Oracle® Cloud

Using Oracle Messaging Cloud Service

E37257-26
May 2020

Oracle Cloud Using Oracle Messaging Cloud Service,

E37257-26

Primary Author: Nisha Singh

Contributing Authors: Poh Lee Tan, Mark Moussa

Contributors: Ian Sutherland, Derek Dalrymple, Rehan Iftikhar

Contents

3 Accessing Oracle Messaging Cloud Service Using REST API

4 Accessing Oracle Messaging Cloud Service Using Java Library

C Code Samples

Preface

Oracle Messaging Cloud Service provides a platform that enables data communication between applications within Oracle Cloud as well as outside of Oracle Cloud.

Topics:

Audience

Using Oracle Messaging Cloud Service is intended for Oracle Cloud developers who want to facilitate data communication between software components.

For example, a company may have orders submitted on an e-commerce web site that go into a queue for processing. After the orders are processed, they go into another queue for shipping at the warehouse.

Related Resources

For more information, see these Oracle resources:

- Oracle Cloud

  `http://cloud.oracle.com`
- *About Oracle Java Cloud Service - SaaS Extension*

Conventions

The following text conventions are used in this document:

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

1

Getting Started with Oracle Messaging Cloud Service

This section describes how to get started with Oracle Messaging Cloud Service for Oracle Cloud administrators and developers.

Topics:

See Oracle Cloud Terminology in *Getting Started with Oracle Cloud* for definitions of terms found in this and other documents in the Oracle Cloud library.

About Oracle Messaging Cloud Service

Oracle Messaging Cloud Service is one of the Infrastructure as a Service (IaaS) offerings. It provides a messaging system for applications to communicate reliably with each other, enabling application developers to share information across multiple applications.

Topics:

Oracle Messaging Cloud Service gives developers an easier and more reliable method to build complex, distributed systems of heterogeneous applications that may have fundamentally different underlying characteristics such as programming platform, system uptime, and network latency. In addition, businesses using Oracle Messaging Cloud Service do not require any special, dedicated hardware, and the service can be accessed by applications from anywhere over the Internet.

Oracle Messaging Cloud Service is heavily influenced by the Java Message Service (JMS) API specification, which is a standard messaging interface for sending and receiving messages between enterprise Java applications. For Java applications, Oracle Messaging Cloud Service provides a Java library that implements and extends the JMS 1.1 interface. The Java library implements the JMS API by acting as a client of the REST API. Any application platform that understands HTTP can also use Oracle Messaging Cloud Service through the REST interface. This means developers can

use a single communication API to build reliable and robust communication between intra-cloud and extra-cloud applications.

Oracle Messaging Cloud Service uses wildcard certificates for HTTPS access. The use of wildcard certificates may require that client environments be configured to accept wildcard certificates in order to access Messaging Cloud Service. For example, if you are using Oracle WebLogic Server, then you can configure WebLogic Server's SSL hostname verifier to accept wildcard certificates by referring to the Using a Custom Host Name Verifier topic in the *Oracle Fusion Middleware Securing Oracle WebLogic Server* guide.

About Messaging Concepts

In a messaging system, information is transmitted between clients (where a client is defined as a running instance of an application) in the form of **messages**. From the sending client, **producers** send messages to a **destination**. On the receiving client, **consumers** retrieve messages from a destination.

A destination is a type of named resource that resides within an Oracle Messaging Cloud Service instance. It is a repository for messages. **Queues** and **topics** are types of destinations to which messages can be sent.

Messages sent to a queue are received by one and only one consumer. A message sent to a queue is kept on the queue until the message is received by a client or until the message expires. This style of messaging, in which every message sent is successfully processed by at most one consumer, is know as point-to-point.

Messages sent to a topic can be received by multiple consumers or none. This style of messaging, in which each message can be processed by any number of consumers (or none at all), is known as publish/subscribe. To receive a message sent to a topic, a consumer that subscribes to the topic (the **subscriber**) must be connected to the topic when the message is sent by the producer (the **publisher**). That is, only clients that have a consumer connected to a topic will receive messages sent to that topic. If there are no consumers on the topic, messages sent to the topic will not be received by anyone, unless there are some durable subscriptions on the topic.

A **durable subscription**, which stores all messages sent to a topic, can be created to ensure that a publish/subscribe application receives all sent messages, even if there is no client currently connected to the topic. For example, if an application goes offline temporarily and has no consumers on the topic, the client will miss any messages sent to the topic. However, if there is a durable subscription, upon restarting the application, the application will be able to receive any messages sent to the topic during the time the application was not running.

Messages sent to a destination can be pushed to another destination or to a user-defined URL using **message push listeners**.

A **connection** and one or more **sessions** from a client associated with the connection are required to send and receive messages. A session sends messages through one or more producers and receives messages through one or more consumers.

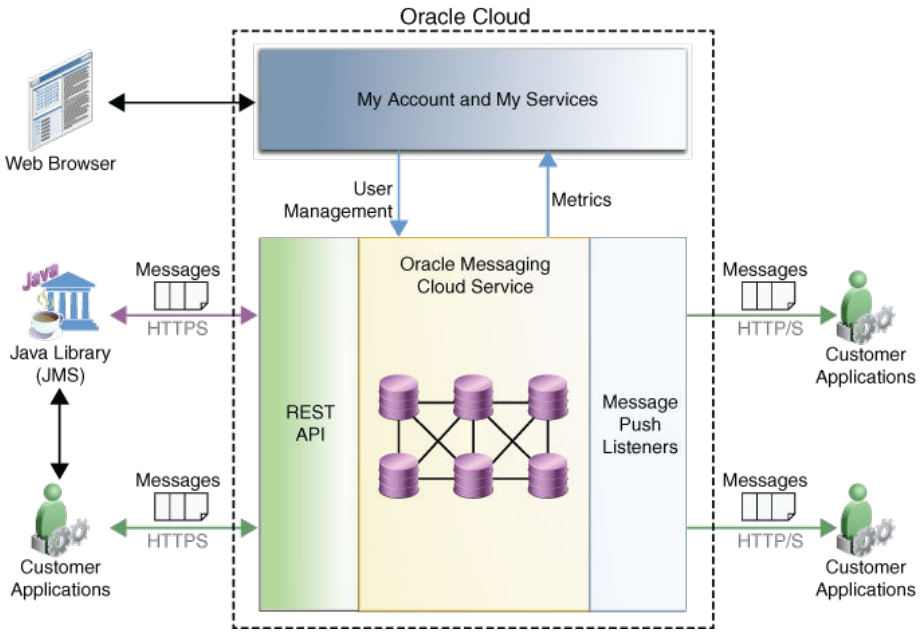
For more information about these and other Oracle Messaging Cloud Service instance resources, see Developing Applications That Use Oracle Messaging Cloud Service.

Architecture Overview

The Oracle Messaging Cloud Service architecture is highly available and fault-tolerant.

Oracle Messaging Cloud Service provides a secure messaging solution for applications that require reliable asynchronous communication. The applications can be within Oracle Cloud as well as outside of Oracle Cloud.

The following diagram presents an architectural overview of Oracle Messaging Cloud Service:



About the Components of Oracle Messaging Cloud Service

Oracle Messaging Cloud Service includes the following components:

- My Account and My Services

  These are applications that allow account administrators and service administrators to manage and monitor their Oracle Cloud service instances, including Oracle Messaging Cloud Service instances. My Account displays information about active, expired, and pending services for an entire account, across multiple data centers and identity domains, and lets administrators monitor the service status. My Services lets administrators monitor and operate all active services within a single identity domain. For more information about the administrative applications, see Overview of Managing Oracle Cloud Accounts and Services in *Getting Started with Oracle Cloud*.

- JMS Broker

  The JMS broker is responsible for handling all administrative and control aspects of the messaging platform, including message persistence, message selection, and session management.

- Message Push Listeners

  These are user-created resources that reside within an Oracle Messaging Cloud Service instance. A message push listener asynchronously receives messages from one destination and either sends the messages to another destination or pushes them to a user-defined URL as an HTTP request.

About the Interfaces to Oracle Messaging Cloud Service

There are two interfaces to Oracle Messaging Cloud Service.

The interfaces to Oracle Messaging Cloud Service are:

- Java library
- REST API

The following table summarizes the interfaces to Oracle Messaging Cloud Service:

Interface	Description	More Information
Java library	The Oracle Messaging Cloud Service Java library provides an implementation of the JMS 1.1 API for sending and receiving messages through the JMS broker. To download the Java library, see Downloading the Oracle Messaging Cloud Service Java SDK. The Java library can be used from any environment connected to the Internet. The Java library also provides APIs for managing instance resources such as queues, topics, durable subscriptions, and message push listeners.	Developing Applications That Use Oracle Messaging Cloud Service Accessing Oracle Messaging Cloud Service Using Java Library Java API Reference for Oracle Messaging Cloud Service
REST API	Oracle Messaging Cloud Service provides a REST API for sending and receiving messages, as well as managing instance resources such as queues, topics, durable subscriptions, and message push listeners. The REST API can be used from any environment connected to the Internet.	Developing Applications That Use Oracle Messaging Cloud Service Accessing Oracle Messaging Cloud Service Using REST API

Note:

Messages sent from one interface can be received through the other interface. Connections, sessions, producers, and consumers cannot be shared between the two interfaces.

About Resource Limits

Paid and trial subscriptions of Oracle Messaging Cloud Service have resource limits.

The following table shows the maximum number of messaging resources per service instance in paid and trial service subscriptions:

Resource	Paid Subscription	Trial Subscription
queues	10,000	5
topics	10,000	5

Resource	Paid Subscription	Trial Subscription
message push listeners	unlimited *	unlimited *
durable subscriptions	10,000	5
persisted messages per destination	Hard Quota: 100,000 Soft Quota: 70,000 For more information, see Hard and Soft Quotas.	Hard Quota: 100 Soft Quota: 70 For more information, see Hard and Soft Quotas.
bytes of persisted messages per destination	Hard Quota: 52,428,800 bytes Soft Quota: 36,700,160 bytes For more information, see Hard and Soft Quotas.	Hard Quota: 52,428,800 bytes Soft Quota: 36,700,160 bytes For more information, see Hard and Soft Quotas.
concurrent connections	10 (additional connections in units of 10 may be purchased)	10 (no upsize)
temporary destinations per connection	50	50

* implicitly capped by the number of concurrent connections

Hard and Soft Quotas

Oracle Messaging Cloud Service restricts both the number of persisted messages that can be sent to a destination but not yet consumed, and the number of bytes of persisted messages that can be sent to a destination but not yet consumed. These restrictions are expressed in terms of a hard quota and a soft quota on both the number and bytes of messages. Clients are allowed to send messages to a destination until the number and bytes of persisted messages on that destination reach the hard quota for either number or bytes of messages. Once the hard quota has been reached for a destination, further attempts to send to that destination will fail with an error until both the number and bytes of message on the destination fall below the soft quota. The hard and soft quotas are the same for all destinations in a given service instance.

For example, suppose the hard quota on number of messages for a queue Q is 100, and the soft quota is 70, there are 100 messages on Q, and a client attempts to send another persistent message, which would put Q over its hard quota on the number of messages. The send will fail, and further sends of persistent messages will continue to fail until at least 31 messages have been consumed from Q, causing it to fall below the soft quota on number of messages. This two-quota algorithm gives consumers on a destination a chance to "catch up" when the destination reaches its hard quota.

See Considerations When Developing Applications That Use Oracle Messaging Cloud Service for additional information about messaging resources.

Before You Begin with Oracle Messaging Cloud Service

Prior to using Oracle Messaging Cloud Service, ensure you are familiar with the following:

- Oracle Cloud

  Create and configure your account on Oracle Cloud. For more information about creating an account on Oracle Cloud, see How to Begin with Oracle Messaging Cloud Service Subscriptions.

- Java library

  The Java library is included in the Oracle Messaging Cloud Service Java SDK that can be downloaded from Oracle Technology Network. To download the Java SDK, see Downloading the Oracle Messaging Cloud Service Java SDK.

  The Java library requires a Java Development Kit (JDK) of version 1.6 or greater for compiling your applications.

  To use the Java library, you must have connectivity to the public Internet.

  The JMS 1.1 API JAR file is also required to compile your web applications. To download the JAR file, accept the Software License Agreement and click on the download link available at following URL:

  `http://download.oracle.com/otndocs/jcp/7542-jms-1.1-fr-doc-oth-JSpec/`

- REST API

  To use the Oracle Messaging Cloud Service REST API, you must have connectivity to the public Internet.

  You should have a strong understanding of the HTTP request/response protocol, specifically:

  – How HTTP cookies are used in response and request headers and when a given cookie is to be included in a request to a particular URL

  – How to read and manipulate HTTP headers and query string parameters

- Message Push Listeners

  If you are using message push listeners to send messages to user-defined URLs, you must have an HTTP server that is reachable from the public Internet and addressable at the provided URL. You must also have the ability to deploy custom applications to the user-defined URL to first verify ownership of the provided URL and then receive messages.

  If you are pushing messages to a user-defined URL over HTTPS, the push target must have a valid Secure Sockets Layer (SSL) certification from Verisign. Communication from Oracle Cloud to external hosts with invalid SSL certificates or self-signed certificates will fail.

Before developing applications that use Oracle Messaging Cloud Service, make sure you understand the following and adhere to the guidelines documented in the relevant sections:

- `JSESSIONID` HTTP cookies

  The Oracle Messaging Cloud Service REST API relies on the use of `JSESSIONID` HTTP cookies to identify and reuse messaging contexts between REST API HTTP requests. Each service instance has a quota of connections that can be created, so it is important to manage messaging contexts, and their associated connections and cookies. For guidelines on using `JSESSIONID` HTTP cookies, see Messaging Context and HTTP Cookies.

- Cross-Site Request Forgery (CSRF) prevention

CSRF is an HTTP client vulnerability in which malicious code attempts to exploit a web server's trust in a user's identity (represented by an HTTP cookie). For information about how Oracle Messaging Cloud Service prevents CSRF attacks and how to manage anti-CSRF tokens generated for connections, see Cross-Site Request Forgery (CSRF) Prevention.

For additional considerations when developing applications with the Oracle Messaging Cloud Service Java library and REST API, see Developing Applications That Use Oracle Messaging Cloud Service.

How to Begin with Oracle Messaging Cloud Service Subscriptions

Here's how to get started with Oracle Messaging Cloud Service trials and paid subscriptions:

To get started with Oracle Messaging Cloud Service, sign up for a free credit promotion, or purchase a subscription. See Requesting and Managing Free Oracle Cloud Promotions or Buying an Oracle Cloud Subscription in *Getting Started with Oracle Cloud*.

About Oracle Messaging Cloud Service Roles and Users

User roles and privileges are described in *Getting Started with Oracle Cloud.*

In addition to the roles and privileges described in Managing User Accounts and Managing User Roles in *Getting Started with Oracle Cloud*, two default account roles are created during provisioning time:

- Messaging Administrator
- Messaging Worker

When the service instance is created during provisioning, the service administrator is given both Messaging Administrator and Messaging Worker roles. The account administrator can create more messaging administrators, messaging workers, or users with both roles, by assigning the appropriate role to users.

The following table summarizes the Oracle Messaging Cloud Service roles used to access, develop, and administer Oracle Messaging Cloud Service and applications.

Role	Description	More Information
Messaging Administrator	Can list and manage all destinations Can list and manage all durable subscriptions Can list and manage all message push listeners Can send and receive messages	Developing Applications That Use Oracle Messaging Cloud Service Accessing Oracle Messaging Cloud Service Using Java Library Accessing Oracle Messaging Cloud Service Using REST API

Role	Description	More Information
Messaging Worker	Can send and receive messages Can list and manage all durable subscriptions Can list and manage all message push listeners Can retrieve properties of individual queues, and topics. Cannot list, create, or delete destinations	Developing Applications That Use Oracle Messaging Cloud Service Accessing Oracle Messaging Cloud Service Using REST API Accessing Oracle Messaging Cloud Service Using Java Library

2

Developing Applications That Use Oracle Messaging Cloud Service

This section provides important information for developers who create applications that use Oracle Messaging Cloud Service through either the Java library or the REST API.

Topics:

Typical Workflow for Using Oracle Messaging Cloud Service

To start developing applications that use Oracle Messaging Cloud Service, refer to the following tasks as a guide:

Task	Description	More Information
Sign up for a free credit promotion, or purchase a subscription.	Provide your information, and sign up for a free credit promotion or purchase a subscription.	Requesting and Managing Free Oracle Cloud Promotions or Buying an Oracle Cloud Subscription in *Getting Started with Oracle Cloud*
Add and manage users and roles	Create accounts for your users and assign them appropriate privileges. Assign the necessary Oracle Messaging Cloud Service roles.	Managing User Accounts and Managing User Roles in *Managing and Monitoring Oracle Cloud*, and About Oracle Messaging Cloud Service Roles and Users
Access the service	Access the service through the REST API and the Java library. To use the Java library, download the Oracle Messaging Cloud Service Java SDK.	Accessing Oracle Messaging Cloud Service
Create and manage destinations	Create, list, and delete destinations. This functionality is only available to users with the Messaging Administrator role.	REST API: Creating and Managing Destinations Java library: Managing Destinations

Task	Description	More Information
Send messages to a destination	Send messages to a destination.	REST API: • Send a Message via a Producer • Sample code: Send a Message to a Topic Java library: • Using JMS to Send and Receive Messages • Sample code: Send a Message to a Topic
Receive messages from a destination	Receive messages from a destination.	REST API: Receiving Messages Java library: Using JMS to Send and Receive Messages
Receive messages using selectors	Use selectors to specify filtered subsets of messages to receive.	REST API: Receive a Message from a Queue with a Selector Java library: Receive a Message from a Queue with an Optional Selector
List durable subscriptions	List information about one or more durable subscriptions.	REST API: List Durable Subscriptions Java library: List Durable Subscriptions
Create and delete durable subscriptions	Create and delete durable subscriptions.	REST API: • Creating and Managing Durable Subscriptions • Sample code: Create a Durable Subscription Java library: Managing Durable Subscriptions
Send and receive messages within transactions	Use transactions to group multiple send and receive operations into atomic operations.	REST API: • Sample code: Process Messages using a Transaction Java library: • Using JMS to Send and Receive Messages • Sample code: Asynchronously Process Messages Within a Transaction
Create and manage message push listeners	Create, list, use, and delete message push listeners.	REST API: • Creating and Managing Message Push Listeners • Sample code: Create a Message Push Listener Java library: • Managing Message Push Listeners • Sample code: Create a Message Push Listener

Accessing Oracle Messaging Cloud Service

You can access Oracle Messaging Cloud Service through the Java library and the REST API.

The Java library is included in the Oracle Messaging Cloud Service Java SDK that can be downloaded from Oracle Technology Network. To download the Java SDK, see Downloading the Oracle Messaging Cloud Service Java SDK.

Before you create applications that make use of Oracle Messaging Cloud Service, be sure to review the guidelines in Considerations When Developing Applications That Use Oracle Messaging Cloud Service.

To learn how to use the Java library and the REST API to access the service for your specific needs, refer to the following documents:

- Using the Java Library and Accessing Oracle Messaging Cloud Service Using Java Library
- Using the REST API and Accessing Oracle Messaging Cloud Service Using REST API

Considerations When Developing Applications That Use Oracle Messaging Cloud Service

Oracle Messaging Cloud Service is largely based on the Java Message Service (JMS) programming model.

Topics:

- About Queues and Topics
- About Message Push and Message Push Listeners
- About Verification of Message Push Listeners
- About Connections
- About Sessions, Acknowledgement Modes, Transactions, and Provisional Messages
- About Producers, Consumers, and Selectors
- About Parts of a Message
- About Persistent and Non-Persistent Messages
- About Authorization
- About Service Termination
- About the Ordering of Message Delivery
- Using Message Groups
- Sending Large Objects as Messages Using Oracle Storage Cloud Service

Oracle Messaging Cloud Service provides the same messaging patterns as JMS and also introduces a new pattern of its own. If you have experience with JMS, the concepts should be familiar. If you are not yet familiar with JMS, see About Messaging Concepts to review basic concepts such as destinations, producers and consumers, and connections and sessions.

The concepts in this section apply to both the REST API and Java library. Regardless of your experience with JMS or REST APIs, and whether you are developing applications with the Java library or REST API, it is important that you review the general guidelines documented in this section first, before referring to the subsequent sections for specific guidelines on Using the Java Library or Using the REST API.

About Queues and Topics

Queues and **topics** are types of **destination** to which **messages** can be sent.

- **Queues**: Messages sent to a queue are received by one and only one consumer.

  When a message is sent to a queue, the message is kept until it is either received or until it expires. This quality of queues removes the timing dependency between

producers and **consumers**, which means producers and consumers don't have to be available and communicating at the same time.

When a message is received, the consumer can either automatically or manually acknowledge message receipt to the messaging platform, indicating whether the message was received or not. See About Sessions, Acknowledgement Modes, Transactions, and Provisional Messages for information about acknowledgement modes.

Each instance of Oracle Messaging Cloud Service has a bound on the number of queues it can have. See About Resource Limits for the maximum number of messaging resources per service instance in paid and trial service subscriptions.

- **Topics**: Messages sent to a topic may be received by multiple consumers or none.

  Consumers must be connected to a topic when a message is sent to the topic in order to receive the message. This quality of topics implies there is a timing dependency between message producers and message consumers. If a client has no consumers on a topic, it will miss any messages sent to the topic until it creates a consumer on the topic.

  If the timing dependency for receiving a message is undesirable, a client can create a **durable subscription** for the topic. A durable subscription stores all messages sent to a topic until each message is received.

  Each instance of Oracle Messaging Cloud Service has a bound on the number of topics and durable subscriptions it can have. See About Resource Limits for the maximum number of messaging resources per service instance in paid and trial service subscriptions.

The time-to-live or maximum time a message can live in Oracle Messaging Cloud Service is 14 days. The time-to-live can be set to a value less than 14 days for any given message. When a message reaches the defined time-to-live value, it is permanently deleted.

About Message Push and Message Push Listeners

Messages sent to a destination can be pushed to either another destination or to a user-defined URL through **message push listeners**.

A user-defined URL's scheme can be either HTTP or HTTPS. For message push to an HTTPS URL to succeed, the server to which messages are pushed must have a valid Secure Sockets Layer (SSL) certification from Verisign. A message push listener asynchronously receives messages from a queue, topic, or durable subscription. When a message is received by a message push listener, the message is pushed to the configured target.

Messages can only be pushed to a user-defined URL via PUT and POST HTTP requests.

Message push listeners can have specific retry and failover policies that define what the message push listener does if delivery of a message to a target fails. Failover policies can push a message to another destination or user-defined URL. If a message push listener cannot deliver a message and no failover policy is specified, then the message push listener discards the message.

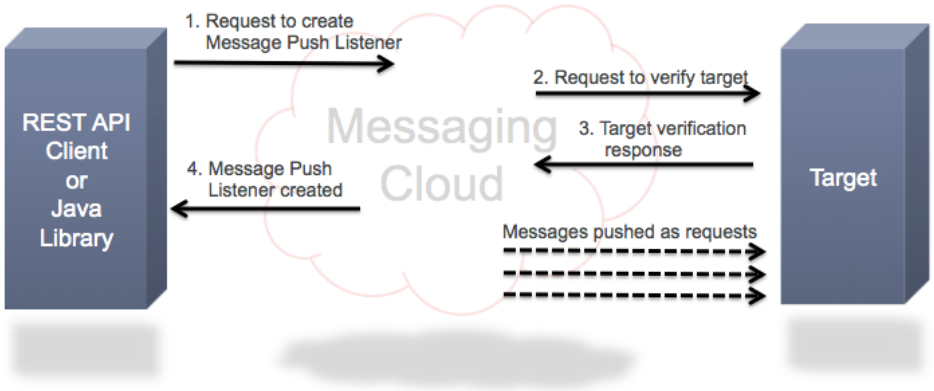
Message push listeners will not follow HTTP redirects. If a message push listener makes an HTTP request to push a message to a user-defined URL, and receives a

redirect response (that is, a response whose status is in the range 300-399), the message push listener will treat it as an error response. Any further handling of the message will be as defined by the message push listener's failover policy. The message push listener will *not* push the message to the location specified in the redirect response.

Each instance of Oracle Messaging Cloud Service cannot be assumed to support more message push listeners than available **connections** (which are required to send and receive messages). See About Connections for information about why it is important to assume that each message push listener will use a dedicated connection.

About Verification of Message Push Listeners

A message can be pushed to a user-defined URL using a message push listener. Before a message push listener can be created, Oracle Messaging Cloud Service must verify all URL targets of the message push listener. In other words, a message push listener is created only after the service has verified that all URLs to which the listener might push a message are willing to receive such pushes, as shown in the following diagram.



The process to verify the URL targets of a message push listener is as follows:

1. A request is made to create a new message push listener, through either the REST API or Java library. Note that the message push listener is not actually created until its URL targets have been verified.
2. Oracle Messaging Cloud Service sends a verification request to every user-defined URL in the listener's definition (primary target and failover targets). Each verification request is an HTTP request that includes the following:
   - An HTTP header with name `X-OC-MPL-CHALLENGE` whose value is a service-generated pseudorandom challenge token
   - An HTTP header with name `X-OC-MPL-VERIFICATION` whose value is a user-provided verification token
3. For a verification request to succeed, the user-defined URL must echo the pseudorandom challenge token as the body of the HTTP response, and the HTTP response must return with a status code of 200. Optionally, the user-defined URL can also validate the value of the user-provided verification token. If the verification request receives a redirect response, the verification request fails.

The verification token is essentially a way for a client creating a message push listener to identify itself to the endpoint as a trusted entity. The verification token could, for example, be a secret string shared between the endpoint and the client, a one-time password, or a cryptographically signed token. The client and endpoint are free to use the value in whatever way they like.

4. If the verification request fails for any user-defined URL, the creation of the message push listener will also fail.

   The purpose of verifying each user-defined URL is to ensure that the owner of the URL is willing to accept pushed messages.

About Destination Deletion

This section provides information about what happens when a client is using, or attempts to use, a non-temporary destination (queue or topic) that is deleted.

Deleting a non-temporary destination is a non-blocking operation. The operation of deleting a destination (either through the REST API or through the Java library) can complete and return control to the client, and the destination can still be in the process of being deleted. Destinations that are in the process of being deleted, but whose deletion is not yet complete, are referred to as being *marked for deletion*. Destinations that are marked for deletion will still be listed when all destinations are listed, and their properties can still be retrieved. They will, however, have status `MARK_FOR_DELETION`. A destination can have status `MARK_FOR_DELETION` for some time.

Any use of a destination that is marked for deletion may fail. This includes sending to it, receiving from it, browsing it (if it is a queue), creating message push listeners on it, having a message push listener push messages from it or to it, etc. This applies both to attempts to use the destination after it is marked for deletion and uses of the destination that began before it was marked for deletion. Sends to a destination that is marked for deletion may succeed or fail. Messages on or sent to a destination that is marked for deletion may or may not be lost. It is recommended that applications be implemented so as to avoid making any use of destinations that are marked for deletion, for example, by using a fixed set of destinations that does not change over time, shutting down all uses of a destination before deleting it, etc.

Message push listeners that listen on a destination that becomes marked for deletion will be deleted automatically. Applications that might be sensitive to the exact time that a message push listener is deleted after its destination is marked for deletion should delete the message push listeners on a destination manually before deleting the destination.

Message push listeners that push to a destination that is marked for deletion or fully deleted will not be deleted automatically when that destination is deleted. Any message push listener that pushes to a destination that might be deleted while the message push listener still exists should be configured with a failure policy that ensures that messages will not be lost if the target destination is deleted.

About Connections

Connections are required to send and receive messages between clients and Oracle Messaging Cloud Service. A connection represents all of the resources needed for communication between clients and the messaging platform.

A client usually uses one connection for all of its sending and receiving operations, though a client can use multiple connections if desired.

Each instance of Oracle Messaging Cloud Service has a quota of concurrent connections. When an instance is using 100% of the connection quota, additional attempts to establish new connections will fail.

A new connection is used when:

- A connection is created through the REST API
- A JMS connection is created through the Java library
- A message push listener is created

Oracle Messaging Cloud Service may allocate message push listeners to connections in different ways in order to optimize performance. The service may have multiple message push listeners use the same connection or have different message push listeners use different connections.

See Messaging Context and HTTP Cookies and Cross-Site Request Forgery (CSRF) Prevention for additional guidelines when using the REST API.

About Sessions, Acknowledgement Modes, Transactions, and Provisional Messages

Once a connection has been created, a **session** must also be created before messages can be sent or received. A session provides a behavioral context that defines what it means for messages to be sent and received between clients and Oracle Messaging Cloud Service.

A single connection can have multiple sessions. Unless explicitly closed, sessions persist until the connection from which they are created is closed.

A message received by a client through a session must be acknowledged before its receipt is treated as final by Oracle Messaging Cloud Service. When a session is created, its acknowledgement mode must be set to one of the following options:

- **Auto-acknowledge**: In **auto-acknowledge** mode, every message received is automatically acknowledged immediately after it is received. Any message received through an auto-acknowledge session is final.
- **Client-acknowledge**: In **client-acknowledge** mode, clients must explicitly acknowledge all message receipts. This allows clients to examine a message to determine if it is prepared to consume the message or not. It is recommended that this mode be used if the session is not transacted and if the Messaging Service is being used for applications in which messages must not be lost.
- **Duplicates-OK**: In **duplicates-OK** mode, message receipts are acknowledged automatically, but lazily, which means messages are not individually acknowledged when they are received but are automatically acknowledged at a later point in time. This mode reduces the communication overhead between clients and the messaging platform, and may increase the rate at which messages are received.

  Using the duplicates-OK mode, however, may result in any given message being delivered multiple times, potentially to more than one client. If a session using this acknowledgement mode unexpectedly closes, the messages delivered to the client since the last acknowledgement may be made available for delivery to other clients.

Sessions created through the REST API are auto-acknowledged by default. Sessions created through the Java library must have their acknowledgement mode specified explicitly.

Instead of specifying an acknowledgement mode, a session can be configured such that sequences of send and receive operations are grouped into atomic operations known as **transactions**. Like acknowledgement modes, sessions are configured to be transacted or not transacted when the session is created. In a transaction, all grouped send and receive operations either complete or do not complete collectively.

At any point in a transaction, the client can call **rollback**, and all previous send and receive operations within that transaction will be cancelled. Transacted sessions must be explicitly committed. When the client calls **commit** on a transaction, all send and receive operations are made permanent and a new transaction is started.

In sessions whose mode is client-acknowledge, the client receiving messages is responsible for explicitly acknowledging that messages have been received. Until the client explicitly acknowledges a message, received messages are considered to be provisional. If the client fails to perform acknowledgement, provisionally received messages are returned to the destination and made available to be received by another client when the session or its connection is closed.

Similarly, in transacted sessions the client sending and receiving messages is responsible for explicitly committing the session. Until the session is committed, all sent and received messages within the transaction are considered to be provisional. Failure to commit transacted sessions will cause provisionally sent messages to be discarded and provisionally received messages to be made available for delivery to other clients. In both client-acknowledge mode and transacted sessions, when a provisionally received message becomes available to be delivered to another client, and when the message is so delivered, it is marked as **redelivered**.

About Producers, Consumers, and Selectors

A session sends messages through a producer and receives messages through a consumer. A session can have multiple producers and multiple consumers. Unless explicitly closed, producers and consumers persist until the session in which they are created is closed.

A producer defines the default characteristics of how messages are sent to and stored within the messaging platform. A producer can specify the destination to which all messages are sent, how sent messages should be stored on the target destination, and how long sent messages can live in the service before they expire.

A consumer defines how messages are received from the messaging platform. A consumer must specify the destination from which messages are received.

Optionally, consumers can select a subset of all available messages to be received from a destination by specifying a **selector**. A selector is an SQL-like expression that specifies a condition that a message must satisfy to be eligible for the consumer to receive the message. Selectors can only select messages based on criteria in the message headers and properties. Selectors cannot select messages based on the contents or type of a message body (for example, Text or Object type). For the syntax of selectors, see the **Message Selectors** section of the Java API reference for the `javax.jms.Message` class. For the syntax of selectors, see the **Message Selectors** section of the Java API reference for the `javax.jms.Message` class at the URL:

`http://docs.oracle.com/javaee/6/api/javax/jms/Message.html`

About Parts of a Message

Messages are unique, discrete units of information that pass between two or more clients through Oracle Messaging Cloud Service. Each message has three parts: headers, properties, and a body.

Topics:

Message Headers

Message headers are predefined key/value pairs associated with a message.

Message headers are used by the messaging platform for message identification and routing purposes. Some headers are client-set and some are broker-set. For client-set headers, ‘Required’ means that the client must supply the header and ‘Optional’ means that client need not supply it. For broker-set headers, ‘Required’ means the broker will always set it, whereas ‘Optional’ means the broker may or may not set it.

The headers, some of which may or may not be present in a message, are:

- Correlation ID: An identifier that can be set by the sending client to correlate multiple messages. The value of this header is set by the sending client.
- Delivery Mode: Required. The persistence type of the message. The value of this header is set by the sending client.
- Destination: Required. The destination to which the message was sent. The value of this header is set by the sending client. For more information, see About Persistent and Non-Persistent Messages.
- Expiration: Required. The time when the message will expire. The value of this header is a long integer, and is interpreted as Unix time. This value is set by the JMS broker, but is partially a function of the message's time-to-live, which is set by the sending client.
- Message ID: Required. The globally unique ID of the message. This value is set by the JMS broker.
- Redelivered: Required. Indicates whether the message has been delivered at least once before. Value is `true` or `false`. This value is set by the JMS broker.
- ReplyTo: A destination to which replies to this message should be sent. Controlled by the sending client.
- Time: The time when the message was sent to the destination. Set by the JMS broker.

Message Properties

Message properties are optional key/value pairs associated with a message. Some message properties are user-defined, and some are set by the system.

Message property values can have the following classes:

- Boolean
- Byte
- Short
- Integer
- Long
- Float
- Double
- String

Selectors can use message properties to restrict the messages received by a consumer. See About Producers, Consumers, and Selectors for information about selectors.

Message Body and Message Size

A message body must have a type, which defines the format and structure of the body.

The message type can be one of the following:

- `PLAIN`: The message has no body. It only has headers and properties.
- `TEXT`: The message body is a String.
- `BYTES`: The message body is an array of bytes.
- `OBJECT`: The message body is a serialized Java object.
- `MAP`: The message body is a set of key/value pairs. Keys are Strings and values are Java objects. Each value can be either a Boolean, Byte, Character, Short, Integer, Long, Float, Double, String, or an array of bytes.
- `STREAM`: The message body is a stream of Java objects. Each object in a stream is either a Boolean, Byte, Character, Short, Integer, Long, Float, Double, String, or an array of bytes.
- `HTTP`: The message body is a serialized Java object that contains a byte array representing the body of an HTTP request or response, and Strings representing the HTTP headers that specify the language and media type of that body.

Messages have a maximum size of 512KB. Send operations with messages larger than 512KB will fail. See Accessing Oracle Messaging Cloud Service Using REST API and Accessing Oracle Messaging Cloud Service Using Java Library for information about how message size is calculated.

About Persistent and Non-Persistent Messages

When a message is sent to a destination, the message delivery mode is marked as **persistent** by default.

Persistent messages are guaranteed to be stored in a durable medium while being processed by Oracle Messaging Cloud Service. This means persistent messages are not lost if Oracle Messaging Cloud Service temporarily goes down.

Optionally, messages can be marked as **non-persistent**. Non-persistent messages may or may not be stored in a durable medium. Since non-persistent messages do not require as much I/O as persistent messages, higher throughput rates may be achieved

by using non-persistent messages. If Oracle Messaging Cloud Service temporarily goes down, however, non-persistent messages may be lost.

A queue or topic cannot have more than 100,000 messages at any given time. Send operations to a destination with 100,000 messages will fail. The 100,000 limit applies to either persistent or non-persistent messages, or a combination of both.

> **Note:**
>
> While persistent messages are always stored in a durable medium, there are rare instances when the durable medium and the messages stored may be lost. Additional copies of mission-critical data should therefore always be stored in secure and reliable locations.

About Authorization

User roles and privileges are described in *Getting Started with Oracle Cloud.*

In addition to the roles and privileges described in Managing User Accounts and Managing User Roles in *Getting Started with Oracle Cloud*, two default account roles, Messaging Administrator and Messaging Worker, are created for Oracle Messaging Cloud Service when the service instance is provisioned. Each role has privileges that define what operations users are authorized to perform in the service instance. Any user with the Messaging Administrator role can potentially delete any destination within the instance. Any user with the Messaging Administrator role or the Messaging Worker role can potentially send or receive messages to any destination within the instance. See About Oracle Messaging Cloud Service Roles and Users for additional privileges associated with each role.

About Service Termination

When an instance of Oracle Messaging Cloud Service is terminated, no customer data is archived. Messages residing on destinations are deleted immediately upon service termination.

Before terminating an instance, be sure to drain and store messages from queues and durable subscriptions if the contents of stored messages are important.

About the Ordering of Message Delivery

Oracle Messaging Cloud Service does not provide any strict guarantees about the order in which messages are delivered.

Applications for which message ordering is critical should use the "redelivered" message header to detect redeliveries, and should consider the use of timestamps or sequence numbers, possibly attached to messages as message properties or in the message bodies, to ensure that messages are processed in the proper order.

Using Message Groups

Message groups can be used to send a message that is larger than 512KB in a set of multiple smaller messages using a queue.

Message groups are used to group different messages that should all be processed by the same consumer. Message groups are created implicitly by sending messages that have a message ID and sequence number set on them to a queue.

A message group is defined by a group ID and a group sequence number. If a message belongs to a group, the group it belongs to is defined by the value of its `JMSXGroupID` property.

Note:

- If you send messages in a specific group to a queue, and the sequence numbers are out of order, the consumer will receive them in the order in which they were sent, not in order of message sequence.
- The first message sent in a group must have sequence number 1.
- Sequence numbers have no effect on message receipt order from queues.
- Sequence numbers are not used to eliminate duplicate messages.
- The use of message groups has no effect when used with topics: all consumers on the topic will get all messages, and receipt order will *not* be affected by sequence numbers.
- Sequence numbers play an important role in the delivery of messages in a message group. For example, consider a scenario such as the following:
  - Messages are sent in group G. Consumer C is chosen to receive group G, and gets some of the messages.
  - After consumer C receives few messages from group G, consumer C is closed. However, more messages are sent in group G.
  - A new consumer, D, is chosen to receive messages in G.
  - When consumer D receives messages that don't start with sequence number 1, D can conclude that some other messages in the group were sent to some other consumer.

Method: `POST`

Path: `/producers/producerName/messages`

Scope: Messaging Context

Authorization: Messaging Administrator or Messaging Worker

Request Parameters:

Parameter	Description
groupId	This parameter is used to set the JMSXGroupID property on the message being sent. This is the name of the message group of which this message is a part, if any. **Note:** If the JMSXGroupID property is set as an HTTP request header, it must be set to an escaped value String or a badParameter error response will be generated. For more information on escaped value Strings, see About Escaped Value Strings. If the JMSXGroupID property is set as a query string parameter, the usual conventions for escaping query string parameters hold.
groupSeq	This parameter is used to set the JMSXGroupSeq property on the message being sent. This is the sequence number of the message within the message group specified by the groupId parameter. The groupSeq parameter must be set to an integer or a badParameter error response will be generated.

Result: Sends multiple messages as a message group using a queue.

Error Responses:

Error Message	Description
badParameter	The groupSeq parameter was not set to an integer.
incompleteGroupProperties	Exactly one of the JMSXGroupID and JMSXGroupSeq properties was set on the message. Either both properties must be set, or neither must be set.

See Use Message Groups for a code sample on sending messages using message groups.

Sending Large Objects as Messages Using Oracle Storage Cloud Service

You can use Oracle Storage Cloud Service to send large message payloads using a message that contains a reference to the payload. This is especially useful for storing and consuming messages with a message size of up to 5 GB, the maximum size of a single object stored in a Storage container.

You can send large payloads with a combination of Oracle Messaging Cloud Service and Oracle Storage Cloud Service and use the following features that the services allow:

- Store the message payload as a storage object with an automatic deletion time.
- Create a temporary URL to access the object containing the payload.
- Send a message containing the temporary URL for the object.
- Fetch the payload from the temporary URL.
- Delete the corresponding message object from an Oracle Storage Cloud container.

About Oracle Storage Cloud Service

Oracle Storage Cloud Service is an Infrastructure as a Service (IaaS) product, which provides an enterprise-grade, large-scale object storage solution for data of any type. Oracle Storage Cloud Service stores data as objects within a flat hierarchy of containers.

Oracle Storage Cloud Service allows you to create temporary URLs for your objects, authenticate storage requests, and auto-delete the objects after a certain period.

To learn more about Oracle Storage Cloud Service, see Features of Oracle Cloud Infrastructure Object Storage Classic in *Using Oracle Cloud Infrastructure Object Storage Classic.*

About Temporary URLs

Temporary URLs are time-limited URLs that expire after a configured time period. You can create temporary URLs to provide a secure, temporary access to protected resources like objects in your Oracle Storage Cloud Service account. A temporary URL specifies both the object and the HTTP method with which the object can be accessed. If you do not have access to Oracle Storage Cloud Service, you can download an object from the service using a temporary URL.

> **Note:**
>
> The auto-delete time set on the object and the time at which the temporary URL expires are not necessarily the same time; they can be set independently.

See Downloading an Object Using a Temporary URL in *Using Oracle Cloud Infrastructure Object Storage Classic.*.

Step-by-Step Procedure to Send a Large Object Stored in a Storage Container as a Message

The example shows a step-by-step procedure for sending a large object stored in a storage container as a message:

> **Note:**
>
> - Only relevant HTTP headers are shown in the example.
> - The values of the secret key, password, etc. mentioned as <***> should be replaced by the appropriate values for the service or account, or by user-chosen values.

1. **Get an authentication token**

   HTTP Request

   ```
   GET /auth/v1.0 HTTP/1.1
   Host: storage.oraclecorp.com
   ```

```
Accept: */*
X-Storage-User: Storage-msnerd:msnerd.Storageadmin
X-Storage-Pass: <account password>
```

HTTP Response

```
HTTP/1.1 200 OK
Server: nginx/1.10.2
Content-Length: 0
X-Auth-Token: AUTH_tk7e51d668b970abcc71f91c64e0fa5e38
X-Storage-Token: AUTH_tk7e51d668b970abcc71f91c64e0fa5e38
X-Storage-Url: https://storage.oraclecorp.com/v1/Storage-msnerd
```

The authentication token and storage-URL generated in this section will be used in the requests for accessing the message payload.

2. **Create a container for your object**

HTTP Request

```
PUT /v1/Storage-msnerd/storage-payload HTTP/1.1
Host: storage.oraclecorp.com
Accept: */*
X-Auth-Token: AUTH_tk7e51d668b970abcc71f91c64e0fa5e38
```

HTTP Response

```
HTTP/1.1 202 Accepted
Server: nginx/1.10.2
```

3. **Set the container key using the information retrieved from steps 1 and 2**

HTTP Request

```
POST /v1/Storage-msnerd/storage-payload HTTP/1.1
Host: storage.oraclecorp.com
Accept: */*
X-Auth-Token: AUTH_tk7e51d668b970abcc71f91c64e0fa5e38
X-Container-Meta-Temp-URL-Key: <container_secret_key>
```

<container_secret_key> is a password key value specific to the account owner.

HTTP Response

```
HTTP/1.1 204 No Content
```

4. **Create the payload, upload the object to the storage container, and set an auto deletion time on the object**

HTTP Request

```
PUT /v1/Storage-msnerd/storage-payload/<UUID> HTTP/1.1
Host: storage.oraclecorp.com
Accept: */*
X-Auth-Token: AUTH_tk7e51d668b970abcc71f91c64e0fa5e38
Content-Type: <object_content_type>
X-Delete-At:  <Time-stamp _for_auto-delete>
Content-Length: <Size_of_the_payload>
[Message payload]
```

Here are some examples of the variables used in the above request:

- *UUID* - To create a payload, generate or use a unique ID that can be used to refer to the payload. For example, `1faf75ea-5619-4a0e-a8a3-764f360ee0eb`.

- *object_content_type* - This is the media type of the object. For example, `application/octet-stream`.
- *Time-stamp _for_auto-delete* - This is the UNIX Epoch timestamp representing the date and time at which the object should be deleted. For example, `1416218400` represents November 17, 2014 10:00:00 GMT. See http://www.epochconverter.com/.
- *Size_of_the_payload* - `3523574` (In Bytes)

HTTP Response

```
HTTP/1.1 201 Created
Server: nginx/1.10.2
Content-Type: <object_content_type>
Content-Length: 0
```

object_content_type is the media type of the object. For example, `application/octet-stream`.

Sample Code to compute a temporary URL

You can refer to the following sample codes to compute a temporary URL:

- Java Code
- Python Code

Here's a sample Java code to compute a temporary URL:

```
import java.io.UnsupportedEncodingException;
import java.security.InvalidKeyException;
import java.security.NoSuchAlgorithmException;
import java.security.SignatureException;

import javax.crypto.Mac;
import javax.crypto.spec.SecretKeySpec;

/**
 *  Utility class for creating HMAC-SHA1 signatures and
 *  OSS temporary URLs.
 */
public class HMACUtils
{
    private static final String algorithm = "HmacSHA1";

    // Lower-case hex digits in order
    private static final char[] hexDigit = "0123456789abcdef".toCharArray();

    /**
     *  Method to convert a byte array into a string of lower-case hex digit
pairs.
     *
     *  @param bytes
     *      Bytes to convert.
     *      May not be <code>null</code>.
     *
     *  @return
     *      String of hex digits corresponding to the input array.
     */
    public static String bytesToHexPairs(byte[] bytes)
    {
        StringBuilder sb = new StringBuilder();
```

```
            for(byte b : bytes)
            {
                sb.append(hexDigit[(((int)b) & 0b11110000) >>> 4]);
                sb.append(hexDigit[((int)b) & 0b1111]);
            }

            return sb.toString();
        }

        /**
         *  Method to compute HMAC-SHA1 signature from arbitrary data.
         *
         *  @param key
         *      Secret key for the signature, set on the OSS account or container.
         *      May not be <code>null</code>.
         *
         *  @param data
         *      Data to be signed.
         *      May not be <code>null</code>.
         *
         *  @return
         *      Bytes for the signature.  Must be rendered as lower-case hex-digit
    pairs in an OSS temporary URL.
         *
         *  @throws SignatureException
         *      An error occurred in generating the signature.
         *
         *  @throws NoSuchAlgorithmException
         *      Should not occur; the algorithm used should
         *      be valid HMAC-SHA1.
         *
         *  @throws InvalidKeyException
         *      An error occurred in processing the key.
         */
        public static byte[] signature(byte[] key, byte[] data)
            throws SignatureException, NoSuchAlgorithmException, InvalidKeyException
        {
            SecretKeySpec signingKey = new SecretKeySpec(key,algorithm);
            Mac mac = Mac.getInstance(algorithm);
            mac.init(signingKey);
            return mac.doFinal(data);
        }

        /**
         *  Method to compute HMAC-SHA1 signature for a method, expiration time, and
    path.
         *
         *  @param key
         *      Secret key for the signature, set on the OSS account or container.
         *      May not be <code>null</code>.
         *
         *  @param method
         *      HTTP method (e.g. GET, POST, DELETE, PUT).
         *      May not be <code>null</code>.
         *
         *  @param expiration
         *      Unix epoch time in seconds at which the signature will expire.
         *      Must be non-negative.
         *
         *  @param path
         *      Path (the part of the URL starting with '/' and coming after the
```

```
host and port) to
     *      which the signature will give temporary access.
     *      May not be <code>null</code>.
     *
     *  @return
     *      Bytes for the signature.  Must be rendered as lower-case hex-digit
pairs in an OSS temporary URL.
     *
     *  @throws SignatureException
     *      An error occurred in generating the signature.
     *
     *  @throws NoSuchAlgorithmException
     *      Should not occur; the algorithm used should
     *      be valid HMAC-SHA1.
     *
     *  @throws InvalidKeyException
     *      An error occurred in processing the key.
     *
     *  @throws UnsupportedEncodingException
     *      Should not occur; only UTF-8 encoding is used to convert characters
to bytes.
     */
    public static byte[] signature(byte[] key, String method, long expiration,
String path)
        throws SignatureException, NoSuchAlgorithmException,
InvalidKeyException, UnsupportedEncodingException
    {
        return HMACUtils.signature(key, (method+'\n'+expiration
+'\n'+path).getBytes("UTF-8"));
    }

    /**
     *  Method to compute a temporary OSS URL.
     *
     *  @param key
     *      Secret key for the signature, set on the OSS account or container.
     *      May not be <code>null</code>.
     *
     *  @param method
     *      HTTP method (e.g. GET, POST, DELETE, PUT).
     *      May not be <code>null</code>.
     *
     *  @param timeToExpiration
     *      Amount of time in seconds after which the
     *      temporary URL will expire.
     *
     *  @param path
     *      Path (the part of the URL starting with '/' and coming after the
host and port) to
     *      which the signature will give temporary access.
     *      May not be <code>null</code>.
     *
     *  @return
     *      Path portion of the temporary URL.
     *
     *  @throws SignatureException
     *      An error occurred in generating the signature.
     *
     *  @throws NoSuchAlgorithmException
     *      Should not occur; the algorithm used should
     *      be valid HMAC-SHA1.
```

```
     *
     *  @throws InvalidKeyException
     *      An error occurred in processing the key.
     *
     *  @throws UnsupportedEncodingException
     *      Should not occur; only UTF-8 encoding is used to convert characters
to to bytes.
     */
    public static String temporaryURLPath(byte[] key, String method, long
timeToExpiration, String path)
        throws SignatureException, NoSuchAlgorithmException,
InvalidKeyException, UnsupportedEncodingException
    {
        long expiration = System.currentTimeMillis()/1000 + timeToExpiration;
        byte[] sig = HMACUtils.signature(key,method,expiration,path);

        return  path +
                ((path.indexOf('?') < 0) ? '?' : '&') +
                "temp_url_sig=" +
                bytesToHexPairs(sig) +
                "&temp_url_expires=" +
                expiration;
    }

    /**
     *  <p>
     *      Command-line interface.  This interface takes
     *      arguments specifying a key and a temporary
     *      access to an object in OSS (such as a large
     *      message payload) and outputs the path portion of
     *      the temporary URL with which to access the object.
     *  </p>
     *  <p>
     *      Specifically, the arguments are as follows:
     *  </p>
     *  <ul>
     *      <li>
     *          the secret key with which to generate
     *          the signature; this must must have been
     *          set previously on the OSS container or
     *          account that will contain the object to
     *          be accessed
     *      </li>
     *      <li>
     *          the HTTP method to put into the signature
     *          (e.g. GET, POST, PUT, DELETE); this will
     *          determine what methods can be used with
     *          the temporary URL
     *      </li>
     *      <li>
     *          the amount of time, in seconds, after
     *          which the temporary URL will no longer
     *          function; the signature generated will
     *          work for that many seconds from the time
     *          at which HMACUtils is run.
     *      </li>
     *      <li>
     *          the path to the OSS object to be accessed
     *          via the temporary URL; this is the part
     *          of the full URL that comes <strong>after</strong>
     *          <nobr><code>https://</code><i>&lt;host&gt;</i><code>:</
```

```
code><i>&lt;port&gt;</i></nobr>
     *          (including any leading '/').
     *      </li>
     *  </ul>
     *  <p>
     *      Usage:
     *      <nobr>
     *          <code>java HMACUtils</code>
     *          <i>&lt;key&gt;</i>
     *          <i>&lt;HTTP method&gt;</i>
     *          <i>&lt;time to expiration&gt;</i>
     *          <i>&lt;OSS object path&gt;</i>
     *      </nobr>
     *  </p>
     *  <p>
     *      The output on standard out will be the
     *      path portion of the temporary URL by which to
     *      access the object, that is, the portion that
     *      comes <strong>after</strong>
     *      <nobr><code>https://</code><i>&lt;host&gt;</i><code>:</
code><i>&lt;port&gt;</i></nobr>.
     *  </p>
     */
    public static void main(String[] argv) throws Exception
    {
        byte[] key = argv[0].getBytes("UTF-8");

        String method = argv[1];
        long timeToExpiration = Long.parseLong(argv[2]);
        String path = argv[3];

        System.out.printf("%s
\n",HMACUtils.temporaryURLPath(key,method,timeToExpiration,path));
    }
}
```

Here's a sample Python code to compute a temporary URL:

User Input: Specify the following parameters:

```
serviceInstanceName = 'Storage' # Leave as is unless your service instance has a
different name
identityDomainName = 'acme' # Name of your identity domain
container = 'myContainer' # Container that has the objects you need the tempURL
for
key = 'mykey' # X-Container-Meta-Temp-Url-Key or X-Account-Meta-Temp-Url-Key
value
object = 'myObject' # Object name that you need the tempURL for. This is
optional if a container-level key is used.
urlDuration = 300 # Seconds for which the temp URL should work
serviceRestEndpoint = 'https://acme.storage.oraclecloud.com/v1/Storage-acme' #
REST endpoint URL of your service instance, as shown in the Service Details page
in My Services
```

Code to generate the temporary URL:

> **Note:**
>
> Do not modify any values in this section.

```
import hmac
from hashlib import sha1
from time import time
path = '/v1/' + serviceInstanceName + '-' + identityDomainName + '/' + container
+ '/' + object
expires = int(time() + urlDuration)
hmac_body = '%s\n%s\n%s' % ('GET', expires, path)
sig = hmac.new(key, hmac_body, sha1).hexdigest()
url = serviceRestEndpoint + '/' + container + '/' + object +'?temp_url_sig=' +
sig + '&temp_url_expires=' + str(expires)
print(url)
```

5. **Use Oracle Messaging Cloud service to send a message with the temporary URL as the payload.**

   See Sending Messages.

6. **Retrieve the payload using the temporary URL.**

   HTTP Request

   ```
   GET /v1/Storage-msnerd/storage-payload/<UUID>?
   temp_url_sig=<temp_url_sig>&temp_url_expires=<temp_url_expire> HTTP/1.1
   Host: storage.oraclecorp.com
   Accept: */*
   ```

   temp_url_sig The sample Java/Python code above generates the full temporary URL. Part of the URL it creates is the value of *temp_url_sig.* This is the HMAC-SHA1 signature of the method, expiration time, and path to the object, signed using the container key set in step 3.

   temp_url_expires is the UNIX Epoch timestamp representing the date and time at which the temporary URL expires. Note that this is not when the object is auto-deleted, but the time when the temporary URL will stop working.

   HTTP Response

   ```
   HTTP/1.1 200 OK
   Content-Type: application/octet-stream
   Content-Length: <Size of Payload>
   X-Delete-At: <Time-stamp for auto-delete>
   [Message payload]
   ```

Using the Java Library

The Oracle Messaging Cloud Service Java SDK provides a Java library for managing service instance resources in addition to sending and receiving messages through the JMS API. Included in the Java SDK is a copy of the Javadocs for the Java library.

Topics:

Before you begin using the Java library, be sure to review the guidelines in this section as well as the general guidelines in Considerations When Developing Applications That Use Oracle Messaging Cloud Service.

Typical Workflow for Using the Java Library

To start using the Java library, refer to the following tasks as a guide:

Task	Description	More Information
Download the Oracle Messaging Cloud Service Java SDK	The Oracle Messaging Cloud Service Java SDK provides a Java library for managing service instance resources in addition to sending and receiving messages through the JMS API.	Downloading the Oracle Messaging Cloud Service Java SDK
Extract the Oracle Messaging Cloud Service Java library from the downloaded Java SDK	Extract the Java library JAR file somewhere onto your Java application's class path.	
Package the Java library into an enterprise Java web application	Import the Java library's classes and interfaces into your Java application.	
Create a Servlet or Web service	Create a Servlet or Web service to send and/or receive messages.	Accessing Oracle Messaging Cloud Service Using Java Library Send a Message to a Topic
Create a standalone application	Create a Java standalone application to send and/or receive messages without an HTTP interface.	Accessing Oracle Messaging Cloud Service Using Java Library Asynchronously Receive Messages with a Durable Subscription

Downloading the Oracle Messaging Cloud Service Java SDK

The Oracle Messaging Cloud Service Java SDK is a downloadable package that contains the following components:

Download the Oracle Messaging Cloud Service Java SDK from Oracle Technology Network at the following URL:

`http://www.oracle.com/technetwork/topics/cloud/downloads/messaging-cloud-service-sdk-2279257.html`

- Java library (`oracle.messaging.cloud.api-14.0.X.jar` where `X` is the latest version number of the Java library)
- Java API Reference documentation for Oracle Messaging Cloud Service

Authentication and Authorization

When using the Java library, a client is authenticated with Oracle Messaging Cloud Service when any of the operations listed below is initiated for the first time:

- Using any destination management function (listing, creating, or deleting destinations).

- Using a message push listener management function (listing, creating, or deleting message push listeners).
- Listing durable subscriptions.
- Creating a connection.

The `MessagingService` object is the entry point for all operations against an instance of Oracle Messaging Cloud Service, including message transmission and resource management. If only one `MessagingService` object is created for all operations, then all operations share the same user authentication and authorization level.

Differences from JMS

The Java library provided in Oracle Messaging Cloud Service implements the JMS 1.1 API. While you may be familiar with JMS in on-premises environments, note the following differences from JMS when using the Java library:

- The Java Naming and Directory Interface (JNDI) is currently not supported for referencing JMS objects such as `ConnectionFactory`, `Queue`, and `Topic`. JMS objects must be instantiated directly using the Java library provided in Oracle Messaging Cloud Service. Since JNDI is not supported, Oracle Messaging Cloud Service cannot be used to implement message-driven beans.
- Oracle Messaging Cloud Service does not enforce message priority on message delivery. Although message priority is a standard JMS message header, the implementation of priority headers is not mandated by the JMS specification. Messages can still be sent with a priority header but this does not influence how a message is delivered. The value of the priority header is set to the default value of "4".

Using the REST API

The REST API implements the same messaging model as the Java library, with the exception that connections, sessions, producers, consumers, queue browsers, and temporary destinations are created as resources on the service instance through the REST API.

Topics:

Before you begin using the REST API, review the guidelines in this section as well as the general guidelines in Considerations When Developing Applications That Use Oracle Messaging Cloud Service.

Typical Workflow for Using the REST API

To start using the REST API, refer to the following tasks as a guide:

Task	Description	More Information
Create a messaging context	Create a messaging context, and manage the lifecycle of messaging contexts through the maximum inactive interval (MII) feature.	Creating and Managing Messaging Contexts
Create a connection	Create connections before sending and receiving messages.	Creating and Managing Connections
Create a message push listener	Create listeners to push messages to a destination or user-defined URL.	Creating and Managing Message Push Listeners
Create a session	Create sessions before sending or receiving messages.	Creating and Managing Sessions
Create a producer	Create producers to send messages.	Sending Messages
Create a consumer	Create consumers to receive messages.	Receiving Messages
Send a message through a producer	Send messages through producers.	Sending Messages
Receive a message through a consumer	Receive message through consumers.	Receiving Messages

Messaging Context and HTTP Cookies

A messaging context is a container of ephemeral objects like connections, sessions, producers, consumers, temporary destinations, and queue browsers.

A messaging context is identified by the `JSESSIONID` cookie. The only API specific to a messaging context is the API for getting and setting the **maximum inactive interval** (MII), which controls the expiration time of the messaging context. When the messaging context expires, all ephemeral objects contained in it are closed and deleted, except for temporary destinations. A temporary destination is closed only if the connection by which the temporary destination was created is closed.

The Oracle Messaging Cloud Service REST API relies critically on the use of `JSESSIONID` HTTP cookies to identify and reuse messaging contexts between REST API HTTP requests.

At least one messaging context must be created by a client in order to access Oracle Messaging Cloud Service. If an HTTP request does not include a `JSESSIONID` cookie for an unexpired messaging context, then a new messaging context is created. The HTTP response includes the header `X-OC-NEW-MESSAGING-CONTEXT: true` if a new messaging context is created. Note that connections and derived objects created in a messaging context cannot be used in other messaging contexts, except for temporary destinations. A temporary destination created in a messaging context can be used in other messaging contexts.

When a messaging context is created through the REST API, the messaging context is assigned an MII. A messaging context expires if it is not accessed for a period of

time longer than the associated MII. By default, all messaging contexts have an MII of 5 minutes. The MII for a given messaging context can be increased to a maximum of 15 minutes by the client. The minimum non-zero value of an MII is 1 second. If the MII is set to 0, the messaging context expires immediately.

Each Oracle Messaging Cloud Service instance has a quota of connections that can be created. When using the REST API, it is important to handle `JSESSIONID` HTTP cookies diligently to manage messaging contexts and their associated connections. If the cookies are not handled, then a new messaging context is created on every HTTP request to the REST API. This can quickly lead to the exhaustion of the instance's connection quota if a connection is created in each messaging context.

In special situations, clients may wish to durably store `JSESSIONID` HTTP cookies associated with active messaging contexts. For example, if a client machine fails, cookies for active messaging contexts can be lost from non-durable storage, and if the application cannot wait for the messaging context to expire, then it is important to durably store cookies. If cookies are durably stored, the active messaging contexts and their connections can be reached after the client comes back up, as long as those messaging contexts have not expired during the client down time.

> **Note:**
>
> If Cross-Site Request Forgery (CSRF) prevention is enabled for a given messaging context, clients may also wish to durably store the anti-CSRF token associated with the messaging context for the messaging context and associated connections to be reachable when the client is back up. See Cross-Site Request Forgery (CSRF) Prevention for details about how anti-CSRF tokens are used by Oracle Messaging Cloud Service.

Authentication

All HTTP requests to the Oracle Messaging Cloud Service REST API require authentication.

Every HTTP request to Oracle Messaging Cloud Service should supply HTTP Basic Authentication credentials through the `Authorization` header.

About HTTP Headers

The Oracle Messaging Cloud Service REST API uses various HTTP headers to send and receive information. The names of these headers are written in capital letters throughout, but HTTP headers are required to be treated as case-insensitive.

Any client of the REST API should be implemented so as to treat HTTP header names in Oracle Messaging Cloud Service responses as case-insensitive. HTTP header names may occur multiple times within a single HTTP response, so any client of the REST API should be implemented so as to handle multiple occurrences of a header with the same name (or with the same name except for differences in case) properly.

Cross-Site Request Forgery (CSRF) Prevention

Cross-Site Request Forgery (CSRF) is a malicious attack on HTTP clients whereby destructive operations may be unknowingly made against a web server.

To prevent CSRF attacks, Oracle Messaging Cloud Service generates pseudorandom anti-CSRF tokens for each messaging context.

When a new messaging context is created, a new anti-CSRF token is generated and returned in the messaging context's first HTTP response as the value of the `X-OC-ID-TOKEN` HTTP header. The token is not returned in subsequent HTTP responses.

Once an anti-CSRF token has been generated, every subsequent HTTP request must include its messaging context's associated token in the `X-OC-ID-TOKEN` HTTP header. If the token is inaccurate or missing, an HTTP response with status code 400 is returned, and the HTTP request is not processed.

See Understanding Anti-CSRF Measures for more information about the generation and use of anti-CSRF tokens in the Oracle Messaging Cloud Service REST API.

If desired, you can disable the CSRF prevention mechanism in these ways:

- To disable the mechanism before the connection's anti-CSRF token is generated, pass the `X-OC-ID-TOKEN-STATUS` HTTP header with the value of `disabled` on the messaging context's first HTTP request.
- To disable the mechanism for a messaging context that has already generated an anti-CSRF token, pass the `X-OC-ID-TOKEN-STATUS` HTTP header with the value of `disabled`, and also pass the `X-OC-ID-TOKEN` HTTP header with the value of the token that was generated for the messaging context.

> **Note:**
>
> If anti-CSRF is enabled, see Messaging Context and HTTP Cookies for information about how Oracle Messaging Cloud Service uses `JSESSIONID` HTTP cookies and why clients may wish to durably store anti-CSRF tokens and cookies associated with active messaging contexts.

Resource Management versus Message Transmission APIs

The operations in the Oracle Messaging Cloud Service REST API can be divided into two functional areas:

- **Resource management**: The Resource Management API provides functionality to create and manage destinations, and message push listeners.
- **Message transmission**: The Message Transmission API provides functionality to create and manage connections, create and manage sessions, send messages through producers, receive messages through consumers, create and delete durable subscriptions, inspect messages through queue browsers, and create and delete temporary destinations.

Message Types

This section provides information about various message types supported by the REST API.

Topics:

The REST API supports the `HTTP` message type in addition to all of the JMS message types such as `TEXT` and `BYTES`.

When sending a message through the REST API, set the value of the `X-OC-MESSAGE-TYPE` HTTP header to specify the message's type. The default message type is `HTTP`.

When messages are sent and received through the REST API, the message body is transmitted as the HTTP request body.

The valid values for the `X-OC-MESSAGE-TYPE` HTTP header along with any required formatting for the HTTP request body are described in the following sections.

`PLAIN`

The message has no body.

Note the following rules for a message of this type:

- When the message is sent through the REST API, the HTTP request's body is ignored.
- When the message is accessed through the Java library, the message is an object of the class `javax.jms.Message` that has no body.
- When the message is received through the REST API, the HTTP response body and `Content-Type` header are empty.
- When the message is pushed to a URL by a message push listener, the HTTP request body and `Content-Type` header are empty.

`TEXT`

The message's body is a String.

Note the following rules for a message of this type:

- When the message is sent through the REST API, the HTTP request's body is converted to a String using the encoding specified by the HTTP request's headers.

- When the message is accessed through the Java library, the message is an object of the class `javax.jms.TextMessage`.
- When the message is received through the REST API, the HTTP response's body is encoded with UTF-8 and the `Content-Type` header is as follows:

  ```
  text/plain; charset=UTF-8
  ```

- When the message is pushed to a URL by a message push listener, the HTTP request's body is encoded with UTF-8 and the `Content-Type` header is as follows:

  ```
  text/plain; charset=UTF-8
  ```

`BYTES`

The message's body is an array of bytes.

Note the following rules for a message of this type:

- When the message is sent through the REST API, the HTTP request's body is handled as an array of bytes.
- When the message is accessed through the Java library, the message is an object of the class `javax.jms.BytesMessage`.
- When the message is received through the REST API, the HTTP response's body is the bytes of the array and the `Content-Type` is as follows:

  ```
  application/octet-stream
  ```

- When the message is pushed to a URL by a message push listener, the HTTP request's body is the bytes of the array and the `Content-Type` is as follows:

  ```
  application/octet-stream
  ```

`OBJECT`

The message's body is a serialized Java object.

Note the following rules for a message of this type:

- When the message is sent through the REST API, the HTTP request's body is the serialization of a Java object. Any serializable object can be sent through the REST API.
- When the message is accessed through the Java library, the message is an object of the class `javax.jms.ObjectMessage`.
- When the message is received through the REST API, the HTTP response's body is the serialization of the Java object and the `Content-Type` is as follows:

  ```
  application/octet-stream
  ```

- When the message is pushed to a URL by a message push listener, the HTTP request's body is the serialization of the Java object and the `Content-Type` is as follows:

  ```
  application/octet-stream
  ```

`HTTP`

The message's body is a representation of the content of an HTTP request, including the metadata of the content's media type and language.

Note the following rules for a message of this type:

- When the message is sent through the REST API, the HTTP request's body, `Content-Type` header, and `Content-Language` header are used to create an `oracle.cloud.messaging.client.HttpContent` object.
- When the message is accessed through the Java library, the message is a `javax.jms.ObjectMessage` whose content is a populated `oracle.cloud.messaging.client.HttpContent` object.
- When the message is received through the REST API, the HTTP response's body, `Content-Type` header, and `Content-Language` header are set from the `oracle.cloud.messaging.client.HttpContent` object.
- When the message is pushed to a URL by a message push listener, the HTTP request's body, `Content-Type` header, and `Content-Language` header are set from the `oracle.cloud.messaging.client.HttpContent` object.

`MAP`

The message's body is a set of name/value pairs that defines a mapping from names to values.

Note the following rules for a message of this type:

- When the message is sent through the REST API, the HTTP request's body is an XML document with the following format:

  ```
  <map>
      <entry>
          <name>name</name>
          <type>type</type>
          <value>value</value>
       </entry>
       …
  </map>
  ```

  The child elements of `<map>` must all be `<entry>`. There may be 0 or more `<entry>` child elements. Every `<entry>` must contain exactly one `<name>` element whose content defines the name for the map entry, and either one `<type>` and one `<value>` or none of either. If there are multiple `<name>` elements with the same content, this is an error and the result is unspecified.

  If the `<type>` and `<value>` pair is not present, the value assigned to *name* is `null`. If the `<type>` and `<value>` pair is present, *type* must be one of the following:

  – `boolean`: *value* must be `true` or `false`

  - `byte`: *value* must be two hexadecimal digits (where a hexadecimal digit is a digit from 0-9 or a letter from A-F)
  - `short`: *value* must be a base-10 representation of a Java short integer
  - `int`: *value* must be a base-10 representation of a Java int
  - `long`: *value* must be a base-10 representation of a Java long integer
  - `float`: *value* must be a String representation of a Java float
  - `double`: *value* must be a String representation of a Java double
  - `string`: *value* must be valid XML character data. An empty `<value>` object is interpreted as the empty String rather than `null`.
  - `char`: *value* must be a single character
  - `bytes`: *value* must be an even-length string of hexadecimal digits (where a hexadecimal digit is a digit from 0-9 or a letter from A-F)
- When the message is accessed through the Java library, the message is an object of the class `javax.jms.MapMessage`. Each entry specifies a name/value pair in the `javax.jms.MapMessage`, with name as *name* and value as specified by *type* and *value*.
- When the message is received through the REST API, the HTTP response's body is an XML document with the same format as the XML document for the HTTP request's body, and the `Content-Type` is `application/xml`.
- When the message is pushed to a URL by a message push listener, the HTTP request's body is an XML document with the same format as that in an HTTP request to create a `MAP` message, and the `Content-Type` is `application/xml`.

`STREAM`

The message's body is a sequence of values.

Note the following rules for a message of this type:

- When the message is sent through the REST API, the HTTP request's body is an XML document with the following format:

```
<stream>
    <item>
        <type>type</type>
        <value>value</value>
    </item>
     …
</stream>
```

  All child elements of `<stream>` must be `<item>`. There may be 0 or more `<item>` child elements. Every `<item>` must contain either one `<type>` and one `<value>` or none of either. If the `<type>` and `<value>` pair is not present, the item in the stream is `null`. If the `<type>` and `<value>` pair is present, *type* and *value* must match one of the following:

  - `boolean`: *value* must be `true` or `false`
  - `byte`: *value* must be two hexadecimal digits (where a hexadecimal digit is a digit from 0-9 or a letter from A-F)
  - `short`: *value* must be a base-10 representation of a Java short integer

  - `int`: *value* must be a base-10 representation of a Java int
  - `long`: *value* must be a base-10 representation of a Java long integer
  - `float`: *value* must be a String representation of a Java float
  - `double`: *value* must be a String representation of a Java double
  - `string`: *value* must be valid XML character data. Empty `<value>` elements are interpreted as representing the empty String rather than `null`.
  - `char`: *value* must be a single character
  - `bytes`: *value* must be an even-length string of hexadecimal digits (where a hexadecimal digit is a digit from 0-9 or a letter from A-F)

  The values in the stream are in the same order as the `<item>` elements in the XML document.

- When the message is accessed through the Java library, the message is an object of the class `javax.jms.StreamMessage`. Each `<item>` specifies a value written into the `javax.jms.StreamMessage`, written in document order.
- When the message is received through the REST API, the HTTP response's body is an XML document with the same format as the XML document for the HTTP request's body, and the `Content-Type` is `application/xml`.
- When the message is pushed to a URL by a message push listener, the HTTP request's body is an XML document with the same format as that in an HTTP request to create a `STREAM` message, and the `Content-Type` is `application/xml`.

Message Headers and Properties

Message headers and properties are treated as HTTP headers when messages are sent and received through the REST API.

The following table maps message headers to the corresponding HTTP headers:

Message Header	HTTP Header
Correlation ID	`X-OC-CORRELATION-ID`
Delivery Mode	`X-OC-DELIVERY-MODE`
Destination	`X-OC-DESTINATION`
Expiration	`X-OC-EXPIRATION`
Message ID	`X-OC-MESSAGE-ID`
Redelivered	`X-OC-REDELIVERED`
Reply To	`X-OC-REPLY-TO`
Timestamp	`X-OC-TIMESTAMP`

When a message is sent through the REST API, message properties are set by specifying HTTP headers that follow a specific naming convention. After a message is sent, the properties of the message are treated as standard JMS message properties. When a message is received through the REST API, the message properties are converted to HTTP headers that follow the same naming convention.

Message properties can be set and read through HTTP headers by using the following convention:

`X-OC-`*`TYPE`*`-PROPERTY-`*`NAME`* where *`NAME`* and *`TYPE`* are strings.

`NAME` is the message property's name. The name can only contain alphanumeric characters and underscores. All characters are made lowercase by the service.

`TYPE` is the message property's type, which must be one of the following types:

Message Property Type	HTTP Header Value
`BOOLEAN`	Must be `true` or `false`
`BYTE`	Must be two hexadecimal digits (where a hexadecimal digit is a digit from 0-9 or a letter from A-F)
`SHORT`	Must be a base-10 representation of a Java short integer
`INT`	Must be a base-10 representation of a Java int
`LONG`	Must be a base-10 representation of a Java long integer
`FLOAT`	Must be a String representation of a Java float
`DOUBLE`	Must be a String representation of a Java double
`STRING`	Must be a legal HTTP header value. See Message Headers in the *Hypertext Transfer Protocol - HTTP/1.1* document.

XML versus JSON Response Types

While some of the Oracle Messaging Cloud Service REST API endpoints support JSON as a response type, all of the endpoints support XML.

The default response type for all endpoints is XML.

REST API endpoints that only support XML as a response type include:

- Creating or viewing a message push listener
- Creating or viewing a durable subscription
- Receiving a Map or Stream message
- Creating temporary destinations
- Listing temporary destinations
- Retrieving the properties of a single temporary destination
- Retrieving the properties of a queue browser

The response type for a REST API endpoint can be controlled by setting the HTTP request's `Accept` header to either `application/json` or `application/xml`. To indicate that a client prefers JSON but is willing to accept XML if it is the only format available, the HTTP request's `Accept` header may be set to, for example, `application/json, application/xml;q=0.5`.

If an HTTP response from the REST API is the result of receiving a message, the `Accept` header is ignored. If the HTTP request is a request to receive a message, but an error response is generated, the `Accept` header is used to determine the format of the error response's body. Otherwise, if the REST API cannot provide a response of the type specified in the `Accept` header, an HTTP response with status code 406 is returned.

3

Accessing Oracle Messaging Cloud Service Using REST API

Oracle Messaging Cloud Service provides a Representational State Transfer (REST) API for sending and receiving messages, as well as managing resources such as queues, topics, durable subscriptions, and message push listeners. This section describes how to use the REST API in applications that make use of Oracle Messaging Cloud Service.

Topics:

Topology API

The topology API provides functionality to obtain the topology of a namespace or messaging context. The topology API is available only through the REST API. The information provided by the topology API will, in general, reflect the most recent topology of a namespace or messaging context, but is not necessarily real-time. That is, changes to a namespace or messaging context that has just happened may not be reflected immediately.

Topics:

Viewing all Messaging Contexts

A user can view all messaging contexts and their encapsulated ephemeral resources.

Method: `GET`

Path: `/messagingcontexts`

Scope: Service Instance

Authorization: Messaging Administrator or Messaging Worker

Result: Returns the topology information of all messaging contexts and their encapsulated ephemeral resources.

Response Body: JSON or XML.

JSON

The JSON output for a `GET` from `/messagingcontexts` is a JSON array. Each element of the array is a JSON object corresponding to a single, unique messaging context.

JSON Object for a Messaging Context

A JSON object for a messaging context will contain at most the following fields:

Field Name	Description
`id`	A service-generated ID for the messaging context. This ID is relevant only to the topology API. Each element of the array will have a different `id` value. This field is always present.
`connections`	A JSON array of JSON objects, with each element representing a distinct connection created in the messaging context. If there are no unclosed connections in the messaging context, this field will not be present.

JSON Object for a Connection

A JSON object for a connection will contain at most the following fields:

Field Name	Description
`name`	The name under which the REST API client created the connection. This field is always present.
`clientId`	The connection's client ID; if the connection does not have a client ID, this field will not be present.
`started`	A boolean that is `true` if the connection is currently started, and `false` otherwise. This field is always present.
`sessions`	A JSON array of JSON objects, with each element representing a distinct session created from this connection. If there are no unclosed sessions currently created from this connection, this field will not be present.
`temporaryQueues`	A JSON array of JSON objects, with each element representing a temporary queue associated with this connection. If there are no undeleted temporary queues associated with this connection, this field will not be present.
`temporaryTopics`	A JSON array of JSON objects, with each element representing a temporary topic associated with this connection. If there are no temporary topics associated with this connection, this field will not be present.

JSON Object for a Session

A JSON object for a session will contain at most the following fields:

Field Name	Description
`name`	The name under which the REST API client created the session. This field is always present.
`transacted`	A boolean that is `true` if the session is transacted, and `false` otherwise. This field is always present.
`ackMode`	A boolean indicating the acknowledgement mode of the session. Its value is `auto`, `client`, or `dups_ok`. This field will be present if `transacted` is `false`; otherwise, this field will not be present.
`sessions`	A JSON array of JSON objects, with each element representing a distinct session created from this connection. If there are no unclosed sessions currently created from this connection, this field will not be present.
`producers`	A JSON array of JSON objects, with each element representing a producer in this session. If there are no unclosed producers in this session, this field will not be present.
`consumers`	A JSON array of JSON objects, with each element representing a consumer in this session. If there are no unclosed consumers in this session (other than durable subscribers), this field will not be present. Consumers that are durable subscribers will not appear in this array.
`durableSubscribers`	A JSON array of JSON objects, with each element representing a durable subscriber in this session. If there are no unclosed durable subscribers in this session, this field will not be present.
`queueBrowsers`	A JSON array of JSON objects, with each element representing a queue browser in this session. If there are no unclosed queue browsers in this session, this field will not be present.

JSON Object for a Producer

A JSON object for a producer will contain at most the following fields:

Field Name	Description
`name`	The name under which the REST API client created the producer. This field is always present.

Field Name	Description
`destination`	The destination to which the producer sends messages. If the producer has no associated destination (so that the destination to which a message is sent must be specified in the send), this field will not be present. If present, it will have one of the following formats: • `/queues/`*`queueName`* • `/topics/`*`topicName`* • `/temporaryQueues/`*`queueName`* • `/temporaryTopics/`*`topicName`*
`timeToLive`	The time-to-live, in milliseconds, that is the default time-to-live for messages sent by this producer. This field will always be present, and will be formatted as an integer, not a String.
`deliveryMode`	A String giving the default delivery mode for messages sent by this producer. This field will always be present, and will have the value `persistent`or `non_persistent`.

JSON Object for a Consumer

A JSON object for a consumer (other than a durable subscriber) will contain at most the following fields:

Field Name	Description
`name`	The name under which the REST API client created the consumer. This field is always present.
`destination`	The destination from which the producer receives messages. This field will always be present, and will have one of the following formats: • `/queues/`*`queueName`* • `/topics/`*`topicName`* • `/temporaryQueues/`*`queueName`* • `/temporaryTopics/`*`topicName`*
`selector`	The selector for the consumer. This field will only be present if the consumer has a selector.
`localMode`	The local mode of the consumer, which defines whether a consumer on a topic will receive message sent on the same connection the consumer was created from. If the consumer is not on a topic or temporary topic, this field will not be present. The default is for topic consumers to receive all messages sent to the topic that match their selector, so this field will only be present if the consumer does not receive messages sent on its connection, in which case the field will be present, and will have value `NO_LOCAL`.

JSON Object for a Durable Subscriber

A JSON object for a durable subscriber will contain at most the following fields:

Field Name	Description
`name`	The name under which the REST API client created the consumer. This field is always present.
`destination`	The destination from which the producer receives messages. This field will always be present, and will have one of the following formats: • `/topics/`*`topicName`* • `/temporaryTopics/`*`topicName`*
`subscriptionName`	The string that is the subscription name corresponding to the subscriber. **Note:** This is different from the `name` field, which gives the name by which REST API clients refer to the subscriber.
`selector`	The selector for the consumer. This field will only be present if the consumer has a selector.
`localMode`	The local mode of the consumer, which defines whether a consumer on a topic will receive message sent on the same connection the consumer was created from. If the consumer is not on a topic or temporary topic, this field will not be present. The default is for topic consumers to receive all messages sent to the topic that match their selector, so this field will only be present if the consumer does not receive messages sent on its connection, in which case the field will be present, and will have value `NO_LOCAL`.

JSON Object for a Queue Browser

A JSON object for a queue browser will contain at most the following fields:

Field Name	Description
`name`	The name under which the REST API client created the queue browser. This field is always present.
`queue`	The queue that the browser browses. This field is always present, and will have the format `/queues/`*`queueName`*.
`selector`	The selector for the browser. This field will only be present if the browser has a selector.

XML

The XML output for a `GET` from `/messagingcontexts` is an XML document with root element `<messagingcontexts>`. The XML output for a given service instance is closely analogous to the JSON in the following sense:

- A field in JSON object with a certain name will be represented in XML by a child element whose name is the same as the field name, and whose content corresponds to that of the value of the field.
- The elements of a JSON array will be represented in XML as a sequence of `<items>` elements, with the content of the nth `<items>` corresponding to the nth value in the JSON array.

Viewing a Messaging Context

A user can view a single messaging context and all of its encapsulated ephemeral resources by providing the messaging context's ID

Method: `GET`

Path: `/messagingcontexts/`*`messagingContextID`*

Scope: Messaging Context

Authorization: Messaging Administrator or Messaging Worker

Result: Returns the topology information of a single messaging context and all of its encapsulated ephemeral resources.

Response Body: JSON or XML.

JSON

The output for a `GET` from `/messagingcontexts/`*`messagingContextID`* is a JSON object that represents the messaging context with the specified ID. The output format is same as described in the JSON output sections for messaging contexts. A 404 response is generated if the service instance has no messaging context with the given ID.

XML

The XML output for a `GET` from `/messagingcontexts/`*`messagingContextID`* is an XML document with root element `<messagingcontexts>` whose content is XML corresponding to the fields of a JSON object representing the messaging context with the specific ID. A 404 response is generated if the service instance has no messaging context with the given ID.

Sample Outputs of Topology API

This section provides sample outputs of the topology API.

Sample JSON Output

Sample JSON output from `GET /messagingcontexts` for a service instance with two messaging contexts:

```
[
  {
    "id":"19CB3BA96A0B068B"
  },
  {
    "id":"7E1B3AAF09D27170",
    "connections":[
      {
```

```
"name":"conn348755961301170266",
"clientId":"topologyCID2211817070735200489",
"started":true,
"sessions":[
  {
    "name":"topologySession6782049576740617368",
    "transacted":true,
    "consumers":[
      {
        "name":"cons1986478156341504515",
        "destination":"/topics/topologyTopic1",
        "selector":"property195=232"
      },
      {
        "name":"cons8406813156667706998",
        "destination":"/topics/topologyTopic2",
        "localMode":"NO_LOCAL"
      }
    ],
    "durableSubscribers":[
      {
        "name":"sub8114000992284070555",
        "destination":"/topics/topologyTopic2",
        "subscriptionName":"subscr668",
         "localMode":"NO_LOCAL"
      },
      {
        "name":"sub82185790999888865396",
        "destination":"/topics/topologyTopic1",
        "subscriptionName":"subscr819"
      }
    ],
    "queueBrowsers":[
      {
        "name":"qb658303539525244693",
        "queue":"/queues/topologyQueue1",
        "selector":"property499=195"
      },
      {
        "name":"qb7987305899602893098",
        "queue":"/queues/topologyQueue2",
        "selector":"property269=410"
      }
    ]
  },
  {
    "name":"topologySession4005620455377307315",
    "transacted":false,
    "ackMode":"auto",
    "producers":[
      {
        "name":"prod2624315421281685702",
        "destination":"/queues/topologyQueue1",
        "timeToLive":1209600000,
        "deliveryMode":"non_persistent",
      }
    ],
    "durableSubscribers":[
      {
        "name":"sub1542678913898814730",
        "destination":"/topics/topologyTopic1",
```

```
              "subscriptionName":"subscr765"
            }
          ],
          "queueBrowsers":[
            {
              "name":"qb83025982287048587 36",
              "queue":"/queues/topologyQueue0"
            },
            {
              "name":"qb43261105545319902 21",
              "queue":"/queues/topologyQueue3",
              "selector":"property197=858"
            }
          ]
        }
      ],
      "temporaryQueues":[
        {
          "name":"68BC612EB343DB9A"
        }
      ],
      "temporaryTopics":[
        {
          "name":"D1970C11C263F97D"
        }
      ]
    }
  ]
 }
]
```

The JSON output of `GET /messagingcontexts/19CB3BA96A0B068B` for this service instance would therefore be:

```
{
    “id”:"19CB3BA96A0B068B"
}
```

Sample XML Output

Sample XML output for `GET /messagingcontexts` in the same service instance as above:

```
<messagingcontexts>
  <items>
    <id>19CB3BA96A0B068B</id>
  </items>
  <items>
    <id>7E1B3AAF09D27170</id>
    <connections>
      <items>
        <name>conn348755961301170266</name>
        <clientId>topologyCID2211817070735200489</clientId>
        <started>true</started>
        <sessions>
          <items>
            <name>topologySession6782049576740617368</name>
            <transacted>true</transacted>
            <consumers>
              <items>
                <name>cons1986478156341504515</name>
```

```
        <destination>/topics/topologyTopic1</destination>
        <selector>property195=232</selector>
      </items>
      <items>
        <name>cons8406813156667706998</name>
        <destination>/topics/topologyTopic2</destination>
        <localMode>NO_LOCAL</localMode>
      </items>
    </consumers>
    <durableSubscribers>
      <items>
        <name>sub81140000992284070555</name>
        <destination>/topics/topologyTopic2</destination>
        <subscriptionName>subscr668</subscriptionName>
        <localMode>NO_LOCAL</localMode>
      </items>
      <items>
        <name>sub8218579099988865396</name>
        <destination>/topics/topologyTopic1</destination>
        <subscriptionName>subscr819</subscriptionName>
      </items>
    </durableSubscribers>
    <queueBrowsers>
      <items>
        <name>qb658303539525244693</name>
        <queue>/queues/topologyQueue1</queue>
        <selector>property499=195</selector>
      </items>
      <items>
        <name>qb7987305899602893098</name>
        <queue>/queues/topologyQueue2</queue>
        <selector>property269=410</selector>
      </items>
    </queueBrowsers>
  </items>
  <items>
    <name>topologySession4005620455377307315</name>
    <transacted>false</transacted>
    <ackMode>auto</ackMode>
    <producers>
      <items>
        <name>prod2624315421281685702</name>
        <destination>/queues/topologyQueue1</destination>
        <timeToLive>1209600000</timeToLive>
        <deliveryMode>non_persistent</deliveryMode>
      </items>
    </producers>
    <durableSubscribers>
      <items>
        <name>sub1542678913898814730</name>
        <destination>/topics/topologyTopic1</destination>
        <subscriptionName>subscr765</subscriptionName>
      </items>
    </durableSubscribers>
    <queueBrowsers>
      <items>
        <name>qb8302598228704858736</name>
        <queue>/queues/topologyQueue0</queue>
      </items>
      <items>
        <name>qb4326110554531990221</name>
```

```
                <queue>/queues/topologyQueue3</queue>
                <selector>property197=858</selector>
              </items>
            </queueBrowsers>
          </items>
        </sessions>
        <temporaryQueues>
          <items>
            <name>68BC612EB343DB9A</name>
          </items>
        </temporaryQueues>
        <temporaryTopics>
          <items>
            <name>D1970C11C263F97D</name>
          </items>
        </temporaryTopics>
      </items>
    </connections>
  </items>
</messagingcontexts>
```

The XML output of `GET /messagingcontexts/19CB3BA96A0B068B` for this service instance would therefore be:

```
<messagingcontext>
    <id>19CB3BA96A0B068B</id>
</messagingcontext>
```

Usage API

The Usage API is a part of the Oracle Messaging Cloud Service REST API that provides information about resource usage for a given Oracle Messaging Cloud Service instance. The API provides information on both how much of certain resources are being used, and on various limits how much those resources may be used.

Topics:

- About Usage API
- Sample Outputs of Usage API

About Usage API

This topic provides information about usage API.

Method: `GET`

Path: `/usage`

Scope: Service Instance

Authorization: Messaging Administrator

Response Body: An XML or JSON representation of resource usage and limits. The format is chosen based on the value of the `Accept` header.

The response body of a call to the usage API contains the following information:

- How many connections, queues, topics, and durable subscriptions are currently being used, and what are the maximum numbers of each that can be used for the given service instance.
- The hard and soft quotas on destination backlogs. Note that the API returns only the quotas, not the backlog sizes. For more information, see Hard and Soft Quotas
- Metering data: How many bytes of data have been sent out through the REST API (the egress data) and the number of calls to the REST API in a particular period of time. The egress data includes all bytes sent in HTTP responses, including status line, headers, and the response body. The number of calls to the REST API includes all HTTP requests made.

The information provided by the usage API will, in general, reflect the most recent usage, but is not necessarily real-time. That is, changes to resource usage, metering data, etc. that have just happened may not be reflected immediately.

The Usage API returns usage information for connections, queues, topics, and durable subscriptions, and hard and soft quota limits for destination backlogs. The Usage API also returns the most recent metering data which contains both the number of API calls and egress bandwidth. The non-metered data is accurate to the last 30 seconds. The metered data, api calls and egress bandwidth, are an aggregate over a time period and each metered data section contains the respective start and end times for the metered data. The metered data start and end times are in Coordinated Universal Time (UTC). The destination backlog data is limited to returning the hard quota and soft quota which applies to all destinations (queues and topics) for a service. The backlog quotas apply to both the number of messages and total bytes of messages. If the hard quota of either is exceeded, then sends will fail until the backlog falls below the soft quota.

JSON Format

Usage information in JSON is expressed as a JSON object. The connection, queue, topic, and durable subscription usage is given by the `"connectionCount"`, `"queueCount"`, `"topicCount"`, and `"durableSubscriptionCount"` fields of the object respectively. The value of each of these fields is a JSON object with a `"max"` field whose value is the maximum number of connections, queues, topics, or durable subscriptions that may be used, and a `"used"` field whose value is the current number of connections, queues, topics, or durable subscriptions in use (subject to the above caveats about the data not being real-time).

Destination backlog limitations on number and number of bytes of messages are given by the `"destinationBacklogMessageCount"` and `"destinationBacklogBytes"` fields respectively. The value of each of these fields is a JSON object with a `"hardQuota"` field whose value is the hard quota and a `"softQuota"` field whose value is the soft quota.

Metering data is given by a `"meteringUsages"` field whose value is a JSON array of JSON objects. Each object in the array contains the metering data for a certain time period, expressed as the following fields:

- `"startTimeUtc"`

  The value of this field is the first minute of the time period for the metering data, expressed in Coordinated Universal Time (UTC).
- `"endTimeUtc"`

The value of this field is the last minute of the time period for the metering data, expressed in Coordinated Universal Time (UTC). The metering data will include all metering for the entire minute. For example, if the end time is `2015-07-09T13:09`, all API calls occurring at or after 13:09 but before 13:10 will be included.

- `"dataCenter"`

  The value of this field is the data center to which the metering data in the element applies.

- `"meteredResourceUsages"`

  A JSON array of JSON objects containing data on a given metered resource for the given period and data center.

  Each object in the `"meteredResourceUsages"` array will have the following fields:

  – `"resourceName"`

    The value of this field is the name of the resource whose data is being reported. The value of this field will be either `"EGRESS_DATA"` or `"API_CALLS"`.

  – `"quantity"`

    The value of this field is an integer giving amount of the resource used.

  – `"units"`

    The value of this field is the unit of measurement for the value of the `"quantity"` field. If the value of `"resourceName"` is `"EGRESS_DATA"`, the value of `"units"` will be `"Bytes"`. If the value of `"resourceName"` is `"API_CALLS"`, the value of `"units"` will be `"API Calls"`.

XML Format

The connection, queue, topic, and durable subscription usage is given by the `<connectionCount>`, `<queueCount>`, `<topicCount>`, and `<durableSubscriptionCount>` elements respectively. Each of these elements contains a `<max>` element whose content is the maximum number of connections, queues, topics, or durable subscriptions that may be used, and a `<used>` element whose content is the current number of connections, queues, topics, or durable subscriptions in use (subject to the above caveats about the data not being real-time).

Destination backlog limitations on number and number of bytes of messages are given by the `<destinationBacklogMessageCount>` and `<destinationBacklogBytes>` elements respectively. Each of these elements contains a `<hardQuota>` element whose content is the hard quota and a `<softQuota>` element whose content is the soft quota.

Metering data is contained in `<meteringUsages>` elements. Each such element contains the metering data for a certain time period, expressed as the following child elements:

- `<startTimeUtc>`

  The content of this element is the first minute of the time period for the metering data, expressed in Coordinated Universal Time (UTC).

- `<endTimeUtc>`

The content of this element is the last minute of the time period for the metering data, expressed in Coordinated Universal Time (UTC). The metering data will include all metering for the entire minute. For example, if the end time is `2015-07-09T13:09`, all API calls occurring at or after 13:09 but before 13:10 will be included.

- `<dataCenter>`

  The content of this element is the data center to which the metering data in the element applies.

- `<meteredResourceUsages>`

  One or more elements containing data on a given metered resource for the given period and data center.

  Each `<meteredResourceUsages>` element will contain the following elements:

  – `<resourceName>`

    The content of this element is the name of the resource whose data is being reported. The content of this element will be either `EGRESS_DATA` or `API_CALLS`.

  – `<quantity>`

    The content of this element is an integer giving amount of the resource used.

  – `<units>`

    The content of this element is the unit of measurement for the content of the `<quantity>` element. If the content of `<resourceName>` is `EGRESS_DATA`, the content of `<units>` will be `Bytes`. If the content of `<resourceName>` is `API_CALLS`, the content of `<units>` will be `API Calls`.

Sample Outputs of Usage API

This section provides sample outputs of the usage API.

Sample JSON Output

Sample JSON output of the usage API:

```
{
    "connectionCount": {
        "max": 200,
        "used": 3
    },
    "queueCount": {
        "max": 10000,
        "used": 156
    },
    "topicCount": {
        "max": 10000,
        "used": 156
    },
    "durableSubscriptionCount": {
        "max": 10000,
        "used": 53
    },
    "destinationBacklogMessageCount": {
        "hardQuota": 100,
        "softQuota": 70
    },
```

```
    "destinationBacklogBytes": {
        "hardQuota ": 52428800,
        "softQuota ": 36700160
    },
    "meteredUsages": [
        {
            "dataCenter": "dc",
            "endTimeUtc": "2015-07-09T13:09",
            "meteredResourceUsages": [
                {
                    "quantity": 2425581,
                    "resourceName": "EGRESS_DATA",
                    "units": "Bytes"
                },
                {
                    "quantity": 3426,
                    "resourceName": "API_CALLS",
                    "units": "API Calls"
                }
            ],
            "startTimeUtc": "2015-07-09T13:00"
        },
        {
            "dataCenter": "dc",
            "endTimeUtc": "2015-07-09T13:19",
            "meteredResourceUsages": [
                {
                    "quantity": 2425521,
                    "resourceName": "EGRESS_DATA",
                    "units": "Bytes"
                },
                {
                    "quantity": 3426,
                    "resourceName": "API_CALLS",
                    "units": "API Calls"
                }
            ],
            "startTimeUtc": "2015-07-09T13:10"
        }
    ]
}
```


Sample XML Output

Sample XML output of the usage API:


```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<usage>
    <connectionCount>
        <max>200</max>
        <used>3</used>
    </connectionCount>
    <queueCount>
        <max>10000</max>
        <used>156</used>
    </queueCount>
    <topicCount>
        <max>10000</max>
        <used>156</used>
    </topicCount>
    <durableSubscriptionCount>
```

```
            <max>10000</max>
            <used>53</used>
        </durableSubscriptionCount>
        <destinationBacklogMessageCount>
          <hardQuota>100</hardQuota>
          <softQuota>70</softQuota>
        </destinationBacklogMessageCount>
        <destinationBacklogBytes>
          <hardQuota>52428800</hardQuota>
          <softQuota>36700160</softQuota>
        </destinationBacklogBytes>
        <meteredUsages>
            <startTimeUtc>2015-07-09T13:00</startTimeUtc>
            <endTimeUtc>2015-07-09T13:09</endTimeUtc>
            <dataCenter>dc</dataCenter>
            <meteredResourceUsages>
                <resourceName>EGRESS_DATA</resourceName>
                <quantity>2425581</quantity>
                <units>Bytes</units>
            </meteredResourceUsages>
            <meteredResourceUsages>
                <resourceName>API_CALLS</resourceName>
                <quantity>3426</quantity>
                <units>API Calls</units>
            </meteredResourceUsages>
        </meteredUsages>
        <meteredUsages>
            <startTimeUtc>2015-07-09T13:10</startTimeUtc>
            <endTimeUtc>2015-07-09T13:19</endTimeUtc>
            <dataCenter>dc</dataCenter>
            <meteredResourceUsages>
                <resourceName>EGRESS_DATA</resourceName>
                <quantity>2425521</quantity>
                <units>Bytes</units>
            </meteredResourceUsages>
            <meteredResourceUsages>
                <resourceName>API_CALLS</resourceName>
                <quantity>3426</quantity>
                <units>API Calls</units>
            </meteredResourceUsages>
        </meteredUsages>
    </usage>
```

About Escaped Value Strings

This topic provides information about escaped value Strings.

An escaped value String is a String that uses the following JSON conventions for representing characters that may not legally occur in HTTP header values:

Escape Sequence	Description
\\	The backslash character (\)
\b	The backspace character (ctrl-H)
\f	The form-feed character (ctrl-L)
\n	The new-line character (ctrl-J)
\r	The carriage return character (ctrl-M)

Escape Sequence	Description
`\t`	The tab character
`\u<4 hexadecimal digits>`	The Unicode character whose encoding as an unsigned base-16 integer is given by the 4 hexadecimal digits.

The following is an example:

Raw String	Corresponding Escaped Value String
`<tab>\ding!<ctrl-G><ctrl-M><ctrl-J>`	`\t\\ding!\u0007\r\n`

Note that if a String *s* is a legal HTTP header value that contains no backslashes, the escaped value String corresponding to *s* is the same as *s*.

About Using the REST API

Oracle Messaging Cloud Service can be accessed using the REST API.

Topics:

Before you begin using the REST API, be sure to review the guidelines in this section as well as the guidelines in Considerations When Developing Applications That Use Oracle Messaging Cloud Service and Using the REST API.

Basics of the REST API

Any application platform that understands HTTP can use Oracle Messaging Cloud Service through the Representational State Transfer (REST) interface.

REST is an architectural style for making distributed resources available through a uniform interface that includes uniform resource identifiers (URIs), well-defined operations, hypermedia links, and a constrained set of media types. Typically, these

operations include reading, writing, editing, and removing, and media types include JSON and XML.

The REST API can be used from any environment connected to the Internet. REST commands use standard HTTP methods to specify whether the resource named by the URI is being read (`GET`), created (`PUT`), modified (`POST`), or deleted (`DELETE`). REST APIs can be used from any platform and by any development technology that can make HTTP requests and receive HTTP responses.

Oracle Messaging Cloud Service provides a REST API by which applications can access the service using HTTP. The REST API resources are modeled after Java Message Service (JMS). Various objects used in JMS, such as `Queue`, `Topic`, `Session`, `MessageProducer`, `MessageConsumer`, and so on, are resources in the REST interface with a subset of the properties and methods in JMS. In the REST interface, however, the resources corresponding to these objects are referred to by names supplied by the client, rather than as Java object references.

Structured information supplied as parameters to or returned by Java methods is represented as query string parameters, HTTP headers, or JSON or XML data in HTTP request or response bodies. All message metadata supported by JMS is represented as custom HTTP headers. In addition, some HTTP metadata, such as the `Content-Type` of an HTTP request to send a message, may be passed along in the payload of the generated JMS message, and is used to set headers of the HTTP request or response that delivers the message to a REST client or HTTP endpoint.

Functional Areas of the REST API

The operations in the Oracle Messaging Cloud Service REST API can be divided into two functional areas:

- Resource Management API

  The Resource Management API is used to manage resources such as **destinations**, and **message push listeners**, and to control expiration of cookies.

- Message Transmission API

  The Message Transmission API involves ephemeral resources (connections, sessions, producers, consumers, queue browsers, and temporary destinations) that are created by explicit client action. If a messaging context expires, or the server that created the messaging context goes down, then a client using Message Transmission API methods may have to be prepared to detect that ephemeral resources have been lost and recreate them.

  It is important to note, however, that client acknowledgements and **transactions** are associated with sessions. If the messaging context containing a session expires, or the server that created the messaging context goes down, then any unacknowledged messages become available for receipt by other clients, and any receives or sends in a transacted session that have not been **committed** are **rolled back**.

Understanding Messaging Context and Cookies

A messaging context is a container of ephemeral objects like connections, sessions, producers, consumers, temporary destinations, and queue browsers.

A messaging context is identified by the `JSESSIONID` cookie. The only API specific to a messaging context is the API for getting and setting the MII, which controls the

expiration time of the messaging context. When the messaging context expires, all ephemeral objects contained in it are closed and deleted, except temporary destinations. A temporary destination is closed only if the connection by which the temporary destination was created is closed.

At least one messaging context must be created by a client in order to access Oracle Messaging Cloud Service. If an HTTP request does not include a `JSESSIONID` cookie for an unexpired messaging context, then a new messaging context is created. The HTTP response includes the header `X-OC-NEW-MESSAGING-CONTEXT: true` if a new messaging context is created.

REST clients can detect if a messaging context has expired or the server that created the messaging context has gone down by looking for this header. Server failures may cause messages to be delivered multiple times, and clients must be aware of, and possibly compensate for, duplicate delivery.

Clients that want to handle client failures gracefully should store the value of the `JSESSIONID` cookie in durable storage so that a client that is reactivated after a failure can reread the `JSESSIONID` cookie value from the durable store and access the messaging context that the client was using before the failure, if the messaging context has not expired. If this is not done, then reactivated clients get a new messaging context, with the same problems described in Messaging Context and HTTP Cookies, that is, unnecessarily holding connections and having no access to previously created ephemeral objects.

Understanding Durable Subscriptions

Clients that use **durable subscriptions** must supply a client ID for a connection which, along with a subscription name, identifies the durable subscription. There are restrictions on the use of a given client ID:

- When a client uses the Resource Management API to set a client ID on a connection, that client ID may not be used by any other connection until the messaging context expires or the connection is deleted. This is an additional reason why clients must store and send the `JSESSIONID` cookie, and store the `JSESSIONID` cookie value in durable storage. If a client ID is set on a connection, and the `JSESSIONID` cookie for that connection's messaging context is lost, then the client ID set on the connection cannot be used by any other client or listener until the messaging context whose cookie was lost expires.
- When a **message push listener** receives messages from a durable subscription, the client ID it uses may not be used by any other connection as long as the message push listener is in existence.

Understanding REST API Operations

The description of an operation of the REST API specifies the method and **path** for the HTTP request, which user roles are authorized to perform the operation (that is, Messaging Administrator or Messaging Worker, or either role) and the expected result of a successful operation. A description may, in addition, specify the following:

- Parameters that may or must be supplied in the HTTP request
- HTTP headers that may or must be present on the HTTP response
- How the HTTP request body is used
- What may or must be returned in the HTTP response body

- What errors may be returned (other than, or more specific than, the generic codes described in REST API HTTP Status Codes and Error Messages Reference)

The path specified for an operation is the part of the URL that comes after *`service name-identity domain name`*`/api/v1`. For example, if a tenant with the identity domain name "CloudBank" has a service with the name "OnlineBanking", and a client for that tenant wants to invoke an operation for path `/queues`, then the following is the HTTP request to a URL:

`https://messaging.`*`dataCenter`*`.oraclecloud.com/OnlineBanking-CloudBank/api/v1/queues`

Components of the path must always consist solely of letters of the Roman alphabet (a through z or A through Z), decimal digits (0 through 9), and underscores ('_'). No other characters are allowed. In particular, these restrictions apply to the names of queues, topics, and listeners.

In this document, the term **parameter** is a value supplied in an HTTP request that is not in the request body. Parameters may be supplied as query string parameters or as HTTP headers. The parameters that are recognized by Oracle Messaging Cloud Service are listed in REST API Parameters Reference. Throughout this document, parameters are listed by their query string name.

For HTTP responses that do not return messages, the format of the body in the HTTP response is determined by the `Accept` header of the HTTP request to which it is a response. For some paths, the service may be able to generate a body in the JSON format, with `Content-Type` header `application/json`, or in an XML format, with `Content-Type` header `application/xml`. XML is generated if both are equally acceptable. All HTTP requests should therefore have their `Accept` headers set so that one or both of `application/json` and `application/xml` are acceptable, and so that `application/xml` is acceptable unless the path also supports the JSON format. For responses that do not return a message, and which can return a response body in JSON or XML, if the media type corresponding to the JSON or XML is not acceptable by the `Accept` header, the response sent will have status code `406 (Not Acceptable)`.

For requests whose response is a message, the `Accept` header is ignored. A message is returned even if the `Content-Type` does not match the `Accept` header, rather than a 406 response being returned. Clients that want to ensure that they do not receive certain messages should use message properties and **selectors** to filter messages they do not want to receive.

Understanding Concurrent Access to Resources

This section provides information about restrictions on concurrent access to Oracle Messaging Cloud Service resources like connections, sessions, producers, consumers, destinations, and queue browsers.

Ephemeral objects like sessions, producers, consumers, and queue browsers should not be accessed by multiple client requests concurrently. For example, there should not be multiple client requests to send a message through the same producer at the same time, nor multiple client requests to receive messages from the same consumer at the same time.

Multiple sessions can be created from a given connection. In this case, different client requests can access different sessions concurrently.

Multiple producers, consumers, and queue browsers can be created with a given session, and different client requests can access different producers and consumers

concurrently, as long as no two requests are accessing the same producer, consumer, or queue browser concurrently.

Connections and destinations can be accessed by multiple client requests concurrently.

In the above statements, the term *access* refers to both invoking a method on a resource's path and referencing a resource as a request parameter. For example, clients should not have two concurrent requests to create a temporary queue that supply the same `session` parameter.

Note that these restrictions are not enforced by the service; they must be followed by the client.

Understanding Error Responses

When a status code greater than or equal to 400 is sent in a response, the body of the response is a JSON or XML document that specifies an error message (unless the response is a 406 error to a request that does not accept `application/json` or `application/xml`). The JSON format would be as follows:

```
{
    "httpStatusCode": statusCode,
    "httpMessage": "status line reason phrase for the code",
    "errorCode": "urn:oracle:cloud:errorcode:messaging:error key",
    "errorMessage": "error message"
    "exceptionClass": " full Java class name of the exception",
    "exceptionMessage": "message embedded in the exception",
}
```

The XML format would be as follows:

```
<error>
    <httpStatusCode>status code</httpStatusCode>
    <httpMessage>status line reason phrase for the code</httpMessage>
    <errorCode>urn:oracle:cloud:errorcode:messaging:error key</errorCode>
    <errorMessage>error message</errorMessage>
    <exceptionClass>full Java class name of the exception</exceptionClass>
    <exceptionMessage>message embedded in the exception</exceptionMessage>
</error>
```

In this document, the error that is returned by a given method, interface, or the API as a whole is specified by giving the **error key** for the error. The error key is the last component of the URN error code; the error code is the value of the `errorCode` field of the JSON format or the content of the `<errorCode>` element of the XML format. The error key determines the HTTP status code, which in turn determines the HTTP message, and determines the format of the error message. The actual error message may contain dynamically determined data.

The `exceptionClass` and `exceptionMessage` fields of the JSON format or the content of the `<exceptionClass>` and `<exceptionMessage>` elements of the XML format may or may not be displayed. These two fields or elements are displayed only if the error was caused by a Java exception thrown on the server side. They indicate the class and message of a Java exception that caused the error response.

Error responses may be specified for the entire REST API (in REST API HTTP Status Codes and Error Messages Reference), for a set of methods in the API, or for a

specific method (for example, the Create a Session method). Each error specification gives the error key, and gives any explanation of the circumstances under which the error occurs that is more specific than that included with the error key listing in Error Keys, Status Codes and Error Messages. For example, in the Create a Session method, there is a specification of the error response, as follows:

`sessionAlreadyExists`

A session with the given name already exists.

This means that if, for example, a request were made to create a session named `s`, and a session with that name already existed, then the HTTP response would have status code 409 and the following response body in JSON:

```
{
    "httpStatusCode": 409,
    "httpMessage": "Conflict",
    "errorCode": "urn:oracle:cloud:errorcode:messaging:sessionAlreadyExists",
    "errorMessage": "Session 's' already exists"
}
```

The response body in XML is as follows:

```
<error>
  <httpStatusCode>409</httpStatusCode>
  <httpMessage>Conflict</httpMessage>
  <errorCode>urn:oracle:cloud:errorcode:messaging:sessionAlreadyExists</errorCode>
  <errorMessage>Session 's' already exists</errorMessage>
</error>
```

Localization of Error Messages

When Oracle Messaging Cloud Service encounters an error, a descriptive error message is returned in the body of the HTTP response. Oracle Messaging Cloud Service supports localized error message, which can be used to provide a preferred language for the error message descriptions. Note that only the content which is displayed in the `errorMessage` field (in JSON)/ the `<errorMessage></errorMessage>` element (in XML) is localized. Other content of the error response such as the exception messages may not be localized.

To specify your preferred language for the error message descriptions, you should use the `Accept-Language` header in the HTTP request.

The following languages are supported:

Language	Code
US English	`en-US`
German	`de`
Spanish	`es`
French	`fr`

Language	Code
Italian	`it`
Japanese	`ja`
Korean	`ko`
Brazilian Portuguese	`pt-BR`
Chinese	`zh-CN`
Taiwan Chinese	`zh-TW`

The default language for error message descriptions is US English. If you use the `Accept-Language` header and specify a language which is currently not supported, then the error message descriptions will be returned in US English by default.

When a client's `Accept-Language` header specifies only a language, Oracle Messaging Cloud Service will localize to a supported locale that includes that language if one exists. For example, if the `Accept-Language` header is set to `en`, the error message descriptions will be returned in `en-US`; if it is set to `pt`, the error message descriptions will be returned in `pt-BR`; if it is set to `zh`, the error message descriptions could be returned either in`zh-CN` or `zh-TW`.

Understanding Anti-CSRF Measures

To prevent Cross-Site Request Forgery (CSRF) attacks, Oracle Messaging Cloud Service generates pseudorandom anti-CSRF tokens for each messaging context.

For requests with methods, the REST API provides the capability to enable the generation and use of an anti-CSRF **ID token** according to the following rules:

- On a client's initial HTTP request to the REST API, if the ID token is not explicitly disabled by the request, then a pseudorandom ID token is generated by the service and returned to the client as the value of the `X-OC-ID-TOKEN` header. The initial request may disable the ID token by including the header `X-OC-ID-TOKEN-STATUS: disabled`.

  If this header is in the initial request, no ID token is generated or stored or returned in the response.

- Whenever a request is received when ID tokens are enabled and an ID token value has been previously generated, the previously generated ID token is compared with the value of the `X-OC-ID-TOKEN` header in the request. If there is no such request header, or if its value does not match the previously generated ID token, then the service sends a response with status code 400 and the error message "Missing or incorrect X-OC-ID-TOKEN" (as listed in REST API HTTP Status Codes and Error Messages Reference), and no further processing of the request is performed.

- If a request is received when ID tokens are enabled and an ID token value has been previously generated, and the request has the correct `X-OC-ID-TOKEN` header value, and the request contains the header `X-OC-ID-TOKEN-STATUS: disabled`, the previously generated ID token is discarded, and the ID token is disabled on subsequent requests.
- If a request is received when the ID token is disabled, and the request contains the header `X-OC-ID-TOKEN-STATUS: enabled` then a pseudorandom ID token is generated by the service and returned to the client as the value of the `X-OC-ID-TOKEN` header, and the ID token is enabled for subsequent requests.

Take note of the following additional rules:

- An `X-OC-ID-TOKEN-STATUS` header that enables the ID token when the ID token is already enabled, or that disables the ID token when it is already disabled, has no effect on the ID token's enabling or disabling.
- The ID token is returned only in response to the request (initial or subsequent) that enables the ID token for the associated messaging context. The client must save the value to be able to submit it in subsequent requests, or the client will be unable to re-use the associated messaging context and any of its encapsulated ephemeral objects, like connections, sessions, producers, and consumers.
- If the ID token is currently enabled, then disabling the token requires that the current ID token be supplied by the request.
- Enabling and disabling the ID token is done "out of band" from the other processing of a given request, prior to the normal processing of the request. Even if a request's normal processing fails (for example, an attempt is made to create a **queue** that already exists, or delete a listener that doesn't exist), the enabling or disabling of the ID token will occur according to the rules described earlier. In particular, even if a request that enables the ID token generates an error response with status code 400 or greater, the response will still have an `X-OC-ID-TOKEN` header containing the new ID token.
- Applications which use the REST API that the developer feels do not need anti-CSRF measures can send the disabling header in the client's initial request to the service and subsequently take no account of anti-CSRF measures. If there is a chance that a connection might expire between accesses by such a client, the client may either check for the generation of an anti-CSRF token and disable it via a subsequent request or send the disabling header on every request.

HTTP Header for Messaging Service Version

The Oracle Messaging Cloud Service provides an HTTP response header `X-OC-MESSAGING-SERVER-VERSION` that specifies the version of the Oracle Messaging Cloud Service being run by the server to which the client is connected.

The value of the header `X-OC-MESSAGING-SERVER-VERSION` is a dot-delimited list of integers, for example, 15.2.5.0.0.

HTTP Header for Messaging Context ID

The Oracle Messaging Cloud Service provides an HTTP response header `X-OC-MESSAGING-CONTEXT-ID` whose value is the ID of the messaging context associated with the response.

The HTTP header `X-OC-MESSAGING-CONTEXT-ID` indicates the internal, pseudorandom ID for the messaging context associated with the response. This header is only returned in the response to the request that creates the messaging context.

Resource Management API

The Resource Management API provides functionality to create and manage destinations, and message push listeners.

Topics:

Creating and Managing Destinations

This section provides information about creating and managing destinations in Oracle Messaging Cloud Service.

Topics:

Create a Destination

This section provides information about creating a destination.

Names of queues or topics must always consist solely of letters of the Roman alphabet (a through z or A through Z), decimal digits (0 through 9), and underscores ('_'). No other characters are allowed.

Method: `PUT`

Path:

- To create a queue, the path is `/queues/queueName`
- To create a topic, the path is `/topics/topicName`

Scope: Service Instance

Authorization: Messaging Administrator

Result: Create a queue or topic with the supplied name.

Error Responses:

Error Message	Description
`destinationAlreadyExists`	A destination of the specified type with the specified name already exists.

Error Message	Description
`maxQueuesReached`	A request was made to create a queue, but the number of queues is already at the maximum for the service.
`maxTopicsReached`	A request was made to create a topic, but the number of topics is already at the maximum for the service.

See Create a Queue and Create a Topic for example HTTP requests which create destinations.

List Destinations

This section provides information about listing destinations.

Method: `GET`

Path:

- To list queues, the path is `/queues`
- To list topics, the path is `/topics`

Scope: Service Instance

Authorization: Messaging Administrator

Result: Returns a listing of all queues or topics in the service instance.

Response Body: XML or JSON. The XML format has one element for each destination of the appropriate type.

For queues, the format in JSON is as follows:

```
{
    "items": [
        {
            "name": "name",
            "status": "status",
            "canonicalLink": "relative path to queue"
        },
        ...
    ],
    "canonicalLink": "relative path to queue list"
}
```

The format for queues in XML is as follows:

```
<queues>
    <items>
        <name>name</name>
        <status>status</status>
        <canonicalLink>relative path to queue</canonicalLink>
    </items>
    ...
    <canonicalLink>relative path to list of queues</canonicalLink>
</queues>
```

For topics, the format in JSON is as follows:

```
{
    "items": [
        {
            "name": "name",
            "status": "status",
            "canonicalLink": "relative path to topic"
        },
        ...
    ],
    "canonicalLink": "relative path to topic list"
}
```

The JSON format for listing topics is the same as that for listing queues except that the `canonicalLink` properties are different.

The format for topics in XML is as follows:

```
<topics>
    <items>
        <name>name</name>
        <status>status</status>
        <canonicalLink>relative path to topic</canonicalLink>
    </items>
    ...
    <canonicalLink>relative path to list of topics</canonicalLink>
</topics>
```

In all cases, *name* is the name of the destination of the given type and *status* is `MARK_FOR_DELETION` if the destination still exists in the JMS broker but has been marked for deletion (and thus is in the process of being deleted), and is `PROVISIONED` otherwise. If a destination has been marked for deletion, then it cannot be used, but it is also not possible to create a destination of a given type with the marked destination's name until the marked destination has been deleted (and thus does not appear in the output of this method).

Retrieve Destination Properties

This section provides information about retrieving destination properties.

Method: `GET`

Path:

- To get the properties of a queue, the path is `/queues/queueName`
- To get the properties of a topic, the path is `/topics/topicName`

Authorization: Messaging Administrator

Scope: Service Instance

Request Parameter:

Parameter	Description
`backlog`	The value must be `true` or `false`. The default value is `false`. A destination's backlog size is the number of messages currently stored for the destination. Every destination in Oracle Messaging Cloud Service has a maximum backlog size of 100,000 messages. Attempts to send messages to a destination with a backlog of 100,000 messages will fail. The value reported for a queue's backlog size may be up to 30 seconds old. Note that the backlog feature is currently available only for queues.

Result: Returns the properties of the destination with the specified name. When retrieving a queue's properties, if `backlog` is set to `true`, the HTTP response body includes a `backlogStats` element for XML and JSON response types, respectively.

Response Body: For queues, the format in JSON is as follows:

```
{
    "name": "name",
    "status": "status",
    "canonicalLink": "relative path to queue"
    "backlogStats" : {
    "current" : "size of the destination's backlog"
    }
}
```

The format in XML is as follows:

```
<queue>
    <name>name</name>
    <status>status</status>
    <canonicalLink>relative path to queue</canonicalLink>
    <backlogStats>
      <current>size of the destination's backlog</current>
    </backlogStats>
</queue>
```

For topics, the JSON format is the same, but with the appropriate value of the `canonicalLink` property. The format in XML is as follows:

```
<topic>
    <name>name</name>
    <status>status</status>
    <canonicalLink>relative path to topic</canonicalLink>
</topic>
```

The value and interpretation of *status* are as in the **List Destinations** method.

Error Response:

Error Message	Description
`destinationNotFound`	The destination whose properties are requested does not exist.

Remove a Destination

This section provides information about removing destinations.

Deleting a destination is a non-blocking operation. For more information, refer to About Destination Deletion.

Method: `DELETE`

Path:

- To delete a queue, the path is `/queues/queueName`
- To delete a topic, the path is `/topics/topicName`

Scope: Service Instance

Authorization: Messaging Administrator

Result: Deletes the destination with the given name.

Error Response:

Error Message	Description
`destinationNotFound`	The destination whose deletion is requested does not exist.

Creating and Managing Message Push Listeners

This section provides information about creating and managing message push listeners in Oracle Messaging Cloud Service.

Topics:

- Create a Listener
- Delete a Listener
- List Listeners
- Retrieve Listener Properties

Create a Listener

This topic provides information about creating a listener. The name of a listener must always consist solely of letters of the Roman alphabet (a through z or A through Z), decimal digits (0 through 9), and underscores ('_'). No other characters are allowed.

Method: `PUT`

Path: `/listeners/listenerName`

Scope: Service Instance

Authorization: Messaging Administrator or Messaging Worker

Request Parameter:

Parameter	Description
`verificationToken`	A token used in message push listener verification. The value of the `verificationToken` parameter is a string which is passed along with the message push listener verification request as the `X-OC-MPL-VERIFICATION` header. **Note:** The `X-OC-MPL-VERIFICATION` header is a header that the service sends to the endpoint.

Request Body: An XML document that specifies the following:

- The URI to which to push messages, with any associated parameters to the push.
- The selector, if any, the listener should apply to filter the messages it receives.
- The existing durable subscription, if any, on which the listener should listen for messages.
- The policy the listener should follow if an attempt to push a message fails.

> **Note:**
>
> The XML document should not contain a `DOCTYPE` declaration. If a `DOCTYPE` declaration is included in the XML document, a `500 operationFailed` response is returned. This is done to prevent certain security and Denial of Service (DoS) attacks.

See Create a Message Push Listener for an example HTTP request which creates a message push listener.

The root of the document is `<listener>`. The root must contain a single `<version>` whose content is the version of the listener XML. For the current release, the version must be 1.0. The root must also contain exactly one `<name>` element whose content is *listener name*. The root may also contain a single `<source>` element whose content specifies the queue or topic on which the listener listens for messages. If present, its content must be one each of the following elements:

- **`<type>`**

  The content must be either `queue` or `topic`.

- **`<name>`**

  The name of the queue or topic.

If the `<source>` element is not present, the `<listener>` element must contain a `<subscription>` element. Note that the `<source>` element implicitly specifies a **non-temporary** queue or topic. Message push listeners may not listen on temporary queues or topics.

The root must contain exactly one <target> element, 0 or 1 <selector> elements, 0 or 1 <subscription> elements, and 0 or 1 <failurePolicy> elements; order of all child elements is irrelevant. If a <subscription> element is present, there can be neither a <source> nor a <selector> element, as both the destination and selector, if any, is determined by the subscription.

The <target> element specifies the URI to which the listener pushes messages; it must contain at most one of each of the following elements:

- **<uri>**

  There must be exactly one such element. The content is the URI to which to push, which must be one of the following types:

  – An HTTP or HTTPS URL

  – A URN of the form urn:oracle:cloud:messaging:queues:*queueName* or urn:oracle:cloud:messaging:topics:*topicName*.

  A URI of the first type indicates that the listener should push to an HTTP or HTTPS endpoint. A URI of the second type indicates that the listener should send the message to a queue or topic; the fifth colon-separated component specifies whether the destination is a queue or topic, and the last colon-separated component specifies the name of the queue or topic. It is expected that targets of the latter form will usually occur as targets to which to push a message after an HTTP or HTTPS push has failed, but this is not required.

  Message push listeners will not follow HTTP redirects. An HTTP redirect response from a user-specified URL will be treated as an error as described in the explanation of the <failurePolicy> element below.

- **<method>**

  The content is the HTTP method to use for the push if the <uri> contains an HTTP or HTTPS URI. Only the POST and PUT methods will work; the default is POST. This element is optional, and *must* be omitted if the URI is not an HTTP or HTTPS URI.

- **<user>**

  The user to use for HTTP authentication if <uri> contains an HTTP or HTTP URI. This element is optional, and must be omitted if the URI is not an HTTP or HTTPS URI. There is no default value.

- **<password>**

  The password to use for HTTP authentication if <uri> contains an HTTP or HTTP URI. This element must be present if and only if the <user> element is present. There is no default value.

The <target> element may contain an arbitrary number (including 0) of <header> elements; these elements are ignored if the target is not an HTTP or HTTPS target. Each <header> element must contain exactly one <name> and <value> element. The content specifies a value for the header with name given by the content of the <name> element and value given by the content of the <value> element. Multiple <header> elements with the same <name> content are allowed; those after the first with a given name add headers rather than overwriting the earlier headers. The <name> element's content may not begin with "X-OC-" (case-insensitive). The <name> element's content must be a legal HTTP header name and the <value> element's content must be a legal HTTP header value. If the <name> element contains Content-Type (case-insensitive), the value is ignored, unless the message being pushed has type HTTP and does not specify a Content-Type. Otherwise, the Content-Type header of a message

push request is determined by the message. If the <name> element contains Content-Language (case-insensitive), the value assigned by the listener is overridden if the message being pushed has type HTTP and specifies the Content-Language.

The content of the <selector> element is used as the JMS selector for the listener. For the syntax of selectors, see the **Message Selectors** section of the Java API reference for the javax.jms.Message class.

The <subscription> element must contain one of each of the following elements:

- **<clientId>**

  The content is the client ID of the durable subscription to use.

- **<name>**

  The content is the name of the subscription.

If present, this element specifies an existing durable subscription whose messages the listener should receive and push.

> **Note:**
>
> If there is a listener listening on a durable subscription, no other client or listener is able to use the same client ID, even with a different subscription name. Thus, any client ID used with a listener should be dedicated to that listener, and only one durable subscription can be used for that client ID.

The <failurePolicy> element, if present, specifies what the listener does if its attempt to push a message to a URI fails. If no <failurePolicy> element is present, a message whose push fails is discarded. If present, <failurePolicy> must contain 0 or more <failure> elements. Each <failure> element specifies the kind of push failures to which it applies, and what to do in case of the given failure. If a push fails, the <failure> element that is first in document order that applies to the failure is used to determine what action is taken. If no <failure> element applies (in particular, if the <failurePolicy> element contains no <failure> elements), the message is discarded.

A <failure> element must contain exactly one <cond>. Its content must be one of the following:

- connection: this value indicates that the <failure> element applies if the target URI was an HTTP or HTTPS URL and the listener was unable to establish a connection to the specified endpoint, or the host and port specified by the URL was not an HTTP endpoint.
- responseCode:*n*-*m*, in which *n* and *m* are positive integers: a value of this form indicates that the <failure> element applies if the target URI was an HTTP or HTTPS URL, a connection was successfully made to the specified HTTP endpoint, and the response code is greater than or equal to *n* and less than or equal to *m*. Note that, regardless of the values of *n* and *m*, the <failure> element will only apply if the response status is 300 or greater. HTTP requests to push messages will not follow redirects, and will generate a responseCode error.
- responseCode: *n*: this is equivalent to responseCode:*n*-*n*.

- `send`: this value indicates that the <failure> element applies if the target URI was a URN specifying a queue or topic, and the attempt to send failed.
- `maxHops`: this value indicates that the <failure> element applies if the target URI was a URN specifying a queue or topic, and the message had already been pushed by a listener to a queue or topic the maximum number of times allowed (currently 8).
- `any`: this value indicates that the <failure> element applies to any of the above failure types.

A <failure> element may contain 0 or 1 <wait> elements.

The <wait> element must contain exactly one <time> element. The content of the <time> element must be a non-negative integer, and is interpreted as a number of milliseconds to wait before attempts to push the message again. The allowed content of a <wait> element, other than <time>, is the same as that of <listener>, except that <target> is not required. If there is no <target> element, it is equivalent to the <target> element being the same as that of the nearest enclosing <listener> or <wait> element. After waiting the specified wait time, the listener then behaves as if it were a listener whose content is that of the <wait>; it attempts to push the message to the URI specified by its <target>, and, if that fails, applies the failure policy specified by its <failurePolicy> child, if any. Note that the <failurePolicy> of the parent is not inherited; successive push attempts use the <failurePolicy> at the appropriate level. Thus, a listener will never retry forever; if all push attempts for a given message fail, the message will eventually be discarded.

If a message received by a listener is pushed to a queue or topic, the type and content of the message is preserved. All properties will also be preserved, with the following exceptions:

- The `X_OC_PushCount` property will either be set to 1, if it is not present, is not an integer, or is a negative integer; otherwise, it will be incremented by 1. This property is used by listeners to track how many times the message has been re-sent to a queue or topic by a listener. If the incoming message's `X_OC_PushCount` is 8 or greater, and the listener is directed to push it to a queue or topic, it will instead cause a failure that can be handled by a <failure> element containing a <cond>maxHops</cond> element.
- The ID of the received message is set as the `X_OC_PastJMSMessageID`*N* property of the outgoing message, where *N* is 1 less than the value of the `X_OC_PushCount` property on the outgoing message.
- The destination header of the received message, expressed as a String of the form `/queues/`*queueName* or `/topics/`*topicName*, is set as the `X_OC_PastJMSDestination`*N* property of the outgoing message (*N* as above).
- The value of the timestamp header of the received message is set as the `X_OC_PastJMSTimestamp`*N* property of the outgoing message (*N* as above).
- The value of the "redelivered" header of the received message is set as the `X_OC_PastJMSRedelivered`*N* property of the outgoing message (*N* as above).

The correlation ID, reply-to, and delivery mode of the received message is set as the corresponding headers of the outgoing message. The message is sent with a time-to-live that will make it expire at roughly the same time as the received message.

For example:

```
<listener>
    <version>1.0</version>
    <name>myListener</name>
    <source>
        <type>queue</type>
        <name>myQueue</name>
    </source>
    <target>
        <uri>http://myHost/receiver</uri>
        <method>PUT</method>
        <user>u</user>
        <password>guest</password>
        <header>
            <name>X-PIN</name>
            <value>123456</value>
        </header>
    </target>
    <selector>(urgency = 'high') AND (count &lt; 5)</selector>
    <failurePolicy>
        <failure>
            <cond>connection</cond>
            <wait>
                <time>5000</time>
                <failurePolicy>
                    <failure>
                        <cond>any</cond>
                        <wait>
                            <time>0</time>
                            <target>
                                <uri>http://myBackupHost/deadLetter</uri>
                            </target>
                        </wait>
                    </failure>
                </failurePolicy>
            </wait>
        </failure>
        <failure>
            <cond>responseCode:500-599</cond>
            <wait>
                <time>0</time>
                <target>
                    <uri>http://myBackupHost/deadLetter</uri>
                </target>
            </wait>
        </failure>
        <failure>
            <cond>responseCode:401-499</cond>
            <wait>
                <time>0</time>
                <target>
                    <uri>urn:oracle:cloud:messaging:queues:unpushed</uri>
                </target>
                <failurePolicy>
                    <failure>
                        <cond>send</cond>
                        <wait>
                            <time>0</time>
                            <target>
                                <uri>urn:oracle:cloud:messaging:topics:backup</uri>
                            </target>
                        </wait>
```

```
                    </failure>
                </failurePolicy>
            </wait>
        </failure>
    </failurePolicy>
</listener>
```

This listener listens for messages sent to the queue `myQueue` whose `urgency` property has the value `high` and whose `count` property has value less than 5. (The `urgency` and `count` properties of a message sent via the REST API, if present, would be set to the value of the `X-OC-`*`type`*`-PROPERTY-urgency` and `X-OC-`*`type`*`-PROPERTY-count` headers, respectively, of the HTTP request that created it, if present. The *`type`* parts of the two property headers might be, for example, `STRING` for `urgency` and `INT` for `count`.) The listener attempts to push messages it receives to `http://myHost/receiver` with method `PUT`, user and password `u` and `guest` respectively, and header `X-PIN: 123456`. If it cannot connect, it waits for 5 seconds and tries again. If that fails in any way, it immediately attempts to push to `http://myBackupHost/deadLetter`, with method `POST`, no HTTP authentication credentials, and no special headers. If the second push fails in any way, the message is discarded. If the initial push fails with response status code in the range 500-599, the listener will immediately attempt a push to `http://myBackupHost/deadLetter` as above. If that fails, the message is discarded. If the initial push fails with response status code in the range 401-499, the listener will immediately attempt a push to a queue with name `unpushed`. If that fails, the listener will attempt to push the message to a topic called `backup`, discarding the message if that fails. If the initial push failed for any other reason (for example, response code of exactly 400), the message is discarded.

The following is an example with a durable subscription:

```
<listener>
    <version>1.0</version>
    <name>myListener</name>
    <target>
        <uri>http://myHost/receiver</uri>
        <method>PUT</method>
        <user>u</user>
        <password>guest</password>
        <header>
            <name>X-PIN</name>
            <value>123456</value>
        </header>
    </target>
    <subscription>
        <clientId>myListenerID</clientId>
        <name>sub</name>
    </subscription>
    <failurePolicy>
        <failure>
            <cond>connection</cond>
            <wait>
                <time>5000</time>
                <failurePolicy>
                    <failure>
                        <cond>any</cond>
                        <wait>
                            <time>0</time>
                            <target>
                                <uri>http://myBackupHost/deadLetter</uri>
```

```
                        </target>
                    </wait>
                </failure>
            </failurePolicy>
        </wait>
    </failure>
    <failure>
        <cond>responseCode:500-599</cond>
        <wait>
            <time>0</time>
            <target>
                <uri>http://myBackupHost/deadLetter</uri>
            </target>
        </wait>
    </failure>
    <failure>
        <cond>responseCode:401-499</cond>
        <wait>
            <time>0</time>
            <target>
                <uri>urn:oracle:cloud:messaging:queues:unpushed</uri>
            </target>
            <failurePolicy>
                <failure>
                    <cond>send</cond>
                    <wait>
                        <time>0</time>
                        <target>
                            <uri>urn:oracle:cloud:messaging:topics:backup</uri>
                        </target>
                    </wait>
                </failure>
            </failurePolicy>
        </wait>
    </failure>
  </failurePolicy>
</listener>
```

This listener is the same as the previous one, except that the topic it listens on and the selector, if any, are taken from an existing durable subscription with client ID `myListenerID` and name `sub`.

Result: Creates a listener with name *`listenerName`*.

Error Responses:

Error Message	Description
`nonexistentNamespace`	The specified namespace does not exist.
`nonexistentNamespaceComponents`	The namespace specified for the request does not exist.
`nonexistentNamespaceUnknown`	The namespace specified in the request URL does not exist.

Error Message	Description
`malformedListener`	One of the following occurred: • Neither a source nor a durable subscription was specified in the XML. • The listener XML did not conform to the syntax described in Create a Listener.
`destinationParameterNotFound`	The source specified on which the listener should listen does not exist.
`subscriptionParameterNotFound`	The request body specified a subscription from which to receive, and the subscription does not exist.
`subscriptionNotFoundNoInfo`	The request body specified a subscription from which to receive, and the subscription does not exist.
`listenerAlreadyExists`	A listener with the given name already exists.
`clientIdFailure`	The listener XML specifies a listener on a durable subscription whose client ID is invalid or in use by some client or other listener.

Error Message	Description
`messagePushListenerVerificationBadResponse`	An HTTP or HTTPS endpoint responded to a verification request with a response body that did not match the challenge token sent by Oracle Messaging Cloud Service.
`messagePushListenerVerificationConnectionFailed`	Oracle Messaging Cloud Service was unable to connect to an HTTP or HTTPS endpoint to send a verification request.
`messagePushListenerVerificationErrorResponse`	An HTTP or HTTPS endpoint responded to a verification request with a status code other than 200.
`messagePushListenerVerificationException`	An exception occurred in attempting to read the response to a verification request.
`messagePushListenerVerificationNoToken`	The listener XML specifies at least one HTTP or HTTPS URL to which to push messages, but no `verificationToken` was supplied.

Error Message	Description
`messagePushListenerVerificationRedirectionDisableFailed`	A failed attempt was made to disable HTTP redirects for the message push listener verification request.
`operationFailed`	A low-level exception was thrown while checking for a pre-existing listener, or while creating the listener.

Delete a Listener

This topic provides information about deleting a listener.

Method: `DELETE`

Path: `/listeners/`*`listenerName`*

Scope: Service Instance

Authorization: Messaging Administrator or Messaging Worker

Result: Deletes listener with name *`listenerName`*

Error Responses:

Error Message	Description
`nonexistentNamespace`	The specified namespace does not exist.
`nonexistentNamespaceComponents`	The namespace specified for the request does not exist.
`nonexistentNamespaceUnknown`	The namespace specified in the request URL does not exist.
`listenerNotFound`	The listener to delete did not exist.
`operationFailed`	A low-level exception was thrown while deleting the listener.

List Listeners

This section provides information about listing listeners.

Method: `GET`

Path: `/listeners`

Scope: Service Instance

Authorization: Messaging Administrator or Messaging Worker

Request Parameter:

Parameter	Description
`destination`	Optional. If present, value must be `/queues/`*`queueName`* or `/topics/`*`topicName`*. Specifies a destination whose listeners should be listed.

Result: Returns a list of all listeners created for the namespace if there is no `destination` parameter, and a list of all listeners on the destination specified by `destination` otherwise.

Response Body: XML format with one `<items>` element for each listener and a `<canonicalLink>` element.

```
<listeners>
    <items>
        <name>listener name</name>
        <canonicalLink>relative path to access /listeners/listener name</
canonicalLink>
    </items>
    ....
    <canonicalLink>relative path to access /listeners</canonicalLink>
</listeners>
```

Error Responses:

Error Message	Description
`malformedDestination`	The `destination` parameter value did not parse as a specification of a queue or topic.
`nonexistentNamespace`	The specified namespace does not exist.
`nonexistentNamespaceComponents`	The namespace specified for the request does not exist.
`nonexistentNamespaceUnknown`	The namespace specified in the request URL does not exist.
`destinationParameterNotFound`	The `destination` parameter specified a destination that does not exist.
`operationFailed`	An exception was thrown while getting the list of listeners or outputting the XML representation of the listeners.

Retrieve Listener Properties

This section provides information about retrieving listener properties.

Method: `GET`

Path: `/listeners/`*`listenerName`*

Scope: Service Instance

Authorization: Messaging Administrator or Messaging Worker

Result: Returns properties of the listener with name *listenerName*.

Response Body: An XML document in the format used in the request body to create the listener, describing the listener with name *listenerName*.

Error Responses:

Error Message	Description
nonexistentNamespace	The specified namespace does not exist.
nonexistentNamespaceComponents	The namespace specified for the request does not exist.
nonexistentNamespaceUnknown	The namespace specified in the request URL does not exist.
listenerNotFound	The specified listener does not exist.
operationFailed	A low-level exception was thrown while retrieving the listener properties.

Message Transmission API

The Message Transmission API provides an interface for sending messages through producers and receiving messages through consumers, including advanced messaging capabilities such as transactions. The Message Transmission API also provides functionality to create and manage messaging contexts, connections, sessions, durable subscriptions, temporary destinations, and queue browsers.

Topics:

Creating and Managing Messaging Contexts

This section provides information about creating and managing messaging contexts in Oracle Messaging Cloud Service.

Topics:

Create a Messaging Context

A messaging context is a container of ephemeral objects like connections, sessions, producers, consumers, temporary destinations, and queue browsers.

Creation of a messaging context is an implicit operation. It is created by the first access to the Oracle Messaging Cloud Service that passes authentication. At least one messaging context must be created by a client in order to access Oracle Messaging Cloud Service. When a new messaging context is created, the HTTP response includes the header `X-OC-NEW-MESSAGING-CONTEXT: true`. Existing messaging contexts are identified in HTTP requests by `JSESSIONID` cookies. If an HTTP request does not include a `JSESSIONID` cookie, or if the HTTP request includes a `JSESSIONID` cookie for an expired messaging context, then a new messaging context is created. For more information about messaging contexts, see Messaging Context and HTTP Cookies.

Each messaging context created by the REST API has a **maximum inactive interval** (MII) associated with it. A messaging context expires if it is not accessed for a period of time longer than the associated MII.

Get Maximum Inactive Interval (MII)

This section provides information about getting the maximum MII allowed to be set by the service or service instance.

Method: `GET`

Path: `/maxInactiveInterval`

Scope: Messaging Context

Authorization: Messaging Administrator or Messaging Worker

Result: Returns the information of the current MII and the maximum allowed MII.

Response Headers:

- `X-OC-MII:` *`positive integer number of seconds`*

  Indicates the current value of the MII.

- `X-OC-MAX-MII:` *`positive integer number of seconds`*

  Indicates the maximum MII allowed to be set by the service or service instance.

Error Response:

Error Message	Description
`operationFailed`	A low-level exception occurred in attempting to obtain the client ID.

Set Maximum Inactive Interval (MII)

This section provides information about setting the Maximum Inactive Interval (MII).

Method: `POST`

Path: `/maxInactiveInterval`

Scope: Messaging Context

Authorization: Messaging Administrator or Messaging Worker

Request Parameter:

Parameter	Description
`mii`	The value must be a non-negative integer, interpreted as the number of seconds to which the client would like to set the MII.

Result: If the value of `mii` in the request is positive, then the MII is set to that value (or 900 if the value is greater than 900). If it is 0, the messaging context expires immediately, and is deleted.

> **Note:**
>
> A value of 0 does not indicate infinity. Clients may not set messaging contexts to never expire.

Response Headers:

- `X-OC-MII:` *`positive integer number of seconds`*

  If the requested MII was positive, this is the new value of the maximum inactive interval, which may be less than the value submitted. The maximum is 900 seconds.

- `X-OC-MAX-MII:` *`positive integer number of seconds`*

  Indicates the maximum MII allowed to be set by the service or service instance. The maximum MII allowed is 900 seconds.

Error Responses:

Error Message	Description
`badParameter`	The value of the `mii` parameter did not parse as an integer or was negative.
`operationFailed`	A low-level exception occurred.

Creating and Managing Connections

A client can associate a **client ID** with a connection. The client ID is a string that, along with a subscription name, identifies a durable subscription to a **topic**. When creating a durable consumer on a topic, the client ID set on the connection is used with the subscription name supplied in the consumer creation method to identify the durable subscription.

An attempt to set the client ID on a connection that generates an error with `InvalidClientIDException` as its `exceptionClass` causes the connection to become unusable. Any connection on which such an error response is received should be closed.

Topics:

Create a Connection

This topic provides information about creating a connection.

Method: `PUT`

Path: `/connections/`*`connectionName`*

Scope: Messaging Context

Authorization: Messaging Administrator or Messaging Worker

Request Parameters:

Parameter	Description
`clientId`	Optional. The value is the client ID that the client would like to set on the connection.
`action`	Optional. To start a connection, value of the `action` parameter must be `start`.

Result: Creates a connection with the name *`connectionName`*. A connection is stopped when it is created (unless the `start` action is specified in the `PUT` request to create the connection).

Response Body:

The response body in XML for creating a connection is as follows:

```
<connection>
    <metadata>
        <JMSXPropertyNames>
            <items>JMSXDeliveryCount</items>
            <items>JMSXGroupID</items>
            <items>JMSXGroupSeq</items>
        </JMSXPropertyNames>
    </metadata>
  <canonicalLink>relative path to newly created connection</canonicalLink>
</connection>
```

Each supported JMSX property name is listed in an `<items>` element.

The response body in JSON for creating a connection is as follows:

```
{
   "metadata":
      {
        "JMSXPropertyNames":
            [
              "JMSXDeliveryCount",
              "JMSXGroupID",
```

```
            "JMSXGroupSeq"
          ]
      }
  "canonicalLink": "relative path to newly created connection"
}
```

For more information, refer to the following links:

- X-OC-DELIVERY-COUNT
- X-OC-GROUP-ID
- X-OC-GROUP-SEQ

Error Responses:

Error Message	Description
`clientIdFailure`	The value that is provided for the connection's client ID is rejected by the JMS provider.
`connectionAlreadyExists`	A connection with the specified name already exists.
`maxConnectionCountUnavailable`	An internal error has occurred in determining the number of connections that a service instance is allowed.
`maxLocalConnectionsReached`	The service instance has exceeded the number of connections it can create on a single virtual machine in the cloud. This usually means that the service instance has reached, or even gone beyond, the maximum number of allowed connections.
`operationFailed`	A low-level exception occurred in attempting to obtain the client ID.

Update Connection Properties

This section provides information about updating connection properties.

Method: `POST`

Path: `/connections/`*`connectionName`*

Scope: Messaging Context

Authorization: Messaging Administrator or Messaging Worker

Request Parameters:

Parameter	Description
`clientId`	Optional. The value is the client ID that the client would like to set on the connection.
`action`	Optional. To start a connection, value of the `action` parameter must be `start`. To stop a connection, the value of the action parameter must be `stop`.

Result: If the `clientId` parameter is present, the client ID of the connection is set to the specified value, provided that none of the following blocking conditions are true:

- The client ID has already been set to a value for the connection.
- The connection was started when it was created.
- The connection has been used for any operation.
- The client ID is in use by some other application.

If any of the blocking conditions is true, a 400 error response is generated.

Note:

If a REST API client sets a client ID on a connection, as long as that connection exists, no other connection will be able to have the same client ID set on it. As a consequence, no other connection will be able to create, delete, or consume from any durable subscription with that client ID. For example, consider the following situations:

- A client fails, or otherwise loses the value of the `JSESSIONID` HTTP cookie that identifies the messaging context that contains a connection with a client ID.
- A client exits without either deleting a connection with a client ID or expiring the messaging context containing it by setting the messaging context's MII to 0.

In either of these cases, no client will be able to create a new connection with the given client ID until the messaging context expires. To avoid these situations, a REST API client that sets a client ID on a connection should do the following:

- Set the MII of any messaging context containing a connection with a client ID to the smallest feasible value, so that, if the client fails or loses the value of the `JSESSIONID` HTTP cookie, the time before the client ID can be used again is minimal.
- Delete connections with client IDs before exiting, either by explicitly performing a `DELETE` on the connection resource or by setting the MII of the messaging context containing such connections to 0, which will close all connections in the context.

Error Responses:

Error Message	Description
`clientIdFailure`	The value that is provided for the connection's client ID is rejected by the JMS provider. The value of the clientId may be in use by another application, including message push listeners.
`connectionNotFound`	The requested connection does not exist.
`clientIdUnsettable`	An attempt was made to set the client ID on a connection on which it cannot be set, either because it has already been set or because an operation has been performed (for example, starting the connection, or creating a session) after which the client ID can no longer be set.
`operationFailed`	A low-level exception occurred in attempting to obtain the client ID, or its value has been rejected.

Delete a Connection

This topic provides information about deleting a connection.

Method: `DELETE`

Path: `/connections/connectionName`

Scope: Messaging Context

Authorization: Messaging Administrator or Messaging Worker

Result: Closes and deletes the connection. This also closes and deletes all sessions, producers, consumers, temporary destinations, and queue browsers created with the connection.

Error Response:

Error Message	Description
`connectionNotFound`	The requested connection does not exist.

Creating and Managing Sessions

Before messages can be sent or received through the Message Transmission API, a connection and a session must be created. A session provides a behavioral context that defines what it means for messages to be sent and received between clients and Oracle Messaging Cloud Service.

Topics:

Create a Session

This topic provides information about creating a session.

Method: `PUT`

Path: `/sessions/sessionName`

Scope: Messaging Context

Authorization: Messaging Administrator or Messaging Worker

Request Parameters:

Parameter	Description
`connection`	Specify the name of the connection used to create the session.

Parameter	Description
`transacted`	Optional. If present, must have value `true` or `false`; default is `false`. Determines whether the session is transacted.
`ackMode`	Optional. If present, must have value `auto`, `client`, or `dups_ok`; default is `auto`. Determines how messages received through the session are acknowledged. This parameter is irrelevant if the `transacted` parameter is `true`. The `auto` value means that received messages are automatically acknowledged as they are received. The `client` value means that messages must be explicitly acknowledged by the client. The `dups_ok` value means that messages are automatically acknowledged, but "lazily", so their acknowledgement may be delayed.

Result: Creates a session.

Error Responses:

Error Message	Description
`missingParameter`	The `connection` parameter was not supplied.
`badParameter`	The value of the `transacted` parameter had a value other than `true` or `false`, or the value of the `ackMode` parameter had a value other than the allowed ones.
`sessionAlreadyExists`	A session with the given name already exists.
`connectionParameterNotFound`	No connection exists with name which is specified for the `connection` parameter.
`operationFailed`	A low-level exception was thrown in creating the session.

Acknowledge, Commit, Rollback, or Recover a Session

This topic provides information about acknowledging, committing, doing a rollback, or recovering a session.

Method: `POST`

Path: `/sessions/`*`sessionName`*`/state`

Scope: Messaging Context

Authorization: Messaging Administrator or Messaging Worker

Request Parameter:

Parameter	Description
`action`	Required. Must have value `acknowledge`, `commit`, `rollback`, or `recover`. Value determines whether the operation is to acknowledge any unacknowledged messages received through the session, commit any uncommitted sends and receives, roll back any uncommitted sends and receives, or recover any unacknowledged messages.

Result: Performs the specified operation. If the action is `acknowledge` or `recover`, and the session is set to acknowledge messages automatically (either with acknowledgement mode `auto` or `dups_ok`), this is a no-op. If the action is `commit` or `rollback`, and the session is not transacted, this is a no-op. If the action is `recover` and the session is transacted, a 500 error response is generated.

See Process Messages using a Transaction for an example HTTP request/response sequence.

Error Responses:

Error Message	Description
`missingParameter`	The `action` parameter was not supplied.
`badParameter`	The value of the action parameter was not `acknowledge`, `commit`, `rollback`, or `recover`.
`sessionNotFound`	There is no session with the specified name.
`operationFailed`	A low-level exception was thrown while attempting to carry out the specified action on the specified session.

Delete a Durable Subscription

This topic provides information about deleting a durable subscription.

Method: `DELETE`

Path: `/sessions/`*`sessionName`*`/subscriptions/`*`subscriptionName`*

Scope: Messaging Context

Authorization: Messaging Administrator or Messaging Worker

Result: Deletes the subscription whose client ID is that set on the connection and whose name is that given on the path.

Error Responses:

Error Message	Description
`sessionNotFound`	Session specified on the path does not exist.
`subscriptionInUse`	The subscription has a consumer on it.
`subscriptionNotFound`	Subscription specified on the path does not exist.
`operationFailed`	A low-level exception was thrown while attempting to delete the subscription.

Close and Delete a Session

This topic provides information about closing and deleting a session.

Method: `DELETE`

Path: `/sessions/`*`sessionName`*

Scope: Messaging Context

Authorization: Messaging Administrator or Messaging Worker

Result: Session with name *sessionName* is closed and deleted. All producers, consumers, and queue browsers associated with the session are implicitly closed and deleted.

Error Responses:

Error Message	Description
`sessionNotFound`	There is no session with the specified name.
`operationFailed`	A low-level exception was thrown while attempting to close and delete the specified session.

Sending Messages

This section provides information about sending messages using the REST API.

Topics:

- Create a Producer
- Set Properties of a Producer
- Close and Delete a Producer
- Send a Message via a Producer

See Send a Message to a Topic for example HTTP requests which send a message to a destination.

Create a Producer

This topic provides information about creating a producer.

Method: `PUT`

Path: `/producers/`*`producerName`*

Scope: Messaging Context

Authorization: Messaging Administrator or Messaging Worker

Request Parameters:

Parameter	Description
`session`	Required. The value is the name of the session in which to create the producer.

Parameter	Description
`destination`	Optional. If present, the value must have one of the following forms: • `/queues/`*`queueName`* • `/topics/`*`topicName`* • `/temporaryQueues/`*`queueName`* • `/temporaryTopics/`*`topicName`* Specifies the default destination of messages sent via the producer. If this parameter is omitted, then the destination to which a message is sent must be specified as a parameter in each send operation performed via this producer.
`messageIdEnabled`	Optional. If present, the value must be `true` or `false`; default is `true`. Determines whether the producer generates IDs for messages sent through it. **Note:** The value of this parameter is a hint to the service, which may disregard the value.
`deliveryMode`	Optional. If present, the value must be `persistent` or `non_persistent`; default is `persistent`. Determines whether messages produced are persistent.
`ttl`	Optional. If present, the value must be a strictly positive long integer or the value `maximum`; default is `maximum`. Determines the time in milliseconds between when a message is dispatched and when the JMS broker may delete it if not yet delivered. The value `maximum` indicates that the maximum time-to-live allowed by the service (the number of milliseconds in 2 weeks) should be used.

Result: Creates a producer.

Error Responses:

Error Message	Description
`missingParameter`	The `session` parameter value was not supplied.
`badParameter`	One of the following occurred: • The `destination` parameter value did not parse as a specification of a queue or topic. • The `messageIdEnabled` parameter had a value other than `true` or `false`. • The value of the `deliveryMode` parameter did not parse as a delivery mode. • The value of the `ttl` parameter was not a valid time-to-live.
`sessionParameterNotFound`	Session by which to create the producer does not exist.

Error Message	Description
`destinationParameterNotFound`	The default destination specified for the producer does not exist.
`producerAlreadyExists`	There is already a producer with the specified name.
`timeToLiveTooLarge`	The `ttl` parameter was supplied, but was an integer that is larger than that permitted by the service (the number of milliseconds in 2 weeks).
`operationFailed`	A low-level exception was thrown while attempting to create the producer.

Set Properties of a Producer

This section provides information about setting properties of a producer.

Method: `POST`

Path: `/producers/`*`producerName`*

Scope: Messaging Context

Authorization: Messaging Administrator or Messaging Worker

Request Parameters:

Parameter	Description
`messageIdEnabled`	Optional. If present, the value must be `true` or `false`. Determines whether the producer generates IDs for messages sent through it. **Note:** The value of this parameter is a hint to the service, which may disregard the value.
`deliveryMode`	Optional. If present, the value must be `persistent` or `non_persistent`. Determines whether messages produced are persistent.
`ttl`	Optional. If present, the value must be a strictly positive long integer or the value `maximum`. Determines the time in milliseconds between when a message is dispatched and when the JMS broker may delete it if not yet delivered. The value `maximum` indicates that the maximum time-to-live allowed by the service (the number of milliseconds in 2 weeks) should be used.

Result: Sets the property associated with the parameter to the specified value for all parameters present.

Error Responses:

Error Message	Description
`badParameter`	One of the following occurred: • The `messageIdEnabled` parameter had a value other than `true` or `false`. • The value of the `deliveryMode` parameter did not parse as a delivery mode. • The value of the `ttl` parameter was not a valid time-to-live.
`producerNotFound`	The specified producer to modify does not exist.
`timeToLiveTooLarge`	The `ttl` parameter was supplied, but was an integer that is larger than that permitted by the service (the number of milliseconds in 2 weeks).
`operationFailed`	A low-level exception was thrown while attempting to modify the producer.

Close and Delete a Producer

This topic provides information about closing and deleting a producer.

Method: `DELETE`

Path: `/producers/`*`producerName`*

Scope: Messaging Context

Authorization: Messaging Administrator or Messaging Worker

Result: Closes and deletes the producer with name *producerName*.

Error Responses:

Error Message	Description
`producerNotFound`	The specified producer to close and delete does not exist.
`operationFailed`	A low-level exception was thrown while attempting to close and delete the producer.

Send a Message via a Producer

This section provides information about sending a message via a producer.

Method: `POST`

Path: `/producers/`*`producerName`*`/messages`

Scope: Messaging Context

Authorization: Messaging Administrator or Messaging Worker

Result: Sends the message specified by the HTTP headers and request body to the destination specified as described below.

Request parameters, headers, and the request body are as described in Properties of HTTP Requests to Send Messages from REST Clients, with the following differences in the request parameters:

Parameter	Description
deliveryMode	The default value of this header is defined by the producer.
ttl	The default value of this header is defined by the producer.
destination	Required if the producer was not created with a destination, in which case the message is sent to the specified destination. If present, must have one of the following forms: • /queues/*queueName* • /topics/*topicName* • /temporaryQueues/*queueName* • /temporaryTopics/*topicName* Forbidden if the producer was created with a destination, in which case the message is sent to the destination with which the producer was created.
groupId	Optional. This parameter is used to set the JMSXGroupID property on the message being sent. This is the name of the message group of which this message is a part, if any. **Note:** • If the JMSXGroupID property is set as an HTTP request header, it must be set to an escaped value String or a badParameter error response will be generated. For more information on escaped value Strings, see About Escaped Value Strings. If the JMSXGroupID property is set as a query string parameter, the usual conventions for escaping query string parameters hold. • This parameter is optional, but should be set if, and only if, groupSeq is set.
groupSeq	Optional. This parameter is used to set the JMSXGroupSeq property on the message being sent. This is the sequence number of the message within the message group specified by the groupId parameter. The groupSeq parameter must be set to an integer or a badParameter error response will be generated. Note that this parameter is optional, but should be set if, and only if, groupId is set.

Response Headers:

Header	Description
X-OC-DESTINATION	One of the following values is set for this header: • /queues/*name of the queue to which the message was sent* • /topics/*name of the topic to which the message was sent* • /temporaryQueues/*name of the temporary queue to which the message was sent* • /temporaryTopics/*name of the temporary topic to which the message was sent*
X-OC-MESSAGE-ID	*message's ID*, if present

Header	Description
`X-OC-DELIVERY-MODE`	*`persistent`* or *`non_persistent`*. For more information, see About Persistent and Non-Persistent Messages.
`X-OC-TIMESTAMP`	*`time at which message was handed off to the JMS broker`*, if present. This is a long integer interpreted as Unix time.
`X-OC-EXPIRATION`	*`message expiration time.`* This is a long integer interpreted as Unix time.
`X-OC-PRIORITY`	This is always the default value, 4.

Error Responses:

Error Message	Description
`badParameter`	One of the following occurred: • The request specified a value for the `destination` parameter that did not parse as a specification of a queue or topic. • The value of the `deliveryMode` parameter did not parse as a delivery mode. • The value of the `ttl` parameter was not a valid time-to-live. • The value of the `replyTo` parameter did not parse as a queue or topic specification. • The value of the `messageType` parameter was not one of the allowed values for this parameter.
`badProperty`	An `X-OC-`*`type`*`-PROPERTY-`*`name`* header had a value that did not fit the format of properties with the given *type*.
`forbiddenContentType`	The `Content-Type` header of the HTTP request had value `application/x-www-form-urlencoded`. See Error Keys, Status Codes and Error Messages for further information.
`multipleDestinations`	Either the producer has no default destination and no destination was specified by the request, or the producer has a default destination *and* a destination was specified by the request.
`destinationParameterNotFound`	The destination specified for the message does not exist.

Error Message	Description
`producerNotFound`	The specified producer by which to send the message does not exist.
`messageHeadersTooLarge`	The request's message-relevant headers exceeded the maximum size.
`messageBodyTooLarge`	The request's body exceeded the maximum size.
`timeToLiveTooLarge`	The `ttl` parameter was supplied, but was an integer that is larger than that permitted by the service (the number of milliseconds in 2 weeks).
`maxMessagesOnTargetDestinationReached`	The service instance already has the maximum number of messages on the specified destination of the message.
`maxMessageBytesOnTargetDestinationReached`	The message could not be sent because the targeted destination reached the hard quota on the number of bytes of messages on it, and has not yet fallen below its soft quota. For more information, see Hard and Soft Quotas.
`operationFailed`	The server was unable to obtain the input stream containing the message body, or a low-level exception was thrown by the JMS broker in trying to send the message.

Receiving Messages

This section provides information about receiving messages using REST API.

Topics:

- Create a Consumer
- Close and Delete a Consumer
- Receive a Message via a Consumer

Create a Consumer

This topic provides information about creating a consumer.

Method: `PUT`

Path: `/consumers/`*`consumerName`*

Scope: Messaging Context

Authorization: Messaging Administrator or Messaging Worker

Request Parameters:

Parameter	Description
`session`	Required. The value is the name of the session in which to create the consumer.
`destination`	Optional. If present, the value must have one of the following forms: • `/queues/`*`queueName`* • `/topics/`*`topicName`* • `/temporaryQueues/`*`queueName`* • `/temporaryTopics/`*`topicName`* Specifies the destination from which the consumer consumes messages. If the `destination` parameter is present, a consumer is created with characteristics given by the other parameters, as described later. If the `destination` parameter is not present, the consumer being created should be a consumer of a topic through an already created durable subscription.
`selector`	Optional. Specifies the subset of messages the consumer will receive. The value of the parameter must be a selector. For the syntax of selectors, see the **Message Selectors** section of the Java API reference for the `javax.jms.Message` class.
`localMode`	Optional. If present, value must be `GET_LOCAL` or `NO_LOCAL`. Specifies whether the consumer on a topic will receive messages sent via the connection that contains the consumer. A value of `GET_LOCAL` means that messages sent via the connection are received, and a value of `NO_LOCAL` means that such messages will not be received. The default is `GET_LOCAL`.
`subscriptionName`	Optional. Specifies the name of the durable subscription. If present, the consumer created will be a durable topic subscriber (and so the `destination` parameter must specify a topic). See Create a Durable Subscription for an example HTTP request/response sequence.

Result: Creates a consumer. If the `destination` parameter is present and it specifies a queue, then the consumer will consume from the queue. If the `destination` parameter is present and it specifies a topic, then the consumer will consume from the topic.

Note:

If the `subscriptionName` parameter is present, then the exact result depends significantly on whether the `destination` parameter is present or not.

If the `destination` parameter is present, a consumer is created with characteristics given by the other parameters, as follows:

- If the `destination` specifies a topic, the client ID has been set for the connection and `subscriptionName` is present, then the consumer will consume messages via a durable subscription to the topic specified by the client ID and the subscription name.

- If no durable subscription with the given client ID and name exists, one is created on the specified topic with the specified selector.
- If a durable subscription for the ID and name already exists, and the topic and selector are the same as in the method that created the durable subscription, the existing durable subscription is used (and, thus, any messages sent to the topic since the subscription was created that have not been consumed through the subscription is available to be consumed).
- If a durable subscription for the ID and name already exists, but either the topic or the selector (or lack thereof) specified in this method are different from the topic and selector (or lack thereof) specified in the method that created the existing subscription, the existing subscription is deleted, and messages saved by it discarded, and a new durable subscription with the specified ID, name and selector on the specified topic is created.

If the `destination` parameter is *not* present, the client ID *must* have been set on the connection, the `subscriptionName` parameter *must* be present, the `selector` parameter must *not* be present, and there must be an existing durable subscription with the given client ID and subscription name. (The semantics of `localMode` are unchanged.) In this case, the consumer created is a consumer on the topic for the existing subscription, with the selector (or lack thereof) of the existing subscription, that uses the existing subscription. In this case, the method will *not* create or delete a subscription.

Response Headers:

Response to creating a consumer on an existing durable subscription contains the following headers that provide the properties of the subscriber:

- `X-OC-DESTINATION`

  Indicates the topic of the subscription.

- `X-OC-SELECTOR`

  Indicates the selector of the subscription, if any.

Error Responses:

Error Message	Description
`missingParameter`	The `session` parameter value was not supplied.
`badParameter`	One of the following occurred: • The `destination` parameter value did not parse as a specification of a queue or topic. • The `selector` parameter value was ill-formed, or contained a disallowed identifier. • The `localMode` parameter value is not one of the allowed values. • The method did not supply a `destination` parameter, but the existing durable subscription from which it was meant to consume had a bad selector.

Error Message	Description
`noDestinationForConsumer`	Neither a destination nor a subscription name were supplied.
`localModeNonTopic`	The `localMode` and `destination` parameters were supplied, but the `destination` parameter does not specify a topic.
`subscriptionInUse`	An attempt was made to create a consumer on a durable subscription when that durable subscription already has a consumer on it.
`subscriptionNonTopic`	Subscription name and destination parameters were supplied, but the `destination` parameter does not specify a topic.
`sessionParameterNotFound`	Session by which to create the producer does not exist.
`destinationParameterNotFound`	The destination from which the consumer should get messages is nonexistent.
`consumerAlreadyExists`	There is already a consumer with the given name.
`subscriptionNotFoundNoInfo`	The method did not supply a `destination` parameter, but the existing durable subscription from which it was meant to consume was not found.
`maxDurableSubscriptionsReached`	The consumer whose creation was attempted was on a durable subscription that does not currently exist, and this method invocation would create the subscription if it does not exist, and the service instance is at its maximum number of durable subscriptions.
`operationFailed`	The method did not supply a `destination` parameter, but the existing durable subscription from which it was meant to consume had a bad destination, or a low-level exception was thrown while creating the consumer.

Close and Delete a Consumer

This topic provides information about closing and deleting a consumer.

Method: `DELETE`

Path: `/consumers/`*`consumerName`*

Scope: Messaging Context

Authorization: Messaging Administrator or Messaging Worker

Result: Closes and deletes the consumer.

Error Responses:

Error Message	Description
`consumerNotFound`	The specified consumer to close does not exist.
`operationFailed`	A low-level exception was thrown while closing and deleting the specified consumer.

Receive a Message via a Consumer

This section provides information about receiving a message via a consumer.

Method: `POST`

Path: `/consumers/consumerName/messages`

Scope: Messaging Context

Authorization: Messaging Administrator or Messaging Worker

Request Parameter:

Parameter	Description
`timeout`	Required. The number of milliseconds in the value must be a strictly positive long integer that is no more than the maximum receive timeout of 5 minutes. Specifies the amount of time in milliseconds to wait for a message to become available from the consumer before returning a null message.

See Receive a Message from a Queue with a Selector and Receive a Message from a Durable Subscription for examples of HTTP request/response sequences.

Result:

- If there is a message on the consumer's queue that satisfies the selector (if it was set on the consumer), or if one enters the queue within the number of milliseconds in `timeout`, it is returned in the HTTP response. Otherwise, a null response is returned.
- If receiving from a topic, if messages have been published to the topic since the consumer was created, or if messages are published to the topic within the number of milliseconds in `timeout`, one of those messages will be return in the HTTP response. Otherwise, a null response is returned.
- If the consumer is consuming from a durable subscription, and if there is a message currently stored in the durable subscription, or if one enters the durable subscription within the number of milliseconds in `timeout`, it is returned in the HTTP response. Otherwise, a null response is returned.

> **Note:**
>
> If the session by which the consumer was created is transacted, or is not transacted but has client acknowledgement set, only one message may be received via that consumer before committing (if the session is transacted) or acknowledging (if the session is not transacted but has client acknowledgement set) the session. Until the appropriate commit or acknowledgement action is taken, receives from that consumer will return a null response, even if there are other messages that could otherwise be received from the consumer's destination.

Response Headers: If a message is returned, the headers will include those determined by the message described in Properties of HTTP Requests and Responses that Deliver Messages. Otherwise, a response with no content with the header `X-OC-NULL: true` is returned.

Error Responses:

Error Message	Description
`missingParameter`	The request had no `timeout` parameter specified.
`badParameter`	The value of the `timeout` parameter did not parse as a long integer value, or the value of the `timeout` parameter was 0 or a negative number.
`consumerNotFound`	The specified consumer from which to receive does not exist.
`timeoutTooLarge`	The value of the `timeout` parameter was larger than the maximum allowed value (5 minutes).
`operationFailed`	A low-level exception was thrown while receiving, or an exception was thrown while attempting to extract information from the received message.

Creating and Managing Durable Subscriptions

This section provides information about creating and managing durable subscriptions in Oracle Messaging Cloud Service.

Topics:

- Create a Durable Subscription
- List Durable Subscriptions
- Delete a Durable Subscription

Create a Durable Subscription

Durable subscriptions are implicitly created by creating a special type of consumer.

To create a durable subscription, see the Create a Consumer section, specifically the notes about `subscriptionName` and `clientId`.

List Durable Subscriptions

This section provides information about listing durable subscriptions.

Method: `GET`

Path: `/subscriptions`

Scope: Messaging Context

Authorization: Messaging Administrator or Messaging Worker

Request Parameters:

Parameter	Description
`subscriptionName`	The name of the subscription.
`topicName`	The name of a topic. Note that the value of this parameter is not of the form `/topics/`*`name`*, but is only the *`name`* part of that form, since durable subscriptions only apply to topics.
`clientId`	A client ID associated with the subscription.

None of these parameters are mandatory, but only certain combinations are allowed, as described below.

Result: Returns information about one or more durable subscriptions depending on the request parameters supplied:

- `clientId` and `subscriptionName`

  Returns an XML or a JSON document specifying the unique durable subscription for the given name and client ID, if one exists, for the namespace specified in the request.

  The XML format is as follows:

  ```
  <subscription>
      <clientId>client ID<clientId>
      <name>subscription name</name>
      <topic>topic name</topic>
      <selector>selector</selector>
      <canonicalLink>relative path to the subscription</canonicalLink>
  </subscription>
  ```

  The JSON format is as follows:

  ```
  {
      "clientId": "client ID",
      "name": "subscription name",
      "topic": "topic name",
      "selector": "selector",
      "canonicalLink": "relative path to the subscription"
  }
  ```

  All child elements will be present except `selector`, which is present if the subscription is associated with a selector. The `clientId` and `name` elements always have the same values as the corresponding parameters.

- `clientId`

  Returns an XML or a JSON document specifying all of the durable subscriptions for the given client ID for the namespace specified in the request.

  The XML format is as follows:

  ```
  <subscriptions>
      <items>
          <clientId>client ID<clientId>
          <name>name</name>
          <topic>topic name</topic>
          <selector>selector</selector>
          <canonicalLink>relative path to the subscription</canonicalLink>
      </items>
      ...
      <canonicalLink>relative path to the subscription list</canonicalLink>
  </subscriptions>
  ```

  The JSON format is as follows:

  ```
  {
     "subscriptions":
        {
          "clientId": "client ID",
          "name": "subscription name"
          "topic": "topic name",
          "selector": "selector",
          "canonicalLink": "relative path to the subscription"
        }
      "canonicalLink": "relative path to the subscription list"
  }
  ```

  This element may be empty if the client ID has no subscriptions. Each child specifies a unique durable subscription. The *client ID* of all `subscription` elements is the same as the value of the corresponding request parameter; each value of *name* will appear only once, as a durable subscription is specified by the client ID and subscription name.

- `topicName`

  Returns an XML or a JSON document specifying all of the durable subscriptions for the topic with the given name for the namespace specified in the request. Note that the value of this parameter is not of the form `/topics/`*name*, but is only the *name* part of that form, since durable subscriptions only apply to topics. The format of the XML document in the response is the same as the XML format described earlier for `clientId`. This element may be empty if the topic has no subscriptions. Each child specifies a unique durable subscription. The `topic>` content of all `subscription` elements is the same as the value of the corresponding request parameter; each pair of values for *client ID* and *name* will appear only once, as a durable subscription is specified by the client ID and subscription name.

- None of the parameters are supplied

  Returns an XML or a JSON document specifying all of the durable subscriptions for the namespace specified in the request. The format is the same as described earlier for `clientId` and `subscriptionName`, `clientId`, and `topicName`. This element may be empty if the namespace has no durable subscriptions. Each child specifies a unique durable subscription. The *topic* values of all `subscription` elements is the same as the value of the corresponding request parameter; each

pair of values for *client ID* and *name* will appear only once, as a durable subscription is specified by the client ID and subscription name.

Error Responses:

Error Message	Description
`disallowedSubscriptionLookup`	A combination of the `subscriptionName`, `clientId`, and `topicName` parameters has been supplied that is not one of those listed earlier.
`destinationParameterNotFound`	The `topicName` parameter was specified, but there is no topic by that name in the namespace.
`subscriptionNotFoundFull`	A client ID and subscription name were specified in the request, but no such subscription exists.
`operationFailed`	A low-level exception occurred.

Delete a Durable Subscription

Durable subscriptions are implicitly created by creating a special type of consumer. A durable subscription stores all messages sent to a topic until each message is received.

For information about deleting durable subscriptions through sessions, see Delete a Durable Subscription.

Creating and Managing Temporary Destinations

This section provides information about creating and managing temporary destinations in Oracle Messaging Cloud Service.

Topics:

- Create a Temporary Destination
- List Temporary Destinations
- Remove a Temporary Destination

Create a Temporary Destination

You can create the following kinds of temporary destinations:

- Temporary Queue: A *TemporaryQueue* is a unique Queue object created for the duration of a connection. Messages may only be consumed from a temporary queue through the connection with which it was created.
- Temporary Topic: A *TemporaryTopic* is a unique Topic object created for the duration of a connection. Messages may only be consumed from a temporary topic through the connection with which it was created.

Method: `POST`

Path:

- To create a temporary queue, the path is `/temporaryQueues`
- To create a temporary topic, the path is `/temporaryTopics`

Scope: Messaging Context

Authorization: Messaging Administrator or Messaging Worker

Request Parameter:

Parameter	Description
`session`	Specify the session name. This parameter is mandatory.

Result: Creates a temporary queue or temporary topic.

Response Header:

`Location`

Contains the URL for the newly created temporary queue or temporary topic.

Response Body:

Destination Type	XML Format	JSON Format

Temporary queue	`<temporaryQueue>`
	`  <name>`*queue name*`</name>`
	`  <connection>`*name of the connection with which the queue is associated*`</connection>`
	`  <canonicalLink>`*relative path to temporary queue*`</canonicalLink>`
	`</temporaryQueue>`
		`{`
		`  "type": "temporaryQueue",`
		`  "name" "`*queue name*`",`
		`  "connection": "`*name of the connection with which the queue is associated*`",`
		`  "canonicalLink": "`*relative path to temporary queue*`"`
		`}`

Temporary topic	`<temporaryTopic>`
	`  <name>`*topic name*`</name>`
	`  <connection>`*name of the connection with which the topic is associated*`</connection>`
	`  <canonicalLink>`*relative path to temporary topic*`</canonicalLink>`
	`</temporaryTopic>`
		`{`
		`  "type": "temporaryTopic",`
		`  "name" "`*topic name*`",`
		`  "connection": "`*name of the connection with which the topic is associated*`",`
		`  "canonicalLink": "`*relative path to temporary topic*`"`
		`}`

Destination Type	XML Format	JSON Format

Temporary queues	<temporaryQueues>
	<name>*queue name*</name>
	<connection>*name of the connection with which the queue is associated*</connection>
	<canonicalLink>*relative path to temporary queue*</canonicalLink>
	...
	<canonicalLink>*relative path to list of temporary queues*</canonicalLink>
	</temporaryQueues>
		{
		"temporaryQueues": [
		{
		"name": "*queue name*",
		"connection": "*name of the connection with which the queue is associated*",
		"canonicalLink": "*relative path to temporary queue*"
		}
		...
		],
		"canonicalLink": "*relative path to list of temporary queues*"
		}

Temporary topics	<temporaryTopics>
	<name>*topic name*</name>
	<connection>*name of the connection with which the topic is associated*</connection>
	<canonicalLink>*relative path to temporary topic*</canonicalLink>
	...
	<canonicalLink>*relative path to list of temporary topics*</canonicalLink>
	</temporaryTopics>
		{
		"temporaryTopics": [
		{
		"name": "*topic name*",
		"connection": "*name of the connection with which the topic is associated*",
		"canonicalLink": "*relative path to temporary topic*"
		}
		...
		],
		"canonicalLink": "*relative path to list of temporary topics*"
		}

The content of the `name` field is a pseudorandom name generated by the service for the newly created temporary queue or topic.

The `connection` field is omitted if the destination was not created from a connection in the messaging context.

Error Response:

Error Message	Description
`sessionNotFound`	A session with the specified name does not exists.

List Temporary Destinations

This section provides information about listing temporary destinations.

- List Temporary Queues or Temporary Topics
- Retrieve Properties of a Single Temporary Queue or a Single Temporary Topic

List Temporary Queues or Temporary Topics

Method: `GET`

Path:

- To list all temporary queues in the messaging context, the path is `/temporaryQueues`
- To list all temporary topics in the messaging context, the path is `/temporaryTopics`

Scope: Messaging Context

Authorization: Messaging Administrator or Messaging Worker

Request Parameter:

Parameter	Description
`connection`	Optional. Specify the `connection` parameter to list all temporary queues or topics associated with the connection with the name *`connection-name`*.

Result: Returns a listing of all temporary queues or temporary topics in the messaging context.

Response Body:

In XML, the format for listing all temporary queues associated with an HTTP cookie is as follows:

```
<temporaryQueues>
    <items>
        <name>queue name</name>
        <connection>name of the connection with which the queue is associated</
connection>
        <canonicalLink>relative path to temporary queue</canonicalLink>
    </items>
    ...
    <canonicalLink>relative path to list of temporary queues</canonicalLink>
</temporaryQueues>
```

In XML, the format for listing all temporary topics associated with an HTTP cookie is as follows:

```
<temporaryTopics>
    <items>
        <name>topic name</name>
        <connection>name of the connection with which the topic is associated</
connection>
        <canonicalLink>relative path to temporary topic</canonicalLink>
    </items>
    ...
    <canonicalLink>relative path to list of temporary topics</canonicalLink>
</temporaryTopics>
```

The content of the <name> element is the *queueName* or *topicName*.The <connection> element is present only if the temporary queue or topic was created from a connection associated with the client's messaging context.

If the temporary queue or topic was created elsewhere and received as, for example, in the Reply-To header of a message, then the <connection> element will not be present.

Error Response:

Error Message	Description
connectionParameterNotFound	A REST API client invoked the method to list all temporary destinations created through a certain named connection, but no connection with that name exists in the messaging context.

Retrieve Properties of a Single Temporary Queue or a Single Temporary Topic

Method: GET

Path:

- To retrieve the properties of a single temporary queue, the path is /temporaryQueues/*queueName*
- To retrieve the properties a single temporary topic, the path is /temporaryTopics/*topicName*

Scope: Messaging Context

Authorization: Messaging Administrator or Messaging Worker

Result: Returns the properties of a temporary queue or a temporary topic with the given name.

Response Body:

The format for listing a single temporary queue is as follows:

```
<temporaryQueue>
    <name>queue name</name>
    <connection>name of the connection with which the queue is associated</
connection>
    <canonicalLink>relative path to temporary queue</canonicalLink>
</temporaryQueue>
```

The format for listing a single temporary topic is as follows:

```
<temporaryTopic>
    <name>topic name</name>
    <connection>name of the connection with which the topic is associated</
connection>
    <canonicalLink>relative path to temporary topic</canonicalLink>
</temporaryTopic>
```

The content of the <name> element is the *queueName* or *topicName*.The <connection> element is present only if the temporary queue or topic was created from a connection associated with the client's messaging context.

If the temporary queue or topic was created elsewhere and received as, for example, in the `Reply-To` header of a message, then the `<connection>` element will not be present.

Error Response:

Error Message	Description
`destinationNotFound`	The temporary destination that is requested does not exist.

Remove a Temporary Destination

This section provides information about removing temporary destinations.

Method: `DELETE`

Path:

- To delete a temporary queue, the path is `/temporaryQueues/`*`queueName`*
- To delete a temporary topic, the path is `/temporaryTopics/`*`topicName`*

Scope: Messaging Context

Authorization: Messaging Administrator or Messaging Worker

Result: Deletes the temporary destination with the given name.

> **Note:**
>
> If the temporary destination that is being deleted was created from a connection in the same messaging context, the temporary destination will be deleted from the back-end. Else, it will be deleted only from the messaging context, and not from the back-end.
>
> After being deleted from the back-end, the temporary destination will not be available for use to other clients.

Error Response:

Error Message	Description
`destinationNotFound`	The temporary destination whose deletion is requested does not exist.

Creating and Managing Queue Browsers

This section provides information about creating and managing queue browsers in Oracle Messaging Cloud Service.

Topics:

- Create a Queue Browser
- Retrieve Queue Browser Properties

- Browse Messages
- Remove a Queue Browser

Create a Queue Browser

A client uses a queue browser to look at messages on a queue without removing them. A queue browser is created from a Session.

A queue browser may be used to look at all the messages in a queue, or only those that match a message selector. Note that if messages are sent to a queue after a browser on that queue is created, those messages may not be visible via the queue browser.

Method: `PUT`

Path: `/queueBrowsers/browserName`

Scope: Messaging Context

Authorization: Messaging Administrator or Messaging Worker

Request Parameters:

Parameter	Description
`session`	The value is the name of the session in which the queue browser needs to be created.
`destination`	Specify the destination name. The value must have the form `/queues/queueName`.
`selector`	Message selector, which is optional.

Result: Creates a queue browser on the `destination` parameter.

Error Responses:

Error Message	Description
`sessionNotFound`	There is no session with the specified name.
`destinationNotFound`	The destination that is requested does not exist.
`queueBrowserAlreadyExists`	A queue browser with the specified name already exists.
`badParameter`	One of the following occurred: • The `destination` parameter value did not parse as a specification of a queue. • The `selector` parameter value was ill-formed, or contained a disallowed identifier.

Retrieve Queue Browser Properties

This section provides information about retrieving queue browser properties.

Method: `GET`

Path: `/queueBrowsers/browserName`

Scope: Messaging Context

Authorization: Messaging Administrator or Messaging Worker

Result: Returns properties of the queue browser with the name *browserName*.

Response Body:

In XML, the format for the properties of a queue browser is as follows:

```
<queueBrowser>
    <name>client-assigned name of the browser</name>
    <queue>path specifying the persistent or temporary queue browsed</queue>
    <selector>selector expression<selector>
    <canonicalLink>relative path to queue browser</canonicalLink>
</queueBrowser>
```

In JSON, the format for the properties of a queue browser is as follows:

```
{
    "name": "client-assigned name of the browser",
    "queue": "path specifying the persistent or temporary queue browsed",
    "selector": "selector expression",
    "canonicalLink": "relative path to queue browser"
}
```

The content of the `queue` element has the form `/queue/queueName` if the browser browses a persistent queue named `queueName`; it has the form `/temporaryQueue/queueName` if the browser browses a temporary queue named `queueName`.

The `selector` element is present only if the queue browser is associated with a selector.

Error Response:

Error Message	Description
`queueBrowserNotFound`	The queue browser that is requested does not exist.

Browse Messages

This topic provides information about browsing messages in a queue browser.

Method: `POST`

Path: `/queueBrowsers/browserName`

Scope: Messaging Context

Authorization: Messaging Administrator or Messaging Worker

Result: If there is a message in the browser, it is returned in the HTTP response. Otherwise, a null response is returned.

Response Header:

If there is an `X-OC-NULL` header with value `true`, then it indicates that there are no more messages in the browser.

Error Response:

Error Message	Description
`queueBrowserNotFound`	The queue browser that is requested does not exist.

Remove a Queue Browser

This section provides information about removing a queue browser.

Method: `DELETE`

Scope: Messaging Context

Path: `/queueBrowsers/browserName`

Authorization: Messaging Administrator or Messaging Worker

Result: Closes and deletes the queue browser.

Error Response:

Error Message	Description
`queueBrowserNotFound`	The queue browser that is requested does not exist.

Properties of HTTP Requests to Send Messages from REST Clients

This section provides information about properties of HTTP requests to send messages from REST clients.

Topics:

Request Parameters

This section provides information about request parameters which can be used when sending messages using the REST API.

`correlationId`

Optional. No default.

This parameter is used to group together (correlate) multiple messages.

For the query string parameter `correlationId`, the equivalent HTTP header is `X-OC-CORRELATION-ID`.

`deliveryMode`

Optional. If present, value must be `persistent` or `non_persistent`; default is `persistent`. Determines whether the message is persistent (that is, stored by the JMS broker in persistent storage until delivered).

For the query string parameter `deliveryMode`, the equivalent HTTP header is `X-OC-DELIVERY-MODE`.

`messageType`

Optional; if omitted, the default is HTTP. This parameter specifies the type of JMS message the HTTP request will cause to be generated. Depending on the value, there may be restrictions on the HTTP request body, and/or special interpretation of that body and/or some of the standard HTTP request parameters.

For the query string parameter `messageType`, the equivalent HTTP header is `X-OC-MESSAGE-TYPE`.

See Message Types for information about the valid values for the `messageType` query string parameter along with any required formatting for the HTTP request body.

`replyTo`

Optional. If present, must have one of the following forms:

- `/queues/`*`queueName`*
- `/temporaryQueues/`*`queueName`*
- `/topics/`*`topicName`*
- `/temporaryTopics/`*`topicName`*

Specifies a destination or a temporary destination to which replies should be sent.

For the query string parameter `replyTo`, the equivalent HTTP header is `X-OC-REPLY-TO`.

`ttl`

Optional. If present, value must be a strictly positive long integer or the value `maximum`. Determines the time in milliseconds between when the message is dispatched and when the service may delete it if not yet delivered. The semantics of these parameters is the same as the corresponding parameters for producers, but the default is defined by the producer, and is overridden by any headers occurring in this operation.

For the query string parameter `ttl`, the equivalent HTTP header is `X-OC-TTL`.

HTTP Headers to Specify Message Properties

The HTTP request headers (when sending messages) can specify messaging properties for the message being sent.

`X-OC-`*`propertyType`*`-PROPERTY-`*`propertyName`*

Optional. This is not a single header, but a family of headers, referred to collectively as **message property headers**, one for each *property type* and *property name*. The *property name* value should consist of alphanumeric characters and underscores, and is made lower-case by the service. This sets the message property with name *property*

name on the message to value *property value*, if the *property type* is one of the allowed values and *property value* is a value allowed for the *property type*. See Message Headers and Properties for information about the allowed values of *property type*.

Limitations on Message Size

Oracle Messaging Cloud Service only allows messages with at most 3K (3072) characters of messaging-relevant HTTP request headers and at most 512K (524,288) bytes or characters of HTTP request body. An attempt to send a message that exceeds these limits will generate an error response, and no message will be sent. Details of this limitation are as follows:

- An HTTP request header is **messaging-relevant** if its name begins with `X-OC-` (case-insensitive); if the message type is `HTTP`, or is not supplied (in which case it defaults to `HTTP`), the `Content-Type` and `Content-Language` headers are also messaging-relevant.
- The number of characters in a messaging-relevant header is the sum of the number of characters in the header name and the number of characters in the header value.
- If the message has no type, or has type `BYTES`, `OBJECT`, or `HTTP`, the size of the HTTP request body is measured in bytes, and is counted as in the HTTP specification.
- If the message has type `TEXT`, `MAP`, or `STREAM`, the size of the HTTP request body is measured in characters according to the character encoding.
- If the message has type `PLAIN`, the request body is ignored, and so has size 0 for purposes of size limitations.

> **Note:**
>
> - To send a message that is larger than 512 KB, see Using Message Groups. This enables you to send messages/files up to 10 MB size in smaller chunks.
> - To send larger message payload, you can use Oracle Storage Cloud Service to store the object and send it as a message. This is especially useful for storing and consuming messages with a message size of up to 5 GB. See Sending Large Objects as Messages Using Oracle Storage Cloud Service

Properties of HTTP Requests and Responses that Deliver Messages

Messages are delivered to clients of the REST API in two ways: in the HTTP response to a receive request, and in an HTTP request made by a message push listener to an HTTP endpoint.

This section describes the headers and body of the responses and requests that describe the message delivered.

For HTTP requests made by message push listeners, some of the characteristics of the requests (the URL pushed to, method used, and so on) are determined by the listener. In addition to the characteristics that are determined by the listener, push requests also have a header `X-OC-LISTENER-NAME` whose value is the name of the listener from which the push originates.

The remainder of this section describes the aspects of a receive response or push request that are determined by the message:

- The body of the request or response and its `Content-Type`, `Content-Language`, and `X-OC-MESSAGE-TYPE` headers are as described in Properties of HTTP Requests to Send Messages from REST Clients.
- The following non-standard headers are set from message headers and properties:
  - `X-OC-CORRELATION-ID: message correlation ID`
  - `X-OC-DELIVERY-MODE: persistent` or `non_persistent`. See About Persistent and Non-Persistent Messages.
  - `X-OC-DESTINATION`: One of the following values is set for this header.
    - `/queues/name of the queue to which the message was sent`
    - `/topics/name of the topic to which the message was sent`
    - `/temporaryQueues/name of the temporary queue to which the message was sent`
    - `/temporaryTopics/name of the temporary topic to which the message was sent`
  - `X-OC-EXPIRATION: message expiration time`. This is a long integer interpreted as Unix time.
  - `X-OC-MESSAGE-ID: message's ID`, if present
  - `X-OC-PRIORITY: message priority`
  - `X-OC-REDELIVERED: true/false`, indicating if message is a re-delivery
  - `X-OC-REPLY-TO`: One of the following values is set for this header.
    - `/queues/name of the queue to which to reply`
    - `/topics/name of the topic to which to reply`
    - `/temporaryQueues/name of the temporary queue to which to reply`
    - `/temporaryTopics/name of the temporary topic to which to reply`
  - `X-OC-TIMESTAMP: time at which message was handed off to the JMS broker`, if present. This is a long integer interpreted as Unix time.
  - `X-OC-LISTENER-NAME: name of the listener that pushed the message`
  - `X-OC-DELIVERY-COUNT: value of the message's JMSXDeliveryCount property`, if present. This is the number of times the service has attempted to deliver this message. The value of this parameter can be 2 or higher if, for example, the message has been received in a transacted session but the receive was not committed.
  - `X-OC-GROUP-ID: the name of the message group of which this message is a part`, if any. This is the value of the `JMSXGroupId` property, with

characters that are not legal in an HTTP header value escaped. The group of a message is set by the client when the message is sent. The value of this header, if present, will be an escaped value String. For more information on escaped value Strings, see About Escaped Value Strings.

– `X-OC-GROUP-SEQ:`*`sequence number of the message within a message group`*, if present. The value will be an integer. This is the value of the `JMSXGroupSeq` property. The sequence number is set by the client when the message is sent.

– For each message property, a header

  `X-OC-`*`property type`*`-PROPERTY-`*`property name`*`: `*`property value`*

  for properties whose names and values conform to the HTTP specification for headers, with the *property type* determined by the type of the Message's properties; for header names and values that do not, a header of the form

  `X-OC-GENERAL-`*`property type`*`-PROPERTY-`*`property name as pairs of hex digits`*`:` *`property value as pairs of hex digits`*

  Any message property with value `null` is not expressed as a property header.

  See Message Headers and Properties for information about message headers and HTTP headers, and message property types.

4

Accessing Oracle Messaging Cloud Service Using Java Library

The Oracle Messaging Cloud Service Java library implements and extends the Java Message Service (JMS) 1.1 interface. This section provides information about how to use the Java library and the differences from JMS when using the Java library.

Topics:

Client-Side Logging

Certain events are logged by the Java library on the Java platform that runs the client software.

Event logging is done via the `java.util.logging` framework, using the logging levels of java.util.logging.Level. Events may be useful in diagnosing the cause of problems, or may be useful input to Support.

Events are logged at the WARNING and SEVERE levels in two packages: `oracle.cloud.messaging.client` and `oracle.cloud.messaging.util`.

Package	Log Level	Log Message	Description
`oracle.cloud.messaging.client`	SEVERE	Exception shutting down thread pool	An exception was thrown when a Connection was being closed. This occurred when an attempt was made to shut down a thread pool used for background operations like executing `MessageListeners`. The exception thrown is logged.
		Exception stopping REST client	An exception was thrown when a Connection was being closed. This occurred when an attempt was made to stop the HTTP client by which the Java library communicates with the REST API. The exception thrown is logged.
		Exception deleting connection	An exception was thrown when attempting to close and delete the server-side Connection associated with a client-side Connection. The exception thrown is logged.
		Exception shutting down messaging context	An exception was thrown when attempting to shut down the server-side messaging context associated with a client-side Connection. The exception thrown is logged.
		Exception disconnecting HTTP connection	An exception was thrown when attempting to disconnect a network connection from the client to the server. The exception thrown is logged.

Package	Log Level	Log Message	Description
		Exception on client-side session close	An exception was thrown when a Connection was being closed. This occurred when an attempt was made to shut down a Session created from the Connection. The exception thrown is logged.
	WARNING	Closing unclosed Connection in finalizer	A Connection was closed when the Connection object was garbage-collected; this indicates that a reference to a Connection was discarded without calling `close()` on the Connection.
`oracle.cloud.messaging.util`	WARNING	Runtime exception thrown from exception listener	The HTTP client encountered an exception attempting to reach the server, and the listener that handles such exceptions threw a RuntimeException. The exception thrown is logged.

Automatic Closing of Connections

This section provides information about situations in which Connections close automatically.

Connections Close if Server is Unreachable for a Sufficiently Long Period of Time

If a Java library client cannot reach the Oracle Messaging Cloud Service server for a sufficiently long period of time, Connections for that client will be closed. The amount of time before a Connection is closed is given by the Connection's timeout, plus an additional margin for error to allow for clock skew between the client and server.

During the period in which the client is unable to reach the server, client operations will throw exceptions, but a Connection will not be closed until the server has been unreachable for longer than the Connection's timeout. When a Connection is closed for this reason, if the Connection has an `ExceptionListener` set on it, a `JMSException` will be dispatched to the Connection's `ExceptionListener`.

Connections Close if Server-Side State is Lost

Connections may detect that their server-side state has expired or been lost. This may be because the client was unable to reach the server for a certain period, or because

the server failed due to other reasons. When the Java library detects that it has lost the server-side state for a Connection, the Connection will be closed. When a Connection is closed for this reason, if the Connection has an `ExceptionListener` set on it, a `JMSException` will be dispatched to the Connection's `ExceptionListener`.

Diagnosing Errors from the Java Library

The Java library offers additional features for diagnosing exceptions beyond what is available from JMS.

When a JMSException (or one of its subclasses) is thrown by a method, or dispatched to an `ExceptionListener`, the exception may have a cause, which can be obtained by invoking `getCause()` on the exception object. If the cause is not `null`, and has class `HttpResponseException`, the cause will contain Messaging-Service-specific information about the error that caused the exception. In particular, it may contain an error key and an error message with further information.

For more information on `HttpResponseException`, see `HttpResponseException` in *Java API Reference for Oracle Messaging Cloud Service*. For details on the meaning of error information obtainable from `HttpResponseException`, see Error Keys, Status Codes and Error Messages.

Using the Re-try Function

When a request from the Java client library to Oracle Messaging Cloud Service receives a response that indicates that it is attempting to access a messaging context that has expired, it re-tries the request, with pauses, before throwing a JMSException.

By default, the library re-tries the request four times, pausing for one second before the first re-try, then two seconds between the first re-try and the second, then four seconds, then eight seconds. At any point, if the response to the re-try indicates that the messaging context has been found again, no further re-tries is made.

The number of re-tries, and pauses between them, can be changed by setting the `oracle.cloud.messaging.client.retryWait` property using the OracleCloudConnectionFactory.setProperty() method. The value set for this property must be a comma-separated list of long integers. The number of integers is the number of re-tries; each integer is the number of milliseconds to pause before the corresponding re-try.

For example, invoking `setProperty("oracle.cloud.messaging.client.retryWait","1000,10000")` on an `OracleCloudConnectionFactory` object will change the re-try behavior to re-trying twice, waiting one second before the first re-try and ten seconds between the first and second re-tries.

Setting the `oracle.cloud.messaging.client.retryWait` property to `null` disables re-tries.

About Using the Java Library

Oracle Messaging Cloud Service can be accessed using the Java library.

Topics:

Prerequisites for Using the Java Library

The Oracle Messaging Cloud Service Java library provides all required Java Message Service (JMS) 1.1 functions, plus additional functions related to JMS and Oracle Messaging Cloud Service for sending and receiving messages through the JMS broker. Previous experience using JMS will be helpful.

The following are required to use the Oracle Messaging Cloud Service Java library:

- An Oracle Messaging Cloud Service instance
- The `oracle.cloud.messaging.api-14.0.X.jar` (where `X` is the number representing the latest version of the Java library)
- The JMS 1.1 API JAR file
- A Java Development Kit (JDK) of version 1.6 or greater

How to Use the Java Library

Before you begin using the Java library, be sure to review the following:

- Considerations When Developing Applications That Use Oracle Messaging Cloud Service
- Authentication and Authorization
- Differences from JMS
- Prerequisites for Using the Java Library
- Check the Version of the Java Library

To use the Oracle Messaging Cloud Service Java library:

1. Download the Java library (see Downloading the Oracle Messaging Cloud Service Java SDK).
2. Add the jar file `oracle.messaging.cloud.api-14.0.X.jar` (where `X` is the latest version number of the Java library) to your Java application's class path.
3. Import the Java library's classes and interfaces into your Java application.

```
import javax.jms.*;
import oracle.cloud.messaging.*;
import oracle.cloud.messaging.client.*;
import oracle.cloud.messaging.common.*;
```

4. Create a `MessagingService` object.

A `MessagingService` object is created from a `MessagingServiceFactory`. When creating a `MessagingService` object, a `MessagingServiceNamespace` object and a `MessagingServiceCredentials` object must be provided.

For more information, see Creating a MessagingService Object.

How to Check the version of the Java Library

The Java library is delivered in two identical .jar files — one containing the version number of the Java library in the file name (for example, `oracle.cloud.messaging.api-14.0.1.jar`,) and one without the version number (for example, `oracle.cloud.messaging.api.jar`.) You can also obtain the information about the version of the Java library after installation, through the class `oracle.cloud.messaging.VersionInfo`.

To check the version of the Java library, run the following command:
`java -jar <jar name>`.

The version number of the Java library is displayed.

The manifest of each of the Java library jars contains the version of the Java library as the `Implementation-Version` attribute of the `oracle.cloud.messaging` package.

Creating a `MessagingService` Object

The `MessagingService` interface is the entry point for all functionality of the Java library.

To create a `MessagingService` object, you must provide an Oracle Messaging Cloud Service instance URL and user credentials to the `MessagingServiceFactory.getMessagingService()` method.

The `MessagingServiceFactory.getMessagingService` method takes two parameters:

- A `MessagingServiceNamespace` object
- A `MessagingServiceCredentials` object

Example to create a `MessagingService` object

```
MessagingServiceFactory factory = MessagingServiceFactory.getInstance();

String serviceUrl = "https://messaging.us2.oraclecloud.com/myservice-mydomain";
String username = "john.doe@example.com";
String password = "myPassword";

Namespace namespace = new MessagingServiceNamespace(serviceUrl);

Credentials credentials = new MessagingServiceCredentials(username, password);

MessagingService service = factory.getMessagingService(namespace, credentials);
```

Once a `MessagingService` object has been created, you can manage **queues**, **topics**, and **message push listeners**. You can also list, and retrieve the properties of **durable subscriptions**.

In addition you can create a `ConnectionFactory` to send and receive messages using the Java library. See ConnectionFactory Creation API for information about how to use the JMS API provided in the Java library to obtain a `ConnectionFactory` object.

Using Messaging Cloud Service from Oracle Java Cloud Service - SaaS Extension

Complete the following steps to use Messaging Service from Oracle Java Cloud Service - SaaS Extension:

1. Subscribe to Oracle Java Cloud Service - SaaS Extension and Oracle Messaging Cloud Service.
2. Create a new user in Messaging Cloud Service to use from Oracle Java Cloud Service - SaaS Extension. Assign the `Messaging Worker` role if the application needs to send and receive messages. Assign the `Messaging Administrator` role if the application needs to manage resources. See About Oracle Messaging Cloud Service Roles and Users.
3. Download the Java library (see Downloading the Oracle Messaging Cloud Service Java SDK).
4. Package the library with your Java application.
5. Write the application code that uses the Java library to create a `ConnectionFactory` object. Use the `ConnectionFactory` object to obtain other JMS objects to send and receive messages. See Using JMS to Send and Receive Messages.
6. Deploy the Java application to Oracle Java Cloud Service - SaaS Extension. See Preparing Applications for Oracle Java Cloud Service - SaaS Extension Deployment.

Resource Management API

The Resource Management API provides functionality to create and manage destinations, message push listeners, and durable subscriptions.

Topics:

- Managing Destinations
- Managing Message Push Listeners
- Managing Durable Subscriptions

Service roles define what messaging resource operations users are authorized to perform in the Oracle Messaging Cloud Service instance. See About Oracle Messaging Cloud Service Roles and Users for information about roles and the privileges associated with each role.

Managing Destinations

This section provides information about managing destinations.

Topics:

Create a Destination

A user with the Messaging Administrator role can create both queues and topics. Names of queues or topics must always consist solely of letters of the Roman alphabet (a through z or A through Z), decimal digits (0 through 9), and underscores ('_'). No other characters are allowed.

Refer to the following for details:

- `MessagingService.createQueue()` in *Java API Reference for Oracle Messaging Cloud Service*
- `MessagingService.createTopic()` in *Java API Reference for Oracle Messaging Cloud Service*

See Create Resources for a code sample.

Delete a Destination

A user with the Messaging Administrator role can delete queues and topics.

Deleting a destination is a non-blocking operation. For more information, refer to About Destination Deletion.

> **Note:**
>
> Deleting a destination will permanently delete all undelivered messages currently residing on the destination. It also deletes all message push listeners on the destination.
>
> Deleting a topic deletes all durable subscriptions on the topic.

Refer to the following for details:

- `MessagingService.deleteQueue()` in *Java API Reference for Oracle Messaging Cloud Service*
- `MessagingService.deleteTopic()` in *Java API Reference for Oracle Messaging Cloud Service*

List Destinations

A user with the Messaging Administrator role can list existing destinations.

Refer to the following for details:

- `MessagingService.listQueueProperties()` in *Java API Reference for Oracle Messaging Cloud Service*
- `MessagingService.listTopicProperties()` in *Java API Reference for Oracle Messaging Cloud Service*

Retrieve a Destination's Properties

A user with the Messaging Administrator role can get an existing queue or topic's properties.

Refer to the following for details:

- `MessagingService.getQueueProperties()` in *Java API Reference for Oracle Messaging Cloud Service*
- `MessagingService.getTopicProperties()` in *Java API Reference for Oracle Messaging Cloud Service*

Note that for queues, you can also retrieve the queue's backlog size which is the number of messages currently stored in the queue.

You can call the `getQueueProperties(String queueName, boolean returnBacklogStats)` method on a `MessagingService` object to get a `QueueProperties` object. If the `returnBacklogStats` parameter is specified as `true`, the returned `QueueProperties` object will return a `BacklogStats` object when the `getBacklogStats()` method is called. The queue's current backlog count can be obtained by calling the `getCurrent()` method on the `BacklogStats` object.

Managing Message Push Listeners

This section provides information about managing message push listeners. The name of a message push listener must always consist solely of letters of the Roman alphabet (a through z or A through Z), decimal digits (0 through 9), and underscores ('_'). No other characters are allowed.

Topics:

Create a Message Push Listener

A user with the Messaging Administrator or Messaging Worker role can create message push listeners.

Refer to the following for details:

- `MessagingService.createListener()` in *Java API Reference for Oracle Messaging Cloud Service*

See Create Resources for a code sample.

When creating a new message push listener, a user needs to supply objects of the following classes (as needed):

- `Medium`: A reference to an existing queue or topic from which the listener will receive messages. This cannot refer to a temporary queue or topic. Message push listeners cannot be created on temporary queues and topics.
- `DurableSubscription`: A reference to an existing durable subscription from which to receive messages.
- `Selector`: A wrapper that holds a JMS Selector expression. A JMS Selector can be used to limit the messages that a message push listener receives.
- `PushURI`: A superclass to define the target to which messages are pushed. Subclasses include `PushMedium`, which indicates that messages should be pushed to another destination, and `PushURL`, which indicates that messages should be pushed to a URL.
- `FailurePolicy`: A definition of the policy for the message push listener to follow if a failure is encountered when pushing to its configured target. Different failure conditions can trigger different actions defined in `FailureResponse` objects.

Delete a Message Push Listener

A user with the Messaging Administrator or Messaging Worker role can delete existing message push listeners.

Refer to the following for details:

- `MessagingService.deleteListener()` in *Java API Reference for Oracle Messaging Cloud Service*

List Message Push Listeners

A user with the Messaging Administrator or Messaging Worker role can list existing message push listeners. All message push listeners for a service instance are listed by default. Optionally, message push listeners for a specific queue or topic can be listed.

For more information, refer to `MessagingService.getListenerNames()` in *Java API Reference for Oracle Messaging Cloud Service.*

Retrieve a Message Push Listener's Properties

A user with the Messaging Administrator or Messaging Worker role can get the definitions of an existing message push listener.

Refer to the following for details:

- `MessagingService.getListener()` in *Java API Reference for Oracle Messaging Cloud Service*

Managing Durable Subscriptions

Durable subscriptions are created and deleted using standard JMS 1.1 mechanisms.

Topics:

- List Durable Subscriptions
- Retrieve a Durable Subscription's Properties

Note that the Oracle Messaging Cloud Service Java library provides two additional `createDurableSubscriber()` methods that are not part of the JMS 1.1 interface. See Safe Durable Subscriptions for information about how to obtain durable subscribers safely.

List Durable Subscriptions

A user with the Messaging Administrator or Messaging Worker role can list existing durable subscriptions. Durable subscriptions can be listed for the entire service instance, a specific client ID, or a specific topic.

Refer to the following for details:

- `MessagingService.getAllDurableSubscriptions()` in *Java API Reference for Oracle Messaging Cloud Service*
- `MessagingService.getDurableSubscriptionsByClientID()` in *Java API Reference for Oracle Messaging Cloud Service*
- `MessagingService.getDurableSubscriptionsByTopic()` in *Java API Reference for Oracle Messaging Cloud Service*

Retrieve a Durable Subscription's Properties

A user with the Messaging Administrator or Messaging Worker role can get the properties of an existing durable subscription.

For details, refer to `MessagingService.getDurableSubscription()` in the *Java API Reference for Oracle Messaging Cloud Service*.

ConnectionFactory Creation API

This topic provides information about the `ConnectionFactory` object.

In the Oracle Messaging Cloud Service Java library, a JMS `ConnectionFactory` object is obtained from a `MessagingService` object by invoking the `getConnectionFactory()` method on the `MessagingService` object. The object returned has class `oracle.cloud.messaging.client.OracleCloudConnectionFactory`, which is the Oracle Messaging Cloud Service extension of `ConnectionFactory`. Note that this is different from the suggested method of obtaining `ConnectionFactory` objects from an instance of the Java Naming and Directory Interface (JNDI) as specified in the JMS 1.1 standard. Oracle Messaging Cloud Service does not currently support JNDI access to `ConnectionFactory` objects.

The `MessagingService` class also has methods `getQueueConnectionFactory()` and `getTopicConnectionFactory()`. These methods are provided purely for backward compatibility with JMS 1.0, in which separate connection factory classes are implemented for queues and topics. The separate connection factory methods should not be used unless it is necessary to use Oracle Messaging Cloud Service with legacy JMS 1.0 code that requires the use of `QueueConnectionFactory` or `TopicConnectionFactory` objects.

The Java library includes methods to create `ConnectionFactory`, `QueueConnectionFactory`, and `TopicConnectionFactory` objects that set a fixed client ID on the `Connection`, `QueueConnection`, or `TopicConnection` objects that they create. If such a `ConnectionFactory` object is created, any Connection it creates must be closed before the `ConnectionFactory` can be used to create another Connection, since there can only be one Connection with a given client ID.

> **Note:**
>
> Connection objects that you create via the Java library should be closed if you are not going to use them further. This is because Connection objects consume computing and network resources on both the client and the server side, and failing to close them wastes these resources.
>
> References to Connection objects that have not been closed should not just be discarded, with the Connection being taken care of by garbage collection. If resources held by Connections are allowed to be released by garbage collection then there may be a substantial delay between when a Connection reference is discarded and when garbage collection occurs. This causes server and network resources to be wasted.

ConnectionFactory Control of Thread Pools

An API has been added to `OracleCloudConnectionFactory` to control the client-side use of thread pools. Some implementations of this interface support setting a thread pool (an instance of `java.util.concurrent.Executor`) for the `ConnectionFactory`. This thread pool is used to obtain threads to execute concurrent operations like processing asynchronously received Message objects via a `MessageListener`.

By default, the thread pool is of the sort returned by `java.util.concurrent.Executors.newCachedThreadPool()`. This is a pool that is initially empty, creates new threads as needed, and re-uses previously started threads in the pool. When a thread is returned to the pool, it remains available for re-use for 60 seconds, after which it is terminated. The threads managed by the default thread pool are all daemon threads.

Connections that are created while a `ConnectionFactory` is using a given thread pool will continue to use threads from that thread pool for concurrent operations. For example, if a `ConnectionFactory` is created, and is used to create five Connections, and the ConnectionFactory's thread pool is then set to a different thread pool, future Connections will use the new thread pool, but the first five Connections will use the default thread pool.

Oracle Cloud Messaging Service implementations of JMS that support setting thread pools will interact with those thread pools solely via the `java.util.concurrent.Executor` interface. They will not shut down or otherwise

manage such thread pools. If a client sets the thread pool of a `ConnectionFactory`, it must do any configuration, startup, shutdown, or other management itself. When a thread pool is used to execute a concurrent operation, if the attempt to start that operation in a thread throws an exception (for example, because the thread has a maximum pool size, and that size has been reached), it will cause a JMSException to be thrown by the JMS operation that attempted to start the concurrent operation (for example, `setMessageListener()` on a `MessageConsumer`). Implementations of this interface that support setting thread pools and are serializable will re-initialize the thread pool to the default thread pool when de-serialized.

Using JMS to Send and Receive Messages

Once a `ConnectionFactory` has been created from the `MessagingService` object, you can use the standard JMS 1.1 API operations for sending and receiving messages.

Refer to Java Message Service Concepts in *Java EE 6 Tutorial* if you need information about how to use the JMS 1.1 API.

> **Note:**
>
> There are several differences between using JMS from the Oracle Messaging Cloud Service Java library and other on-premises environments. See Differences from JMS for a complete list.

Refer to the following for code samples:

- Send a Message to a Topic
- Receive a Message from a Queue with an Optional Selector
- Asynchronously Receive Messages with a Durable Subscription
- Asynchronously Process Messages Within a Transaction
- Create Resources
- Use Message Groups

Using Extensions to the JMS API

The Oracle Messaging Cloud Service Java library extends the Java Message Service (JMS) 1.1 interface.

Topics:

- Safe Durable Subscriptions
- Strong Typing for JMS
- Connection Timeout
- Obtaining Service Version
- Obtaining Messaging Context ID

Safe Durable Subscriptions

The Oracle Messaging Cloud Service Java library provides a method to obtain a consumer on a durable subscription safely in the sense that the creation of the consumer will not create a subscription that does not already exist and will not destroy an existing subscription.

In JMS 1.1, a durable subscription is not directly represented as a Java object. What is represented is a **consumer** on a durable subscription. Such consumers are called **durable subscribers**, and are created by invoking one of the two `createDurableSubscriber()` methods on a JMS `Session`.

An invocation of a `createDurableSubscriber()` method is always passed a `Topic` object (representing the topic on which to create a durable subscriber) and a name, which is the name of the subscription. The `Session` object on which the method is invoked must have been created from a `Connection` object that had a client ID set on it prior to the session's creation. One of the `createDurableSubscriber()` methods also takes a selector that will define a subset of messages that will be delivered to the durable subscriber.

A durable subscription is uniquely identified by its associated client ID and subscription name. The topic subscribed to and the selector, if any, are *properties* of the durable subscription.

The first unsafe aspect of the JMS methods is that creating a durable subscription with a given client ID and subscription name will create a durable subscription corresponding to the client ID and subscription if one does not exist when the method is called. If a client means to access an existing durable subscription, but gets the wrong client ID or subscription name (for example, due to a typographical error in code or data), it will simply create and begin to access a new durable subscription. Meanwhile, the durable subscription it means to access may be holding messages that the client will miss.

The second unsafe aspect of the JMS methods is their behavior when a durable subscriber is created on an existing durable subscription, but the topic or selector specified is different from that of the existing durable subscription. In this case, the JMS 1.1 standard specifies that the existing durable subscription be deleted, discarding all messages it is holding, and a new durable subscription be created on the new topic or with the new selector, or both. This may cause messages to be lost if a mistake in code or data causes a client to change the topic or selector of an existing durable subscription inadvertently.

The Oracle Messaging Cloud Service extension of the JMS `Session` interface is `oracle.cloud.messaging.client.OracleCloudSession` has two additional `createDurableSubscriber()` methods that are not part of the JMS 1.1 interface. Each method takes a subscription name, but no `Topic` object or selector. The methods return a consumer on an *existing* durable subscription with the client ID from the `Connection` object and the subscription name supplied to the invocation. If no durable subscription with the specified client ID and subscription name exists, the method will throw an exception.

Strong Typing for JMS

The Oracle Messaging Cloud Service Java library adds several classes that can make code more strongly typed and self-documenting, which can improve code quality.

The use of the Java library classes, and the methods that take them as parameters, is optional, except in the case of a method for safe durable subscriptions, which uses the `LocalMode` class.

The strong typing additions fall into two categories: Java enumeration classes and wrapper classes.

Enumerations

In several methods in the JMS 1.1 API, parameters that indicate one of a small number of alternatives are represented by Java `boolean` or `short` values. In the Oracle Messaging Cloud Service Java library, these parameters can be represented by Java enumeration types.

The enumeration classes are:

- **AcknowledgementMode**: Represents whether the acknowledgement mode of a session is auto-acknowledge, client-acknowledge, or duplicates-ok, and replaces the use of codes of type `short`.
- **DeliveryMode**: Represents whether a message is persistent or non-persistent, and replaces the use of codes of type `short`.
- **LocalMode**: Represents whether a consumer on a topic will receive messages sent through the connection that the consumer uses, and replaces the use of `boolean`.
- **TransactionMode**: Represents whether a session is transacted or not, and replaces the use of `boolean`.

Wrapper Classes

In several methods in the JMS 1.1 API, parameters with specific meanings and limitations on their values are represented by Java types or classes such as `int`, `long`, or `String`.

In the Oracle Messaging Cloud Service Java library, these parameters can be represented by classes that "wrap" the Java types and classes used in JMS 1.1. These "wrapper" classes enforce some of the restrictions on the wrapped values that are required by JMS 1.1, and generally improve the typing discipline of JMS. With the wrapper classes, you can supply `null` to the `send()` methods that take those classes and have the default used. Also, when using the `send()` methods that take the wrapper classes as parameters, you can supply `null` for a parameter and the producer-specified default will be used.

The wrapper classes are:

- **TimeToLive**: Represents the time-to-live of a message, and replaces the use of `long`. Use of this type ensures that inappropriate or disallowed values, such as values less than or equal to 0, cannot be passed. It also allows clients to specify the maximum time-to-live allowed by Oracle Messaging Cloud Service without having to know what that maximum is.
- **Selector**: Represents a selector for a consumer, durable subscription, or message push listener, and replaces the use of `String`. Currently, this class is a simple wrapper for a `String`. In future versions, the wrapper may incorporate syntax checking of the `String` and support programmatic mechanisms for constructing a selector that is guaranteed to be well-formed.

- **Priority**: Represents the priority of a message, and replaces the use of `int`. Use of this type ensures that only `int` values that correspond to a JMS 1.1 priority value can be passed. Oracle Messaging Cloud Service does not currently support priorities other than the default (4), so supplying priority values other than the default will be ignored. Priorities may be supported in a future release.

Connection Timeout

The Oracle Messaging Cloud Service Java library provides methods for controlling a Connection's timeout.

The Oracle Messaging Cloud Service extension of the JMS `Connection` interface is oracle.cloud.messaging.client.OracleCloudConnection. Some implementations of this interface support setting a timeout on the Connection. If a client fails or loses network connectivity to the messaging service, after the timeout has passed, the service will release resources reserved for the client. In particular, if a Connection is created with a client ID set on it, and the client fails or loses network connectivity to the messaging service, that client ID will continue to be reserved (and cannot be set on another connection) until the timeout has passed. As long as the client is running and has connectivity to the service, the timeout should not expire. The timeout is measured in seconds, must be positive, is initialized to 300 (5 minutes), and can be set to a maximum of 900 (15 minutes).

Refer to the following methods for controlling a connection's timeout in the Java library:

- `isTimeoutSupported()` in *Java API Reference for Oracle Messaging Cloud Service*
- `getTimeoutInSeconds()` in *Java API Reference for Oracle Messaging Cloud Service*
- `setTimeoutInSeconds(int)` in *Java API Reference for Oracle Messaging Cloud Service*

Obtaining Service Version

The Oracle Messaging Cloud Service Java library provides the `getServerVersion()` method of the `OracleCloudConnection` class to obtain the version of the Oracle Messaging Cloud Service being run by the server to which the client is connected.

Refer to the following for details:

- `getServerVersion()` in *Java API Reference for Oracle Messaging Cloud Service.*

Obtaining Messaging Context ID

The Oracle Messaging Cloud Service Java library provides the `getMessagingContextId()` method of the `OracleCloudConnection` class to obtain the messaging context ID for a given Connection.

Refer to the following for details:

- `getMessagingContextId()` in *Java API Reference for Oracle Messaging Cloud Service.*

Limitations on Message Size and Time-to-Live

Oracle Messaging Cloud Service imposes limitations on the size and time-to-live of messages sent.

The internal, server-side representation of messages may be no larger than 512KB. The internal size correlates roughly with the size of the body for, for example, `BytesMessage` and `TextMessage` objects. For other `Message` subclasses, such as `MapMessage` and `StreamMessage`, the correlation is less precise. Message headers and properties also contribute to the message size. If a message is sent whose internal size exceeds 512KB by more than a certain margin for imprecision, the Java library will throw a `JMSException` whose message will indicate the internal size of the message.

The time-to-live (TTL) of a message may not be more than two weeks, plus a certain margin for clock skew between the client and server machines' clocks. A TTL of 0 is interpreted as the maximum, two-week TTL rather than an infinite TTL. This limitation is enforced at sending time; setting a producer's default TTL to a value that is too large will not throw an exception, but sends via that **producer** will throw a `JMSException`.

5

Troubleshooting Oracle Messaging Cloud Service

This section describes common problems that you might encounter when using Oracle Messaging Cloud Service and provides tips on possible solutions.

Topics:

Java Library

The following troubleshooting tips pertain to the Java library.

I am unable to use JNDI or message-driven beans

The Oracle Messaging Cloud Service Java library currently cannot be used with the Java Naming and Directory Interface (JNDI) or messaging-driven beans (MDBs).

I see a lot of threads being created or network connections being made from machines running the Java library

Make sure you are explicitly calling the `close()` method on all Connection objects you create, rather than discarding references to them and allowing the objects to be reclaimed by garbage collection. The Java library's Connection objects have resources, both threads and network connections, associated with them; discarding references to Connection objects without calling `close()` may cause significant delays in those resources being reclaimed.

Messages

The following troubleshooting tips pertain to sending and receiving messages.

I am not receiving messages; I send messages, but I never receive them

Remember that you must **start** a connection before you can receive messages through it. When using the Java library, remember to call the `start()` method on the connection object. When using the REST API, remember to pass the HTTP header `X-OC-Action: start` to the connection resource.

I am unable to create a durable subscription or subscriber to a topic

Make sure that the client ID for the durable subscription is not being used by any of the following:

- Another JMS connection created using the Java library
- A connection created using the REST API
- A message push listener

I am receiving an error stating that I am sending a message that is too large

Reduce the size of the message you are sending. Consider using Oracle Storage Cloud Service to store the large message content and send a reference to the stored content in a message. You may also opt to break the message into smaller pieces and use message properties or correlation IDs to indicate grouping.

> **Note:**
>
> For all service instances, the maximum size of a message is 512KB.

I am receiving an error stating that a destination has reached its limit of messages

If the destination is a queue, remove messages from the queue at a faster rate by adding more consumers, or slow the rate of sending.

If the destination is a topic, look for slow consumers.

> **Note:**
>
> You cannot purchase any additional queues, topics, or durable subscriptions for a given service instance. All service instances have a fixed limit of messaging resources. See About Resource Limits for the maximum number of resources in paid and trial service subscriptions.

I lost the HTTP cookie associated with a messaging context

Ensure that you store the `JSESSIONID` cookie in persistent storage for a high availability client. The messaging context and its associated connections, sessions, producers, consumers, queue browsers, and temporary destinations will remain open on the server until the messaging context expires.

> **Note:**
>
> If you lose the `JSESSIONID` cookie, you lose the ability to access the associated messaging context and all connections, sessions, producers, consumers, queue browsers, and temporary destinations created from it.

My messages are being redelivered

If you receive messages in a client-acknowledge mode session that are subsequently redelivered, be sure to acknowledge the messages (or, in the REST API, the session through which they were received).

If you receive messages in a transacted session that are subsequently redelivered, be sure to commit the session through which the messages were received.

> **Note:**
>
> If a client-acknowledge mode session or a transacted session is closed (or, in the REST API, has its messaging context expired), any messages received through that session that are unacknowledged or uncommitted will become available for redelivery.

My messages are not appearing at the target destination

If your message was sent through a transacted session, be sure to commit the session before the session expires or is closed, or the sent messages will be lost.

My messages are not being received by a consumer on a queue

Check if there is another consumer on the same queue, or a message push listener that is receiving messages from that queue. A message on a queue will only be delivered to one consumer.

I am receiving messages whose `Content-Type` does not match my client's `Accept` header

The REST API ignores the `Accept` header of requests to receive a message. Use message properties and selectors to ensure that you do not receive messages of a type your client cannot handle.

I am unable to use message selectors

Ensure that you are using the correct JMS selector syntax defined in the JMS specification. For the syntax of selectors, see the **Message Selectors** section of the Java API reference for the `javax.jms.Message` class.

Destinations

The following troubleshooting tips pertain to destinations.

I am receiving errors stating that my service instance does not have any available queues, topics, temporary queues, temporary topics, or durable subscriptions

Ensure that you delete any unneeded queues, topics, temporary queues, temporary topics, or durable subscriptions.

> **Note:**
>
> You cannot purchase any additional queues, topics, temporary queues, temporary topics, or durable subscriptions. All service instances have a fixed limit of messaging resources. See About Resource Limits for the maximum number of resources in paid and trial service subscriptions.

Miscellaneous

The following troubleshooting tips pertain to scenarios not covered in other sections.

My connections are not released after I stop using the Java library or REST API

A connection created through the REST API is not released until it is deleted or its messaging context expires. If a connection is created through the Java library, and the client that created it crashes without invoking its `close()` method, the connection will not be released until its timeout has expired. Ensure that you always delete or close connections when they are no longer in use.

Refer to the following methods for controlling a connection's timeout in the Java library:

- `isTimeoutSupported()` in *Java API Reference for Oracle Messaging Cloud Service*
- `getTimeoutInSeconds()` in *Java API Reference for Oracle Messaging Cloud Service*
- `setTimeoutInSeconds(int)` in *Java API Reference for Oracle Messaging Cloud Service*

I am receiving the error message "Missing or incorrect X-OC-ID-TOKEN"

You must send the `X-OC-ID-TOKEN` header on every HTTP request to the REST API. You may also disable the checking of this token. See Understanding Anti-CSRF Measures.

I am receiving an error stating that my service instance does not have any available connections

Ensure that your connections are being used effectively by storing and sending the `JSESSIONID` cookie with the REST API and closing connections from the Java library. Delete any unneeded message push listeners. You may also opt to buy more connections.

I am getting a "404 Not Found" response when I try to access connections, sessions, producers, consumers, or queue browsers that I just created

Be sure to store the `JSESSIONID` cookie sent in HTTP responses and send it back in subsequent responses. If you do not do this, connections, sessions, producers, consumers, and queue browsers created by previous requests will be inaccessible by subsequent responses.

Check the value of the Maximum Inactive Interval (MII) to ensure that your REST API messaging context is not expiring before you use it. The default value of the MII is 5 minutes, and it can be set to at most 15 minutes. If your application needs to create and hold messaging contexts through the REST API for more than 15 minutes, send a "heartbeat" request periodically to keep the messaging context from expiring. For example, a `GET` request to `/maxInactiveInterval` keeps the messaging context alive without side effects.

I am receiving an error message of the form "Internal error; log reference: <*pseudorandom string*>"

This is an indication that an internal exception was thrown. Details of the exception can be found in the server log, in a log entry that can be found by searching for the <*pseudorandom string*>. Note that the server log can only be accessed by Oracle Support. The error message, including the pseudorandom string, should be included in any communication about the error to Oracle Support.

A

Best Practices

This topic provides information on best practices for using the Oracle Messaging Cloud Service effectively.

Topics:

Learn JMS 1.1

The Java library provided in Oracle Messaging Cloud Service implements the JMS 1.1 API.

Before using the Java library in Oracle Messaging Cloud Service for creating applications, it is recommended that you familiarize yourself with the JMS 1.1 standard or the JMS 1.1 Javadocs. This will help you get a better understanding of important topics such as `ExceptionListener`.

Effective Pooling of Resources

It is recommended to avoid constant construction and destruction of JMS objects.

JMS applications are intended to set up connections, sessions, producers, and consumers, and hold them while processing.

It is recommended not to repeatedly perform actions such as:

- Sending a message by creating a connection, creating a session, creating a producer, sending the message, and then destroying those objects.
- Receiving a message by creating a connection, creating a session, creating a consumer, doing a receive, and then destroying those objects.

If your application requires such actions to be performed repeatedly, the application should try to avoid creating and destroying JMS objects, and should instead hold objects in pools and re-use them. Note that connections are the most expensive objects to create, followed by sessions, consumers, and producers.

Using Transacted and/or Client-Acknowledged Sessions

In some cases it may be critical to applications that messages not be lost, in such cases it may be desirable to use transacted and/or client-acknowledged sessions. However, frequent acknowledgements for each and every operation should be avoided.

Requestor Sessions

It is important that different requesters use different sessions. Acknowledging one message acknowledges all currently unacknowledged messages received in the same session.

If multiple requesters use the same session, and one receives its response successfully, while another sees a failure and wishes to recover and re-try, if the first acknowledges before the second can recover, the second requester's message will be acknowledged and purged from Oracle Messaging Cloud Service, and the second requester's re-try will receive a null message.

Responder Sessions

It is important that different responders use different sessions. If multiple responders use the same session, and one has received a request, processed it, and sent the response while another responder has only received a request, committing the session will commit both the first responder's request and response and the second responder's request.

Diagnosing Exceptions in the Java Library

The Oracle Messaging Cloud Service Java library provides support for diagnosing problems that are indicated by a JMSException (or subclass thereof) being thrown by an invocation of a Java library method or being dispatched to an `ExceptionListener`.

The Java library is an implementation of JMS 1.1. As such, the problem indicated by an exception can sometimes be inferred from the class of the exception. For example, if an invocation of the `setClientID()` method of the `Connection` class throws `InvalidClientIDException`, the JMS standard and Javadoc indicates that this may indicate that there may be an existing Connection with the same client ID. If you receive this exception, you should implement logic to check for and handle this case, or you should modify the application’s code to eliminate such a possibility.

In some cases, the JMS 1.1 specification may not define the problem indicated by an exception. This will be true, for example, if the exception thrown is of the generic class `JMSException`, or if the exception is a `MessagingException` thrown from a (non-JMS) method of the `MessagingService` interface. In such cases, Oracle Messaging Cloud Service may provide additional information about the problem behind the exception.

Subclasses of MessagingException for Specific Problems

In some cases, a non-JMS method may be declared to throw `MessagingException`, and documented to throw subclasses of `MessagingException` to indicate specific problems. When this is the case, it is documented in the Java library Javadocs.

Examples:

- The `createQueue()` method of `MessagingService` creates a persistent queue with a particular name. It can throw a generic `MessagingException`, but it can also throw an exception of the `DestinationExistsException` subclass, which indicates that there is already a queue with the given name in the service instance.
- The `getTopicProperties()` method of `MessagingService` returns information about a persistent topic with a particular name. It can throw a generic `MessagingException`, but it can also throw an exception of the `DestinationNotFoundException` subclass, which indicates that there is no topic with the given name in the service instance.

HttpResponseException

The Java library interacts with the Oracle Messaging Cloud Service as a client of the Oracle Messaging Cloud Service REST API. Thus, many of the exceptions thrown by the Java library are caused by receiving an HTTP error response from the REST API. `HttpResponseException` is a subclass of `MessagingException` that provides programmatic access to information provided by an HTTP error response received by the Java library. The Javadocs for this exception give full details. However, here are some key points to be noted:

- The response code of the HTTP response from the REST API can be obtained by invoking the exception's `getResponseCode()` method. This is of particular interest because, if the response code is in the `400-499` range, this generally indicates a problem that the client code must correct, whereas a response code in the `400-499` range indicates a problem on the server side.
- The error key in the HTTP response of the REST API (the `errorCode` part of the Oracle Messaging Cloud Service REST API error response) can be obtained as an `ErrorKey` object by invoking the exception's `getErrorKey()` method. If the return value of this method is not `null`, it will provide additional information beyond what is given by the the HTTP response code.

  Note: If `getErrorKey()` returns null, this indicates that the HTTP response received by the Java library was not generated by Oracle Messaging Cloud Service. In such cases, the response may have been generated by Oracle Cloud infrastructure, such as a load balancer, or it may even have come from some part of the user's infrastructure, such as an HTTP proxy. If `getErrorKey()` returns null, the content of the HTTP response may be examined to attempt to diagnose the problem.
- The error message in the HTTP response of the REST API (the `errorMessage` part of the Oracle Messaging Cloud Service REST API error response) can be obtained by the `getErrorMessage()` method. The `errorMessage` generally contains additional information beyond what is given by the error key.
- The exception message for `HttpResponseException` (that is, the return value of the standard `getMessage()` method) is the full HTTP response, including the status line, headers, and body of the error response. The error message text will be in the exception's message, but it will be embedded in the HTTP response with other information.

HttpResponseExceptions in Non-JMS Methods

When a `MessagingException` is thrown by a non-JMS method (e.g. the `MessagingService` methods), and its class is not one of the subclasses that indicate a specific problem, its class may be `HttpResponseException`. In such a case, it will contain the information alluded to in the previous section.

If a `MessagingException` is thrown whose class is not one of the problem-specific classes or `HttpResponseException`, it generally indicates that the exception was thrown purely on the client side. Users must not assume that a thrown `MessagingException` is an `HttpResponseException` if its class is not one of the problem-specific exception classes.

HttpResponseExceptions in JMS Methods

When a JMSException is thrown by the JMS part of the client library, if it's thrown because of an HTTP error response from Oracle Messaging Cloud Service, its cause (the value returned by the standard `getCause()` method) will be an `HttpResponseException` which can be used in the same way as above. Again, the cause must be checked to ensure that it's an `HttpResponseException`.

Using Exception Listeners

Note that JMS 1.1. only allows an exception to be dispatched to an `ExceptionListener` to indicate some problem that is not already indicated by a JMS method throwing an exception.

Recovery Strategies

When operations fail, throwing an exception, use the guidelines provided in Diagnosing Exceptions in the Java Library to attempt to diagnose the seriousness of the problem. In such situations, you should try to avoid attempting the operation repeatedly, deleting the connection, and recreating objects.

Alternative to Selectors

Selectors are expensive, and they may slow down or delay the process of receiving messages, so they should be avoided unless absolutely necessary.

For example, if you are using a single queue with selectors to receive response messages that satisfy a single specific criteria, it is better to use temporary queues for receiving such responses.

You should avoid using selectors to make a single queue look like multiple queues, for example, to implement priorities. In such cases, it is recommended to have multiple queues, for example, a high-, middle-, and low-priority queue.

B

REST API Reference

This section lists the request parameters that can be sent and the status codes and error responses that can be received by a REST API client of Oracle Messaging Cloud Service.

Topics:

REST API Parameters Reference

The table in this section lists the parameters that can be submitted to the REST API.

The parameters are listed by their names as query string parameters, along with their corresponding HTTP headers and a short general description of their meanings. The query string parameter is always a camel-case identifier beginning with a lowercase letter. The corresponding HTTP header is the query string parameter with `X-OC-` prepended, and with case-change word boundaries replaced by dashes.

> **Note:**
>
> Except as noted in the methods for sending messages, every parameter may be supplied either as a query string parameter or as an HTTP header. If supplied as both, the header value will be ignored.

Query String Parameter/ HTTP Header	Possible Values	Description

Query string parameter:
`ackMode`
 	
HTTP header:
`X-OC-ACK-MODE`
	`auto`, `client`, `dups_ok`	Acknowledgement mode for sessions

Query string parameter:
`action`
 	
HTTP header:
`X-OC-ACTION`
	`acknowledge`, `commit`, `rollback`, `recover`, `start`, `stop`	Action to invoke on a session for transaction commit/rollback or message acknowledgement/ recovery, or action to start or stop a connection

Query String Parameter/ HTTP Header	Possible Values	Description
Query string parameter: `clientId` HTTP header: `X-OC-CLIENT-ID`	String	Client ID for connection/durable subscriptions
Query string parameter: `connection` HTTP header: `X-OC-CONNECTION`	String	Name of the connection
Query string parameter: `correlationId` HTTP header: `X-OC-CORRELATION-ID`	String	Correlation ID for a message
Query string parameter: `deliveryMode` HTTP header: `X-OC-DELIVERY-MODE`	`persistent`, `non_persistent`	Delivery mode for a message
Query string parameter: `destination` HTTP header: `X-OC-DESTINATION`	`/queues/`*`queueName`* `/topics/`*`topicName`* `/temporaryQueues/`*`queueName`* `/temporaryTopics/`*`topicName`*	Destination (queue, topic, temporary queue, or temporary topic)
Query string parameter: `groupId` HTTP header: `X-OC-GROUP-ID`	*escaped value String*	Used to set the `JMSXGroupID` property on the message being sent. This is the name of the message group of which this message is a part, if any. **Note:** If the `JMSXGroupID` property is set as an HTTP request header, it must be set to an escaped value String or a `badParameter` error response will be generated. For more information on escaped value Strings, see About Escaped Value Strings. If the `JMSXGroupID` property is set as a query string parameter, the usual conventions for escaping query string parameters hold.

Query String Parameter/ HTTP Header	Possible Values	Description

Query string parameter:
`groupSeq`
 	
HTTP header:
`X-OC-GROUP-SEQ`
	Integer	Used to set the `JMSXGroupSeq` property on the message being sent. This is the sequence number of the message within the message group specified by the `groupId` parameter. It should be set if, and only if, `groupId` is set. The `X-OC-GROUP-SEQ` header must be set to an integer or a `badParameter` error response will be generated.

Query string parameter:
`localMode`
 	
HTTP header:
`X-OC-LOCAL-MODE`
	`GET_LOCAL`, `NO_LOCAL`	Local mode (whether to receive messages sent to a topic via the same connection through which the messages were sent)

Query string parameter:
`messageIdEnabled`
 	
HTTP header:
`X-OC-MESSAGE-ID-ENABLED`
	`true`, `false`	Whether message IDs are enabled on a producer

Query string parameter:
`messageType`
 	
HTTP header:
`X-OC-MESSAGE-TYPE`
	`PLAIN`, `TEXT`, `BYTES`, `OBJECT`, `HTTP`, `MAP`, `STREAM`	The type of the message

Query string parameter:
`mii`
 	
HTTP header:
`X-OC-MII`
	Integer greater than or equal to 0	Maximum Inactive Interval for the messaging context

Query string parameter:
`replyTo`
 	
HTTP header:
`X-OC-REPLY-TO`
	`/queues/`*`queueName`*
	`/topics/`*`topicName`*
	`/temporaryQueues/`*`queueName`*
	`/temporaryTopics/`*`topicName`*
		Destination to which to direct replies to a message

Query string parameter:
`selector`
 	
HTTP header:
`X-OC-SELECTOR`
	For the syntax of selectors, see the Message Selectors section of the Java API reference for the `javax.jms.Message` class.	Selector for filtering messages

Query String Parameter/ HTTP Header	Possible Values	Description

Query string parameter:
`session`
 	
HTTP header:
`X-OC-SESSION`
	String	Session identifier

Query string parameter:
`subscriptionName`
 	
HTTP header:
`X-OC-SUBSCRIPTION-NAME`
	String	Name of a durable subscription

Query string parameter:
`timeout`
 	
HTTP header:
`X-OC-TIMEOUT`
	Integer strictly greater than 0	Timeout for a blocking receive operation

Query string parameter:
`topicName`
 	
HTTP header:
`X-OC-TOPIC-NAME`
	String	Name of a topic

Query string parameter:
`transacted`
 	
HTTP header:
`X-OC-TRANSACTED`
	`true`, `false`	Whether a session is transacted or not

Query string parameter:
`ttl`
 	
HTTP header:
`X-OC-TTL`
	Integer strictly greater than 0 or `maximum`	A time-to-live of the message

Query string parameter:
`verificationToken`
 	
HTTP header:
`X-OC-VERIFICATION-TOKEN`
	String	The value of the verification token to send with the message push listener verification request.

REST API HTTP Status Codes and Error Messages Reference

This section provides information about the status codes and error messages that can be received by a REST API client of Oracle Messaging Cloud Service.

Topics:

Generic Meanings of HTTP Response Status Codes

The following table lists HTTP response status codes and their meanings:

Response Status Code	Meaning
200 Ok	Successful requests other than creations and deletions.
201 Created	Successful creation of a queue, topic, temporary queue, temporary topic, session, producer, consumer, listener, queue browser, or message.
204 No Content	Successful deletion of a queue, topic, session, producer, or listener.
400 Bad Request	The path info doesn't have the right format, or a parameter or request body value doesn't have the right format, or a required parameter is missing, or values have the right format but are invalid in some way (for example, `destination` parameter does not exist, content is too big, or client ID is in use).
403 Forbidden	The invoker is not authorized to invoke the operation.
404 Not Found	The object referenced by the path does not exist.
405 Method Not Allowed	The method is not one of those allowed for the path.
409 Conflict	An attempt was made to create an object that already exists.
500 Internal Server Error	The execution of the service failed in some way.

Response bodies for status codes greater than or equal to 400 are either empty or contain an error response in JSON or XML format. For more information, see Understanding Error Responses.

Error Keys, Status Codes and Error Messages

This section gives a list of the distinct error responses that can be generated by the REST API.

Topics:

The error responses are listed first by their associated HTTP status code. Within each status code, the error responses are listed by their *key*. The error key is the last component of the error code returned in the response. That is, each error code has the form:

`urn:oracle:cloud:errorcode:messaging:`*`error key`*

For example, the error with key `methodNotAllowed` has error code `urn:oracle:cloud:errorcode:messaging:methodNotAllowed`.

For each error key, we give the associated error message followed by further explanation if the error message is not self-explanatory.

Errors with HTTP Status Code 400 (Bad Request)

This section provides information about errors with HTTP status code `400`.

Error Message	Description
`badAntiCsrf` `Missing or incorrect X-OC-ID-TOKEN.`	The anti-CSRF token is enabled, but the request either did not submit the token in header `X-OC-ID-TOKEN` or submitted a value that does not match the previously generated value.
`badContentType` `Content-Type must be one of the following:` *`list of media types.`*	The method and URL path must have content of a particular media type in its body, and the `Content-Type` header does not match any of the expected media types.

Error Message	Description
badParameter Bad parameter '*query string parameter*'/header '*header name*': '*parameter value*'.	The value submitted for a parameter is malformed (for example, it should be an integer but doesn't parse as an integer, or should specify a destination but doesn't start with /queues/ or /topics/), not in the proper range (for example, a timeout that parses as an integer but is 0 or negative), or has some other syntactic problem. This error does not indicate that the value is well-formed but, for example, refers to an entity that doesn't exist. The error message specifies the parameter, giving both the query string parameter and header names, and the bad value.
badProperty Bad property header '*header name*' with value '*value*'.	The value submitted for a message property via an X-OC-*type*-PROPERTY-*name* header is malformed. The error message specifies the header and value.
badSelector Bad selector: '*selector*'.	A message selector was found to be invalid.
clientIdFailure Client ID '*client ID*' is invalid or in use by a client or listener.	A client ID was submitted that is invalid or is already in use, either by a listener, another client, or by the current client.
clientIdUnsettable Connection state does not allow setting client ID.	The client attempted to set a client ID on the connection after performing an operation that puts the connection in a state where its client ID is no longer settable. This includes creating a session and any other (successful) operation that requires the prior creation of a session.
connectionParameterNotFound Connection *'connection name'* does not exist.	The client attempted to create a session with a connection name that is not the name of an existing connection.

Error Message	Description
destinationParameterNotFound Destination '*destination name*' of type '*queue or topic*' does not exist.	A queue or topic submitted in the request does not exist. This key is used when the destination is *not* the resource specified in the URL path, but rather when it is specified as, for example, a `destination` parameter or `X-OC-DESTINATION` header.
disallowedSubscriptionLookup Disallowed combination of parameters: *submitted parameters.*	The method to list durable subscriptions and their properties specified a combination of parameters that is not one of the supported combinations. The error message will give a space-delimited list consisting of some collection of the strings `subscriptionName`, `clientId`, and `topicName`.
durableSubscriberOnTemporaryTopic A consumer cannot be created on a temporary topic.	An attempt was made to create a consumer on a temporary topic that uses a durable subscription. Durable subscriptions are not allowed on temporary topics.
forbiddenContentType The Content-Type header has value '*Content-Type value*', which is not allowed for the requested operation.	A send request was made whose `Content-Type` header had value `application/x-www-form-urlencoded`. This content type is not allowed in send requests, as the combination of the POST method and this content type may cause the web server to consume the content, attempting to parse it as if it were a `<form>` submission from an HTML browser, making the content unavailable to be put into the message.
forbiddenParameter Parameter '*query string parameter*'/header '*header name*' not allowed with method '*method*' on path '*URL path*'.	A parameter was submitted that is not allowed with a given method and URL path. The error message gives both the query string parameter and the header name for the parameter.
incompleteGroupProperties Exactly one of the JMSXGroupID and JMSXGroupSeq properties was set on the message. Either both properties must be set, or neither must be set.	One of the mandatory parameters was not set when sending messages using message groups.

Error Message	Description
`invalidPath` `Invalid path: '`*`URL path`*`'.`	The URL path of a request is not a supported path or is malformed. Specific problems include the following: • The URL does not contain the path component for the service name and identity domain name. • The component of the request path after the service name/identity domain name component is not one of those handled by the service. • Some path component after the service and identity domain names contains characters other than a-z, A-Z, decimal digits, or underscores ('_').
`localModeNonTopic` `Local mode on non-topic.`	A request specified a local mode (whether messages sent to a destination via this connection should be received by consumers on this connection) with a destination that is not a topic (in which case a local mode is not appropriate).
`malformedAcceptHeader` `Invalid Accept header: '`*`Accept header`*`'.`	The service was unable to process the `Accept` header of a request to determine whether the client can accept a response with content of a given media type.
`malformedDestination` `Malformed destination: '`*`destination`*`'.`	A destination specification did not have the proper syntax (for example, it did not start with one of the following forms): • `/queues/` • `/topics/` • `/temporaryQueues/` • `/temporaryTopics/`
`malformedListener` `The XML for the message push listener is malformed; the malformed XML is as follows: '`*`listener XML`*`'.`	The message push listener specification did not have the proper syntax. For examples of well-formed XML and the proper syntax to use, see Create a Listener.

Error Message	Description
maxConnectionsReached The attempted operation could not be completed because the service instance is currently using all of its '*maximum number of concurrent connections*' available connections.	An operation has been attempted that would create a new JMS connection (sending a message, receiving a message, setting the client ID on an connection, and so on), and the service instance is already at the maximum number of concurrent connections it's allowed.
maxDurableSubscriptionsReached The requested durable subscription could not be created because the service instance has reached its maximum number of durable subscriptions. This service instance may have no more than '*maximum number of durable subscriptions*' durable subscriptions.	Self-explanatory.
maxLocalConnectionsReached The attempted operation could not be completed because the service instance may have exceeded its available connections.	The service instance has exceeded the number of connections it can create on a single virtual machine in the cloud. This usually means that the service instance has reached, or even gone beyond, the maximum number of allowed connections.
maxMessagesOnTargetDestinationReached The message could not be sent because the targeted destination reached its maximum number of messages. Each destination on this service instance may not have more than *'hard quota on number of messages'* messages. The number of messages is currently *'current backlog size'*. The number of messages must drop below *'soft quota on number of messages'* before further sends are allowed.	Self-explanatory.
maxMessageBytesOnTargetDestinationReached The message could not be sent because the targeted destination reached its maximum number of bytes of messages. Each destination on this service instance may not have more than *'hard quota on bytes'* bytes of messages. The number of bytes of messages is currently *'current backlog bytes'*. The number of bytes of messages must drop below *'soft quota on bytes'* before further sends are allowed.	Self-explanatory.

Error Message	Description
maxQueuesReached The requested queue could not be created because the targeted service instance has reached its maximum number of queues. This service instance may have no more than *'maximum number of queues'* queues.	Self-explanatory.
maxTempDestinationsOnConnectionReached The requested temporary destination could not be created because the targeted service instance has reached its maximum number of temporary destinations for this connection. Each connection on this service instance may not have more than *'maximum number of temporary destinations'* temporary destinations.	Self-explanatory.
maxTopicsReached The requested topic could not be created because the service instance has reached its maximum number of topics. This service instance may have no more than *'maximum number of topics'* topics.	Self-explanatory.
messageHeadersTooLarge The size of the messaging-relevant headers of the send request exceeded the maximum, *maximum header size*.	The request's message-relevant headers exceeded the maximum size.
messageBodyTooLarge The size of the body of the send request exceeded the maximum, *maximum body size*.	The request's body exceeded the maximum size.
messageTotalTooLarge The internal message representation has size *internal size*, which exceeds the maximum internal size, *maximum internal size*.	A message whose headers and body were within the limitations exceeded the JMS broker's threshold for the size of the internal representation of a message.
messagePushListenerVerificationBadResponse Verification that the endpoint '*<URL of an HTTP/S endpoint>*' is willing to receive messages failed. The endpoint's response body did not match the challenge token.	An HTTP or HTTPS endpoint responded to a verification request with a response body that did not match the challenge token sent by Oracle Messaging Cloud Service.

Error Message	Description
messagePushListenerVerificationConnectionFailed Verification that the endpoint '*<URL of an HTTP/S endpoint>*' is willing to receive messages failed. The service instance could not connect to the endpoint.	Oracle Messaging Cloud Service was unable to connect to an HTTP or HTTPS endpoint to send a verification request.
messagePushListenerVerificationErrorResponse Verification that the endpoint '*<URL of an HTTP/S endpoint>*' is willing to receive messages failed. The endpoint's response had status *<non-200 status>*.	An HTTP or HTTPS endpoint responded to a verification request with a status code other than 200.
messagePushListenerVerificationException Verification that the endpoint '*<URL of an HTTP/S endpoint>*' is willing to receive messages failed. An exception occurred in attempting to read the response.	An exception occurred in attempting to read the response to a verification request.
messagePushListenerVerificationNoToken The message push listener had an HTTP/S target, but no verification token was supplied.	The listener XML specifies at least one HTTP or HTTPS URL to which to push messages, but no `verificationToken` was supplied.
missingParameter Missing parameter '*query string parameter*'/ header '*header name*'.	A parameter that is required for the method and URL path of the request was not supplied, either as a query string parameter or a header. The error specifies the query string parameter and header name for the missing parameter.
multipleDestinations No destination or multiple destinations specified.	A send via a producer was requested, but either the request specified no destination and the producer had no default destination or the request specified a destination and the producer had a default destination.
noContentType No Content-Type.	A request was made that must have a specific media type, but no `Content-Type` header was supplied.
noDestinationForConsumer Neither a destination nor a subscription name were specified.	A request was made to create a consumer, but no destination was specified, and no existing durable subscription (from which a destination could be extracted) was specified.

Error Message	Description
nonexistentNamespace Nonexistent namespace: '*namespace identifier*'.	The namespace (specified by a service name and an identity domain name) specified for the request did not exist.
nonexistentNamespaceComponents There is no service instance with service name '*service name*' and identity domain name '*identity domain name*'.	The namespace specified for the request did not exist.
nonexistentNamespaceUnknown There is no service instance with the specified service and identity domain name.	The namespace specified in the request URL did not exist.
serviceInstanceChanged The client has attempted to use the service instance '*first path component*' with a messaging context that was started with service instance '*original first path component*'.	The client request that created the current messaging context specified a service instance by using a URL path whose first component had the form *service name of the instance-identity domain name of the instance* but the current request has supplied a different first component on its URL path. The service instance may not be changed for a given messaging context. If a REST client wishes to access multiple service instances simultaneously, it must use multiple messaging contexts with different JSESSIONID values.
sessionParameterNotFound Session '*session name*' does not exist.	This error is generated when a nonexistent session is specified as a query string parameter or a header, rather than via the URL path.
subscriptionInUse The durable subscription with client ID *'client ID'* and name *'subscription name'* has a consumer on it.	This error is generated in two circumstances: • An attempt is made to create a consumer on a durable subscription when that durable subscription already has a consumer on it. • An attempt is made to delete a durable subscription that has a consumer on it.

Error Message	Description
subscriptionNonTopic Durable subscription on non-Topic.	An attempt has been made to create a consumer with a durable subscription with a specified destination that is not a topic. This error is generated when it is possible to determine from the request alone that the destination is not a topic (for example, if the parameter specifying the consumer starts with /queues/).
subscriptionNotFoundNoInfo Durable subscription does not exist.	A durable subscription was not found, but no information about that durable subscription was available.
subscriptionParameterNotFound Durable subscription with client ID '*client ID*' and name '*subscription name*' does not exist.	This error is generated when the nonexistent subscription is specified as a query string parameter or a header, rather than via the URL path.
timeoutTooLarge Receive timeout (*requested timeout* milliseconds) is larger than the maximum allowed (*service maximum timeout* milliseconds).	The value of the timeout parameter was larger than the maximum allowed value (5 minutes).
timeToLiveTooLarge Time to live (*requested time-to-live* milliseconds) is larger than the maximum allowed (*service maximum time-to-live* milliseconds).	A requested time-to-live, either for a producer being created, a producer being modified, or a message being sent, is longer than the maximum allowed (2 weeks).

Errors with HTTP Status Code 403 (Forbidden)

This section provides information about errors with HTTP status code 403.

Error Message	Description
adminRequired Administrator authorization required.	Self-explanatory.
noRoles User must be either messaging worker or messaging admin.	The user has authenticated, but has neither of the roles (Messaging Administrator or Messaging Worker) required for any of the messaging operations.

Errors with HTTP Status Code 404 (Not Found)

All errors in this category are returned when a resource specified on the URL path does not exist.

Error Message	Description
`connectionNotFound` `Connection 'connection name' does not exist.`	Self-explanatory.
`consumerNotFound` `Consumer 'consumer name' does not exist.`	Self-explanatory.
`destinationNotFound` `Destination 'name' of type 'queue or topic' does not exist.`	Self-explanatory.
`listenerNotFound` `Message push listener 'listener name' does not exist.`	Self-explanatory.
`producerNotFound` `Producer 'producer name' does not exist.`	Self-explanatory.
`queueBrowserNotFound` `Queue browser 'queue browser name' does not exist.`	Self-explanatory.
`sessionNotFound` `Session 'session name' does not exist.`	Self-explanatory.
`subscriptionNotFound` `Durable subscription with name 'subscription name' does not exist.`	An attempt has been made to delete a nonexistent subscription via a session.
`subscriptionNotFoundFull` `Durable subscription with client ID 'client ID' and name 'subscription name' does not exist.`	As attempt has been made to retrieve the properties of a nonexistent durable subscription.

Errors with HTTP Status Code 405 (Method Not Allowed)

This section provides information about errors with HTTP status code `405`.

Error Message	Description
methodNotAllowed Method '*method*' not allowed on path '*URL path*'.	Self-explanatory.

Errors with HTTP Status Code 406 (Not Acceptable)

This section provides information about errors with HTTP status code 406.

Error Message	Description
unacceptable Client's Accept header '*Accept header*' cannot accept any of the following content types: *space-separated list of content types.*	Self-explanatory.

Errors with HTTP Status Code 409 (Conflict)

This section provides information about errors with HTTP status code 409.

Error Message	Description
connectionAlreadyExists Connection *'connection name'* already exists.	Self-explanatory.
consumerAlreadyExists Consumer '*consumer name*' already exists.	Self-explanatory.
destinationAlreadyExists Destination '*destination name*' of type '*queue or topic*' already exists.	Self-explanatory.
listenerAlreadyExists Message push listener '*listener name*' already exists.	Self-explanatory.
producerAlreadyExists Producer '*producer name*' already exists.	Self-explanatory.
queueBrowserAlreadyExists Queue browser *'queue browser name'* already exists.	Self-explanatory.

Error Message	Description
sessionAlreadyExists Session '*session name*' already exists.	Self-explanatory.
subscriptionAlreadyExists A durable subscription with client ID '*client ID*' and name '*subscription name*' already exists.	Self-explanatory.

Errors with HTTP Status Code 500 (Internal Server Error)

This section provides information about errors with HTTP status code 500.

Error Message	Description
failedGetConnectionProps Failed to get Connection properties.	A failed attempt was made to get properties related to connections to the JMS broker.
failedGetDestinationConnectionProps Failed to get connection properties for destination '*destination name*' of type '*queue or topic*'.	A failed attempt was made to get properties related to connections to the JMS broker for a specific destination.
failedGetDestinationProps Failed to get properties for destination '*destination name*' of type '*queue or topic*'.	A failed attempt was made to get properties related to a specific destination.
failedGetServiceProps Failed to get Service properties.	A failed attempt was made to get properties related to a specific service instance.
maxConnectionCountUnavailable The maximum number of connections allowed for the service instance cannot be determined.	An internal error has occurred in determining the number of connections that a service instance is allowed.
messagePushListenersInterrupted Message push listener functionality was unexpectedly interrupted. Please try again.	Self-explanatory.

Error Message	Description
`messagePushListenerVerificationRedirectionDisableFailed`	A failed attempt was made to disable HTTP redirects for the message push listener verification request.

`operationFailed`
 	
`Operation failed.`
	A low-level failure occurred in attempting to carry out the latest request.

C

Code Samples

The code examples in this section demonstrate how to send and receive messages through the REST API and the Java library when developing applications that use Oracle Messaging Cloud Service.

Topics:

REST API

Note that the code samples provided in this section do not contain all HTTP headers, or responses. The code samples contain HTTP headers, and responses which are relevant to the given example.

In the examples, request parameters are provided as query string parameters. Note that HTTP headers can also be provided as request parameters.

Topics:

Create a Queue

The example shows an HTTP request and response for a queue named `Preprocessing` being created.

Request to create a queue named `Preprocessing`.

```
PUT /MCSService03-MCSOracle3/api/v1/queues/Preprocessing HTTP/1.1
X-OC-ID-TOKEN-STATUS: disabled
Authorization: Basic YXd1c2VyOldlbGNvbWVfMQ==
Accept: application/json, application/xml;q=0.8, */*;q=0.5
```

Note:

- The `X-OC-ID-TOKEN-STATUS` header indicates that the anti-CSRF token is disabled.
- The `Authorization` header indicates that the authentication type is `Basic`.
- The `Accept` header accepts JSON (most preferable), or XML (second preference), or anything else (least preferable).

Successful response.

```
HTTP/1.1 201 Created
Cache-Control: no-cache;no-store;must-revalidate;max-age=0
Content-Length: 0
X-OC-NEW-MESSAGING-CONTEXT: true
Set-Cookie:
JSESSIONID=ppNgTrjQLWY1ZgSfDyq1JWQh5px13gxYqbhJvvyPLrd2G2TCpl82!-1133198190; path=/;
HttpOnly
```

Note:

- A queue named `Preprocessing` is created by the request.
- The `Cache-Control` header indicates that the response should not be cached.
- The `X-OC-NEW-MESSAGING-CONTEXT` header indicates that the request created a new messaging context.
- The Set-Cookie header provides the `JSESSIONID` cookie that must be sent in future requests in order to use the same connection.

Create a Topic

The example shows an HTTP request and response for a topic named `Incoming` being created.

Request to create a topic `Incoming`.

```
PUT /MCSService03-MCSOracle3/api/v1/topics/Incoming HTTP/1.1
X-OC-ID-TOKEN-STATUS: disabled
Authorization: Basic YXd1c2VyOldlbGNvbWVfMQ==
Accept: application/json, application/xml;q=0.8, */*;q=0.5
```

Note:

- The `X-OC-ID-TOKEN-STATUS` header indicates that the anti-CSRF token is disabled.
- The `Authorization` header indicates that the authentication type is `Basic`.
- The `Accept` header accepts JSON (most preferable), or XML (second preference), or anything else (least preferable).

Successful response.

```
HTTP/1.1 201 Created
Cache-Control: no-cache;no-store;must-revalidate;max-age=0
Content-Length: 0
X-OC-NEW-MESSAGING-CONTEXT: true
Set-Cookie:
JSESSIONID=CsSxTrjR50JDJS0vqdFfrKCQFdTwJsyTmVkC6gfsbLXVtZFyHfkG!-1133198190; path=/;
HttpOnly
```

Note:

- A topic named `Incoming` is created by the request.

- The `Cache-Control` header indicates that the response should not be cached.
- The `X-OC-NEW-MESSAGING-CONTEXT` header indicates that the request created a new messaging context.
- The Set-Cookie header provides the `JSESSIONID` cookie that must be sent in future requests in order to use the same connection.

Create a Durable Subscription

The example shows an HTTP request and response for a durable subscription named `Audit` being created.

1. Request to create a connection named `conn`, and set the connection's client ID to `cid`.

```
PUT /MCSService03-MCSOracle3/api/v1/connections/conn?clientId=cid&action=start
HTTP/1.1
X-OC-ID-TOKEN-STATUS: disabled
Authorization: Basic YXd1c2VyOldlbGNvbWVfMQ==
Accept: application/json, application/xml;q=0.8, */*;q=0.5
```

Note:

- The `X-OC-ID-TOKEN-STATUS` header indicates that the anti-CSRF token is disabled.
- The `Authorization` header indicates that the authentication type is `Basic`.
- The `Accept` header accepts JSON (most preferable), or XML (second preference), or anything else (least preferable).
- The `clientId` query string parameter sets the connection's client ID to `cid`.
- The `action` query string parameter indicates that the connection should be started.

Successful response.

```
HTTP/1.1 201 Created
Cache-Control: no-cache;no-store;must-revalidate;max-age=0
Content-Length: 0
X-OC-NEW-MESSAGING-CONTEXT: true
Set-Cookie:
JSESSIONID=hFZBTrjSdmx5MdSGyWjwG6bvQLCt8gnfHgfn5cL7HYQy8sSqnLfQ!-1133198190;
path=/; HttpOnly
```

Note:

- A connection named `conn` is created by the request.
- The `Cache-Control` header indicates that the response should not be cached.
- The `X-OC-NEW-MESSAGING-CONTEXT` header indicates that the request created a new messaging context.
- The Set-Cookie header provides the `JSESSIONID` cookie that must be sent in future requests in order to use the same connection.

2. Request to create a session named `s`.

```
PUT /MCSService03-MCSOracle3/api/v1/sessions/s?connection=conn HTTP/1.1
Cookie:
JSESSIONID=hFZBTrjSdmx5MdSGyWjwG6bvQLCt8gnfHgfn5cL7HYQy8sSqnLfQ!-1133198190
```

```
X-OC-ID-TOKEN-STATUS: disabled
Authorization: Basic YXd1c2VyOldlbGNvbWVfMQ==
Accept: application/json, application/xml;q=0.8, */*;q=0.5
```

Note:

- The `connection` query string parameter is set to `conn`, which indicates that the service should use the connection created by the previous request to create the session.
- The `Cookie` header sends the `JSESSIONID` cookie which is associated with the messaging context and its encapsulated ephemeral objects, like connections, sessions, producers, and consumers. This cookie is required for the messaging context and any of its ephemeral objects to be available for the current request.
- The `X-OC-ID-TOKEN-STATUS` header indicates that the anti-CSRF token is disabled.
- The `Authorization` header indicates that the authentication type is `Basic`.
- The `Accept` header accepts JSON (most preferable), or XML (second preference), or anything else (least preferable).

Successful response.

```
HTTP/1.1 201 Created
Cache-Control: no-cache;no-store;must-revalidate;max-age=0
Content-Length: 0
```

Note:

- A session named `s` is created by the request.
- The `Cache-Control` header indicates that the response should not be cached.

3. Request to create a consumer named `c` on a durable subscription.

```
PUT /MCSService03-MCSOracle3/api/v1/consumers/c?session=s&destination=%2Ftopics
%2FIncoming&subscriptionName=Audit HTTP/1.1
Cookie:
JSESSIONID=hFZBTrjSdmx5MdSGyWjwG6bvQLCt8gnfHgfn5cL7HYQy8sSqnLfQ!-1133198190
X-OC-ID-TOKEN-STATUS: disabled
Authorization: Basic YXd1c2VyOldlbGNvbWVfMQ==
Accept: application/json, application/xml;q=0.8, */*;q=0.5
```

Note:

- The `session` query string parameter is set to `s`, which indicates that a consumer `c` will be created in the session `s`.
- The `destination` query string parameter is set to the topic `Incoming`, which indicates that the consumer will consume from the topic `Incoming`.
- The `subscriptionName` query string parameter indicates that the consumer will be a durable subscriber on a durable subscription named `Audit`, and the client ID will be `cid` from the connection `conn`.
- The `Cookie` header sends the `JSESSIONID` cookie which is associated with the messaging context and its encapsulated ephemeral objects, like connections, sessions, producers, and consumers. This cookie is required for the messaging context and any of its ephemeral objects to be available for the current request.

- The `X-OC-ID-TOKEN-STATUS` header indicates that the anti-CSRF token is disabled.
- The `Authorization` header indicates that the authentication type is `Basic`.
- The `Accept` header accepts JSON (most preferable), or XML (second preference), or anything else (least preferable).

Successful response.

```
HTTP/1.1 201 Created
Cache-Control: no-cache;no-store;must-revalidate;max-age=0
Content-Length: 0
```

Note:

- The creation of the consumer will create a durable subscription on the topic `Incoming` with client ID `cid` and name `Audit`, unless a subscription with that client ID and name already exists.

  If a subscription with the specified client ID and name already exists, then the action taken is as follows:

  – If the existing durable subscription is on the topic `Incoming`, and has no selector, then this consumer will consume from the existing durable subscription.

  – If the existing durable subscription is not on the topic `Incoming`, then the existing durable subscription will be deleted (unless there is another consumer on it) and a new durable subscription will be created with the given client ID and name, on the `Incoming` topic, with no selector. Note that when a durable subscription is deleted, all messages that were saved in the existing durable subscription are discarded.

  – If the existing durable subscription is on the topic `Incoming`, but has a selector, the existing durable subscription will be deleted (unless there is another consumer on it) and a new durable subscription will be created with the given client ID and name, on the topic `Incoming`, with no selector. Note that when a durable subscription is deleted, all messages that were saved in the existing durable subscription are discarded.
- The `Cache-Control` header indicates that the response should not be cached.

Create a Message Push Listener

The example shows an HTTP request and response for a message push listener named `l` being created.

Request to create a listener named `l`.

```
PUT /MCSService03-MCSOracle3/api/v1/listeners/l HTTP/1.1
X-OC-ID-TOKEN-STATUS: disabled
Authorization: Basic YXd1c2VyOldlbGNvbWVfMQ==
Content-Type: application/xml
Accept: application/json, application/xml;q=0.8, */*;q=0.5
Content-Length: 286

<?xml version="1.0" encoding="UTF-8"?>
<listener>
    <version>1.0</version>
    <name>l</name>
```

```
    <source>
        <type>topic</type>
        <name>Incoming</name>
    </source>
    <target>
        <uri>urn:oracle:cloud:messaging:queues:Preprocessing</uri>
    </target>
</listener>
```

Note:

- The `X-OC-ID-TOKEN-STATUS` header indicates that the anti-CSRF token is disabled.
- The `Authorization` header indicates that the authentication type is `Basic`.
- The `Accept` header accepts JSON (most preferable), or XML (second preference), or anything else (least preferable).
- The `Content-Type` header indicates that body of the request is in XML.
- The XML body of the request indicates the properties of the listener. The listener receives messages from the topic `Incoming` and sends them to the queue `Preprocessing`.

> **Note:**
>
> The XML document should not contain a `DOCTYPE` declaration. If a `DOCTYPE` declaration is included in the XML document, a `500 operationFailed` response is returned. This is done to prevent certain security and Denial of Service (DoS) attacks.

Successful response.

```
HTTP/1.1 201 Created
Cache-Control: no-cache;no-store;must-revalidate;max-age=0
Content-Length: 0
X-OC-NEW-MESSAGING-CONTEXT: true
Set-Cookie:
JSESSIONID=MLyKTrjTh62Hb1PGkwCFGrrLc5p2sVdl8vv1WD5CzdNXTtppJJCZ!-1133198190; path=/;
HttpOnly
```

Note:

- A listener named `l` is created by the request.
- The `Cache-Control` header indicates that the response should not be cached.
- The `X-OC-NEW-MESSAGING-CONTEXT` header indicates that the request created a new messaging context.
- The Set-Cookie header provides the `JSESSIONID` cookie that must be sent in future requests in order to use the same connection.

Receive a Message from a Durable Subscription

The example shows an HTTP request and response for a message being received from a durable subscription named `Audit`.

1. Request to create a connection named `conn`, and set the connection's client ID to `cid`.

```
PUT /MCSService03-MCSOracle3/api/v1/connections/conn?clientId=cid&action=start
HTTP/1.1
X-OC-ID-TOKEN-STATUS: disabled
Authorization: Basic YXd1c2VyOldlbGNvbWVfMQ==
Accept: application/json, application/xml;q=0.8, */*;q=0.5
```

Note:

- The `X-OC-ID-TOKEN-STATUS` header indicates that the anti-CSRF token is disabled.
- The `Authorization` header indicates that the authentication type is `Basic`.
- The `Accept` header accepts JSON (most preferable), or XML (second preference), or anything else (least preferable).
- The `clientId` query string parameter sets the connection's client ID to `cid`.
- The `action` query string parameter indicates that the connection should be started.

Successful response.

```
HTTP/1.1 201 Created
Cache-Control: no-cache;no-store;must-revalidate;max-age=0
Content-Length: 0
X-OC-NEW-MESSAGING-CONTEXT: true
Set-Cookie:
JSESSIONID=MQm1TrjV2zLv7b0GV5kny1XwBcd7hZv0LhJSWxKvyJthq31jVH7L!-1133198190;
path=/; HttpOnly
```

Note:

- A connection named `conn` is created by the request.
- The `Cache-Control` header indicates that the response should not be cached.
- The `X-OC-NEW-MESSAGING-CONTEXT` header indicates that the request created a new messaging context.
- The Set-Cookie header provides the `JSESSIONID` cookie that must be sent in future requests in order to use the same connection.

2. Request to create a session named `s`.

```
PUT /MCSService03-MCSOracle3/api/v1/sessions/s?connection=conn HTTP/1.1
Cookie:
JSESSIONID=MQm1TrjV2zLv7b0GV5kny1XwBcd7hZv0LhJSWxKvyJthq31jVH7L!-1133198190
X-OC-ID-TOKEN-STATUS: disabled
Authorization: Basic YXd1c2VyOldlbGNvbWVfMQ==
Accept: application/json, application/xml;q=0.8, */*;q=0.5
```

Note:

- The `connection` query string parameter is set to `conn`, which indicates that the service should use the connection created by the previous request to create the session.
- The `Cookie` header sends the `JSESSIONID` cookie which is associated with the messaging context and its encapsulated ephemeral objects, like connections, sessions, producers, and consumers. This cookie is required for the messaging context and any of its ephemeral objects to be available for the current request.
- The `X-OC-ID-TOKEN-STATUS` header indicates that the anti-CSRF token is disabled.
- The `Authorization` header indicates that the authentication type is `Basic`.
- The `Accept` header accepts JSON (most preferable), or XML (second preference), or anything else (least preferable).

Successful response.

```
HTTP/1.1 201 Created
Cache-Control: no-cache;no-store;must-revalidate;max-age=0
Content-Length: 0
```

Note:

- A session named `s` is created by the request.
- The `Cache-Control` header indicates that the response should not be cached.

3. Request to create a durable subscriber named `c`.

```
PUT /MCSService03-MCSOracle3/api/v1/consumers/c?session=s&subscriptionName=Audit
HTTP/1.1
Cookie:
JSESSIONID=MQm1TrjV2zLv7b0GV5kny1XwBcd7hZv0LhJSWxKvyJthq31jVH7L!-1133198190
X-OC-ID-TOKEN-STATUS: disabled
Authorization: Basic YXd1c2VyOldlbGNvbWVfMQ==
Accept: application/json, application/xml;q=0.8, */*;q=0.5
```

Note:

- The `session` query string parameter is set to `s`, which indicates that a consumer `c` will be created on the session `s`.
- The request contains a `subscriptionName` query string parameter without a `destination` query string parameter. This indicates that the consumer will be created on an existing durable subscription with client ID `cid` and subscription name `Audit`. If a 400 error response is returned, it indicates that the durable subscription with client ID `cid` and subscription name `Audit` does not exist.
- The `Cookie` header sends the `JSESSIONID` cookie which is associated with the messaging context and its encapsulated ephemeral objects, like connections, sessions, producers, and consumers. This cookie is required for the messaging context and any of its ephemeral objects to be available for the current request.
- The `X-OC-ID-TOKEN-STATUS` header indicates that the anti-CSRF token is disabled.
- The `Authorization` header indicates that the authentication type is `Basic`.

- The `Accept` header accepts JSON (most preferable), or XML (second preference), or anything else (least preferable).

Successful response.

```
HTTP/1.1 201 Created
Cache-Control: no-cache;no-store;must-revalidate;max-age=0
Content-Length: 0
X-OC-DESTINATION: /topics/Incoming
```

Note:

- A 201 response indicates that the durable subscription with client ID `cid` and subscription name `Audit` exists.
- The `X-OC-DESTINATION` header indicates that the existing durable subscription is on the topic `Incoming`.
- The response does not contain a `X-OC-SELECTOR` header. This indicates that the existing durable subscription does not have a selector, and it will store all the messages which are sent to `Incoming`.
- The `Cache-Control` header indicates that the response should not be cached.

4. Request to receive a message from the consumer `c`.

```
POST /MCSService03-MCSOracle3/api/v1/consumers/c/messages?timeout=1000 HTTP/1.1
Cookie:
JSESSIONID=MQm1TrjV2zLv7b0GV5kny1XwBcd7hZv0LhJSWxKvyJthq31jVH7L!-1133198190
X-OC-ID-TOKEN-STATUS: disabled
Authorization: Basic YXd1c2VyOldlbGNvbWVfMQ==
Accept: application/json, application/xml;q=0.8, */*;q=0.5
```

Note:

- The `timeout` query string parameter indicates that the service should return a response indicating that no message has been received if no message is currently stored in the durable subscription and no new message arrives within `1000` milliseconds.
- The `X-OC-ID-TOKEN-STATUS` header indicates that the anti-CSRF token is disabled.
- The `Authorization` header indicates that the authentication type is `Basic`.
- The `Accept` header accepts JSON (most preferable), or XML (second preference), or anything else (least preferable).

Response returns the following message:

```
HTTP/1.1 200 OK
Cache-Control: no-cache;no-store;must-revalidate;max-age=0
Transfer-Encoding: chunked
Content-Type: text/plain; charset=UTF-8
X-OC-DESTINATION: /topics/Incoming
X-OC-PRIORITY: 4
X-OC-DELIVERY-MODE: persistent
X-OC-TIMESTAMP: 1403757556763
X-OC-EXPIRATION: 1404967156763
X-OC-MESSAGE-TYPE: TEXT
X-OC-MESSAGE-ID: ID:adc00onb-44216-1403559824551-509:1:1:1:1
X-OC-REDELIVERED: false

0008
```

```
request

0000
```

Note:

- The `X-OC-MESSAGE-TYPE` header indicates that the message is a `TEXT` message, and hence the `Content-Type` is `text/plain; charset=UTF-8`.
- The `Transfer-Encoding` header indicates that the response body is being sent in chunks as defined by the HTTP 1.1 RFC (`http://www.w3.org/Protocols/rfc2616/rfc2616-sec3.html#sec3.6.1`).
- In this particular message, the content of the message is sent in one chunk, which is 8 bytes long, and consists of the letters `request` followed by a new-line character.
- The `Cache-Control` header indicates that the response should not be cached.

Receive a Message from a Queue with a Selector

The example shows an HTTP request and response for a message being received from a queue named `Postprocessing` with a selector.

1. Request to create a connection named `conn`.

```
PUT /MCSService03-MCSOracle3/api/v1/connections/conn HTTP/1.1
X-OC-ID-TOKEN-STATUS: disabled
Authorization: Basic YXd1c2VyOldlbGNvbWVfMQ==
Accept: application/json, application/xml;q=0.8, */*;q=0.5
```

Note:

- The `X-OC-ID-TOKEN-STATUS` header indicates that the anti-CSRF token is disabled.
- The `Authorization` header indicates that the authentication type is `Basic`.
- The `Accept` header accepts JSON (most preferable), or XML (second preference), or anything else (least preferable).
- The `PUT` request to create the connection does not contain the `action` query string parameter which starts the connection. Hence, the connection `conn` is not started when it is created.

Successful response.

```
HTTP/1.1 201 Created
Cache-Control: no-cache;no-store;must-revalidate;max-age=0
Content-Length: 0
X-OC-NEW-MESSAGING-CONTEXT: true
Set-Cookie:
JSESSIONID=szv9TrjYZxKBrgntMn3g5KnCrHtQfYyGLBkTQnR1d0R55XKQHcvf!-1133198190;
path=/; HttpOnly
```

Note:

- A connection named `conn` is created by the request.
- The `Cache-Control` header indicates that the response should not be cached.
- The `X-OC-NEW-MESSAGING-CONTEXT` header indicates that the request created a new messaging context.

- The Set-Cookie header provides the `JSESSIONID` cookie that must be sent in future requests in order to use the same connection.

2. Request to create a session named `s`.

```
PUT /MCSService03-MCSOracle3/api/v1/sessions/s?connection=conn HTTP/1.1
Cookie:
JSESSIONID=szv9TrjYZxKBrgntMn3g5KnCrHtQfYyGLBkTQnR1d0R55XKQHcvf!-1133198190
X-OC-ID-TOKEN-STATUS: disabled
Authorization: Basic YXd1c2VyOldlbGNvbWVfMQ==
Accept: application/json, application/xml;q=0.8, */*;q=0.5
```

Note:

- The `connection` query string parameter is set to `conn`, which indicates that the service should use the connection created by the previous request to create the session.
- The `Cookie` header sends the `JSESSIONID` cookie which is associated with the messaging context and its encapsulated ephemeral objects, like connections, sessions, producers, and consumers. This cookie is required for the messaging context and any of its ephemeral objects to be available for the current request.
- The `X-OC-ID-TOKEN-STATUS` header indicates that the anti-CSRF token is disabled.
- The `Authorization` header indicates that the authentication type is `Basic`.
- The `Accept` header accepts JSON (most preferable), or XML (second preference), or anything else (least preferable).

Successful response.

```
HTTP/1.1 201 Created
Cache-Control: no-cache;no-store;must-revalidate;max-age=0
Content-Length: 0
```

Note:

- A session named `s` is created by the request.
- The `Cache-Control` header indicates that the response should not be cached.

3. Request to create a consumer named `c` with the previously created session.

```
PUT /MCSService03-MCSOracle3/api/v1/consumers/c?session=s&destination=%2Fqueues
%2FPostprocessing&selector=palindrome HTTP/1.1
Cookie:
JSESSIONID=szv9TrjYZxKBrgntMn3g5KnCrHtQfYyGLBkTQnR1d0R55XKQHcvf!-1133198190
X-OC-ID-TOKEN-STATUS: disabled
Authorization: Basic YXd1c2VyOldlbGNvbWVfMQ==
Accept: application/json, application/xml;q=0.8, */*;q=0.5
```

Note:

- The `session` query string parameter is set to `s`, which indicates that a consumer `c` will be created in the session `s`.
- The `destination` query string parameter is set to the queue `Postprocessing`, which indicates that the consumer will consume from the queue `Postprocessing`.

- The consumer uses a selector, indicated by the `selector` query string parameter. The selector, `palindrome`, indicates that the consumer should only receive messages with a boolean property `palindrome` that has the value `true`.
- The `Cookie` header sends the `JSESSIONID` cookie which is associated with the messaging context and its encapsulated ephemeral objects, like connections, sessions, producers, and consumers. This cookie is required for the messaging context and any of its ephemeral objects to be available for the current request.
- The `X-OC-ID-TOKEN-STATUS` header indicates that the anti-CSRF token is disabled.
- The `Authorization` header indicates that the authentication type is `Basic`.
- The `Accept` header accepts JSON (most preferable), or XML (second preference), or anything else (least preferable).

Successful response.

```
HTTP/1.1 201 Created
Cache-Control: no-cache;no-store;must-revalidate;max-age=0
Content-Length: 0
```

4. Request to start the connection `conn`.

```
POST /MCSService03-MCSOracle3/api/v1/connections/conn?action=start HTTP/1.1
Cookie:
JSESSIONID=szv9TrjYZxKBrgntMn3g5KnCrHtQfYyGLBkTQnR1d0R55XKQHcvf!-1133198190
X-OC-ID-TOKEN-STATUS: disabled
Authorization: Basic YXd1c2VyOldlbGNvbWVfMQ==
Accept: application/json, application/xml;q=0.8, */*;q=0.5
```

Note:

- Sessions and consumers can be created without starting the connection. Connections can be started later using the `action` query string parameter.
- A connection must be started to receive messages from consumers on the connection.
- The `action` query string parameter is used to start the connection.
- The `X-OC-ID-TOKEN-STATUS` header indicates that the anti-CSRF token is disabled.
- The `Authorization` header indicates that the authentication type is `Basic`.
- The `Accept` header accepts JSON (most preferable), or XML (second preference), or anything else (least preferable).

Successful response.

```
HTTP/1.1 200 OK
Cache-Control: no-cache;no-store;must-revalidate;max-age=0
Content-Length: 0
```

5. Request to receive a message through the consumer `c`.

```
POST /MCSService03-MCSOracle3/api/v1/consumers/c/messages?timeout=1000 HTTP/1.1
Cookie:
JSESSIONID=szv9TrjYZxKBrgntMn3g5KnCrHtQfYyGLBkTQnR1d0R55XKQHcvf!-1133198190
X-OC-ID-TOKEN-STATUS: disabled
```

```
Authorization: Basic YXd1c2VyOldlbGNvbWVfMQ==
Accept: application/json, application/xml;q=0.8, */*;q=0.5
```

Note:

- The receive timeout of 1000 milliseconds is indicated by the `timeout` query string parameter.
- The `X-OC-ID-TOKEN-STATUS` header indicates that the anti-CSRF token is disabled.
- The `Authorization` header indicates that the authentication type is `Basic`.
- The `Accept` header accepts JSON (most preferable), or XML (second preference), or anything else (least preferable).

Response returns a message, as follows:

```
HTTP/1.1 200 OK
Cache-Control: no-cache;no-store;must-revalidate;max-age=0
Transfer-Encoding: chunked
Content-Type: text/plain; charset=UTF-8
X-OC-DESTINATION: /queues/Postprocessing
X-OC-PRIORITY: 4
X-OC-DELIVERY-MODE: persistent
X-OC-TIMESTAMP: 1403757560091
X-OC-EXPIRATION: 1404967160091
X-OC-MESSAGE-TYPE: TEXT
X-OC-MESSAGE-ID: ID:adc00onb-44216-1403559824551-513:1:1:1:1
X-OC-REDELIVERED: false
X-OC-BOOLEAN-PROPERTY-palindrome: true

0008
racecar

0000
```

Note:

- The `X-OC-MESSAGE-TYPE` header indicates that the message is a `TEXT` message, and hence the `Content-Type` is `text/plain; charset=UTF-8`.
- The `Transfer-Encoding` header indicates that the response body is being sent in chunks as defined by the HTTP 1.1 RFC (http://www.w3.org/Protocols/rfc2616/rfc2616-sec3.html#sec3.6.1).
- In this particular message, the content of the message is sent in one chunk, which is 8 bytes long, and consists of the letters `racecar` followed by a new-line character.
- The `X-OC-BOOLEAN-PROPERTY-palindrome` header indicates that the message has a boolean property `palindrome` that has the value `true`.

Send a Message to a Topic

The example shows HTTP requests to send a message to a topic named `Incoming`.

1. Request to create a connection named `conn`.

```
PUT /MCSService03-MCSOracle3/api/v1/connections/conn HTTP/1.1
X-OC-ID-TOKEN-STATUS: disabled
```

```
Authorization: Basic YXd1c2VyOldlbGNvbWVfMQ==
Accept: application/json, application/xml;q=0.8, */*;q=0.5
```

Note:

- The `X-OC-ID-TOKEN-STATUS` header indicates that the anti-CSRF token is disabled.
- The `Authorization` header indicates that the authentication type is `Basic`.
- The `Accept` header accepts JSON (most preferable), or XML (second preference), or anything else (least preferable).
- The `PUT` request to create the connection does not contain the `action` query string parameter which starts the connection. Hence, the connection `conn` is not started when it is created.

Successful response.

```
HTTP/1.1 201 Created
Cache-Control: no-cache;no-store;must-revalidate;max-age=0
Content-Length: 0
X-OC-NEW-MESSAGING-CONTEXT: true
Set-Cookie:
JSESSIONID=yvz2Trjhh3J2qvZMp2QTlCPDQqydsFlQPzBD1v92xGYwy6TbXjR5!-1133198190;
path=/; HttpOnly
```

Note:

- A connection named `conn` is created by the request.
- The `Cache-Control` header indicates that the response should not be cached.
- The `X-OC-NEW-MESSAGING-CONTEXT` header indicates that the request created a new messaging context.
- The Set-Cookie header provides the `JSESSIONID` cookie that must be sent in future requests in order to use the same connection.

2. Request to create a session named `s`.

```
PUT /MCSService03-MCSOracle3/api/v1/sessions/s?connection=conn HTTP/1.1
Cookie:
JSESSIONID=yvz2Trjhh3J2qvZMp2QTlCPDQqydsFlQPzBD1v92xGYwy6TbXjR5!-1133198190
X-OC-ID-TOKEN-STATUS: disabled
Authorization: Basic YXd1c2VyOldlbGNvbWVfMQ==
Accept: application/json, application/xml;q=0.8, */*;q=0.5
```

Note:

- The `connection` query string parameter is set to `conn`, which indicates that the service should use the connection created by the previous request to create the session.
- The `Cookie` header sends the `JSESSIONID` cookie which is associated with the messaging context and its encapsulated ephemeral objects, like connections, sessions, producers, and consumers. This cookie is required for the messaging context and any of its ephemeral objects to be available for the current request.
- The `X-OC-ID-TOKEN-STATUS` header indicates that the anti-CSRF token is disabled.
- The `Authorization` header indicates that the authentication type is `Basic`.

- The `Accept` header accepts JSON (most preferable), or XML (second preference), or anything else (least preferable).

Successful response.

```
HTTP/1.1 201 Created
Cache-Control: no-cache;no-store;must-revalidate;max-age=0
Content-Length: 0
```

Note:

- A session named `s` is created by the request.
- The `Cache-Control` header indicates that the response should not be cached.

3. Request to create a producer named `p` with the session `s`.

```
PUT /MCSService03-MCSOracle3/api/v1/producers/p?session=s&destination=%2Ftopics
%2FIncoming HTTP/1.1
Cookie:
JSESSIONID=yvz2Trjhh3J2qvZMp2QTlCPDQqydsFlQPzBD1v92xGYwy6TbXjR5!-1133198190
X-OC-ID-TOKEN-STATUS: disabled
Authorization: Basic YXd1c2VyOldlbGNvbWVfMQ==
Accept: application/json, application/xml;q=0.8, */*;q=0.5
```

Note:

- The `destination` query string parameter indicates that a destination is specified for the producer, and the producer will send messages to the topic `Incoming`.
- All messages sent via this producer will be sent to the specified destination.
- The `X-OC-ID-TOKEN-STATUS` header indicates that the anti-CSRF token is disabled.
- The `Authorization` header indicates that the authentication type is `Basic`.
- The `Accept` header accepts JSON (most preferable), or XML (second preference), or anything else (least preferable).

Successful response.

```
HTTP/1.1 201 Created
Cache-Control: no-cache;no-store;must-revalidate;max-age=0
Content-Length: 0
```

4. Request to send a `TEXT` message through the producer.

```
POST /MCSService03-MCSOracle3/api/v1/producers/p/messages?messageType=TEXT HTTP/
1.1
Cookie:
JSESSIONID=yvz2Trjhh3J2qvZMp2QTlCPDQqydsFlQPzBD1v92xGYwy6TbXjR5!-1133198190
X-OC-ID-TOKEN-STATUS: disabled
Authorization: Basic YXd1c2VyOldlbGNvbWVfMQ==
Content-Type: text/plain; charset=UTF-8
Accept: application/json, application/xml;q=0.8, */*;q=0.5
Content-Length: 15

A text message
```

Note:

- The `messageType` query string parameter indicates that the message is a TEXT message, and hence the `Content-Type` is `text/plain; charset=UTF-8`.
- The content of the message is `A text message`, followed by a new-line character.
- The `X-OC-ID-TOKEN-STATUS` header indicates that the anti-CSRF token is disabled.
- The `Authorization` header indicates that the authentication type is `Basic`.
- The `Accept` header accepts JSON (most preferable), or XML (second preference), or anything else (least preferable).

Successful response.

```
HTTP/1.1 201 Created
Cache-Control: no-cache;no-store;must-revalidate;max-age=0
Content-Length: 0
X-OC-DESTINATION: /topics/Incoming
X-OC-MESSAGE-ID: ID:adc00onb-44216-1403559824551-517:1:1:1:1
X-OC-DELIVERY-MODE: persistent
X-OC-TIMESTAMP: 1403757563379
X-OC-EXPIRATION: 1404967163379
X-OC-PRIORITY: 4
```

Note:

- The message's metadata is indicated by various headers. The *`X-OC-*`* headers indicate the message headers that are set by the sending operation, except the `X-OC-PRIORITY` header which is sent by the service.
- The `X-OC-DESTINATION` header indicates the message is sent to the topic `Incoming`.

Process Messages using a Transaction

The example shows an HTTP request and response for creating a transacted session, receiving a message from a queue named `Preprocessing`, sending a message to a queue named `Postprocessing`, and committing the session.

1. Request to create a connection named `conn`.

```
PUT /MCSService03-MCSOracle3/api/v1/connections/conn?action=start HTTP/1.1
X-OC-ID-TOKEN-STATUS: disabled
Authorization: Basic YXd1c2VyOldlbGNvbWVfMQ==
Accept: application/json, application/xml;q=0.8, */*;q=0.5
```

Note:

- The `X-OC-ID-TOKEN-STATUS` header indicates that the anti-CSRF token is disabled.
- The `Authorization` header indicates that the authentication type is `Basic`.
- The `Accept` header accepts JSON (most preferable), or XML (second preference), or anything else (least preferable).
- The `action` query string parameter is used to start the connection.

Successful response.

```
HTTP/1.1 201 Created
Cache-Control: no-cache;no-store;must-revalidate;max-age=0
Content-Length: 0
X-OC-NEW-MESSAGING-CONTEXT: true
Set-Cookie:
JSESSIONID=v1cyTrjbScfh2Cy62pyD9615xPbCx33vvQPT6JyJ5RG2TjvfG2Md!-1133198190;
path=/; HttpOnly
```

Note:

- A connection named `conn` is created by the request.
- The `Cache-Control` header indicates that the response should not be cached.
- The `X-OC-NEW-MESSAGING-CONTEXT` header indicates that the request created a new messaging context.
- The Set-Cookie header provides the `JSESSIONID` cookie that must be sent in future requests in order to use the same connection.

2. Request to create a session named `s`.

```
PUT /MCSService03-MCSOracle3/api/v1/sessions/s?connection=conn&transacted=true
HTTP/1.1
Cookie:
JSESSIONID=v1cyTrjbScfh2Cy62pyD9615xPbCx33vvQPT6JyJ5RG2TjvfG2Md!-1133198190
X-OC-ID-TOKEN-STATUS: disabled
Authorization: Basic YXd1c2VyOldlbGNvbWVfMQ==
Accept: application/json, application/xml;q=0.8, */*;q=0.5
```

Note:

- The `connection` query string parameter is set to `conn`, which indicates that the service should use the connection created by the previous request to create the session.
- The `Cookie` header sends the `JSESSIONID` cookie which is associated with the messaging context and its encapsulated ephemeral objects, like connections, sessions, producers, and consumers. This cookie is required for the messaging context and any of its ephemeral objects to be available for the current request.
- The `transacted` query string parameter indicates that the session is transacted.
- The `X-OC-ID-TOKEN-STATUS` header indicates that the anti-CSRF token is disabled.
- The `Authorization` header indicates that the authentication type is `Basic`.
- The `Accept` header accepts JSON (most preferable), or XML (second preference), or anything else (least preferable).

Successful response.

```
HTTP/1.1 201 Created
Cache-Control: no-cache;no-store;must-revalidate;max-age=0
Content-Length: 0
```

Note:

- A transacted session named `s` is created by the request.
- The `Cache-Control` header indicates that the response should not be cached.

3. Request to create a consumer named `c` on the queue `Preprocessing`.

```
PUT /MCSService03-MCSOracle3/api/v1/consumers/c?session=s&destination=%2Fqueues
%2FPreprocessing HTTP/1.1
Cookie:
JSESSIONID=v1cyTrjbScfh2Cy62pyD9615xPbCx33vvQPT6JyJ5RG2TjvfG2Md!-1133198190
X-OC-ID-TOKEN-STATUS: disabled
Authorization: Basic YXd1c2VyOldlbGNvbWVfMQ==
Accept: application/json, application/xml;q=0.8, */*;q=0.5
```

Note:

- The `session` query string parameter is set to `s`, which indicates that a consumer `c` will be created in the session `s`.
- The `destination` query string parameter is set to the queue `Preprocessing`, which indicates that the consumer will consume from the queue `Preprocessing`.
- The `Cookie` header sends the `JSESSIONID` cookie which is associated with the messaging context and its encapsulated ephemeral objects, like connections, sessions, producers, and consumers. This cookie is required for the messaging context and any of its ephemeral objects to be available for the current request.
- The `X-OC-ID-TOKEN-STATUS` header indicates that the anti-CSRF token is disabled.
- The `Authorization` header indicates that the authentication type is `Basic`.
- The `Accept` header accepts JSON (most preferable), or XML (second preference), or anything else (least preferable).

Successful response.

```
HTTP/1.1 201 Created
Cache-Control: no-cache;no-store;must-revalidate;max-age=0
Content-Length: 0
```

4. Request to create a producer named `p` with the session `s`.

```
PUT /MCSService03-MCSOracle3/api/v1/producers/p?session=s&destination=%2Fqueues
%2FPostprocessing HTTP/1.1
Cookie:
JSESSIONID=v1cyTrjbScfh2Cy62pyD9615xPbCx33vvQPT6JyJ5RG2TjvfG2Md!-1133198190
X-OC-ID-TOKEN-STATUS: disabled
Authorization: Basic YXd1c2VyOldlbGNvbWVfMQ==
Accept: application/json, application/xml;q=0.8, */*;q=0.5
```

Note:

- The `session` query string parameter is set to `s`, which indicates that a producer `p` will be created in the session `s`.
- The `destination` query string parameter indicates that a destination is specified for the producer, and the producer will send messages to the queue `Postprocessing`.
- The `X-OC-ID-TOKEN-STATUS` header indicates that the anti-CSRF token is disabled.
- The `Authorization` header indicates that the authentication type is `Basic`.

- The `Accept` header accepts JSON (most preferable), or XML (second preference), or anything else (least preferable).

Successful response.

```
HTTP/1.1 201 Created
Cache-Control: no-cache;no-store;must-revalidate;max-age=0
Content-Length: 0
```

5. Request to receive a message through the consumer `c`.

```
POST /MCSService03-MCSOracle3/api/v1/consumers/c/messages?timeout=1000 HTTP/1.1
Cookie:
JSESSIONID=v1cyTrjbScfh2Cy62pyD9615xPbCx33vvQPT6JyJ5RG2TjvfG2Md!-1133198190
X-OC-ID-TOKEN-STATUS: disabled
Authorization: Basic YXd1c2VyOldlbGNvbWVfMQ==
Accept: application/json, application/xml;q=0.8, */*;q=0.5
```

Note:

- The receive timeout of 1000 milliseconds is indicated by the `timeout` query string parameter.
- The `X-OC-ID-TOKEN-STATUS` header indicates that the anti-CSRF token is disabled.
- The `Authorization` header indicates that the authentication type is `Basic`.
- The `Accept` header accepts JSON (most preferable), or XML (second preference), or anything else (least preferable).

Response returns a message, as follows:

```
HTTP/1.1 200 OK
Cache-Control: no-cache;no-store;must-revalidate;max-age=0
Transfer-Encoding: chunked
Content-Type: text/plain; charset=UTF-8
X-OC-DESTINATION: /queues/Preprocessing
X-OC-PRIORITY: 4
X-OC-DELIVERY-MODE: persistent
X-OC-TIMESTAMP: 1403757556777
X-OC-EXPIRATION: 1404967156773
X-OC-MESSAGE-TYPE: TEXT
X-OC-MESSAGE-ID: ID:adc00onb-44216-1403559824551-365:1:5:1:1
X-OC-REDELIVERED: false

0010
Post-processing

0000
```

Note:

- The `X-OC-MESSAGE-TYPE` header indicates that the message is a `TEXT` message, and hence the `Content-Type` is as follows:

  ```
  text/plain; charset=UTF-8
  ```

- The `Transfer-Encoding` header indicates that the response body is being sent in chunks as defined by the HTTP 1.1 RFC (http://www.w3.org/Protocols/rfc2616/rfc2616-sec3.html#sec3.6.1).

- In this particular message, the content of the message is sent in one chunk, which is 16 bytes long, and consists of the letters `Post-processing` followed by a new-line character.
- The message is unavailable for receipt by other consumers. The receipt of the message is provisional because the session is transacted.
- If the session is rolled back, then the message will be available for receipt by other consumers, and will have the header `X-OC-REDELIVERED: true` on any subsequent receipts.

6. Request to send a message through the producer `p`.

```
POST /MCSService03-MCSOracle3/api/v1/producers/p/messages?messageType=TEXT HTTP/
1.1
Cookie:
JSESSIONID=v1cyTrjbScfh2Cy62pyD9615xPbCx33vvQPT6JyJ5RG2TjvfG2Md!-1133198190
X-OC-ID-TOKEN-STATUS: disabled
Authorization: Basic YXd1c2VyOldlbGNvbWVfMQ==
Content-Type: text/plain; charset=UTF-8
Accept: application/json, application/xml;q=0.8, */*;q=0.5
X-OC-BOOLEAN-PROPERTY-palindrome: true
Content-Length: 15

racecar
```

Note:

- The `messageType` query string parameter indicates that the message is a TEXT message, and hence the `Content-Type` is as follows:

  ```
  text/plain; charset=UTF-8
  ```

- Content of the message is `racecar`, followed by a new-line character.
- The message has a boolean property `palindrome` that has the value `true`.
- The `X-OC-ID-TOKEN-STATUS` header indicates that the anti-CSRF token is disabled.
- The `Authorization` header indicates that the authentication type is `Basic`.
- The `Accept` header accepts JSON (most preferable), or XML (second preference), or anything else (least preferable).

Successful response.

```
HTTP/1.1 201 Created
Cache-Control: no-cache;no-store;must-revalidate;max-age=0
Content-Length: 0X-OC-DESTINATION: /queues/Postprocessing
X-OC-MESSAGE-ID: ID:adc00onb-44216-1403559824551-519:1:1:1:1
X-OC-DELIVERY-MODE: persistent
X-OC-TIMESTAMP: 1403757567007
X-OC-EXPIRATION: 1404967167007
X-OC-PRIORITY: 4
```

Note:

- The message is sent provisionally because the session is transacted. The message is on the server, but it is not put on the queue, and is not available for the consumers to consume.

7. Request to commit the session.

```
POST /MCSService03-MCSOracle3/api/v1/sessions/s/state?action=commit HTTP/1.1
Cookie:
JSESSIONID=v1cyTrjbScfh2Cy62pyD9615xPbCx33vvQPT6JyJ5RG2TjvfG2Md!-1133198190
X-OC-ID-TOKEN-STATUS: disabled
Authorization: Basic YXd1c2VyOldlbGNvbWVfMQ==
Accept: application/json, application/xml;q=0.8, */*;q=0.5
```

Note:

- The `action` query string parameter indicates that the session should be committed.
- The `X-OC-ID-TOKEN-STATUS` header indicates that the anti-CSRF token is disabled.
- The `Authorization` header indicates that the authentication type is `Basic`.
- The `Accept` header accepts JSON (most preferable), or XML (second preference), or anything else (least preferable).

Successful response.

```
HTTP/1.1 200 OK
Cache-Control: no-cache;no-store;must-revalidate;max-age=0
Content-Length: 0
```

Note:

- Sending and receiving of messages is committed. This indicates that the messages sent by the producer `p` can be received by other consumers.

Cookie Management

It is important to manage HTTP cookies created by the REST API. HTTP cookies are used by HTTP clients to identity existing messaging contexts.

Messaging contexts are containers for ephemeral messaging objects like connections, sessions, producers, and consumers. The Oracle Messaging Cloud Service's Java library manages HTTP cookies for you. If you use the REST API directly (from Java or another programming platform) you need to manage the cookies yourself.This example shows how capture, store, and re-use HTTP cookies created by the REST API using standard Java platform classes from the java.net package. Most modern programming platforms which support HTTP provide support for managing HTTP cookies. Please consult your platform's manual for further guidance.

The example class `CookieManagement` includes an example in the `main` method.

```
package oracle.cloud.messaging.docs;

import java.io.IOException;
import java.net.CookieManager;
import java.net.CookiePolicy;
import java.net.CookieStore;
import java.net.HttpCookie;
import java.net.HttpURLConnection;
import java.net.URI;
import java.net.URL;
import java.net.URLConnection;
import java.util.List;
import java.util.Map;
```

```
import oracle.cloud.messaging.util.Base64Util;

/**
 *  Class to manage cookies, and demonstrate cookie management.
 *  Uses cookie management classes from java.net.
 */
public class CookieManagement
{
    // java.net CookieManager to store and manage cookies
    private CookieManager mgr = null;

    /**
     *  Create a CookieManagement Object
     *  with a given CookiePolicy.  See the
     *  javadoc for the java.net.CookiePolicy class
     *  (http://docs.oracle.com/javase/6/docs/api/java/net/CookiePolicy.html)
     *  for pre-defined values.
     */
    public CookieManagement(CookiePolicy policy)
    {
        this.mgr = new CookieManager();
        this.mgr.setCookiePolicy(policy);
    }

    /**
     *  Store cookies from the response headers from a
     *  fetch of a URL.
     *
     *  @param uri URI fetched
     *  @param responseHeaders Map encoding HTTP response headers
     *  @throws IOException
     */
    public void storeCookies(URI uri, Map<String,List<String>> responseHeaders)
throws IOException
    {
        synchronized(this.mgr)
        {
            this.mgr.put(uri,responseHeaders);
        }
    }

    /**
     *  Store cookies from the response headers from a
     *  fetch of a URL.
     *
     *  @param uri URI fetched
     *  @param connection A connection to the "uri" parameter
     *                    that has already been connected,
     *                    so that response headers are
     *                    available.
     *  @throws IOException
     */
    public void storeCookies(URI uri, URLConnection connection) throws IOException
    {
        this.storeCookies(uri,connection.getHeaderFields());
    }

    /**
     *  Set Cookie request headers corresponding to cookies
     *  from the cookie store that are appropriate to send
     *  to a URL.
```

```
     *
     *  @param uri URI to which we are about to connect
     *
     *  @param connection A connection to the "uri" parameter
     *                    that has not yet been connected,
     *                    so that request headers may
     *                    still be set.
     */
    public void setCookies(URI uri, URLConnection connection)
    {
        synchronized(this.mgr)
        {
            CookieStore store = this.mgr.getCookieStore();

            // Get list of appropriate cookies
            List<HttpCookie> cookies = store.get(uri);

            // Set cookies on the connection
            for(HttpCookie cookie : cookies)
            {
                connection.addRequestProperty("Cookie",cookie.toString());
            }
        }
    }

    /*
     *  Sample code that uses CookieManagement to
     *  connect to the OMCS REST API.
     *
     *  Usage:
     *
     *      java oracle.cloud.messaging.docs.CookieManagement \
     *          [Service URL, up to and including "/api/v1"] \
     *          [Identity domain name] \
     *          [user name]:[password] \
     *          [queue name]
     */
    public static void main(String[] argv) throws Exception
    {
        String baseURL = argv[0];
        String identityDomainName = argv[1];
        String userPassword = argv[2];
        String queueName = argv[3];

        // Queue encoded for use as a query string parameter
        String queueArg = "%2Fqueues%2F" + queueName;

        // Accept all cookies
        CookieManagement cm = new CookieManagement(CookiePolicy.ACCEPT_ALL);

        // Base64-encoded user:password
        String authorizationHeader = "Basic " +
Base64Util.to64(userPassword.getBytes("UTF-8"));

        // Connection to the service URL
        HttpURLConnection connection;

        // Whether to delete the queue used after using it
        boolean deleteAfter;

        // Create queue if it doesn't already exist
```

```
        URL createQueueURL = new URL(baseURL + "/queues/" + queueName);

        connection = (HttpURLConnection)createQueueURL.openConnection();

        connection.setRequestMethod("PUT");

        // Disable anti-CSRF token on the first access
        connection.setRequestProperty("X-OC-ID-TOKEN-STATUS","disabled");
        connection.setRequestProperty("X-ID-TENANT-NAME",identityDomainName);

        // Set Basic authentication header
        connection.setRequestProperty("Authorization",authorizationHeader);

        // Keep the connection open for future HTTP requests
        connection.setRequestProperty("Connection", "keep-alive");

        // Accept JSON most preferably, then XML, then anything else
        connection.setRequestProperty("Accept", "application/json, application/
xml;q=0.8, */*;q=0.5");

        System.out.println("Creating queue");
        connection.connect();

        // Store cookie on first access
        cm.storeCookies(createQueueURL.toURI(),connection);

        if (connection.getResponseCode() == HttpURLConnection.HTTP_CREATED)
        {
            System.out.println("Queue created, so will be deleted at the end of the
test");
            deleteAfter = true;
        }
        else
        if (connection.getResponseCode() == HttpURLConnection.HTTP_CONFLICT)
        {
            System.out.println("Queue already exists");
            deleteAfter = false;
        }
        else
        {
            throw new Exception("Queue creation failed");
        }

        // Create a connection and start it

        URL createConnectionURL = new URL(baseURL + "/connections/myConnection?
action=start");

        connection = (HttpURLConnection)createConnectionURL.openConnection();

        connection.setRequestMethod("PUT");

        connection.setRequestProperty("X-ID-TENANT-NAME",identityDomainName);
        connection.setRequestProperty("Authorization",authorizationHeader);
        connection.setRequestProperty("Connection", "keep-alive");
        connection.setRequestProperty("Accept", "application/json, application/
xml;q=0.8, */*;q=0.5");

        // Not the first access, so set cookies stored from previous accesses
        cm.setCookies(createConnectionURL.toURI(),connection);
```

```
        System.out.println("Creating connection");
        connection.connect();

        cm.storeCookies(createConnectionURL.toURI(),connection);

        if (connection.getResponseCode() != HttpURLConnection.HTTP_CREATED)
        {
            throw new Exception("Connection creation failed");
        }

        // Create a session

        URL createSessionURL = new URL(baseURL + "/sessions/mySession?
connection=myConnection");

        connection = (HttpURLConnection)createSessionURL.openConnection();

        connection.setRequestMethod("PUT");

        connection.setRequestProperty("X-ID-TENANT-NAME",identityDomainName);
        connection.setRequestProperty("Authorization",authorizationHeader);
        connection.setRequestProperty("Connection", "keep-alive");
        connection.setRequestProperty("Accept", "application/json, application/
xml;q=0.8, */*;q=0.5");

        cm.setCookies(createSessionURL.toURI(),connection);

        System.out.println("Creating session");
        connection.connect();

        cm.storeCookies(createSessionURL.toURI(),connection);

        if (connection.getResponseCode() != HttpURLConnection.HTTP_CREATED)
        {
            throw new Exception("Session creation failed");
        }

        // Create a producer with no default destination

        URL createProducerURL = new URL(baseURL + "/producers/myProducer?
session=mySession");

        connection = (HttpURLConnection)createProducerURL.openConnection();

        connection.setRequestMethod("PUT");

        connection.setRequestProperty("X-ID-TENANT-NAME",identityDomainName);
        connection.setRequestProperty("Authorization",authorizationHeader);
        connection.setRequestProperty("Connection", "keep-alive");
        connection.setRequestProperty("Accept", "application/json, application/
xml;q=0.8, */*;q=0.5");

        cm.setCookies(createProducerURL.toURI(),connection);

        System.out.println("Creating producer");
        connection.connect();

        cm.storeCookies(createProducerURL.toURI(),connection);

        if (connection.getResponseCode() != HttpURLConnection.HTTP_CREATED)
```

```
        {
            throw new Exception("Producer creation failed");
        }

        // Create a consumer on the queue specified on the command line

        URL createConsumerURL = new URL(baseURL + "/consumers/myConsumer?
session=mySession&destination=" + queueArg);

        connection = (HttpURLConnection)createConsumerURL.openConnection();

        connection.setRequestMethod("PUT");

        connection.setRequestProperty("X-ID-TENANT-NAME",identityDomainName);
        connection.setRequestProperty("Authorization",authorizationHeader);
        connection.setRequestProperty("Connection", "keep-alive");
        connection.setRequestProperty("Accept", "application/json, application/
xml;q=0.8, */*;q=0.5");

        cm.setCookies(createConsumerURL.toURI(),connection);

        System.out.println("Creating consumer");
        connection.connect();

        cm.storeCookies(createConsumerURL.toURI(),connection);

        if (connection.getResponseCode() != HttpURLConnection.HTTP_CREATED)
        {
            throw new Exception("Consumer creation failed");
        }

        // Send a PLAIN message to the queue via the producer

        URL sendMessageURL = new URL(baseURL + "/producers/myProducer/messages?
messageType=PLAIN&destination=" + queueArg);

        connection = (HttpURLConnection)sendMessageURL.openConnection();

        connection.setRequestMethod("POST");

        connection.setRequestProperty("X-ID-TENANT-NAME",identityDomainName);
        connection.setRequestProperty("Authorization",authorizationHeader);
        connection.setRequestProperty("Connection", "keep-alive");
        connection.setRequestProperty("Accept", "application/json, application/
xml;q=0.8, */*;q=0.5");

        cm.setCookies(sendMessageURL.toURI(),connection);

        System.out.println("Sending blank message");
        connection.connect();

        cm.storeCookies(sendMessageURL.toURI(),connection);

        if (connection.getResponseCode() != HttpURLConnection.HTTP_CREATED)
        {
            throw new Exception("Send failed");
        }

        // Receive from the queue 5 times, with a timeout of 1 second, and report
        //    whether a message was received each time.
```

```
        URL receiveURL = new URL(baseURL + "/consumers/myConsumer/messages?
timeout=1000");

        for(
            int receiveIndex = 0;
            receiveIndex < 5;
            receiveIndex++
        )
        {
            connection = (HttpURLConnection)receiveURL.openConnection();

            connection.setRequestMethod("POST");

            connection.setRequestProperty("X-ID-TENANT-NAME",identityDomainName);
            connection.setRequestProperty("Authorization",authorizationHeader);
            connection.setRequestProperty("Connection", "keep-alive");
            connection.setRequestProperty("Accept", "application/json, application/
xml;q=0.8, */*;q=0.5");
            cm.setCookies(receiveURL.toURI(),connection);

            System.out.print("Receiving ... ");
            connection.connect();

            cm.storeCookies(receiveURL.toURI(),connection);

            if (connection.getResponseCode() != HttpURLConnection.HTTP_OK)
            {
                throw new Exception("Receive failed");
            }

            // Check for X-OC-NULL: true header indicating that no message was
received
            if ("true".equals(connection.getHeaderField("X-OC-NULL")))
            {
                System.out.println("No message received");
            }
            else
            {
                System.out.println("Message received");
            }
        }

        // Close connection, and so the session, producer, and consumer

        URL closeConnectionURL = new URL(baseURL + "/connections/myConnection");

        connection = (HttpURLConnection)closeConnectionURL.openConnection();

        connection.setRequestMethod("DELETE");

        connection.setRequestProperty("X-ID-TENANT-NAME",identityDomainName);
        connection.setRequestProperty("Authorization",authorizationHeader);
        connection.setRequestProperty("Connection", "keep-alive");
        connection.setRequestProperty("Accept", "application/json, application/
xml;q=0.8, */*;q=0.5");

        cm.setCookies(closeConnectionURL.toURI(),connection);

        System.out.println("Closing connection");
        connection.connect();
```

```
        cm.storeCookies(closeConnectionURL.toURI(),connection);

        if (connection.getResponseCode() != HttpURLConnection.HTTP_NO_CONTENT)
        {
            throw new Exception("Connection close failed");
        }

        if (deleteAfter)
        {
            // Delete the queue created at the beginning

            URL deleteQueueURL = new URL(baseURL + "/queues/" + queueName);

            connection = (HttpURLConnection)deleteQueueURL.openConnection();

            connection.setRequestMethod("DELETE");

            connection.setRequestProperty("X-ID-TENANT-NAME",identityDomainName);
            connection.setRequestProperty("Authorization",authorizationHeader);
            connection.setRequestProperty("Connection", "keep-alive");
            connection.setRequestProperty("Accept", "application/json, application/
xml;q=0.8, */*;q=0.5");

            cm.setCookies(deleteQueueURL.toURI(),connection);

            System.out.println("Deleting queue");
            connection.connect();

            cm.storeCookies(deleteQueueURL.toURI(),connection);

            if (connection.getResponseCode() != HttpURLConnection.HTTP_NO_CONTENT)
            {
                throw new Exception("Queue deletion failed");
            }
        }

        // Expire the Messaging Context; this will clean up any ephemeral
        // resources not already cleaned up.
        URL expirURL = new URL(baseURL + "/maxInactiveInterval?mii=0");

        connection = (HttpURLConnection)expirURL.openConnection();

        connection.setRequestMethod("POST");

        connection.setRequestProperty("X-ID-TENANT-NAME",identityDomainName);
        connection.setRequestProperty("Authorization",authorizationHeader);
        connection.setRequestProperty("Accept", "application/json, application/
xml;q=0.8, */*;q=0.5");

        cm.setCookies(expirURL.toURI(),connection);

        System.out.println("Expiring Messaging Context");
        connection.connect();

        if (connection.getResponseCode() != HttpURLConnection.HTTP_OK)
        {
            System.out.println("Expiring failed");
        }
    }
}
```

Java Library

The information provided in this section applies to Java library.

Topics:

Create Resources

The example shows a command-line program that creates the following resources when the program is started and its `main` method is run:

- Two queues named `Preprocessing` and `Postprocessing`
- One topic named `Incoming`
- One message push listener named `Forwarder`, which receives messages from the topic named `Incoming` and sends them to the queue named `Preprocessing`

```
package oracle.cloud.messaging.samples;

import oracle.cloud.messaging.*;
import oracle.cloud.messaging.client.*;
import oracle.cloud.messaging.common.*;

import java.io.*;

public class CreateResources {

    public static void main(String[] argv) {

        MessagingServiceFactory factory = MessagingServiceFactory.getInstance();
        try {

            Namespace ns = new MessagingServiceNamespace("https://
messaging.us2.oraclecloud.com/mymessaging-john");
            Credentials creds = new
MessagingServiceCredentials("john.doe@oracle.com", "fFHG04x7");
            MessagingService ms = factory.getMessagingService(ns, creds);

            try {
                ms.createQueue("Preprocessing");
            } catch (DestinationExistsException ex) {
                System.out.println("Preprocessing queue already exists.");
            }

            try {
                ms.createQueue("Postprocessing");
```

```
            } catch (DestinationExistsException ex) {
                System.out.println("Postprocessing queue already exists.");
            }

            try {
                ms.createTopic("Incoming");
            } catch (DestinationExistsException ex) {
                System.out.println("Incoming already exists.");
            }

            String name = "Forwarder";
            Medium topic = Medium.getMedium(MediumType.TOPIC,"Incoming");
            PushMedium target =
PushMedium.getInstance(Medium.getMedium(MediumType.QUEUE,"Preprocessing"));
            MessagePushListener mpl = MessagePushListener.getInstance(name,
                                                                      topic,
                                                                      null,     //
No selector
                                                                      target,
                                                                      null      //
No failure policy
                                                                     );
            try {
                ms.createListener(null,     // No verification token; allowed
because no URLs pushed to
                                  mpl
                                 );
            } catch (ListenerExistsException ex) {
                System.out.println("Forwarder MPL already exists");
            }

        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

Send a Message to a Topic

The example shows a JAX-RS web service that sends a `TextMessage` to a topic named `Incoming`. The body of the HTTP request is used as the body of the `TextMessage`.

```
package oracle.cloud.messaging.samples;

import javax.ws.rs.*;
import javax.ws.rs.core.*;

import javax.jms.*;
import oracle.cloud.messaging.*;
import oracle.cloud.messaging.client.*;
import oracle.cloud.messaging.common.*;

import java.io.*;

@Path("/sendMessageToTopic")
public class sendMessageToTopic {

  @PUT
  @Produces(MediaType.TEXT_PLAIN)
```

```
  @Consumes(MediaType.TEXT_PLAIN)
  public String sendMessage(String body) {

      Connection conn = null;
      MessagingServiceFactory factory = MessagingServiceFactory.getInstance();
      try {

          Namespace ns = new MessagingServiceNamespace("https://
messaging.us2.oraclecloud.com/mymessaging-john");
          Credentials creds = new MessagingServiceCredentials("john.doe@oracle.com",
"fFHG04x7");
          MessagingService ms = factory.getMessagingService(ns, creds);

          ConnectionFactory cf = ms.getConnectionFactory();
          conn = cf.createConnection();
          conn.start();
          Session session = conn.createSession(false, Session.AUTO_ACKNOWLEDGE);
          Topic topic = session.createTopic("Incoming");
          MessageProducer producer = session.createProducer(topic);
          Message message = session.createTextMessage(body);
          producer.send(message);
      } catch (Exception ex) {

          StringWriter sw = new StringWriter();
          PrintWriter pw = new PrintWriter(sw);
          ex.printStackTrace(pw);
          return sw.toString();
      } finally {

          try {
              if (conn != null) {
                  conn.close();
              }
          } catch (Exception jmsex) {
              jmsex.printStackTrace();
          }
      }

      return "Message sent to topic successfully \n";

  }

}
```

Receive a Message from a Queue with an Optional Selector

The example shows a JAX-RS web service that receives a message from a queue named `Postprocessing`.

If the optional query parameter `palindromeOnly=true` is passed with the request, the consumer will use a selector to receive only messages that have the boolean message property `palindrome` set to `true`. If a message was received from the queue, the body of the message is returned in the body of the HTTP response.

```
package oracle.cloud.messaging.samples;

import javax.ws.rs.*;
import javax.ws.rs.core.*;

import javax.jms.*;
```

```
import oracle.cloud.messaging.*;
import oracle.cloud.messaging.client.*;
import oracle.cloud.messaging.common.*;

import java.io.*;

@Path("/receiveMessageFromQueue")
public class receiveMessageFromQueue {

  @POST
  @Produces(MediaType.TEXT_PLAIN)
  @Consumes(MediaType.TEXT_PLAIN)
  public String receiveMessage(@QueryParam("palindromeOnly") String palindromeOnly) {

      Connection conn = null;
      MessagingServiceFactory factory = MessagingServiceFactory.getInstance();
      try {

          Namespace ns = new MessagingServiceNamespace("https://
messaging.us2.oraclecloud.com/mymessaging-john");
          Credentials creds = new MessagingServiceCredentials("john.doe@oracle.com",
"fFHG04x7");
          MessagingService ms = factory.getMessagingService(ns, creds);

          ConnectionFactory cf = ms.getConnectionFactory();
          conn = cf.createConnection();
          conn.start();
          Session session = conn.createSession(false, Session.AUTO_ACKNOWLEDGE);
          Queue queue = session.createQueue("Postprocessing");
          MessageConsumer consumer;

          if(palindromeOnly != null && palindromeOnly.equals("true")) {
              consumer = session.createConsumer(queue, "palindrome");
          } else {
              consumer = session.createConsumer(queue);
          }

          Message message = consumer.receive(1000);

          if (message != null) {
              if (message instanceof TextMessage) {
                  return (((TextMessage) message).getText());
              } else {
                  return "A message not of type TextMessage was received\n";
              }
          } else {
              return "No message on queue\n";
          }

      } catch (Exception ex) {
          StringWriter sw = new StringWriter();
          PrintWriter pw = new PrintWriter(sw);
          ex.printStackTrace(pw);
          return sw.toString();
      } finally {
          try {
              if (conn != null) {
                  conn.close();
              }
          } catch (Exception jmsex) {
              jmsex.printStackTrace();
```

```
            }
        }

    }

}
```

Asynchronously Receive Messages with a Durable Subscription

The example shows a command-line program that also implements the `MessageListener` interface.

When the program is started and its `main` method is run, a durable subscription named `audit` is either created or reconnected to if it already exists. Messages are asynchronously received from the durable subscription and printed until an input is made to the program's standard input (`System.in`).

```
package oracle.cloud.messaging.samples;

import javax.jms.*;

import oracle.cloud.messaging.*;
import oracle.cloud.messaging.client.*;
import oracle.cloud.messaging.common.*;

import java.io.*;

public class AsyncReceiveFromDurableSubscription implements MessageListener {

    public static void main(String[] argv) {

        Connection conn = null;

        try {
            MessagingServiceFactory factory = MessagingServiceFactory.getInstance();

            Namespace ns = new MessagingServiceNamespace("https://
messaging.us2.oraclecloud.com/mymessaging-john");
            Credentials creds = new
MessagingServiceCredentials("john.doe@oracle.com", "fFHG04x7");
            MessagingService ms = factory.getMessagingService(ns, creds);

            ConnectionFactory cf = ms.getConnectionFactory();
            conn = cf.createConnection();
            conn.setClientID("AuditClient");
            Session session = conn.createSession(false, Session.AUTO_ACKNOWLEDGE);
            Topic topic = session.createTopic("Incoming");
            MessageConsumer consumer = session.createDurableSubscriber(topic,
"audit");

            consumer.setMessageListener(new AsyncReceiveFromDurableSubscription());
            conn.start();

            System.out.println("Hit RETURN to exit");
            System.in.read(new byte[1024]);
        } catch (Exception ex) {
            ex.printStackTrace();
        } finally {
            if (conn != null) {
                try {
```

```
                    conn.close();
                } catch (Exception jmsex) {
                    jmsex.printStackTrace();
                }
            }
        }
    }

    @Override
    public void onMessage(Message message) {
        if (!(message instanceof TextMessage)) {
            System.err.println("A message not of type TextMessage was received");
        } else {
            try {
                System.out.println("Message Received from durable subscription: " +
                                   ((TextMessage)message).getText());
            } catch (Exception ex) {
                ex.printStackTrace();
            }
        }
    }
}
```

Asynchronously Process Messages Within a Transaction

The example shows a command-line program that also implements the `MessageListener` interface.

When the program is started and its `main` method is run, a transacted session is created. Messages are asynchronously received from the queue named `Preprocessing` via the `MessageListener`. When a message is received, the contents of the message are reversed. A new message is created from the reversed contents of the original message and sent to the queue named `Postprocessing`. If the original message and the reversed message are identical, then a boolean message property named `palindrome` is set to `true`. After the send completes, the transacted session is committed. Messages are processed until an input is made to the program's standard input (`System.in`).

```
package oracle.cloud.messaging.samples;

import javax.jms.*;

import oracle.cloud.messaging.*;
import oracle.cloud.messaging.client.*;
import oracle.cloud.messaging.common.*;

import java.io.*;

public class AsyncTransactionProcessing implements MessageListener {

    private Session session;
    private MessageProducer producer;

    public AsyncTransactionProcessing(Session session, MessageProducer producer) {
        this.session = session;
        this.producer = producer;
    }

    public static void main(String[] argv) {
```

```
        Connection conn = null;

        try {
            MessagingServiceFactory factory = MessagingServiceFactory.getInstance();
            Namespace ns = new MessagingServiceNamespace("https://
messaging.us2.oraclecloud.com/mymessaging-john");
            Credentials creds = new
MessagingServiceCredentials("john.doe@oracle.com", "fFHG04x7");
            MessagingService ms = factory.getMessagingService(ns, creds);

            ConnectionFactory cf = ms.getConnectionFactory();
            conn = cf.createConnection();
            Session session = conn.createSession(true, Session.AUTO_ACKNOWLEDGE);
            Queue preprocessingQueue =
session.createQueue("Preprocessing");
            MessageConsumer consumer = session.createConsumer(preprocessingQueue);
            Queue postprocessingQueue = session.createQueue("Postprocessing");
            MessageProducer producer = session.createProducer(postprocessingQueue);

            consumer.setMessageListener(new
AsyncTransactionProcessing(session,producer));
            conn.start();

            System.out.println("Hit RETURN to exit");
            System.in.read(new byte[1024]);
        } catch (Exception ex) {
            ex.printStackTrace();
        } finally {
            if (conn != null) {
                try {
                    conn.close();
                } catch (Exception jmsex) {
                    jmsex.printStackTrace();
                }
            }
        }
    }

    @Override
    public void onMessage(Message message) {

        if (!(message instanceof TextMessage)) {
            System.err.println("A message not of type TextMessage was received");
            return;
        }

        try {

            String body = ((TextMessage)message).getText();
            String reversed_body = new StringBuilder(body).reverse().toString();
            TextMessage outgoingMessage = session.createTextMessage(reversed_body);

            if (body.equals(reversed_body)) {
                outgoingMessage.setBooleanProperty("palindrome", true);
            }

            producer.send(outgoingMessage);
            session.commit();
        } catch (Exception ex) {
            try {
```

```
                session.rollback();
                ex.printStackTrace();
            } catch (JMSException jmsex) {
                jmsex.printStackTrace();
            }
        }
    }

}
```

Use Message Groups

The example explains the step-by-step process to send a large message as a group of smaller messages. The example includes sample code to demonstrate how messages can be grouped and sequenced for the consumer.

Steps

1. Create a queue.
2. Create the consumers.
3. Create a producer.
4. Send a large message over the queue:
   a. Divide the large message into multiple smaller messages.
   b. Set the message `groupId` and `groupSeq`.
   c. Send the messages over the queue.
5. Consume the messages.

   Consolidate the messages with the same `groupId` into a single large message/file.
6. Send a large message over the queue.
7. Divide the large message into multiple smaller messages.
8. Set the message `groupId` and `groupSeq`.

Here's a sample code:

```
import java.io.*;
import javax.jms.*;
import oracle.cloud.messaging.client.*;
import oracle.cloud.messaging.common.DestinationExistsException;


/**
 *  Sends a file in chunks, optionally grouped using
 *  message properties.
 */
public class Sender
{
    private static void send(
        String fileName,
        String messageGroupID,
        MessageProducer prod,
        Session sess
    )
        throws Exception
    {
```

```
    FileReader in = null;

    try
    {
        int messageGroupSeq = 1;

        in = new FileReader(fileName);

        char[] buffer = new char[102400];

        for(
            int numRead = in.read(buffer);
            numRead > 0;
            numRead = in.read(buffer)
        )
        {
            TextMessage tmessage =
                sess.createTextMessage(new String(buffer,0,numRead));

            if (messageGroupID != null)
            {
                tmessage.setStringProperty("JMSXGroupID",messageGroupID);
                tmessage.setIntProperty("JMSXGroupSeq",messageGroupSeq++);
            }

            System.err.printf("Sending %d character\n",numRead);
            System.err.flush();

            prod.send(tmessage);
        }

        System.err.printf("Sending EOF messages\n");
        System.err.flush();

        for(
            int i = 0;
            i < 5;
            i++
        )
        {
            TextMessage tmessage = sess.createTextMessage();

            if (messageGroupID != null)
            {
                tmessage.setStringProperty("JMSXGroupID",messageGroupID);
                tmessage.setIntProperty("JMSXGroupSeq",messageGroupSeq++);
            }

            prod.send(tmessage);
        }
    }
    finally
    {
        if (in != null)
        {
            in.close();
        }
    }
}

public static void main(String[] argv)
```

```
    {
        if (argv.length < 5)
        {
            System.err.printf(
                "<URL> <user> <password> <queue name> <file name> [<message group
ID>]\n"
            );
            return;
        }

        OracleCloudConnection conn = null;

        try
        {
            OracleCloudConnectionFactory fact =
                MessagingServiceFactory
                    .getInstance()
                    .getMessagingService(
                        new MessagingServiceNamespace(argv[0]),
                        new MessagingServiceCredentials(argv[1],argv[2])
                    )
                    .getConnectionFactory();

            conn = fact.createConnection();

            OracleCloudSession sess =
                conn.createSession(
                    TransactionMode.NON_TRANSACTED,
                    AcknowledgementMode.AUTO_ACKNOWLEDGE
                );

            OracleCloudQueue q = sess.createQueue(argv[3]);

            OracleCloudMessageProducer prod = sess.createProducer(q);
            prod.setTimeToLive(TimeToLive.timeInMilliseconds(10000));

            conn.start();

            Sender.send(
                argv[4],
                ((argv.length >= 6) ? argv[5] : null),
                prod,
                sess
            );
        }
        catch(Exception exc)
        {
            exc.printStackTrace();
        }
        finally
        {
            if (conn != null)
            {
                try
                {
                    conn.close();
                }
                catch(Exception exc)
                {
                    exc.printStackTrace();
                }
```

```
            }
        }
    }
}

/*
 *  Receives a file sent in chunks and accumulates them
 *  into a file.
 */
public class Receiver
{
    public static void main(String[] argv)
    {
        if (argv.length < 5)
        {
            System.err.printf(
                "<URL> <user> <password> <queue name> <file name for output>\n"
            );
            return;
        }

        OracleCloudConnection conn = null;

        try
        {
            OracleCloudConnectionFactory fact =
                MessagingServiceFactory
                    .getInstance()
                    .getMessagingService(
                        new MessagingServiceNamespace(argv[0]),
                        new MessagingServiceCredentials(argv[1],argv[2])
                    )
                    .getConnectionFactory();

            conn = fact.createConnection();

            OracleCloudSession sess =
                conn.createSession(
                    TransactionMode.NON_TRANSACTED,
                    AcknowledgementMode.AUTO_ACKNOWLEDGE
                );

            OracleCloudQueue q = sess.createQueue(argv[3]);

            OracleCloudMessageConsumer cons = sess.createConsumer(q);

            Accumulator accumulator = new Accumulator(cons,argv[4]);

            conn.start();

            accumulator.start();

            System.err.printf("RETURN to stop receiving\n");
            System.err.flush();
            System.in.read(new byte[32]);

            accumulator.interrupt();
            while(accumulator.isAlive());
        }
        catch(Exception exc)
        {
```

```
            exc.printStackTrace();
        }
        finally
        {
            if (conn != null)
            {
                try
                {
                    conn.close();
                }
                catch(Exception exc)
                {
                    exc.printStackTrace();
                }
            }
        }
    }
}

/**
 *  This creates the queue to use for the demo, and then
 *  deletes it after a RETURN on the console.  It can
 *  probably be omitted from the sample code.
 */
public class Initializer
{
    public static void main(String[] argv) throws Exception
    {
        if (argv.length < 4)
        {
            System.err.printf(
                "<URL> <user> <password> <queue name>\n"
            );
            return;
        }

        MessagingService ms =
            MessagingServiceFactory
                .getInstance()
                .getMessagingService(
                    new MessagingServiceNamespace(argv[0]),
                    new MessagingServiceCredentials(argv[1],argv[2])
                );

        try
        {
            ms.createQueue(argv[3]);
        }
        catch(DestinationExistsException deexc)
        {
            // Already exists; ignore
        }

        System.err.printf("RETURN to delete the queue");
        System.in.read(new byte[32]);

        ms.deleteQueue(argv[3]);
    }
}

import java.io.*;
```

```
import javax.jms.*;

/*
 *  Thread that repeatedly consumes from a queue,
 *  concatenating received text payloads into a file until
 *  it receives an empty payload, at which point it closes
 *  the file.
 */
public class Accumulator extends Thread
{
    private MessageConsumer consumer = null;
    private byte[] buffer = new byte[102400];
    private FileWriter out = null;
    private String outName = null;

    public Accumulator(MessageConsumer consumer, String outName)
    {
        this.outName = outName;
        this.consumer = consumer;
    }

    @Override
    public void run()
    {
        while(!Thread.currentThread().isInterrupted())
        {
            try
            {
                this.process();
            }
            catch(Exception exc)
            {
                exc.printStackTrace();
                break;
            }
        }
    }

    public void process() throws IOException, JMSException
    {
        Message message = this.consumer.receive(1000);

        if (message instanceof TextMessage)
        {
            TextMessage tmessage = (TextMessage)message;

            String payload = tmessage.getText();

            if (payload == null)
            {
                if (this.out != null)
                {
                    System.err.printf(
                        "Closing file '%s'\n",
                        this.outName
                    );
                    System.err.flush();

                    this.out.close();
                    this.out = null;
                }
```

```
            }
            else
            {
                if (this.out == null)
                {
                    System.err.printf(
                        "Opening file '%s'\n",
                        this.outName
                    );
                    System.err.flush();

                    this.out = new FileWriter(this.outName);
                }

                System.err.printf(
                    "Writing %d chars to '%s'\n",
                    payload.length(),
                    this.outName
                );

                this.out.write(payload);
                this.out.flush();
            }
        }
    }
}
```

Receive Messages from a Queue Using a MessageListener

This example shows sample code to receive messages from a queue using a `MessageListener`.

The following is a command-line client to set up the `MessageListener`:

```
package oracle.cloud.messaging.demo;

import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.JMSException;
import javax.jms.MessageConsumer;
import javax.jms.Queue;
import javax.jms.Session;

import oracle.cloud.messaging.client.MessagingService;
import oracle.cloud.messaging.client.MessagingServiceCredentials;
import oracle.cloud.messaging.client.MessagingServiceFactory;
import oracle.cloud.messaging.client.MessagingServiceNamespace;

import oracle.cloud.messaging.MessagingException;

/**
 *  Oracle Messaging Service client code to receive
 *  messages from a queue using a MessageListener.
 */
public class MessageToFileClient
{
    private String urlWithNamespace = null;
    private String user = null;
    private String password = null;
    private String queueName = null;
```

```
private String dir = null;

private boolean started = false;

private Connection connection = null;
private Session session = null;
private MessageConsumer consumer = null;

/**
 *  @param urlWithNamespace
 *      Namespace URL for the messaging service
 *      instance to use
 *
 *  @param user
 *      User with admin privileges for the service
 *      instance
 *
 *  @param password
 *      Password for the user
 *
 *  @param queueName
 *      Name of the queue for the listener to listen on
 *
 *  @param dir
 *      Name of the directory into which to put text
 *      message payloads as files
 */
public MessageToFileClient(
    String urlWithNamespace,
    String user,
    String password,
    String queueName,
    String dir
)
{
    if (urlWithNamespace == null)
    {
        throw new IllegalArgumentException("urlWithNamespace == null");
    }

    if (user == null)
    {
        throw new IllegalArgumentException("user == null");
    }

    if (password == null)
    {
        throw new IllegalArgumentException("password == null");
    }

    if (queueName == null)
    {
        throw new IllegalArgumentException("queueName == null");
    }

    if (dir == null)
    {
        throw new IllegalArgumentException("dir == null");
    }

    this.urlWithNamespace = urlWithNamespace;
```

```
    this.user = user;
    this.password = password;
    this.queueName = queueName;
    this.dir = dir;
}

/**
 *  Start the client. If already started, this is
 *  a no-op. On return, the client is set up and
 *  listening.
 */
public void start() throws MessagingException, JMSException
{
    synchronized(this)
    {
        if (this.started)
        {
            return;
        }

        MessagingServiceFactory factory = MessagingServiceFactory.getInstance();

        MessagingServiceNamespace ns =
            new MessagingServiceNamespace(this.urlWithNamespace);
        MessagingServiceCredentials cred =
            new MessagingServiceCredentials(this.user,this.password);

        MessagingService ms = factory.getMessagingService(ns,cred);

        ConnectionFactory cf = ms.getConnectionFactory();

        this.connection = cf.createConnection();

        this.session =
            this.connection.createSession(
                false,      // Not transacted
                Session.AUTO_ACKNOWLEDGE
            );

        Queue q = this.session.createQueue(this.queueName);

        this.consumer = this.session.createConsumer(q);

        this.consumer
            .setMessageListener(
                new MessageToFileListener(this.dir)
            );

        this.connection.start();

        this.started = true;
    }
}

/**
 *  Return whether the client is started.
 */
public boolean isStarted()
{
    synchronized(this)
    {
```

```
        return this.started;
    }
}

/**
 *  Pause the listener. This will cause the
 *  listener to stop receiving messages until {@link
 *  #restart()} is called. If the client has not been
 *  started, IllegalStateException is thrown.
 */
public void pause() throws JMSException
{
    synchronized(this)
    {
        if (this.started)
        {
            this.connection.stop();
        }
        else
        {
            throw new IllegalStateException("Client unstarted");
        }
    }
}

/**
 *  Make the listener resume receiving messages.
 *  If the listener is not paused, this is a
 *  no-op. If the client has not been started,
 *  IllegalStateException is thrown.
 */
public void restart() throws JMSException
{
    synchronized(this)
    {
        if (this.started)
        {
            this.connection.start();
        }
        else
        {
            throw new IllegalStateException("Client unstarted");
        }
    }
}

/**
 *  Stop the client. If the client has not been
 *  started, this is a no-op. Once the client has
 *  been stopped, it cannot be re-started.
 */
public void stop() throws JMSException
{
    synchronized(this)
    {
        if (this.started)
        {
            this.connection.close();
        }
    }
}
```

```
    /**
     *  Run the client from the command line. The first
     *  5 arguments to the command line are the 5 inputs
     *  to the constructor, in order. After starting,
     *  the client will run until a newline is input to
     *  System.in, after which
     *  it will stop itself.
     */
    public static void main(String[] argv) throws Exception
    {
        MessageToFileClient client =
            new MessageToFileClient(
                argv[0],    // urlWithNamespace,
                argv[1],    // user,
                argv[2],    // password,
                argv[3],    // queueName,
                argv[4]     // dir
            );

        System.err.printf("Starting client ... ");
        System.err.flush();
        client.start();
        System.err.printf("started\n");
        System.err.flush();

        byte[] buffer = new byte[1024];

        System.in.read(buffer);

        System.err.printf("Stopping client ... ");
        System.err.flush();
        client.stop();
        System.err.printf("stopped\n");
        System.err.flush();
    }

    protected void finalize() throws Throwable
    {
        try
        {
            this.stop();
        }
        finally
        {
            super.finalize();
        }
    }
}
```

The following is the `MessageListener` class:

```
package oracle.cloud.messaging.demo;

import java.io.File;
import java.io.FileOutputStream;

import java.util.UUID;

import java.util.logging.Level;
```

```
import java.util.logging.Logger;

import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.ObjectMessage;

import oracle.cloud.messaging.client.HttpContent;

/**
 *  Client-side asynchronous message listener. This
 *  listener assumes that the messages it receives are
 *  ObjectMessages with HttpContent payloads, as would
 *  be the case if they had been sent via the REST
 *  API with message type HTTP, or no message type.
 *  Any messages not of this form are logged and discarded.
 *  The listener puts the body content of messages it processes
 *  into files in a specified directory.
 */
public class MessageToFileListener implements MessageListener
{
    private static final Logger logger =
        Logger.getLogger(MessageToFileListener.class.getName());

    private String dir = null;

    /**
     *  @param dir
     *      Path to the directory in which files containing
     *      message payloads will be put; may not be null
     */
    public MessageToFileListener(String dir)
    {
        if (dir == null)
        {
            throw new IllegalArgumentException("dir == null");
        }

        this.dir = dir;
    }

    @Override
    public void onMessage(Message message)
    {
        if (message instanceof ObjectMessage)
        {
            ObjectMessage omessage = (ObjectMessage)message;
            byte[] body = null;

            try
            {
                Object payload = omessage.getObject();

                if (payload instanceof HttpContent)
                {
                    String type = ((HttpContent)payload).getContentType();
                    body = ((HttpContent)payload).getContent();

                    System.err.printf("Got object message with '%s' content:
\n",type);
                    System.err.flush();
                    System.err.write(body);
```

```
                    FileOutputStream out =
                        new FileOutputStream(
                            this.dir +
                            File.separator +
                            UUID.randomUUID().toString() +
                            ".dat"
                        );

                    out.write(body);
                    out.flush();
                    out.close();
                }
                else
                {
                    MessageToFileListener.logger.log(
                        Level.SEVERE,
                        "Message delivered to listener is an ObjectMessage, but
payload is not HttpContent; payload class is '" +
                        payload.getClass().getName() +
                        "'"
                    );
                }
            }
            catch(Exception exc)
            {
                MessageToFileListener.logger.log(
                    Level.SEVERE,
                    "Exception writing message to file",
                    exc
                );
            }
        }
        else
        {
            MessageToFileListener.logger.log(
                Level.SEVERE,
                "Message delivered to listener is not an ObjectMessage; class is '" +
                message.getClass().getName() +
                "'"
            );
        }
    }
}
```