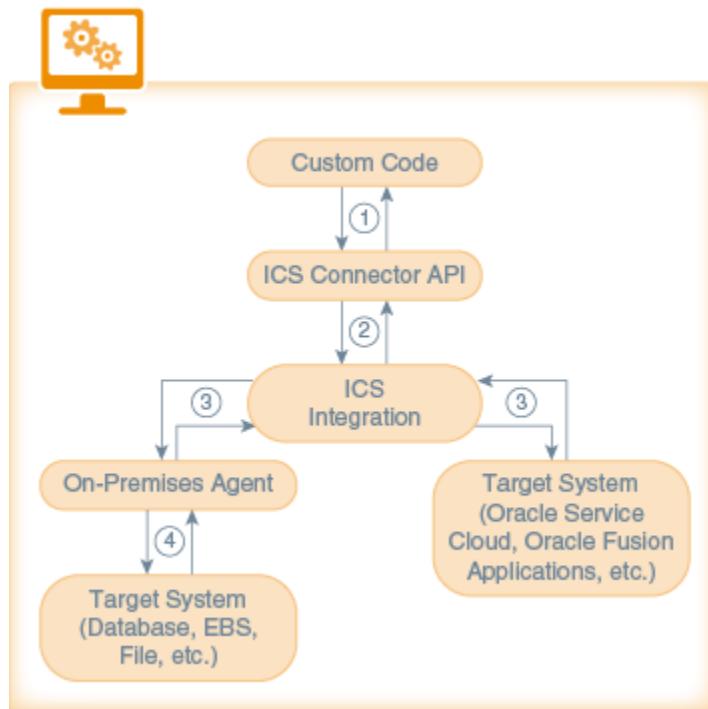
ICS Connector API Flow

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.** AMCe 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:** AMCe 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

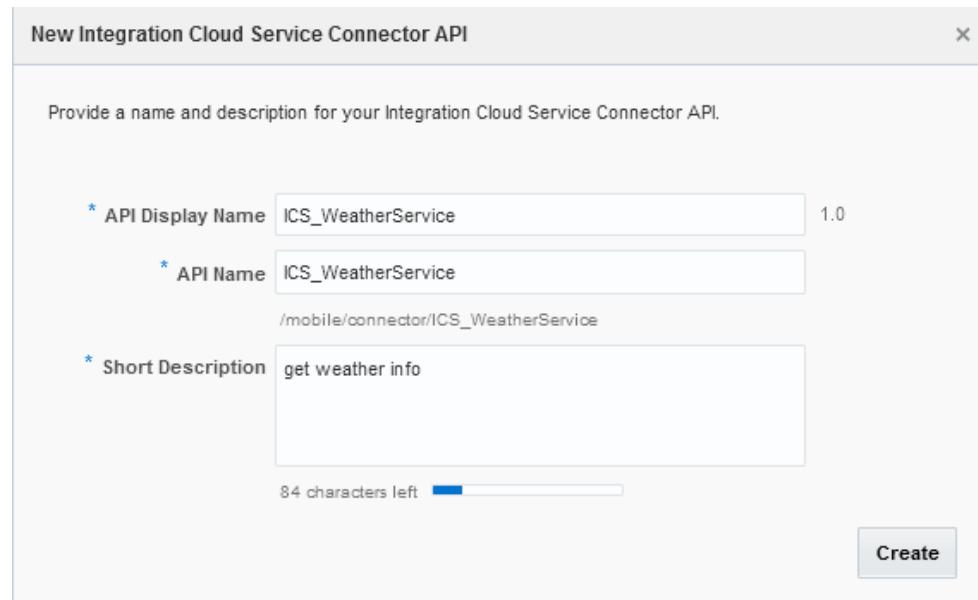
Before you begin configuring your connector, you must provide some initial basic information like the connector API name, a brief description, and connection timeout settings.

1. Click  and select **Development > 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.

2. 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.



The screenshot shows the 'New Integration Cloud Service Connector API' dialog. It contains the following fields:

- * API Display Name: ICS_WeatherService
- * API Name: ICS_WeatherService
- /mobile/connector/ICS_WeatherService
- * Short Description: get weather info
- 84 characters left

A 'Create' button is located at the bottom right.

3. 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.

4. Click **Create.**

The General page of the ICS Connector API wizard is displayed.

5. 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'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 [Oracle Autonomous Mobile Cloud Enterprise Policies](#).

Remote Service Connection Settings

The screenshot shows a configuration interface for 'Remote Service Connection Settings'. It features two input fields: 'HTTP Read Timeout' and 'HTTP Connection Timeout', both currently set to 26,000. Each field includes a numeric input box, up/down arrow buttons for adjustment, and a unit indicator 'Milliseconds'.

HTTP Read Timeout	26,000	Milliseconds
HTTP Connection Timeout	26,000	Milliseconds

- 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.

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 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.

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 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.

Integrations

Select an integration Cloud Service instance, or create a new one by entering the instance information and credentials below. A successful connection will be saved so you can just select it next time.

Select Connection (or)

* Connection Name: My_ICS_Instance

* Service URL: https://cloud.oracle.com/ics

* User Name: icsdeveloper

* Password: *****

Remember My Credentials

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.orgs](#), otherwise, even if your 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 AMCe user and aren't provided if another AMCe 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.

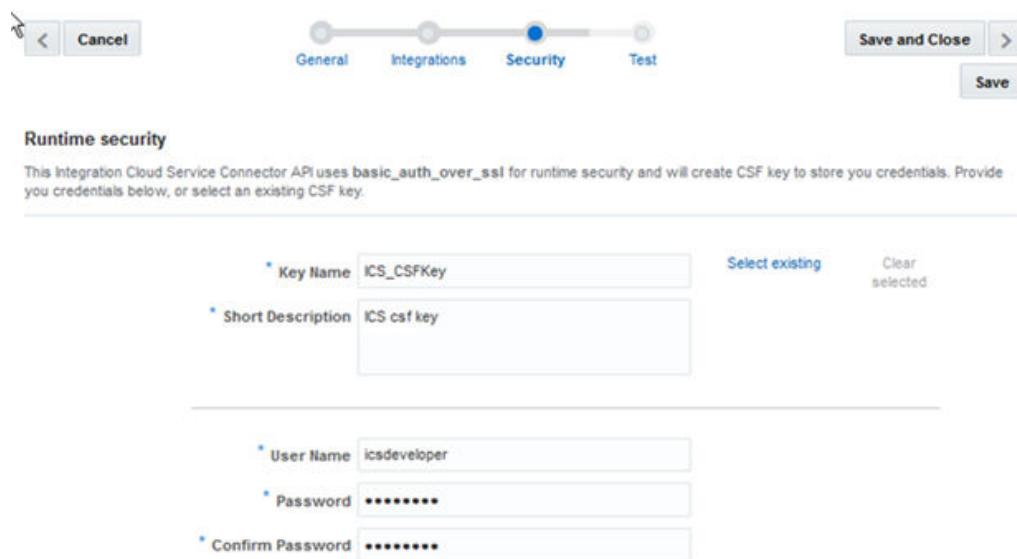
 **Note:**

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 AMCe 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.



The screenshot shows the 'Security' tab selected in a four-step wizard. The steps are: General, Integrations, Security, and Test. The 'Security' tab is highlighted with a blue dot. The 'Runtime security' section contains the following fields:

* Key Name	ICS_CSFKey	Select existing	Clear selected
* Short Description	ICS csf key		
<hr/>			
* User Name	icsdeveloper		
* Password	*****		
* Confirm Password	*****		

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.

The screenshot shows the 'Connector API Testing' interface. The top navigation bar has four tabs: General, Integrations, Security, and Test, with 'Test' being the active tab. Below the tabs is a message: 'You can test any method by specifying an endpoint and parameters for the local URL. You can also optionally include HTTP headers or a request body. If you're happy with your new API, click 'Done' to return to the list of connector APIs.' A yellow button labeled 'Default Test Credentials' is visible. The main content area shows a green header with a cloud icon and the text 'POST GetCityWeatherByZIP /mobile/connector/ICS_WeatherService/GetCityWeatherByZIP'. Below this is a section titled 'Examples' with a sub-section 'View examples of Request, Response, and Fault payloads'. Under 'Request', there is a 'Payload Example' code block:

```

Payload Example {
    "Header": null,
    "Body": {
        "GetCityWeatherByZIP": {
            "ZIP": "ZIP15"
        }
    }
}

```

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.

The screenshot shows the 'HTTP Headers' and 'HTTP Message Body' sections of the test configuration. The 'HTTP Headers' section has a 'Standard' dropdown and a 'Content-Type' dropdown set to 'application/json; charset=UTF-8'. A 'Add HTTP Header' button is available. The 'HTTP Message Body' section has a 'Create a Body for testing or use the sample JSON provided.' link and a 'Learn about XML to JSON translation' link. It contains a 'HTTP Body' code block:

```

HTTP Body {
    "schema": "http://json-schema.org/draft-04/schema#",
    "title": "Object",
    "description": "An object for this service",
    "type": "object"
}

```

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.

The screenshot shows a 'Test Response Status: 200' page. It has tabs for Request and Response. The Request tab shows a JSON object with headers like X-ORACLE-DV5-ECID and Date. The Response tab shows a JSON body with a 'Body' key containing a 'GetCityWeatherByZIPResponse' object. This object includes a 'GetCityWeatherByZIPResult' object with fields such as Success (true), ResponseText ("City Found"), State ("CA"), City ("Redwood City"), WeatherStationCity ("Hayward"), WeatherID (14), Description ("Cloudy"), Temperature ("63"), RelativeHumidity ("93"), Wind ("W"), Pressure ("29.908"), Visibility (null), WindChill (null), and Remarks ("trueCity FoundCA Redwood City/Redwood City2013-01-03T00:00:00ZPartly Cloudy56102013-01-04T00:00:00ZPartly Cloudy385700002013-01-05T00:00:00ZPartly Cloudy395000102013-01-06T00:00:00ZPartly Cloudy4254202013-01-07T00:00:00ZPartly Cloudy395810102013-01-08T00:00:00ZPartly Cloudy415810102013-01-09T00:00:00ZPartly Cloudy41571010Z").

```

{
  "Body": {
    "GetCityWeatherByZIPResponse": {
      "GetCityWeatherByZIPResult": {
        "Success": true,
        "ResponseText": "City Found",
        "State": "CA",
        "City": "Redwood City",
        "WeatherStationCity": "Hayward",
        "WeatherID": 14,
        "Description": "Cloudy",
        "Temperature": "63",
        "RelativeHumidity": "93",
        "Wind": "W",
        "Pressure": "29.908",
        "Visibility": null,
        "WindChill": null,
        "Remarks": "trueCity FoundCA Redwood City/Redwood City2013-01-03T00:00:00ZPartly Cloudy56102013-01-04T00:00:00ZPartly Cloudy385700002013-01-05T00:00:00ZPartly Cloudy395000102013-01-06T00:00:00ZPartly Cloudy4254202013-01-07T00:00:00ZPartly Cloudy395810102013-01-08T00:00:00ZPartly Cloudy415810102013-01-09T00:00:00ZPartly Cloudy41571010Z"
      }
    }
  }
}

```

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:

- 2xx: indicates a successful connection
- 4xx: indicates a user error occurred

- 5xx: indicates a server error occurred

The following table lists the most common status messages you'll see:

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.
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, AMCe 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 AMCe 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 AMCe, 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](#).

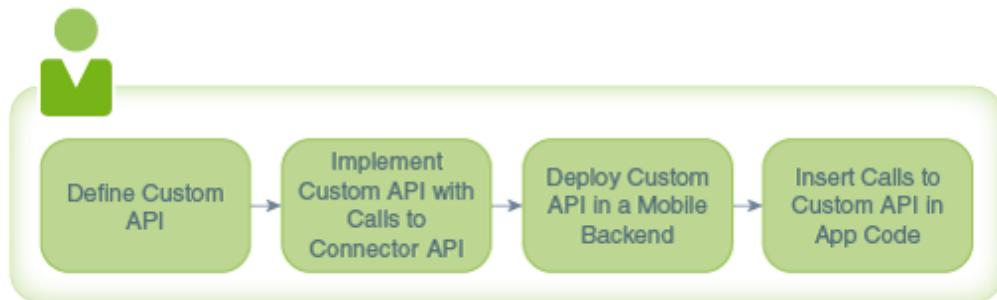
CSF keys and their values are specific to the environment in which they're defined. That is, if the Development environment is selected, then only the CSF keys and certificates for the security policies in use by artifacts in that environment are listed in the CSF Keys dialog. A different set of keys 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 (4xx) and server (5xx) 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 AMCe.

- 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, AMCe itself, the ICS instance that AMCe 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 [AMCe Policies and 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.

Fusion Applications Connector APIs

Oracle Autonomous Mobile Cloud Enterprise (AMCe) 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.

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.

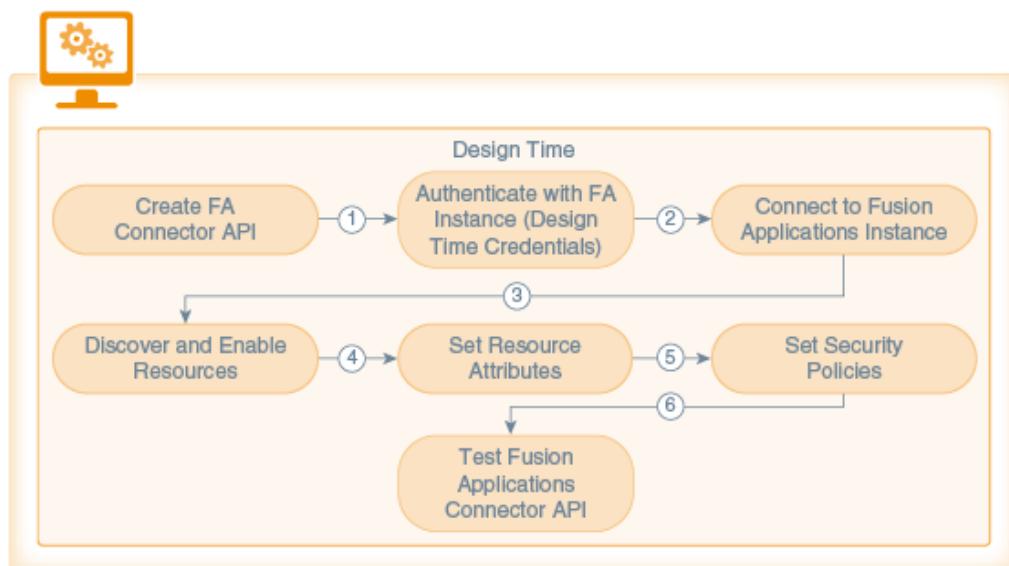
Use the Fusion Applications Connector API wizard to quickly and easily create a connector API with a customized selection of resources from a Fusion Applications service or Fusion-based service.

Here are 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:



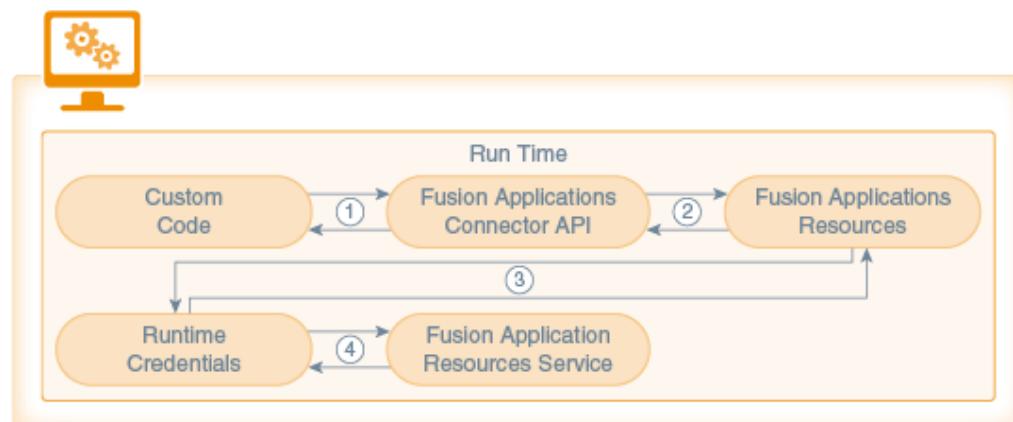
1. **Connector Creation phase.** An unbound Fusion Applications Connector API is created with the Fusion Applications Connector API wizard.
2. **Connection phase.** Design time credentials are passed and a connection to the Fusion Applications instance is made. The design time credentials are saved in the Credentials Store Framework (CSF) in AMCe. The Fusion Applications service description, the Fusion Applications Describe, is retrieved from the external service.
3. **Resource Discovery phase.** AMCe locates the Fusion Applications instance via the Describe URL provided. When authentication is confirmed, AMCe 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

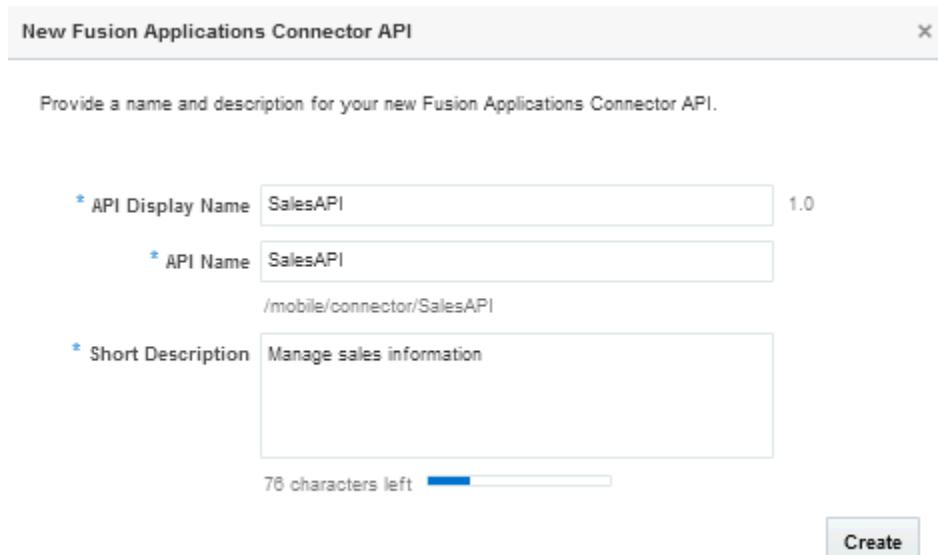
Before you begin configuring your connector, you must provide some initial basic information like the connector API name, a brief description, and a local URI (from which the connector API will be available to the custom code):

1. Click  and select **Development > 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.

2. 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.



New Fusion Applications Connector API

Provide a name and description for your new Fusion Applications Connector API.

* API Display Name: SalesAPI

* API Name: SalesAPI

/mobile/connector/SalesAPI

* Short Description: Manage sales information

78 characters left

Create

3. 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.

4. 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'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 Policies in *Managing Oracle Autonomous Mobile Cloud Enterprise*.

Remote Service Connection Settings

HTTP Read Timeout: 25.000 Milliseconds

HTTP Connection Timeout: 25.000 Milliseconds

- **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.
5. 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.
 6. 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.

General Resources Attributes Runtime Sec... Test Done Save Save and Close

Resources

Enter the Metadata URL and credentials to connect to an Oracle Fusion Applications instance. Once connected, we'll display a list of Resources which you can choose to expose from this Connector API. Resources are not selected by default.

* Describe URL: https://my-saaspaas-host.oracle.com/salesApi/resources/11.1.10/describe

* User Name: mary.jane-p1crm10

* Password: [REDACTED]

Connect

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, AMCe 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

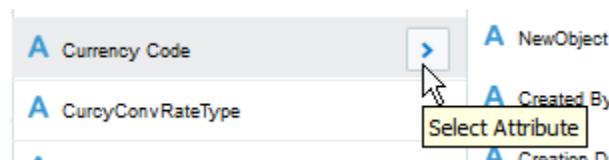
On the Attributes page, you can select the optional attributes you want for each of your selected resources. Any required attributes are automatically added to the configuration. Select a resource from the Resources list, view the available attributes for the resource in the next column, and then select the specific attributes you want to include in the connector configuration:

1. Click the **Attributes** navigation link.

Resources	Attributes	Select All	Selected Attributes	Remove All
NewObject	A LastUpdateLogin A CpdrfVerSor A UserLastUpdateDate A CpdrfLastUpd A CpdrfVerPillar	<input type="checkbox"/> Select All	# Record ID* A RowType* A NewObject Name* A Created By* A Creation Date*	<input type="button" value="Remove All"/>
ViewAccessor_CurrencyC...				
Forecast Territory Details				
Forecast Item	A JobDefinitionPackage A RequestId	<input type="checkbox"/> Select All	A Last Updated By* A Last Update Date*	<input type="button" value="Remove All"/>
Sales Lookup SDO	A JobDefinitionName A Currency Code	<input type="checkbox"/> Select All		<input type="button" value="Remove All"/>
LookupPVO	A CurcyConvRateType	<input type="checkbox"/> Select All		<input type="button" value="Remove All"/>
forecastProducts	A CorpCurrencyCode A SourceType	<input type="checkbox"/> Select All		<input type="button" value="Remove All"/>

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:

The screenshot shows a user interface for managing selected attributes. At the top, there is a header labeled "Selected Attributes" and a "Remove All" button. Below this, there is a list box containing one item: "Record ID". To the left of the list box is a small icon of a person with a plus sign. Underneath the list box are two input fields: "Name" which contains "Record ID" and "Description" which is empty.

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 you 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.

Available Policies		Selected Policies	
HTTP Token Client Policy (Basic Auth Policy)		SAML Bearer Token Over SSL Client Policy	
OAuth2.0 Config Client Policy		>	
OAuth2.0 Token Client Policy		>>	<
OAuth2.0 Token Over SSL Client Policy		<<	

Policy Description		Policy Overrides	
<p>Includes an 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 a oauth2_config_client_policy together. This version of the policy enforces that connections are made over https.</p>		<p><code>oracle/wss_saml_token_bearer_over_ssl_client_policy</code></p> <p>?user.attributes (Default value null)</p> <p>?saml.issuer.name (Default value www.oracle.com)</p> <p>?user.roles.include (Default value false)</p> <p>?csf-key (Default is basic.credentials)</p>	

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 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 Credentials (CSF Keys and Certificates) in *Managing Oracle Autonomous Mobile Cloud Enterprise*.

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.

The screenshot shows the Oracle Fusion Applications Connector API documentation. On the left, a sidebar lists various endpoints under the 'NewObject' category, including 'GET List NewObject' which is currently selected. The main panel displays the 'List NewObject' endpoint details. It includes a 'Request' tab and a 'Response' tab. Under the 'Request' tab, there are two parameters: 'q' and 'offset'. The 'q' parameter is described as a string that queries the resource collection using expressions. An example provided is 'Deptno>=10 and <= 30;Loc!=NY'. The 'offset' parameter is described as an integer defining the starting position of the resource collection. An example provided is '20'. There is also a 'Schema' link for each parameter. At the top right, there is a yellow button labeled 'Default Test Credentials'.

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, `PersonId`, `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* in *Managing Oracle Autonomous Mobile Cloud Enterprise*.

Getting the Test Results

Test results are displayed at the bottom of the Test page. The result indicator is the response status:

- 2xx: indicates a successful connection
- 4xx: indicates a user error occurred
- 5xx: 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	AMCe 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 AMCe 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 AMCe, 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 [Credentials \(CSF Keys and Certificates\)](#) in *Managing Oracle Autonomous Mobile Cloud Enterprise*.

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 (4xx) and server (5xx) 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 [AMCe Policies and 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.

21

Diagnostics

The Diagnostics features of AMCe 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?

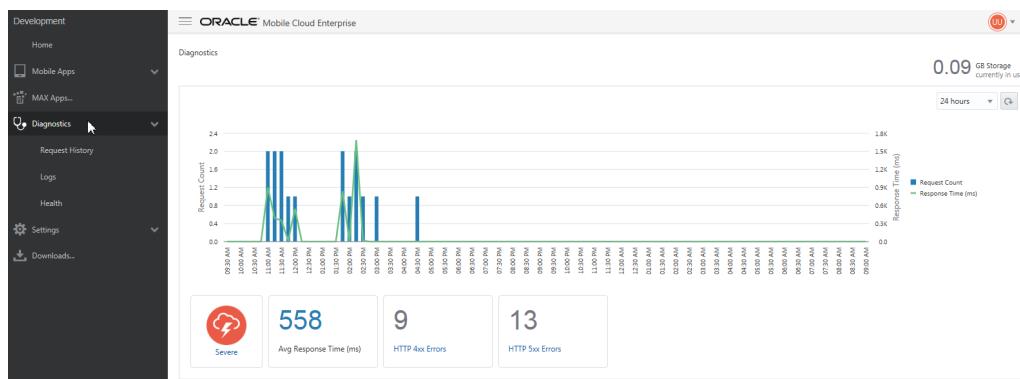
Whether you're a developer tracing errors in custom code, or an administrator who notices a flurry of 5xx responses, Diagnostics lets you easily find out what's going on by providing you with increasingly detailed levels of logging messages.

The Diagnostics landing page provides a high-level view that includes a traffic-light indicator that conveys 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. This page provides 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. To see specific logs, check out [Viewing Underperforming Requests](#).

Although admins and developers can both benefit from diagnostics, each uses it differently. As pointed out in [Monitoring Environments for a Selected Mobile Backend](#), developers typically use a backend's diagnostics as the starting point in their debugging efforts. To get an idea how developers go through their paces see [Use Case: Using Correlation to Diagnose Custom Code](#). While developers focus on a backend, administrators instead monitor the big picture for a system. For an example of how an 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 on the Diagnostics page depict the overall health of an environment for the last hour, or other selected time period.



AMCe bases this at-a-glance view on the fine-grained health metrics for that environment. When the percentage of error responses exceed configured thresholds for the selected time period, the traffic light indicator changes from green (normal) to amber (warning) or red (severe).

The time period for analysis can be changed using the menu, with additional choices for 2, 6, 12, or 24 hours.

Viewing Server Load

As part of the overall portrait of health at any given moment, the Diagnostics page includes a timeline that plots a recent history of the number of requests and response times.

Viewing Errors

The Diagnostics page notes the number of client (4xx) and server (5xx) errors that have occurred within the last hour or other selected time period. See [Viewing Status Codes for API Calls and Outbound Connector Calls](#).

Viewing Underperforming Requests

The high-level data shown on the Diagnostics page is the entry point for increasingly detailed levels of analysis. When you hover over an indicator, the traffic light indicator shows the percentage of failed requests. This data is derived from the last hour or other selected time period of the system's behavior and highlights the severity of an issue 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, and by viewing the API history log data to 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, AMCe 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, 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. See [Relating Log Messages](#). For more information about the various logs

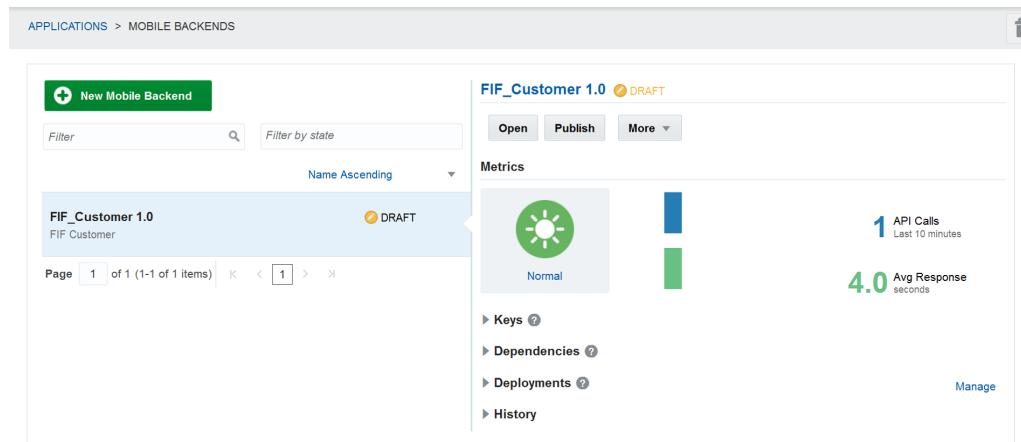
generated by Diagnostics (such as the API History, Connector History, Custom Code, and System logs), see [Viewing Log Messages](#).

Viewing Storage Usage

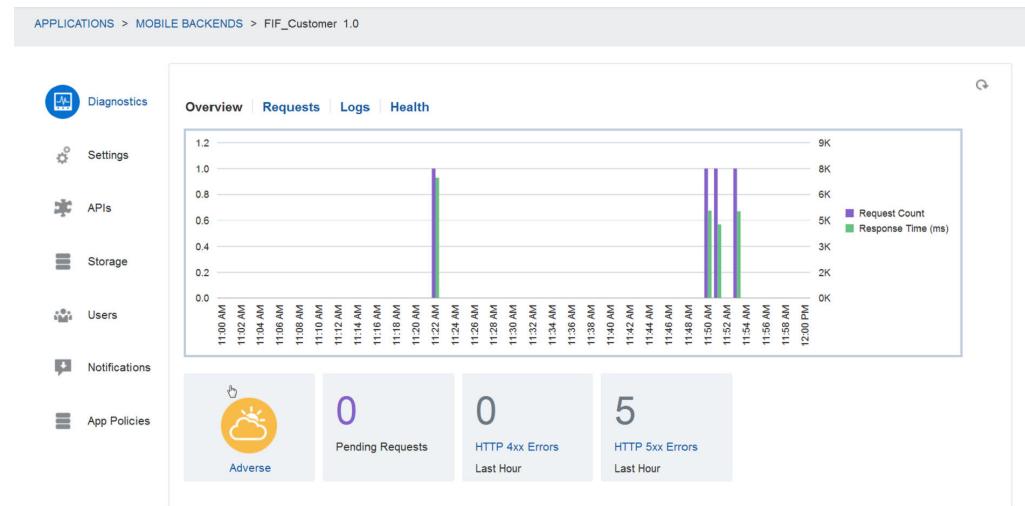
In addition to showing API request data, the Diagnostics page shows you how much database storage, shown in gigabytes, the environment is currently using. You can see this information in the top right corner of the page.

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 Diagnostics 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 backend, you can focus your attention where it's needed: on specific performance issues or problems with the API implementations and connectors used by the 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 backend or to an entire environment. For each API, AMCe records the same error and request handling metrics that it applies to a backend. You can drill down to see how the API endpoints behave in terms of these performance metrics.

From Diagnostics, click Health to view the APIs for a backend . You can also open this page by clicking the traffic light indicator on the Diagnostics page. If the traffic light indicator is amber or red, then you can quickly investigate the cause of the problem by using the Health page.

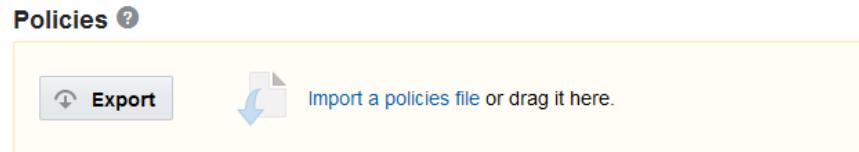
Diagnostics > Health					
<input type="text" value="Filter by Backend name"/> <input type="button" value="Search"/> 24 hours <input type="button" value="Reset"/>					
Name	2xx Count	4xx Count	5xx Count	Avg Response Time (ms)	Error (%)
DefaultMBE_86461_Oak	27850	16097	16097	1752.6	53
DefaultAPI_86461_sawUp_Oak	27850	16097	16097	1752.6	53
/default	16094	0	0	8.4	0
/defaultWithAutomaticPause	11756	0	0	4140.5	0
/status4xx	0	16097	0	0	100
/status5xx	0	0	16097	0	100

Page 1 of 1 (1 of 1 items) | < 1 > | ⌂

Adjusting the Performance Threshold Configurations

The default thresholds may not apply at all phases of the backend's lifecycle and may not always reflect your interpretation of a healthy environment. To adjust the

thresholds, administrators can get 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.



For more information on policies, see [Oracle Autonomous Mobile Cloud Enterprise 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 screenshot shows the Request History page of the Oracle Autonomous Mobile Cloud Enterprise Diagnostics section. The left sidebar has 'Development' selected, with 'Request History' highlighted. The main area has a title 'ORACLE Mobile Cloud Enterprise' and 'Diagnostics > Request History'. It includes a filter bar for 'backend or API', time range buttons (5m, 1h, 1d, 10/06/1, 10/07/1), and status code buttons (1xx, 2xx, 3xx, 4xx, 5xx). The 'Request History' tab is selected. The table displays the following data:

Related	Timestamp	Status	Call	Path	Time	Req Length	Resp Length
	10/6/17 4:16:44.647 pm -0700	400	PizzaStore 1.0 > custom/PizzaBotAPI 1.0	POST /components/{componentName}	1004	633	1356
	10/6/17 4:15:57.372 pm -0700	500	PizzaStore 1.0 > custom/PizzaBotAPI 1.0	POST /components/{componentName}	2471	679	0
	10/6/17 3:54:27.547 pm -0700	500	PizzaStore 1.0 > custom/PizzaBotAPI 1.0	POST /components/{componentName}	1302	679	0
	10/6/17 3:53:39.379 pm -0700	500	PizzaStore 1.0 > custom/PizzaBotAPI 1.0	POST /components/{componentName}	1387	680	103
	10/6/17 3:52:01.483 pm -0700	500	PizzaStore 1.0 > custom/PizzaBotAPI 1.0	POST /components/{componentName}	1591	679	102
	10/6/17 3:48:34.964 pm -0700	500	PizzaStore 1.0 > custom/PizzaBotAPI 1.0	POST /components/{componentName}	903	508	102
	10/6/17 3:46:54.188 pm -0700	500	PizzaStore 1.0 > custom/PizzaBotAPI 1.0	POST /components/{componentName}	5048	480	0
	10/6/17 3:44:55.900 pm -0700	400	PizzaStore 1.0 > custom/PizzaBotAPI 1.0	POST /components/{componentName}	875	460	1198

At the bottom left, there is a 'Page' indicator showing '1 of 1 (0 of 24 items)' with a small navigation icon.

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-sect10.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.

Time	Req Length	Resp Length	Related
5883	171	79	
3942	171		 Log Messages Related by API Request Log Messages Related by Mobile App Session Log Messages Related by Mobile Device Log Messages Related by User
5013	171		
4284	171		
5055	171	79	

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 AMCe 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

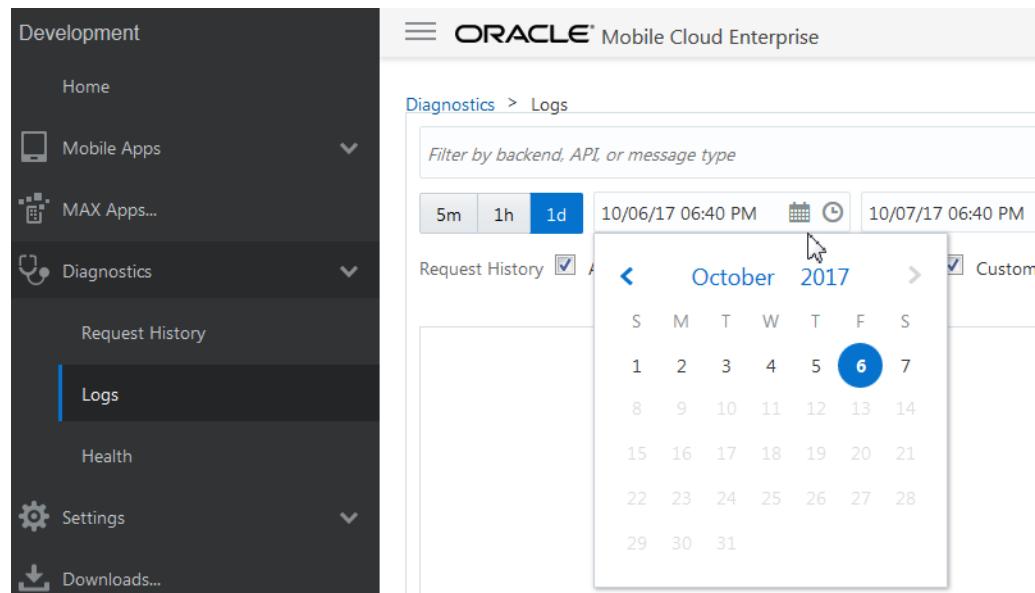
You can access this page by selecting from the logging options in the Related column in the Request History page, or by clicking **Logs** on the top-level health page.

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** in Diagnostics. 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 by AMCe (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.

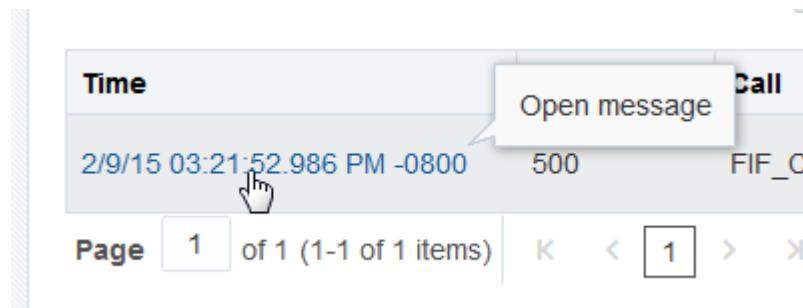


You can also apply filters, so that you can view messages by message type, backend, backend version, and API name. You can add filters by selecting from the drop down list, or by entering some criteria in the **Filters** field. For example, if you're interested in a particular backend, then enter its 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](#).

Message Detail

Message 1 of 7 Filter logs by ...

Overview | Headers

API Request: incidentreport 1.0 - HTTP 500 (anonymous)

incidentreport 1.0.1

POST /internal-tools/env/1.0/ui-tooling/endpoint/mobile/custom/incidentreport/contacts

Message The API invocation ended.
Text

Request

Mobile Backend FIF_Customer 1.0
API incidentreport 1.0
Service incidentreport 1.0.1
URI /internal-rt/mobile/custom/incidentreport/contacts
HTTP Method POST

Performance

Elapsed Time 3942 ms
Service Time 3813 ms
Start Time
End Time
Request Length 171 bytes
Response 79 bytes
Length

Message

Message Id MOBILE-38592
Message Level SEVERE
Status 500 - Internal Server Error

Context

Username anonymous
Request 00586ui0mDfFW7Y5Hrc9yf0007z'0000w2
Correlation Id

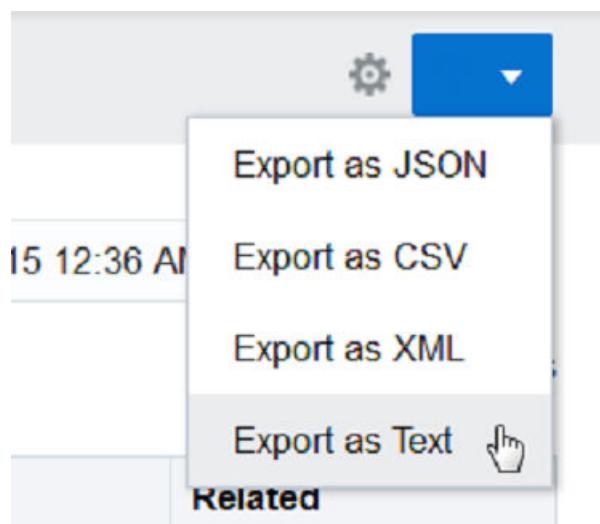
To get further diagnostics data from the Oracle stack (and any system, API, or connector messages that may have been logged with the same Correlation ID), click the **Request Correlation ID funnel** to view the logging messages that have been tagged with the request's ID. You can control the volume and 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.

The screenshot shows the 'Message Detail' page in the Oracle Mobile Cloud Service. At the top, it says 'APPLICATIONS > MOBILE BACKENDS > FIF_Customer 1.0'. On the right, it shows 'Development uimcs@mobile.oracle.com'. Below the header, there's a 'Message Detail' section with a sub-section titled 'Request Headers'. This section lists various HTTP headers: content-type, connection, oracle-mobile-api-version, host, content-length, accept, user-agent, accept-encoding, ecid-context, Date, Vary, Content-Length, oracle-mobile, cc-unhandled-error, Connection, Content-Type, and X-Powered-By. Below the 'Request Headers' is another section titled 'Response Headers' which contains similar header information.

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: testMobileUser0fff081190f4cbc89ef0189f1ec9e8a].
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 the AMCe core runtime. For example, tid:61.
ECID	The execution context in which the request has been dispatched by the AMCe core runtime.
RID	The Relationship ID of the execution context. The RID tracks any subrequests called by the AMCe 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:-8000-000000000000c2ba7,0]
[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 the AMCe service consumed by the API. For example, [SVC_NAME: storage].
SVC_TYPE	The AMCe service type.
SVC_VER	The version of the AMCe service consumed by the API.
SVC_PARAM	The service parameters of the AMCe 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 to AMCe 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 the AMCe server spent processing the request. This includes dispatching time and service time.
SVC_TIME	The total time (in milliseconds) that the AMCe 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	The AMCe error message ID, which is supplied by the AMCe 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:-8000-000000000001caa,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 to AMCe 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	PROTCL: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
ENDPOINT_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 5xx 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.

- 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 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:

- Click **HTTP 5xx Errors** to open the Request History page.
- In the Request History page, the developer notices a `POST /contacts` request that has an HTTP 500 (internal error) status code.

Related	Timestamp	Status	Call	Path	Time	Req Length	Resp Length
	10/6/17 4:16:44.647 pm -0700	400	PizzaStore 1.0 > custom/PizzaBotAPI 1.0	POST /components/{componentName}	1004	633	1356
	10/6/17 4:15:57.372 pm -0700	500	PizzaStore 1.0 > custom/PizzaBotAPI 1.0	POST /components/{componentName}	2471	679	0

- 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.

Request	Performance
Mobile Backend FIF_Customer 1.0 API incidentreport 1.0 Service incidentreport 1.0.1 URI /internal-rt/mobile/custom/incidentreport/contacts HTTP Method POST	Elapsed Time 1408 msec Service Time 1294 msec Start Time 2/9/15 03:21:51.578 PM -0800 End Time 2/9/15 03:21:52.986 PM -0800 Request Length 170 bytes Response Length 79 bytes

- 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.
 - 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."

MOBILE APPS > BACKENDS > FIF 1.0

Diagnostics Overview Requests Logs Health

ecid:1000000000 X Server Settings

5m 1h 1d 10/10/17 01:23 pm 10/10/17 02:23 pm Search Message text Search

Settings Clients Security APIs Storage Notifications App Policies

Request History APIs Connectors Diagnostic Logs Custom Code System

Related	Timestamp	Message Type	Call	Message
≡	10/10/17 3:14:52.402 pm +01:00	SEVERE	FIF 1.0 > custom/FIFSampleAPI 1.0 > GET /customer	The API invocation ended.
≡	10/10/17 3:14:52.401 pm +01:00	WARNING	FIF 1.0 > custom/FIFSampleAPI 1.0	The Node server running on port 60502 has closed.
≡	10/10/17 3:14:52.400 pm +01:00	WARNING	FIF 1.0 > custom/FIFSampleAPI 1.0	The Node server running on port 60502 has stopped taking requests.
≡	10/10/17 3:14:52.397 pm +01:00	SEVERE	FIF 1.0 > custom/FIFSampleAPI 1.0	Custom Code Problem: request: get /mobile/custom/FIFSa...
≡	10/10/17 3:14:52.394 pm +01:00	FINE	FIF 1.0 > custom/FIFSampleAPI 1.0	/mobile/custom/FIFSampleAPI/customer
≡	10/10/17 3:14:52.373 pm +01:00	INFO	FIF 1.0 > custom/FIFSampleAPI 1.0	The Node server is listening at port 60502
≡	10/10/17 3:14:51.946 pm +01:00	INFO	FIF 1.0 > custom/FIFSampleAPI 1.0 > GET /customer	The API invocation ended.

Page 1 of 1 (1-7 of 7 items) K < 1 > >>

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.error('settings are zero');
186     }
187 }

```

- 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:

- Click **HTTP 5xx Errors** on the Diagnostics 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.

Select filter criteria							5m	1h	1d	02/09/15 02:21 PM	02/09/15 03:23 PM	10 messages
Request History		APIs	Connectors	1xx	2xx	3xx	4xx	5xx				
Time	Status	Call	Path	Time (ms)	Req Length...	Resp Length...	Related Logs					
2/9/15 06:01:22.641 PM -0800	500	FIF_Customer 1.0 > incidentreport 1.0	GET /incidents/{id}	27371	140	140						
2/9/15 06:01:22.534 PM -0800	500	FIF_Customer 1.0 > RightNow 1.0	POST /GetIncidentById	26130	61	61						
2/9/15 06:00:52.630 PM -0800	500	FIF_Customer 1.0 > incidentreport 1.0	GET /incidents/{id}	27143	140	140						
2/9/15 06:00:52.528 PM -0800	500	FIF_Customer 1.0 > RightNow 1.0	POST /GetIncidentById	25665	61	61						
2/9/15 06:00:22.637 PM -0800	500	FIF_Customer 1.0 > incidentreport 1.0	GET /incidents/{id}	31672	140	140						
2/9/15 06:00:22.532 PM -0800	500	FIF_Customer 1.0 > RightNow 1.0	POST /GetIncidentById	30530	61	61						
2/9/15 05:56:32.637 PM -0800	500	FIF_Customer 1.0 > incidentreport 1.0	GET /incidents/{id}	29556	140	140						
2/9/15 05:56:32.531 PM -0800	500	FIF_Customer 1.0 > RightNow 1.0	POST /GetIncidentById	28420	61	61						
2/9/15 05:55:22.660 PM -0800	500	FIF_Customer 1.0 > incidentreport 1.0	GET /incidents/{id}	34142	140	140						
2/9/15 05:55:22.530 PM -0800	500	FIF_Customer 1.0 > RightNow 1.0	POST /GetIncidentById	31634	61	61						

- 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.

The screenshot shows the 'Message Details' interface. At the top, it says 'Message 1 of 10'. Below that are tabs for 'Overview' and 'Headers'. The 'Overview' tab is selected, displaying the following details:

- API Request:** incidentreport 1.0 - HTTP 500 (anonymous)
- incidentreport 1.0.1**
- GET /internal-rt/mobile/custom/incidentreport/incidents/1**
- Message Text:** The API invocation ended.

Below this, there are two main sections: 'Request' and 'Performance'.

Request:

- Mobile Backend FIF_Customer 1.0
- API incidentreport 1.0
- Service incidentreport 1.0.1
- URI /internal-rt/mobile/custom/incidentreport/incidents/1
- HTTP Method GET
- HTTP Parameters

Performance:

- Elapsed Time 27371 msec
- Service Time 27274 msec
- Start Time 2/9/15 06:00:55.270 PM -0800
- End Time 2/9/15 06:01:22.641 PM -0800
- Response Length 140 bytes

At the bottom, under 'Message' and 'Context', it shows:

- Message Id MOB-38592
- Message Level SEVERE
- Status 500 - Internal Server Error
- Username anonymous
- Request Correlation Id 7ce737488d3b695ff610dbf10:14b3a999c9a-8000-000000000009ee92

- 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.

- 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 MOB-16006) and provides the service name (incidentService) and port (7002) along with a tip: Check the validity of the SOAP connector configuration.

The screenshot shows the 'Message Details' interface. At the top, it says 'Message 3 of 4'. Below that are tabs for 'Overview' and 'Headers'. The 'Overview' tab is selected, displaying the following details:

- API Request:** RightNow 1.0 - HTTP 500 (anonymous)
- RightNow 1.0**
- POST /internal-rt/mobile/connector/RightNow/GetIncidentByld**

Below this, there are two main sections: 'Request' and 'Performance'.

Request:

- Mobile Backend FIF_Customer 1.0
- API RightNow 1.0
- Service RightNow 1.0
- URI /internal-rt/mobile/connector/RightNow/GetIncidentByld
- HTTP Method POST
- HTTP Parameters

Performance:

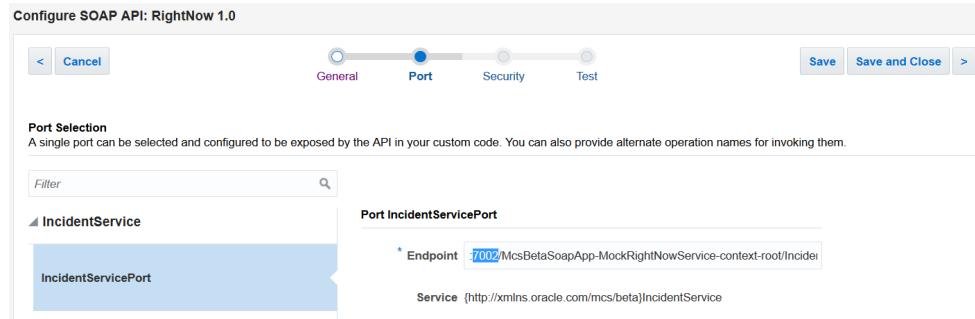
- Elapsed Time 26130 msec
- Service Time 26031 msec
- Start Time 2/9/15 06:01:22.534 PM -0800
- End Time 2/9/15 06:01:22.534 PM -0800
- Request Length 61 bytes

A blue callout box labeled 'Problematic Port' points to the 'Port' field in the 'Performance' section.

At the bottom, under 'Message' and 'Context', it shows:

- Message Id MOB-38592
- Message Level SEVERE
- Status 500 - Internal Server Error
- Username anonymous
- Request Correlation Id 7ce737488d3b695ff610dbf10:14b3a999c9a-8000-000000000009ee92

- 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.

GET /incidents/{id}
https://fif.oracle.com/{version}/mobile/custom/incidentreport/incidents/{id}

Retrieves the incident report with the specified id.

▶ **HTTP 200 OK.**
HTTP 404 The requested resource could not be found but may be available again in the future.

Parameters	Description	Test Console
PATH		
id Required	integer the unique id of the incident report	<input type="text" value="1"/>

▶ **Authentication**

▶ **Response Status: 200**

Request | **Response**

Request

```
HTTP/1.1 200 OK
Oracle-Mobile-Sync-Resource-Type: item
X-ORACLE-DMS-ECID: ce737488d3b695ff:610dbf10:14b3a999c9a:-8000-000000000009f985
Date: Tue, 10 Feb 2015 03:02:32 GMT
Transfer-Encoding: chunked
Content-Length: 225
oracle-mobile-runtime-version: 14.2.0.0.0
Connection: keep-alive
Content-Type: application/json
X-Powered-By: Express
X-Powered-By: Servlet/2.5 JSP/2.1
```

Response

```
{
  "id": 1,
  "title": "Title1",
  "createdon": "2015-01-30 07:11:08 PST",
  "status": "New",
  "priority": "Medium",
  "imagelink": "ImageLink3",
  "technician": "jill@fixit.com",
  "contact": {
```

Packages

Oracle Autonomous Mobile Cloud Enterprise (AMCe) lets you share and move bundles of related artifacts built in AMCe to another instance of AMCe. You do this by exporting artifacts along with their dependencies, which creates a package, and importing that package to other instances of AMCe.

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 AMCe. If artifacts in the current instance of AMCe 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.

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 AMCe. 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 AMCe, you can *export* the artifacts (that is, create a package) in the current instance and import them into the target instance of AMCe 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 AMCe 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 AMCe. 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.

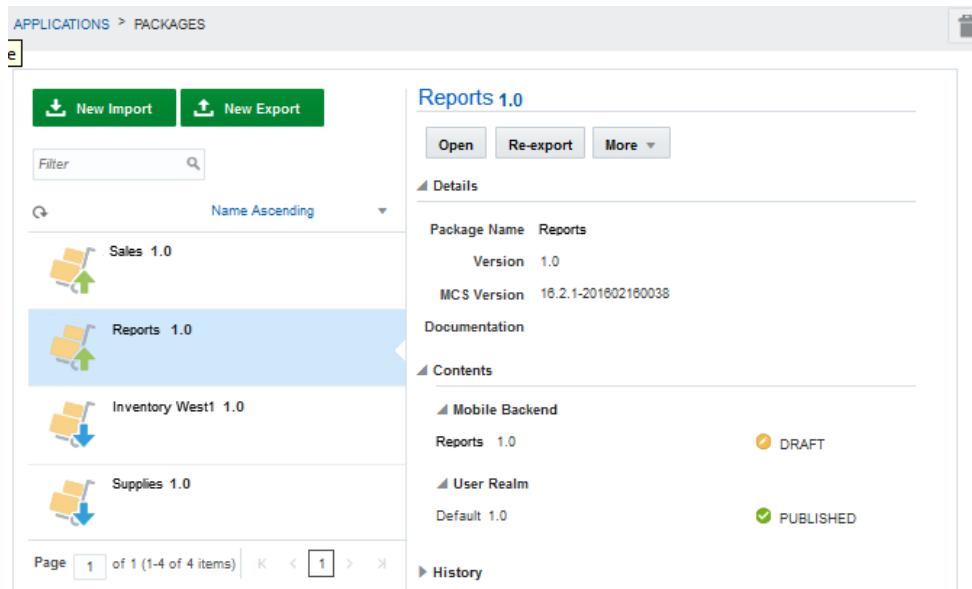
The Export Package wizard walks you through the following steps to export a package:

- [Adding Artifacts to the Package](#)
- [Reviewing Dependencies During Export](#)
- [Setting Environment Policies During Export](#)
- [Completing the Export](#)

Adding Artifacts to the Package

1. Click  and select **Applications > Packages** from the side menu.

If there are existing import and export packages, you'll see a list of packages.



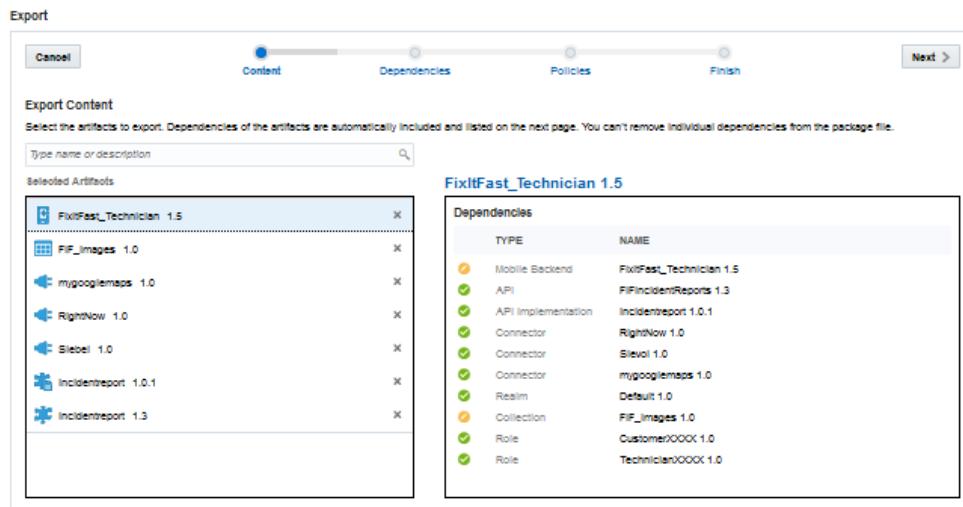
The screenshot shows the AMCe interface with the 'APPLICATIONS > PACKAGES' path selected. On the left, a list of packages is shown with icons for import/export: Sales 1.0 (import), Reports 1.0 (selected, export), Inventory West1 1.0 (import), and Supplies 1.0 (import). The 'Reports 1.0' package is selected, and its details are displayed on the right. The 'Details' section shows the package name 'Reports', version '1.0', and MCS Version '16.2.1-201602160038'. The 'Contents' section lists 'Mobile Backend' (Reports 1.0, DRAFT) and 'User Realm' (Default 1.0, PUBLISHED).

Up 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.

2. Click **New Export**.
3. 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.



4. 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.

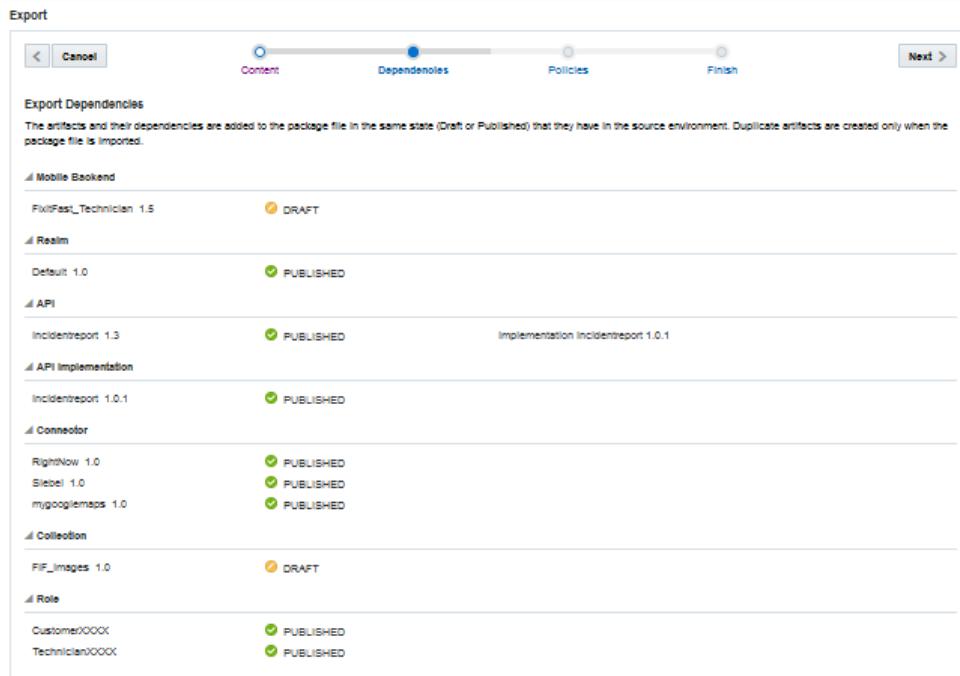
5. Click **Next (>)** to go to the next step.

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 All** 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.

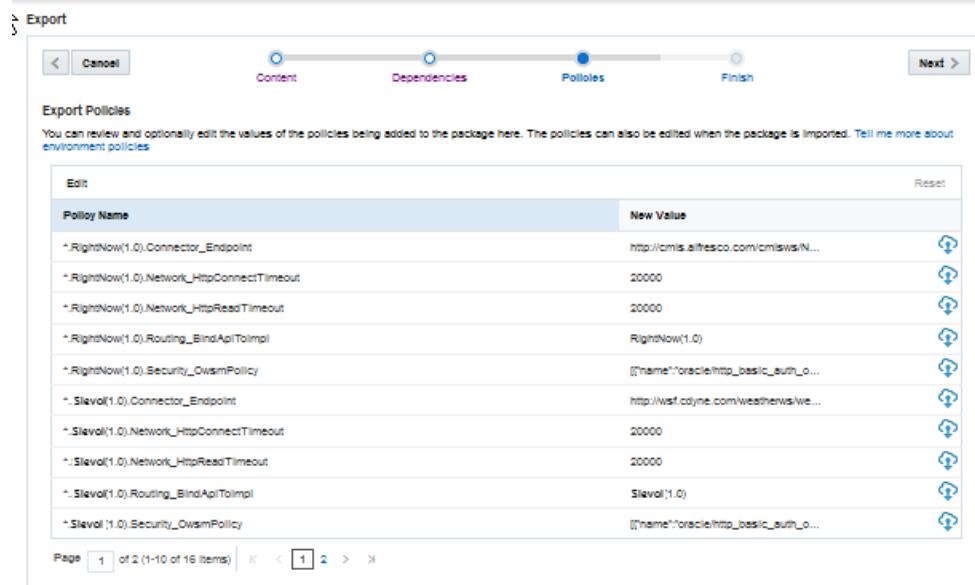
You can save some time by setting values now if you know what values will be required. For example, if a connector API is in the package, you may want to change the security policy. If a mobile backend is being exported, you may want to reset the `Sync_CollectionTimeoutToLive` policy. Another example 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 [AMCe Policies and Values](#).

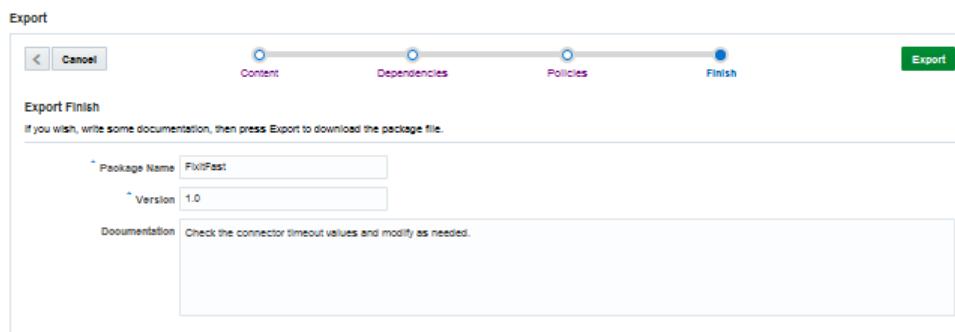
Completing the Export

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 policies, 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.

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

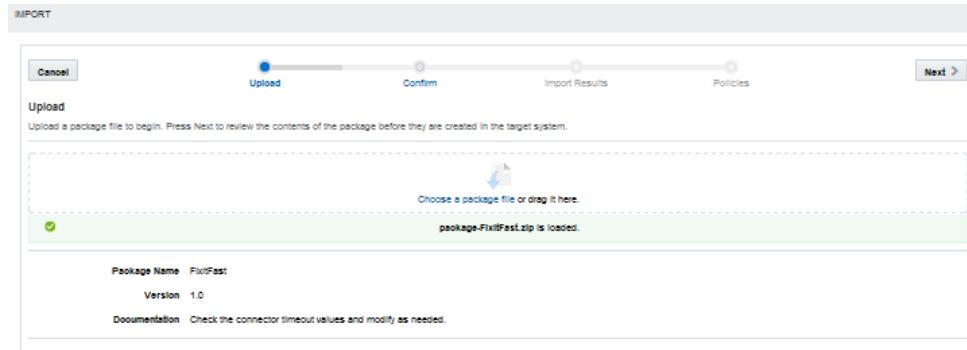
When you upload the package, the contents of the package are immediately installed in the target environment unless a conflict or some other error occurs during the import. You can view the contents of the package and whether or not all of the contents were successfully imported on the next page of the 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 [Notification Profiles and Client Apps](#).

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 AMCe.

The Import Results page shows the artifacts that have been installed.

The screenshot shows the 'Import' interface with the 'Import Results' step selected. The results table lists various artifacts and their status:

Category	Name	Status	Notes
Mobile Backend	FixItFast_Technician 1.5	IMPORTED	
User Realm	Default 1.0	EXISTS	
API	IncidentReport 1.3	IMPORTED	Implementation IncidentReport 1.0.1
API Implementation	IncidentReport 1.0.1	IMPORTED	
Connector	RightNow 1.0	IMPORTED	
	Globe 1.0	IMPORTED	
	mygooglegroups 1.0	IMPORTED	
Collection	FIF_Images 1.0	IMPORTED	
Role	CustomerXXXX	IMPORTED	
	TechnicianXXXX	IMPORTED	

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.

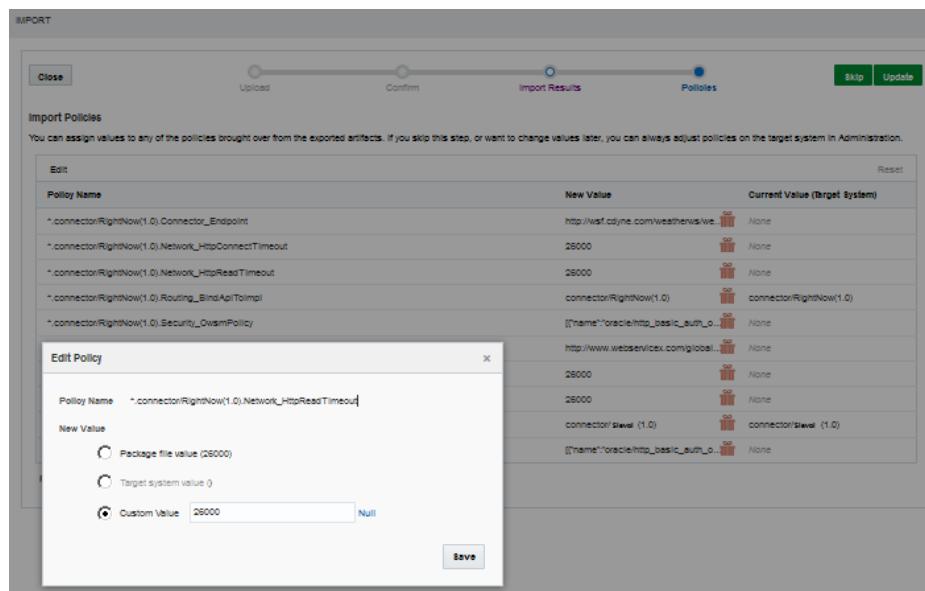
Even if you don't modify values for existing environment policies, any policies associated with the artifacts in the package that are new to the target environment are added for you when you update.

Check the documentation included in the package to see if any recommended values or policies are described. For descriptions of policies, see [AMCe Policies and Values](#).

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 AMCe 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 AMCe 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.
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 AMCe and you want those upgrades in another instance of AMCe? 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 AMCe 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 [AMCe Policies and Values](#).

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 Autonomous Mobile Cloud Enterprise (AMCe) 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 Autonomous Mobile Cloud Enterprise (AMCe) 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 AMCe, 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	<p>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.</p> <p>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.</p>	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	<p>The ID of the mobile backend issued by AMCe, which enables a mobile application to access APIs associated with that mobile backend.</p> <p>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.</p>	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':SSSZ.	Diagnostics
Oracle-Mobile-Content-Disposition	A request 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 Autonomous Mobile Cloud Enterprise Policies

This chapter lists the policies that you can configure in Oracle Autonomous Mobile Cloud Enterprise (AMCe). 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, or they can affect an individual artifact.

 **Note:**

The scope value shown is the narrowest level at which the property can be set.

AMCe Policies and Values

Policies determine the behavior of various aspects of AMCe. As AMCe administrator, you can view and modify the policies in the `policies.properties` file by exporting the file from the Administration page.

Policy	Type / Description	Default Value	Scope / Affects
Analytics_ApplicationGuid	String. Stores an association between the backend and the Analytics application. The value is the Application ID.	There is no default value for this policy.	Scope: Backend Affects: Backend
Analytics_BaiduMapCsfKeyName	String. Stores the name of the CSF key that stores the Baidu application key (ak).	There is no default value for this policy.	Scope: Backend Affects: Backend
Asset_AllowPurge	String. Controls whether or not Draft and Published artifacts in the trash can be purged (deleted permanently). Valid values are: <ul style="list-style-type: none">• All• None• Draft• Published	All	Scope: Instance Affects: Backend, Custom API, API Implementation, Connector, and Collection

Policy	Type / Description	Default Value	Scope / Affects
Asset_AllowTrash	<p>String. Controls whether or not Draft and Published artifacts can be moved to the trash.</p> <p>Valid values are:</p> <ul style="list-style-type: none"> • All • None • Draft • Published 	All	Scope: Instance Affects: Backend, Custom API, API Implementation, Connector, and Collection
Asset_AllowUntrash	<p>String. Controls whether or not Draft and Published artifacts can be restored from the trash.</p> <p>Valid values are:</p> <ul style="list-style-type: none"> • All • None • Draft • Published 	All	Scope: Instance Affects: Backend, Custom API, API Implementation, Connector, and Collection
Asset_DefaultInitialVersion	<p>String. Sets the default version for all newly created artifacts.</p> <p>Note: Generally, the default value should be used.</p>	1.0	Scope: Instance Affects: All artifacts that have versions
CCC_DefaultNodeConfiguration	<p>String. Sets the default node.js configuration used by the API implementation (custom code). The custom code implementation can override the default node configuration in its package.json.</p> <p>Valid values are:</p> <ul style="list-style-type: none"> • 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. <p>For the related JavaScript library versions, see What's the Foundation for the Custom Code Service?</p>	For AMCe instances provisioned before 18.2.5, the default is 6.10. For instances provisioned as 18.2.5 or above, the default is 8.9.4.	Scope: Instance Affects: Custom Code

Policy	Type / Description	Default Value	Scope / Affects
CCC_LogBody	Boolean. Determines whether to log the body of a request in custom code. Bodies will be logged in the following circumstances: <ul style="list-style-type: none"> • Logging level == FINEST or there is an uncaught exception. • This property is set to true. 	false	Scope: Backend Affects: Custom Code
CCC_LogBodyMaxLength	Integer. Sets the maximum number of characters to log if the custom code is logging the request body.	512	Scope: Backend Affects: Custom Code
CCC_SendStackTraceWithError	Boolean. 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.	false	Scope: Backend Affects: Custom Code
Connectors_Endpoint	String. Stores the endpoint URL of the particular connector instance. Set this policy by uncommenting the policy.	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	String. Identifies the JSON document representing connections to each configured ICS instance.	null	Scope: Instance Affects: ICS Connector

Policy	Type / Description	Default Value	Scope / Affects
Database_CreateTables Policy	<p>String. Controls whether the Database API can create, alter, or drop tables from custom code or SQL. The default value (allow) enables calls from custom code that perform implicit operations and also explicit query operations from raw SQL.</p> <p>Setting this policy to implicitOnly enables these operations and JSON from custom code calls, and prohibits SQL operations. Setting the policy to explicitOnly enables these operations using the Database Management Service API, and prohibits non-SQL operations from custom code. Setting the policy to none curtails implicit and explicit table creation, deletion, and updates.</p>	allow	Scope: Instance Affects: Database Service
Database_MaxRows	<p>Integer. Sets the maximum number of rows that can be returned by a single database query.</p>	1000	Scope: Instance Affects: Database Service
Database_QueryTimeout	<p>Integer. Sets the number of seconds to wait for a database query to return before canceling it.</p>	20	Scope: Instance Affects: Database Service
Diagnostics_ExcludedHttpHeadersInLogs	<p>String. Creates a list of headers that shouldn't be logged with each API request in the API History log file.</p>	Authorization header, cookie name	Scope: Instance Affects: Administration
Diagnostics_RequestPercentageErrorThreshold	<p>Double. Sets the percentage of requests returning error codes compared with total request above which the service will report an error condition.</p> <p>Set this value higher than the one set for the Diagnostics_RequestPercentageWarningThreshold policy, which sets the adverse level of system health.</p>	10	Scope: Instance Affects: Administration

Policy	Type / Description	Default Value	Scope / Affects
Diagnostics_RequestPercentageWarningThreshold	Double. Sets the percentage of requests returning error codes compared with total request above which the service will report a warning condition.	1	Scope: Instance Affects: Administration
Logging_Level	Integer. Sets the logging level.	800	Scope: Backend Affects: Custom APIs, Storage
Network_HttpConnectTimeout	Integer. 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.	There is no default value for this policy. The initial value is set when the connector is created.	Scope: Instance, Backend, Connector, Fully-Qualified Connector Affects: Connectors
Network_HttpReadTimeout	Integer. Sets the maximum time (in milliseconds) spent waiting to read data. The value should be less than the value of Network_HttpRequestTimeout.	There is no default value for this policy. The initial value is set when the connector is created.	Scope: Instance, Backend, Connector, Fully-Qualified Connector Affects: Connectors
Network_HttpRequestTimeout	Integer. Sets the amount of time in milliseconds (ms) on an HTTP request before it times out.	40,000 ms	Scope: Instance Affects: Custom APIs
Notifications_DeviceCountWarningThreshold	Double. 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 as needed.	70.0 Note: For testing purposes only, consider setting this value to 100.0 (100%).	Scope: Instance Affects: Notifications
Routing_BindAPIToImpl	String. Determines which core service to use to resolve the API request.	There is no default value for this policy.	Scope: API Affects: Custom APIs, Connectors
Routing_BindAPIToMock	Boolean. Resolves the API request to a mock service instead of the implementation that's bound to the API.	false Note: Do not modify this policy.	Scope: Fully-Qualified API Affects: Backend, Custom APIs

Policy	Type / Description	Default Value	Scope / Affects
Routing_DefaultImplementation	String. Specifies the default implementation for the initially created API (that is, the mock service).	MockService/1.0	Scope: Instance Affects: Custom APIs Note: Do not modify this policy.
Routing_RouteToBackend	String. Reroutes mobile API calls made to a backend to the target backend specified.	There is no default value for this policy.	Scope: Backend Affects: Dispatcher
Security_AllowOrigin	<p>String. Enables Cross Origin Resource Sharing (CORS) from HTML5 clients on an external domain.</p> <p>Supported values are:</p> <ul style="list-style-type: none"> • disallow • url1, url2, url3 - specifies a whitelist of URLs from which cross-site requests to 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. <p>The wildcard character, *, can be used when providing URL values but doesn't apply across dot (.), forward slash (/), or colon (:) characters.</p>	disallow	Scope: Instance Note: When dealing with browser-based applications, it's highly recommended that cross-site access to 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).
Security_AuthTokenConfiguration	JSON Object. Provides a configuration to integrate with third-party identity providers that support JWT, which mobile app users can use to authenticate.		Scope: Environment Affects: Security

Policy	Type / Description	Default Value	Scope / Affects
Security_CollectionsAnonymousAccess	A comma-separated list of storage collections following this pattern: <pre><collection1_name>[(<version> *) [,<collection2_name >[(<version> *)] [, ...]</pre>	No default value	Scope: Storage Collections Affects: Only the listed Collections
Security_ExposeHeaders	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.	""	Scope: Instance Affects: All cross origin calls to a given instance
Security_IdentityProviders	String. 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.	Indicates that no response headers are to be exposed to the browser.	Scope: Instance Affects: Security
Security_IgnoreHostNameVerification	Boolean. Disables the SSL host name verification. To be applied to connectors (in development) that call outbound services using SSL certificates with an invalid or incomplete hostname.	false	Scope: Instance Affects: REST, SOAP, ICS, and Fusion Applications Connectors

Policy	Type / Description	Default Value	Scope / Affects
Security_OwsmPolicy	Object. Sets the security policy used for outbound security.	There is no default value for this policy. The initial value is set when the connector is created.	Scope: Connector Affects: Connectors
Security_SsoRedirectWhitelist	String. Lists the URL patterns for the SSO redirect_uri parameter values that are permitted.	disallow	Scope: Instance, Backend Affects: SSO Token Relay
Security_TokenExchangeTimeoutPolicy	String. Defines the policy that governs the expiration time for tokens generated and issued as a result of token exchange. Valid values are: <ul style="list-style-type: none">• FromTimeoutSecs - token expiry time is governed by the Security_TokenExchangeTimeoutSecs policy.• FromExternalToken - token expiry time is set to the same time as the external token expiry time.• FromExternalTokenLimitedByTimeoutSecs - token expiry time is set to the value determined from the Security_TokenExchangeTimeoutSecs policy or the external token expiry time, whichever comes first.	FromTimeoutSeconds	Scope: Instance Affects: SSO Token Exchange
Security_TokenExchangeTimeoutSecs	Integer. Sets the token expiration time for SSO login.	216000 s	Scope: Instance Affects: SSO Token Relay

Policy	Type / Description	Default Value	Scope / Affects
Security_TransportSecurityProtocols	<p>String. Specifies a list of the TLS/SSL protocols that should be used for the outbound connection for the specific connector. By default, only TLSv1.2 protocols are used for outbound connections.</p> <p>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.</p> <p>Caution: Use this property carefully as older protocols are more vulnerable to security exploits.</p> <p>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.</p> <p>Supported protocols are: SSLv2Hello, TLSv1, TLSv1.1, TLSv1.2.</p>	No default value	<p>Scope: Connectors, Fully-qualified Connectors</p> <p>Affects: All Connectors</p>
Sync_CollectionTimeToLive	<p>Integer. 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.</p>	86400 s Set this policy as needed.	<p>Scope: Instance</p> <p>Affects: Storage</p>
Url_PercentEncodeQueryParameterSpaces	<p>Boolean. 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.</p>	false	<p>Scope: Connector</p> <p>Affects: REST Connector</p>

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
<code>http_basic_auth_over_ssl_client_policy</code>	Includes user name and password in an HTTP Basic Authorization header.
<code>http_jwt_token_client_policy</code>	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.
<code>http_jwt_token_identity_switch_client_policy</code>	Includes JWT token in the HTTP header. Similar to <code>http_jwt_token_client_policy</code> 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
<code>http_jwt_token_over_ssl_client_policy</code>	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.
<code>http_saml20_token_bearer_client_policy</code>	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.
<code>http_saml20_token_bearer_over_ssl_client_policy</code>	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.
<code>oauth2_config_client_policy</code>	Provides information about the OAuth2 server, which performs authorization and issues the access tokens. You must set both this policy and <code>oracle/http_oauth2_token_client_policy</code> together.
<code>http_oauth2_token_client_policy</code>	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 <code>oracle/oauth2_config_client_policy</code> together.
<code>http_oauth2_token_over_ssl_client_policy</code>	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 <code>oracle/oauth2_config_client_policy</code> 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 policies 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_no_timestamp_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 vouchers 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_message_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 performs 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: <ul style="list-style-type: none"> • 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: <ul style="list-style-type: none"> • * in any location • /* at the end of the URI • .* at the end of the URI
saml.enveloped.signature.require.d	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	<p>REST security policies:</p> <p>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</p> <p>SOAP security policies:</p> <p>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</p> <p>Fusion Applications security policies:</p> <p>wss_http_token_client_policy wss_saml_token_bearer_over_ssl_client_policy</p> <p>ICS security policies:</p> <p>http_basic_auth_over_ssl_client_policy</p>

Property	Security Policies Containing the Property
csf.map	ICS security policies: <code>http_basic_auth_over_ssl_client_policy</code>
federated.client.token	Fusion Applications security policies: <code>wss_http_token_client_policy</code> <code>wss_saml_token_bearer_over_ssl_client_policy</code> REST security policy: <code>http_jwt_token_identity_switch_client_policy</code> REST security policies: <code>http_oauth2_token_client_policy</code> <code>http_oauth2_token_over_ssl_client_policy</code> Fusion Applications security policies: <code>http_oauth2_token_client_policy</code> <code>http_oauth2_token_over_ssl_client_policy</code> REST security policies: <code>http_jwt_token_client_policy</code> <code>http_jwt_token_identity_switch_client_policy</code> <code>http_jwt_token_over_ssl_client_policy</code> <code>http_oauth2_token_client_policy</code> <code>http_oauth2_token_over_ssl_client_policy</code> Fusion Applications security policies: <code>http_oauth2_token_client_policy</code> <code>http_oauth2_token_over_ssl_client_policy</code> REST security policies: <code>http_jwt_token_client_policy</code> <code>http_jwt_token_identity_switch_client_policy</code> <code>http_jwt_token_over_ssl_client_policy</code> <code>http_oauth2_token_client_policy</code> <code>http_oauth2_token_over_ssl_client_policy</code> Fusion Applications security policies: <code>http_oauth2_token_client_policy</code> <code>http_oauth2_token_over_ssl_client_policy</code> REST security policies: <code>http_jwt_token_client_policy</code> <code>http_jwt_token_identity_switch_client_policy</code> <code>http_jwt_token_over_ssl_client_policy</code> <code>http_oauth2_token_client_policy</code> <code>http_oauth2_token_over_ssl_client_policy</code> Fusion Applications security policies: <code>http_oauth2_token_client_policy</code> <code>http_oauth2_token_over_ssl_client_policy</code>
include.certificate	REST security policies: <code>http_jwt_token_client_policy</code> <code>http_jwt_token_identity_switch_client_policy</code> <code>http_jwt_token_over_ssl_client_policy</code> <code>http_oauth2_token_client_policy</code> <code>http_oauth2_token_over_ssl_client_policy</code> Fusion Applications security policies: <code>http_oauth2_token_client_policy</code> <code>http_oauth2_token_over_ssl_client_policy</code> REST security policies: <code>http_jwt_token_client_policy</code> <code>http_jwt_token_identity_switch_client_policy</code> <code>http_jwt_token_over_ssl_client_policy</code> <code>http_oauth2_token_client_policy</code> <code>http_oauth2_token_over_ssl_client_policy</code> Fusion Applications security policies: <code>http_oauth2_token_client_policy</code> <code>http_oauth2_token_over_ssl_client_policy</code> REST security policies: <code>http_jwt_token_client_policy</code> <code>http_jwt_token_identity_switch_client_policy</code> <code>http_jwt_token_over_ssl_client_policy</code> <code>http_oauth2_token_client_policy</code> <code>http_oauth2_token_over_ssl_client_policy</code> Fusion Applications security policies: <code>http_oauth2_token_client_policy</code> <code>http_oauth2_token_over_ssl_client_policy</code> REST security policies: <code>http_jwt_token_client_policy</code> <code>http_jwt_token_identity_switch_client_policy</code> <code>http_jwt_token_over_ssl_client_policy</code> <code>http_oauth2_token_client_policy</code> <code>http_oauth2_token_over_ssl_client_policy</code> Fusion Applications security policies: <code>http_oauth2_token_client_policy</code> <code>http_oauth2_token_over_ssl_client_policy</code>
issuer.name	REST security policies: <code>http_jwt_token_client_policy</code> <code>http_jwt_token_identity_switch_client_policy</code> <code>http_jwt_token_over_ssl_client_policy</code> <code>http_oauth2_token_client_policy</code> <code>http_oauth2_token_over_ssl_client_policy</code> Fusion Applications security policies: <code>http_oauth2_token_client_policy</code> <code>http_oauth2_token_over_ssl_client_policy</code> REST security policies: <code>http_jwt_token_client_policy</code> <code>http_jwt_token_identity_switch_client_policy</code> <code>http_jwt_token_over_ssl_client_policy</code> <code>http_oauth2_token_client_policy</code> <code>http_oauth2_token_over_ssl_client_policy</code> Fusion Applications security policies: <code>http_oauth2_token_client_policy</code> <code>http_oauth2_token_over_ssl_client_policy</code>

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	<p>REST security policies:</p> <p>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</p> <p>SOAP security policies:</p> <p>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</p> <p>Fusion Applications security policies:</p> <p>http_oauth2_token_client_policy http_oauth2_token_over_ssl_client_policy wss_saml_bearer_token_over_ssl_client_policy</p>

Property	Security Policies Containing the Property
oauth2.client.csf.key	<p>REST security policies:</p> <p><code>http_oauth2_token_client_policy</code> <code>http_oauth2_token_over_ssl_client_policy</code></p> <p>Fusion Applications security policies:</p> <p><code>http_oauth2_token_client_policy</code> <code>http_oauth2_token_over_ssl_client_policy</code></p>
propagate.identity.context	<p>REST security policies:</p> <p><code>http_jwt_token_client_policy</code> <code>http_jwt_token_identity_switch_client_policy</code> <code>http_jwt_token_over_ssl_client_policy</code> <code>http_saml20_token_bearer_client_policy</code> <code>http_saml20_token_bearer_over_ssl_client_policy</code> <code>http_oauth2_token_client_policy</code> <code>http_oauth2_token_over_ssl_client_policy</code></p> <p>SOAP security policies:</p> <p><code>wss_saml_token_bearer_client_policy</code> <code>wss_saml_token_bearer_over_ssl_client_policy</code> <code>wss_saml20_token_bearer_over_ssl_client_policy</code> <code>wss_saml20_token_over_ssl_client_policy</code> <code>wss10_saml_token_client_policy</code> <code>wss10_saml_token_with_message_integrity_client_policy</code> <code>wss10_saml_token_with_message_protection_client_policy</code> <code>wss10_saml20_token_client_policy</code> <code>wss10_saml20_token_with_message_protection_client_policy</code> <code>wss10_saml_token_with_message_protection_ski_bas_ic256_client_policy</code> <code>wss11_saml_token_with_message_protection_client_policy</code> <code>wss11_saml20_token_with_message_protection_client_policy</code></p> <p>Fusion Applications security policies:</p> <p><code>http_oauth2_token_client_policy</code> <code>http_oauth2_token_over_ssl_client_policy</code> <code>wss_saml_token_bearer_over_ssl_client_policy</code></p>

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_protectio_n_client_policy wss10_username_token_with_message_protection_cli_ent_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_cli_ent_policy

Property	Security Policies Containing the Property
saml.audience.uri	<p>REST security policies:</p> <p><code>http_saml20_token_bearer_client_policy</code> <code>http_saml20_token_bearer_over_ssl_client_policy</code></p> <p>SOAP security policies:</p> <p><code>wss_saml_token_bearer_client_policy</code> <code>wss_saml_token_bearer_over_ssl_client_policy</code> <code>wss_saml20_token_bearer_over_ssl_client_policy</code> <code>wss_saml20_token_over_ssl_client_policy</code> <code>wss10_saml_token_client_policy</code> <code>wss10_saml_token_with_message_integrity_client_policy</code> <code>wss10_saml_token_with_message_protection_client_policy</code> <code>wss10_saml20_token_client_policy</code> <code>wss10_saml20_token_with_message_protection_client_policy</code> <code>wss10_saml_token_with_message_protection_ski_bas_ic256_client_policy</code> <code>wss11_saml_token_identity_switch_with_message_protection_client_policy</code> <code>wss11_saml_token_with_message_protection_client_policy</code> <code>wss11_saml20_token_with_message_protection_client_policy</code></p> <p>Fusion Applications security policies:</p> <p><code>wss_saml_token_bearer_over_ssl_client_policy</code></p>
saml.enveloped.signature.required	<p>REST security policies:</p> <p><code>http_saml20_token_bearer_client_policy</code> <code>http_saml20_token_bearer_over_ssl_client_policy</code></p> <p>SOAP security policies:</p> <p><code>wss_saml_token_bearer_client_policy</code> <code>wss_saml_token_bearer_over_ssl_client_policy</code> <code>wss_saml20_token_bearer_over_ssl_client_policy</code></p> <p>Fusion Applications security policies:</p> <p><code>wss_saml_token_bearer_over_ssl_client_policy</code></p>

Property	Security Policies Containing the Property
saml.issuer.name	<p>REST security policies:</p> <ul style="list-style-type: none"> http_saml20_token_bearer_client_policy http_saml20_token_bearer_over_ssl_client_policy <p>SOAP security policies:</p> <ul style="list-style-type: none"> 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_bas_ic256_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 <p>Fusion Applications security policies:</p> <ul style="list-style-type: none"> wss_saml_token_bearer_over_ssl_client_policy
scope	<p>REST security policies:</p> <ul style="list-style-type: none"> http_oauth2_token_client_policy http_oauth2_token_over_ssl_client_policy <p>Fusion Applications security policies:</p> <ul style="list-style-type: none"> http_oauth2_token_client_policy http_oauth2_token_over_ssl_client_policy

Property	Security Policies Containing the Property
subject.precedence	<p>REST security policies:</p> <p>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</p> <p>SOAP security policies:</p> <p>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_bas ic256_client_policy wss11_saml_token_identity_switch_with_message_pr otection_client_policy wss11_saml_token_with_message_protection_client policy wss11_saml20_token_with_message_protection_clien t_policy</p> <p>Fusion Applications security policies:</p> <p>wss_saml_token_bearer_over_ssl_client_policy</p>
token.uri	<p>REST security policy:</p> <p>oauth2_config_client_policy</p> <p>Fusion Applications security policies:</p> <p>http_oauth2_config_client_policy</p>

Property	Security Policies Containing the Property
user.attributes	<p>REST security policies:</p> <p>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</p> <p>SOAP security policies:</p> <p>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_bas_ic256_client_policy wss11_saml_token_with_message_protection_client_policy wss11_saml20_token_with_message_protection_client_policy</p> <p>Fusion Applications security policies:</p> <p>http_oauth2_token_client_policy http_oauth2_token_over_ssl_client_policy wss_saml_token_bearer_over_ssl_client_policy</p>

Property	Security Policies Containing the Property
user.roles.include	<p>REST security policies:</p> <p>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</p> <p>SOAP security policies:</p> <p>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_bas ic256_client_policy wss11_saml_token_identity_switch_with_message_pr otection_client_policy wss11_saml_token_with_message_protection_client_
 policy wss11_saml20_token_with_message_protection_clien t_policy</p> <p>Fusion Applications security policies:</p> <p>http_oauth2_token_client_policy http_oauth2_token_over_ssl_client_policy wss_saml_token_bearer_over_ssl_client_policy</p>

Property	Security Policies Containing the Property
user.tenant.name	<p>REST security policies:</p> <p>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</p> <p>SOAP security policies:</p> <p>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_bas_ic256_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</p> <p>ICS security policies:</p> <p>http_basic_auth_token_over_ssl_client_policy http_username_token_over_ssl_client_policy</p> <p>Fusion Applications security policies:</p> <p>wss_http_token_client_policy</p>

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

Writing Swift Applications Using the iOS Client SDK

You can also use the Oracle Autonomous Mobile Cloud Enterprise 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 [Libraries and 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 AMCE 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 "OMCMobileManager.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 "OMCAalytics.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 OAuthView.h, OMCAuthorization.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 AMCe 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 AMCe 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);
    };
}
```

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 Autonomous Mobile Cloud Enterprise 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)

Identity Provider Integration

Here are the steps you need to follow to integrate various third-party identity providers with AMCe.

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 AMCe roles by setting Role Attribute rules in the Keys and Certificates dialog.
 - 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 AMCe, 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 and ensures that you can exchange an externally-issued SAML token that you need to call AMCe APIs. Set this value to the AMCe SSO token exchange 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 AMCe, 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 the token issuer in **Settings > Credentials** in AMCe. See Adding a Token Issuer in *Managing Oracle Autonomous Mobile Cloud Enterprise*.

You'll also want to get token certificate contents from this page and paste them in the Certificate panel for the certificate in **Settings > Credentials**. See Configuring a Web Service or Token Certificate in the same guide.

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 AMCe so that AMCe can validate the token. These are the token certificates that will be associated with the token issuer in AMCe.

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 AMCe, these user groups are mapped to existing AMCe 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 AMCe.

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 AMCe.

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 AMCe. The user assigned to the group would then be assigned to the corresponding AMCe 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 AMCe, 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.