Using APIs for External Messaging Platforms

Version 2024.1
2024-05-02

InterSystems Corporation 1 Memorial Drive Cambridge MA 02142 www.intersystems.com

Using APIs for External Messaging Platforms
InterSystems IRIS Data Platform Version 2024.1 2024-05-02

For Support questions about any InterSystems products, contact:

InterSystems Worldwide Response Center (WRC)
Tel: +1-617-621-0700
Tel: +44 (0) 844 854 2917
Email: support@InterSystems.com

Table of Contents

1

Using the Messaging APIs

InterSystems IRIS provides messaging APIs that can be used to communicate directly with a variety of messaging platforms: JMS, Kafka, RabbitMQ, Amazon Simple Notification Service (SNS), and Amazon Simple Queue Service (SQS). These API classes are available in the %External.Messaging package.

Most of the code flow is the same regardless of the platform. This page discusses this common flow and the common classes and methods. Other articles provide details for specific messaging platforms; see the links above.

(In addition to the APIs described in this page, InterSystems provides specialized classes for use in interoperability productions. These classes enable productions to communicate directly with the same messaging platforms; follow the links above for details.)

1.1 Connecting to a Messaging Platform

Before you can send or receive messages, your code needs to:

1. Create a *settings object*, which contains information about connecting to the platform. This object is an instance of one of the settings classes. InterSystems provides a set of platform-specific settings classes, which are all subclasses of %External.Messaging.Settings. Create an instance of the applicable class and set its properties as needed.

   For example (Kafka):

   ObjectScript

   ```
   Set settings = ##class(%External.Messaging.KafkaSettings).%New()
   Set settings.username = "amandasmith"
   Set settings.password = "234sdsge"
   Set settings.servers = "100.0.70.179:9092, 100.0.70.089:7070"
   Set settings.clientId = "BazcoApp"
   // If Consumer, specify a consumer group
   Set settings.groupId = "G1"
   ```

2. Create the *messaging client object*, which is also specific to the platform. This object is an instance of one of the client classes, which are all subclasses of %External.Messaging.Client.

   To create the messaging client object, call the **CreateClient()** method of %External.Messaging.Client, passing the settings object as the first argument. For example:

   ObjectScript

   ```
   Set client = ##class(%External.Messaging.Client).CreateClient(settings, .tSC)
   // if tSC is an error, handle error scenario
   ```

The method returns a status code by reference as the second argument. Your code should check the status before proceeding.

CreateClient() returns an instance of the appropriate client class. For example, if the *settings* variable is an instance of %External.Messaging.KafkaSettings, the returned object is an instance of %External.Messaging.KafkaClient.

Similarly, if *settings* is an instance of %External.Messaging.SNSSettings, the returned object is an instance of %External.Messaging.SNSClient.

1.2 Creating Topics/Queues

Before you can send messages, you may need a topic or queue depending on which is relevant to your specific messaging platform.

Platform-specific settings for a topic/queue are passed into the common method as a string, but these settings are defined using an object, so the object's **ToJSON()** method must be called when passing in the settings. The following code creates a new Kafka topic using the common method:

ObjectScript

```
Set topic = "quick-start-events"
Set queueSettings = ##class(%External.Messaging.KafkaTopicSettings).%New()
Set queueSettings.numberOfPartitions = 1, queueSettings.replicationFactor = 1
Set tSC = client.CreateQueueOrTopic(topic, queueSettings.ToJSON())
```

Be aware that some messaging platforms have their own method for creating a topic/queue that may include advanced capabilities.

1.3 Sending Messages

Once you have a messaging client object and any relevant topic/queue object, call the methods of the client object to send messages, as follows:

1. Create a message object, which is an instance of a message class. InterSystems provides a set of platform-specific message classes, which are all subclasses of %External.Messaging.Message.

   For example (Kafka):

   ObjectScript

   ```
   Set msg = ##class(%External.Messaging.Messaging).%New()
   Set msg.topic = "quick-start-events"
   Set msg.value = "body of the message"
   Set msg.key = "somekey"
   ```

2. Call the **SendMessage()** method of the client object, passing the message object as the argument. For example:

   ObjectScript

   ```
   Set tSC = client.SendMessage(msg)
   // if tSC is an error, handle error scenario
   ```

   The method returns a status code, which your code should check before proceeding.

3. When your application no longer needs a topic/queue, your code can safely delete it using the DeleteTopicOrQueue() method of the client object.

   ObjectScript

   ```
   Do client.DeleteQueueOrTopic(topic)
   ```

   Be aware that some messaging platforms have their own method for deleting a topic/queue that may include advanced capabilities.

1.4 Receiving Messages

For messaging platforms that can receive messages, use the **ReceiveMessage()** method of the client object to receive messages. The first argument for this method is the name of the topic/queue. The second argument is an instance of %ListOfObjects. The method returns messages by reference using this second argument.

ReceiveMessage() also accepts a JSON-formatted string of *settings* as an optional third argument. Available settings vary by platform, and are defined as properties of the ReceiveSettings class for each platform. To define these settings, create a new instance of the ReceiveSettings class and set properties as desired. Then, use the **ToJSON()** method inherited by the ReceiveSettings object to provide these settings to the **ReceiveMessage()** method.

The **ReceiveMessage()** method returns a status code, which your program should check before it continues.

In the following example, a JMS client (*client*) receives messages from a JMS queue (*queue*), with the timeout for message retrieval set to 200 milliseconds:

ObjectScript

```
Set rset = ##class(%External.Messaging.JMSReceiveSettings).%New()
Set rset.receiveTimeout = 200

#dim messages As %ListOfObjects
Set tSC = client.ReceiveMessage(queue, .messages, rset.ToJSON())
// if tSC is an error, handle error scenario
```

The %ListOfObjects instance *messages* contains the messages obtained from the topic/queue.

1.5 Deleting Topics/Queues

When your application no longer needs a topic/queue, your code can safely delete it using the DeleteTopicOrQueue() method of the client object.

ObjectScript

```
Do client.DeleteQueueOrTopic(topic)
```

Be aware that some messaging platforms have their own method for deleting a topic/queue that may include advanced capabilities.

1.6 Disconnecting from a Messaging Platform

When your code is done communicating with the messaging platform, call the **Close()** method of the client object. For example:

ObjectScript

```
Do:client'="" client.Close()
```

2

Using the JMS Messaging API

InterSystems provides an API you can use to produce and consume messages using a Java Messaging Service (JMS). Your code acts as a producer or consumer by creating a client, then calling the client's methods to perform actions like sending and receiving messages. InterSystems IRIS also provides methods to create and delete JMS queues and topics.

The JMS API is based on the common messaging classes that are shared by other messaging platforms. This page describes platform-specific variations in the work flow which these common classes establish.

In addition to the API described here, InterSystems provides specialized classes that you can use to send messages to a JMS and retrieve messages from a JMS as part of an interoperability production.

2.1 Connecting to JMS

To create a connection to a JMS application:

1. Create a settings object. To do this, create an instance of %External.Messaging.JMSSettings and set its properties as needed:

   ObjectScript

   ```
   Set settings = ##class(%External.Messaging.JMSSettings).%New()
   Set settings.url = "messaging.bazco.com"
   Set settings.connectionFactoryName = "connectionFactory"
   Set settings.initialContextFactoryName = "eventContextFactory"
   Set settings.username = "briannawaller"
   Set settings.password = "824yvpi"
   ```

   For a full list of the properties available, see the JMSSettings class reference page.

2. Create the *messaging client object*. To do this, call the **CreateClient()** method of the generic %External.Messaging.Client class, passing the settings object as the first argument. For example:

   ObjectScript

   ```
   Set client = ##class(%External.Messaging.Client).CreateClient(settings, .tSC)
   If $$$ISERR(tSC) {
         //handle error scenario
   }
   ```

   The method returns a status code by reference as the second argument. Your code should check the status before proceeding.

   Because the *settings* object is an instance of %External.Messaging.JMSSettings, the returned object (*client*) is an instance of %External.Messaging.JMSClient.

2.2 JMS Producers

InterSystems IRIS can act as a producer within a JMS application by calling API methods to create messages and then send them. If you need to create the queues or topics which will route your messages, see Working with Queues and Topics. The following flow uses the client object to interact with a JMS application as a producer:

Set Message Properties (Optional)

The JMS specification allows you to attach a variety of metadata to a message, using both specified message headers and custom message properties. To create and set properties for a JMS message, you must create a %ListOfObjects collection of JMS message property objects. A JMS message object (described in the next section) accepts this list as an optional property. For each message property you want to define:

1. Create a new instance of the %External.Messaging.JMSMessageProperty object.
2. Set the properties of the message property object:
   - key, the message property key
   - type, the data type of the message property value
   - value, a string representation of the message property value
3. Add the message property object to your list of message property objects.

Refer to the JMS documentation for more information about message properties. The following example prepares a custom time stamp property:

ObjectScript

```
Set propList = ##class(%ListOfObjects).%New()

Set key = "odbcUtcTimestamp"
Set type = "String"
Set value = $zdatetime($horolog, 3, 8)

Set msgProp1 = ##class(%External.Messaging.JMSMessageProperty).%New()
Set msgProp1.key = key
Set msgProp1.type = type
Set msg.value = value

Set tSC = propList.Insert(msgProp1)
```

Create Message

To prepare a message to be sent, create a new instance of the %External.Messaging.JMSMessage object. Then, define properties for that message object. You must specify the name of the destination and the type (`"Text"` or `"Bytes"`) of the message. Depending on the message type, use the textBody property or the bytesBody property to set the message body. If you have created a list of message property objects as described in the previous section, provide that list as the properties property.

ObjectScript

```
Set destination = "quick-start-events-queue"
Set type = "Text"
Set textBody = "MyMessage"

Set msg = ##class(%External.Messaging.JMSMessage).%New()
Set msg.destination = destination
Set msg.type = type
Set msg.textBody = textBody
Set msg.properties = propList
```

Send Message

After creating a message, you can send it to the destination queue or topic by executing the **SendMessage()** method for the JMS client object. For example:

ObjectScript

```
set tSC = client.SendMessage(msg)
if $$$ISERR(tSC) {
      //handle error scenario
}
```

2.3 JMS Consumers

InterSystems IRIS can act as an a consumer within a JMS application by calling an API method to retrieve messages for a topic. The following flow uses the client to interact with a JMS application as a consumer:

Configure Settings for Message Retrieval (Optional)

The JMS client object can use the **ReceiveMessage()** method to act as a consumer of JMS messages. This method allows you to specify settings for the message retrieval operation by providing a JSON-formatted string as an optional argument. To do so, create a new instance of the %External.Messaging.JMSReceiveSettings class and set properties as desired. The following properties are available:

- receiveTimeout, an integer specifying the duration of time (in milliseconds) before the retrieval operation times out. If not set, the default value is `100`.
- subscriber (optional), a string containing a subscriber name to identify the client

For example:

ObjectScript

```
Set rset = ##class(%External.Messaging.JMSReceiveSettings).%New()
Set rset.receiveTimeout = 200
```

Retrieve Messages

To retrieve messages, invoke the **ReceiveMessage()** method inherited by the JMS client object. This method takes the name of a queue or a topic as an argument and returns messages as a %ListOfObjects by reference. If you have specified message retrieval settings as described in the preceding section, provide these settings as a third argument using the **ToJSON()** method for the settings object.

ObjectScript

```
#dim messages As %ListOfObjects
Set tSC = client.ReceiveMessage(queue, .messages, rset.ToJSON())
```

2.4 Working with Queues and Topics

InterSystems IRIS provides an API to manage destinations for JMS messages. For point-to-point messaging, you can create or delete a queue. For publisher-to-subscriber messaging, you can create or delete a topic.

2.4.1 Create or Delete an Queue

Create an Queue

A JMS client object includes a **CreateQueue()** method for creating a queue. **CreateQueue()** accepts the queue name as an argument:

ObjectScript

```
Set queueName = "quick-start-events"
Set tSC = client.CreateQueue(queueName)
```

As an alterative, you can create the queue with a method that is common to all messaging platforms. See %External.Messaging.Client.CreateQueueOrTopic() for details.

Delete an Queue

You can delete a JMS queue with a method that is common to client objects for all messaging platforms supported by the API. See %External.Messaging.Client.DeleteQueueOrTopic() for details.

ObjectScript

```
Set tSC = client.DeleteQueueOrTopic(queueName)
```

2.4.2 Create or Delete a Topic

Create a Topic

A JMS client object includes a **CreateTopic()** method for creating a topic. **CreateTopic()** accepts the topic name as an argument:

ObjectScript

```
Set topicName = "alerts_urgent"
Set tSC = client.CreateTopic(topicName)
```

As an alterative, you can create the topic with a method that is common to all messaging platforms. See %External.Messaging.Client.CreateQueueOrTopic() for details.

Delete a Topic

You can delete a JMS topic with a method that is common to client objects for all messaging platforms supported by the API. See %External.Messaging.Client.DeleteQueueOrTopic() for details.

ObjectScript

```
Set tSC = client.DeleteQueueOrTopic(topicName)
```

2.5 Close Client

An InterSystems IRIS application that is done communicating with a JMS application should close the client with the **Close()** method for the client object. For example:

ObjectScript

```
Do:client'="" client.Close()
```

3

Using the Kafka Messaging API

InterSystems provides an API you can use to produce and consume Kafka messages. Your code acts as a producer or consumer by creating a client, then calling the client's methods to perform actions like sending and receiving messages. InterSystems IRIS also provides methods to create and delete Kafka topics.

The Kafka API is based on the common messaging classes that are shared by other messaging platforms. This page describes platform-specific variations in the work flow these common classes establish.

In addition to the API described here, InterSystems provides specialized classes that you can use in interoperability productions. These classes enable your productions to send messages to Kafka and retrieve messages from Kafka.

3.1 Connecting to Kafka

To create a connection to Kafka:

1. Create a settings object. To do this create an instance of %External.Messaging.KafkaSettings and set its properties as needed. Most of the properties, listed below, apply to both producers and consumers; the exception is the groupId setting, which is used only to assign a consumer to a consumer group.
   - username and password define the client's Kafka credentials.
   - servers defines a comma-separated list of IP address:port entries that identify servers in your Kafka broker cluster.
   - clientId optionally defines the client ID of the Kafka producer or consumer.
   - groupId defines the consumer group ID of a Consumer.
   - securityprotocol specifies the security protocol which secures connections to your Kafka broker cluster. Currently, this property supports two values:
     - `SASL_PLAINTEXT`, which performs SASL authentication of the client over an unencrypted channel.
     - `SASL_SSL`, which uses the truststore and keystore information you provide to establish an SSL/TLS connection over which SASL authentication takes place.
   - saslmechanism specifies the SASL authentication mechanism used to authenticate the client. Currently, only `PLAIN` is supported.
   - truststorelocation (optional) specifies the file system path to the truststore which contains the certificate authority certificates necessary to validate a certificate from your Kafka broker cluster and establish an SSL/TLS connection.
   - truststorepassword (optional) defines the password which provides access to the truststore at the location specified by truststorelocation.

- keystorelocation (optional) specifies the file system path to the keystore which contains the keys necessary to establish an SSL/TLS connection with your Kafka broker cluster.
- keystorepassword (optional) defines the password which provides access to the keystore at the location specified by keystorelocation.
- keypassword (optional) defines the password which provides access to a private key within the keystore at the location specified by keystorelocation.

For example:

ObjectScript

```
Set settings = ##class(%External.Messaging.KafkaSettings).%New()
Set settings.username = "amandasmith"
Set settings.password = "234sdsge"
Set settings.servers = "100.0.70.179:9092, 100.0.70.089:7070"
Set settings.clientId = "BazcoApp"
// If Consumer, specify a consumer group
Set settings.groupId = "G1"
Set settings.securityprotocol="SASL_SSL"
Set settings.saslmechanism="PLAIN"
Set settings.truststorelocation="/etc/kafkatls/trustcerts.jks"
Set settings.truststorepassword="F00B&r!"
```

2. Create the *messaging client object*. To do this, call the **CreateClient()** method of %External.Messaging.Client, passing the settings object as the first argument. For example:

ObjectScript

```
Set client = ##class(%External.Messaging.Client).CreateClient(settings, .tSC)
// if tSC is an error, handle error scenario
```

The method returns a status code by reference as the second argument. Your code should check the status before proceeding.

Because the *settings* object is an instance of %External.Messaging.KafkaSettings, the method returns an instance of %External.Messaging.KafkaClient for *client*.

3.2 Kafka Producers

InterSystems IRIS can act as a Kafka producer by calling API methods to create messages and send them. If the application needs to create the topic where messages will be sent, see Working with Topics. The following flow uses the client to interact with Kafka as a producer:

Configure Client as Producer (Optional)

After creating the Kafka client but before sending messages, the application can customize the Producer using standard Apache ProducerConfig configuration options. The client defaults to a standard Producer configuration, but modifications can be made by passing the Apache options as a JSON string to the **UpdateProducerConfig()** method. For a list of supported options, refer to the Apache Kafka documentation. For example, the following code configures the Kafka client with an Apache configuration option:

ObjectScript

```
 Set client = ##class(%External.Messaging.Client).CreateClient(kafkaSettings, .tSC)

 Set producerSettings =
"{""key.serializer"":""org.apache.kafka.common.serialization.StringSerializer""}"
 Set tSC = client.UpdateProducerConfig(producerSettings)
```

Create Message

A message sent to Kafka must contain a topic and value, and can optionally contain a key, which acts as a tag for the value. To prepare a message to be sent to a topic, create a new Kafka message object and then define these properties. For example:

ObjectScript

```
Set topic = "quick-start-events"
Set value = "MyMessage", key = "OptionalTag"

Set msg = ##class(%External.Messaging.KafkaMessage).%New()
Set msg.topic = topic
Set msg.value = value
Set msg.key = key
```

To support Kafka configurations which allow for the exchange of messages which exceed the length of an InterSystems IRIS %String, store your message content as an appropriately encoded binary stream using the property binaryValue instead of value. The binaryValue property can be arbitrarily long.

Send Message

After creating a message, you can send it to the topic by executing the **SendMessage()** method. For example:

ObjectScript

```
Set tSC = client.SendMessage(msg)
```

3.3 Kafka Consumers

InterSystems IRIS can act as a Kafka consumer by calling APIs to retrieve messages for a topic. Be sure to define the groupId property when defining the settings of the client in order to identify the consumer group of the Consumer. The following flow uses the client to interact with Kafka as a Consumer:

Configure Client as Consumer (Optional)

After creating the Kafka client but before retrieving messages, the application can customize the Consumer using standard Apache ConsumerConfig configuration options. The client defaults to a standard Consumer configuration, but modifications can be made by passing the Apache options as a JSON string to the **UpdateConsumerConfig()** method. For a list of supported options, refer to the Apache Kafka documentation. For example, the following code configures the Kafka client with an Apache configuration option:

ObjectScript

```
Set client = ##class(%External.Messaging.Client).CreateClient(kafkaSettings, .tSC)

Set consumerSettings =
"{""key.deserializer"":""org.apache.kafka.common.serialization.StringDeserializer""}"
Set tSC = client.UpdateConsumerConfig(consumerSettings)
```

Configure Settings for Message Retrieval (Optional)

The Kafka client can use the **ReceiveMessage()** method to act as a Kafka Consumer. This method allows you to set the poll timeout for the message retrieval operation (in milliseconds) by providing a JSON-formatted string as an optional argument. If you wish to do so, create a new instance of the %External.Messaging.KafkaReceiveSettings class and set the pollTimeout property:

ObjectScript

```
Set rset = ##class(%External.Messaging.KafkaReceiveSettings).%New()
Set rset.pollTimeout = 500
```

If not set, the default poll timeout duration is 100 milliseconds.

Retrieve Messages

To retrieve messages, invoke the **ReceiveMessage()** method for the Kafka client. This method takes the name of the topic as an argument and returns messages as a %ListOfObjects by reference. If you have specified message retrieval settings using a KafkaReceiveSettings object (as described in the preceding section), provide these settings as a third argument using the **ToJSON()** method for the settings object. Kafka returns only new messages, that is, messages that have not been previously retrieved by the Consumer. For example, to retrieve and display messages for a topic:

ObjectScript

```
#dim messages As %ListOfObjects
Set tSC = client.ReceiveMessage(topic, .messages, rset.ToJSON())

For i=1:1:messages.Size {
       Set msg = messages.GetAt(i)
       Write "Message: ", msg.ToJSON(), !
}
```

For messages which exceed the length of an InterSystems IRIS %String, the Kafka message class also includes a binaryValue property which can store messages of arbitrary length as appropriately encoded binary streams.

3.4 Defining AdminClient Configs

An application can customize the Kafka client using standard Apache AdminClient Configs. The client defaults to a standard configuration, but modifications can be made by passing the Apache options as a JSON string to the UpdateAdminConfig() method. For a list of supported options, refer to the Apache Kafka documentation. For example:

ObjectScript

```
Set client = ##class(%External.Messaging.Client).CreateClient(kafkaSettings, .tSC)

Set adminSettings = "{""ssl.protocol"":""TLSv1.3""}"
Set tSC = client.UpdateAdminConfig(adminSettings)
```

3.5 Working with Topics

InterSystems IRIS provides an API that can be used to create a new Kafka topic, and another that deletes a topic. When creating a topic, the number of partitions and the replication factor are passed in as arguments. For an introduction to partitions and replication factors, see the Apache Kafka documentation. To create a topic:

ObjectScript

```
Set client = ##class(%External.Messaging.Client).CreateClient(kafkaSettings, .tSC)

Set topic = "quick-start-events"
Set numberOfPartitions = 1
Set replicationFactor = 3
Set tSC = client.CreateTopic(topic,numberOfPartitions,replicationFactor)
```

As an alterative, you can create the topic with a method that is common to all messaging platforms. When using this alternative, the topic settings are defined in an instance of the %External.Messaging.KafkaTopicSettings class and then passed to the method as a JSON object using the **ToJSON()** method. See %External.Messaging.Client.CreateQueueOrTopic() for details.

3.5.1 Delete a Topic

An application can delete a Kafka topic using the **DeleteTopic()** method.

ObjectScript

```
Set tSC = client.DeleteTopic(topic)
```

As an alterative, you can delete the topic with a method that is common to all messaging platforms. See %External.Messaging.Client.DeleteQueueOrTopic() for details.

3.6 Close Client

An InterSystems IRIS application that is done communicating with Kafka should close the client with the **Close()** method. For example:

ObjectScript

```
Do:client'="" client.Close()
```

4

Using the RabbitMQ Messaging API

InterSystems provides an API you can use to publish and consume RabbitMQ messages. Your code acts as a publisher or consumer by creating a client, then calling the client's methods to perform actions like sending and receiving messages. InterSystems IRIS also provides methods to manage RabbitMQ queues and exchanges, and to set bindings between the two.

The RabbitMQ API is based on the common messaging classes that are shared by other messaging platforms. This page describes platform-specific variations in the work flow which these common classes establish.

In addition to the API described here, InterSystems provides specialized classes that you can use to send messages to RabbitMQ and retrieve messages from RabbitMQ as part of an interoperability production.

Note: The InterSystems RabbitMQ API corresponds with the RabbitMQ implementation of the AMQP 0–9–1 protocol. To send and receive messages with an external messaging service using the AMQP 1.0 protocol, InterSystems recommends connecting to a JMS implementation of the protocol such as Apache Qpid JMS. InterSystems provides a JMS API for working with JMS applications.

4.1 Connecting to RabbitMQ

To create a connection to RabbitMQ:

1. Create a settings object. To do this create an instance of %External.Messaging.RabbitMQSettings and set its properties as needed:

   - username and password define the client's RabbitMQ credentials. If you do not set the value for these properties, `"guest"` is provided as the default.
   - host defines the name for the host server. If not defined, `"localhost"` is the default.
   - port is an integer which defines the port number being used on the host server. The default is `5672`.
   - virtualHost optionally defines the name of the virtual host.

   For example:

   ObjectScript

   ```
   Set settings = ##class(%External.Messaging.RabbitMQSettings).%New()
   Set settings.username = "ronaldkellogg"
   Set settings.password = "449!ds%t"
   Set settings.host = "bazco.com"
   Set settings.port = 5693
   ```

2. (Optional.) If you want to connect to RabbitMQ using SSL/TLS, set the enableSSL property for the settings object to 1, and then set the following properties according to your TLS configuration:
   - tlsVersion, a string that specifies the version of the TLS protocol you are using. The default value is "TLSv1.2"
   - enableHostnameVerification, a boolean which determines whether the peer verification process includes a verification that the hostname of the server matches the name on the server certificate
   - clientKeyFile, a string specifying the path to the client's private key file (if the server is configured to perform peer verification)
   - keyPassword, the password string which is required to access the client key file (if the key file is secured)
   - keyStoreFile, a string specifying the path to the key store file
   - keyStorePassword, the password string which is required to access the key store file (if the key store file is secured)

3. Create the *messaging client object*. To do this, call the **CreateClient()** method of the generic %External.Messaging.Client class, passing the settings object as the first argument. For example:

   ObjectScript

   ```
   Set client = ##class(%External.Messaging.Client).CreateClient(settings, .tSC)
   If $$$ISERR(tSC) { //handle error scenario }
   ```

   The method returns a status code by reference as the second argument. Your code should check the status before proceeding.

   Because the *settings* object is an instance of %External.Messaging.RabbitMQSettings, the returned object (*client*) is an instance of %External.Messaging.RabbitMQClient.

4.2 RabbitMQ Publishers

InterSystems IRIS can act as a RabbitMQ publisher by calling API methods to create messages and send them. If the application needs to create the exchanges where messages will be sent or the queues where the exchanges will route them, see Working with Exchanges and Queues. The following flow uses the client object to interact with RabbitMQ as a publisher:

Create Message

To prepare a message to be sent, create a new instance of the %External.Messaging.RabbitMQMessage object. Then, define the properties for that message object as needed. For a full description of the available message properties, refer to the RabbitMQ documentation. Set the content of the message by invoking the **SetEncodedContent()** method (for UTF-8) or the **SetContent()** method (for other values of contentEncoding). For example:

ObjectScript

```
set exchange = "events_handler"
set routingKey = "quick_start"
set deliveryMode = 2
set body = "MyMessage"

set msg = ##class(%External.Messaging.RabbitMQMessage).%New()
set msg.exchange = exchange
set msg.routingKey = routingKey
set msg.deliveryMode = deliveryMode
do msg.SetEncodedContent(body)
```

Send Message

After creating a message, you can send it to the topic by executing the **SendMessage()** method for the RabbitMQ client object. For example:

ObjectScript

```
set tSC = client.SendMessage(msg)
if $$$ISERR(tSC) { //handle error scenario }
```

4.3 RabbitMQ Consumers

InterSystems IRIS can act as a RabbitMQ consumer by calling APIs to retrieve messages from a queue. The following flow uses the client object to interact with RabbitMQ as a consumer:

Configure Settings for Message Retrieval (Optional)

The RabbitMQ client can use the **ReceiveMessage()** method to act as a RabbitMQ consumer. This method allows you to specify settings for the message retrieval operation by providing a JSON-formatted string as an optional argument. To do so, create a new instance of the %External.Messaging.RabbitMQReceiveSettings class and set properties as desired. The following properties are available:

- autoAck, a boolean. If true, the server considers messages acknowledged automatically once they are delivered. If false, the server expects explicit, manual acknowledgements from the consumer. If not set, autoAck defaults to `false`.
- ackMultiple, a boolean. If true, a manual acknowledgement acknowledges the batch of all messages up to and including the message corresponding to the delivery tag which the acknowledgement supplies. If false, an acknowledgement only acknowledges the message corresponding to the delivery tag it supplies. If not set, ackMultiple defaults to `false`.

For example:

ObjectScript

```
Set rset = ##class(%External.Messaging.RabbitMQReceiveSettings).%New()
Set rset.autoAck = 0
```

Retrieve Messages

To retrieve messages, invoke the **ReceiveMessage()** method inherited by the RabbitMQ client. This method takes the name of a queue as an argument and returns messages as a %ListOfObjects by reference. If you have specified message retrieval settings as described in the preceding section, provide these settings as a third argument using the **ToJSON()** method. For example:

ObjectScript

```
 #dim messages As %ListOfObjects
 Set tSC = client.ReceiveMessage(queue, .messages, rset.ToJSON())

 For i=1:1:messages.Size {
        Set msg = messages.GetAt(i)
        Write "Message: ", msg.ToJSON(), !
 }
```

4.4 Working with Exchanges and Queues

InterSystems IRIS provides an API for managing RabbitMQ exchanges and queues. This includes:

- Creating or deleting an exchange
- Creating or deleting a queue
- Binding a queue to an exchange

This section describes how to use the API to perform these tasks. For a detailed explanation of exchanges and queues, refer to the introduction to the AMQP 0–9–0 protocol which is provided in the RabbitMQ documentation.

4.4.1 Create or Delete an Exchange

Per the AMQP 0–9–0 specification, all RabbitMQ messages must be sent to an exchange, which routes them to their destination queue (or queues).

Create an Exchange

A RabbitMQ client object includes a **CreateExchange()** method for creating an exchange. **CreateExchange()** accepts the following arguments, in order:

1. *exchangeName*, the name you wish to assign to the exchange.
2. *exchangeType*, a string specifying one of the four AMQP 0–9–1 exchange types: `"direct"`, `"fanout"`, `"topic"`, or `"headers"`.
3. *durable*, a boolean. If true, the exchange survives after the server restarts. If false, the exchange must be created again.
4. *autoDelete*, a boolean. If true, the exchange is deleted when all queues are unbound from it. If false, the exchange persists.

For example:

```
Set exchangeName = "broadcast"
Set exchangeType = "fanout"
Set durable = 1
Set autoDelete = 0

Set tSC = client.CreateExchange(exchangeName, exchangeType, durable, autoDelete)
```

Delete an Exchange

An application can delete a RabbitMQ exchange by invoking the **DeleteExchange()** method of the RabbitMQ client object, providing the name of the exchange as an argument.

```
Set tSC = client.DeleteExchange(exchangeName)
```

4.4.2 Create or Delete a Queue

A RabbitMQ consumer receives messages when an exchange routes them to a queue to which the consumer is subscribed.

Create a Queue

To create a queue, invoke the **CreateQueue()** method of the RabbitMQ client object. **CreateQueue()** accepts the following arguments, in order:

1. *queueName*, the name you wish to assign to the queue.
2. *durable*, a boolean. If true, the queue persists after the server restarts. If false, the exchange must be created again after a restart.
3. *exclusive*, a boolean. If true, the queue is used by only one connection and is deleted when the connection closes.
4. *autoDelete*, a boolean. If true, the queue is deleted when all consumers unsubscribe. If false, the queue persists.

For example:

ObjectScript

```
Set queue = "quick-start-events"
Set durable = 1
Set exclusive = 0
Set autoDelete = 0
Set tSC = client.CreateQueue(queue, durable, exclusive, autoDelete)
```

As an alterative, you can create the queue with a method that is common to all messaging platforms. When using this alternative, the queue settings are defined in an instance of the %External.Messaging.RabbitMQQueueSettings class and then passed to the method as a JSON object using the **ToJSON()** method. See %External.Messaging.Client.CreateQueueOrTopic() for details.

Delete a Queue

An application can delete a RabbitMQ queue by invoking the **DeleteQueue()** method of the RabbitMQ client object, providing the name of the queue as an argument.

```
Set tSC = client.DeleteQueue(queueName)
```

As an alterative, you can delete the queue with a method that is common to all messaging platforms. See %External.Messaging.Client.DeleteQueueOrTopic() for details.

4.4.3 Bind a Queue to an Exchange

To route messages to a queue, the queue must be bound to an exchange. An application can bind a queue to an exchange by invoking the **BindQueue()** method of the RabbitMQ client object.

As its arguments, the **BindQueue()** method accepts the queue name, the exchange name, and a string containing the binding keys separated by commas. A comma which is part of a binding key can be escaped by preceding it with the backslash character (\); backslashes which are part of a binding key can be escaped using a second backslash character.

For example, if the two desired binding keys were `"event-log,critical"` and `"event-log,urgent/important"`, application code could bind a queue as follows:

```
Set bindingKeys = "event-log\,critical,event-log\,urgent\\important"
Set tSC = client.BindQueue(queueName, exchangeName, bindingKeys)
```

4.5 Close Client

An InterSystems IRIS application that is done communicating with RabbitMQ should close the client with the **Close()** method. For example:

ObjectScript

```
Do:client'="" client.Close()
```

5

Using the Amazon SNS Messaging API

InterSystems provides an API you can use to publish messages using the Amazon Simple Notification Service (SNS). Your code acts as a publisher by creating a client and then calling the client's methods to send messages. InterSystems IRIS also provides methods to create and delete Amazon SNS topics.

The Amazon SNS API is based on the common messaging classes that are shared by other messaging platforms. This page describes platform-specific variations in the work flow which these common classes establish.

In addition to the API described here, InterSystems provides specialized classes that you can use to send messages to Amazon SNS as part of an interoperability production.

5.1 Connecting to Amazon SNS

To create a connection to Amazon SNS:

1. Create a settings object. To do this create an instance of %External.Messaging.SNSSettings and set its properties as follows:

   - credentialsFile, a string specifying the location of your Amazon Simple Storage Service (S3) credentials file.
   - accessKey, a string containing your Amazon S3 access key. If you have specified a credentialsFile, you do not need to set this property.
   - secretKey, a string containing your Amazon S3 secret key. If you have specified a credentialsFile, you do not need to set this property.
   - sessionToken, a string containing an Amazon S3 session token. If you have specified a credentialsFile which includes a session token, you do not need to set this property.
   - region, a string specifying an Amazon S3 region.

   For example:

   ObjectScript

   ```
   Set settings = ##class(%External.Messaging.SNSSettings).%New()
   Set settings.credentialsFile = "~/.aws/credentials/cred.ini"
   Set settings.region = "us-east-1"
   ```

2. Create the *messaging client object*. To do this, call the **CreateClient()** method of the generic %External.Messaging.Client class, passing the settings object as the first argument. For example:

ObjectScript

```
Set client = ##class(%External.Messaging.Client).CreateClient(settings, .tSC)
// If tSC is an error, handle error scenario
```

The method returns a status code by reference as the second argument. Your code should check the status before proceeding.

Because the *settings* object is an instance of %External.Messaging.SNSSettings, the returned object (*client*) is an instance of %External.Messaging.SNSClient.

5.2 Amazon SNS Publishers

InterSystems IRIS can act as an Amazon SNS publisher by calling API methods to create messages and then send them. If the application needs to create the topics where messages will be sent, see Working with Topics. The following flow uses the client object to interact with Amazon SNS as a publisher:

Create Message

To prepare a message to be sent, create a new instance of the %External.Messaging.SNSMessage object. Then, define properties for that message object. You must specify the Amazon Resource Name (ARN) for the topic where the message will be sent (the topicARN property) and a message body (the message property). You can also specify an optional subject for the message.

ObjectScript

```
Set topicARN = "arn:aws:sns:us-east-1:123456789012:quick-start-events"
Set message = "MyMessage"
Set subject = "EventNotification"

Set msg = ##class(%External.Messaging.SNSMessage).%New()
Set msg.topicARN = topicARN
Set msg.message = message
Set msg.subject = subject
```

Send Message

After creating a message, you can send it to the topic by executing the **SendMessage()** method for the Amazon SNS client object. For example:

ObjectScript

```
set tSC = client.SendMessage(msg)
if $$$ISERR(tSC) {
      //handle error scenario
}
```

5.3 Working with Topics

InterSystems IRIS provides an API that can be used to create and delete Amazon SNS topics.

Create a Topic

To create a topic, invoke the **CreateTopic()** method of the client object. The method accepts the topic name as an argument, and returns an ARN for the topic by reference. For example:

ObjectScript

```
Set topicName = "quick-start-events"
Set topicARN = ""
Set tSC = client.CreateQueue(topicName, .topicARN)
```

As an alterative, you can create the topic with a method that is common to all messaging platforms: %External.Messaging.Client.CreateQueueOrTopic(). However, this generic method does not return the ARN for the new topic, which is required by the API methods for deleting a topic and for sending a message. To use the API to perform these tasks with a topic created using **CreateQueueOrTopic()**, you must obtain the ARN for the topic manually.

Delete a Topic

An application can delete an Amazon SNS topic by invoking the **DeleteTopic()** method of the client object, providing the ARN of the topic as an argument.

ObjectScript

```
Set tSC = client.DeleteTopic(topicARN)
```

As an alterative, you can delete the topic with a method that is common to all messaging platforms. See %External.Messaging.Client.DeleteQueueOrTopic() for details.

5.4 Close Client

An InterSystems IRIS application that is done communicating with Amazon SNS should close the client with the **Close()** method. For example:

ObjectScript

```
Do:client'="" client.Close()
```

6

Using the Amazon SQS Messaging API

InterSystems provides an API you can use to produce and consume messages using the Amazon Simple Queue Service (SQS). Your code acts as a producer or consumer by creating a client, then calling the client's methods to perform actions like sending and receiving messages. InterSystems IRIS also provides methods to create and delete Amazon SQS queues.

The Amazon SQS API is based on the common messaging classes that are shared by other messaging platforms. This page describes platform-specific variations in the work flow which these common classes establish.

In addition to the API described here, InterSystems provides specialized classes that you can use to send messages to Amazon SQS and retrieve messages from Amazon SQS as part of an interoperability production.

6.1 Connecting to Amazon SQS

To create a connection to Amazon SQS:

1. Create a settings object. To do this create an instance of %External.Messaging.SQSSettings and set its properties as follows:

   - credentialsFile, a string specifying the location of your Amazon Simple Storage Service (S3) credentials file.
   - accessKey, a string containing your Amazon S3 access key. If you have specified a credentialsFile, you do not need to set this property.
   - secretKey, a string containing your Amazon S3 secret key. If you have specified a credentialsFile, you do not need to set this property.
   - sessionToken, a string containing an Amazon S3 session token. If you have specified a credentialsFile which includes a session token, you do not need to set this property.
   - region, a string specifying an Amazon S3 region.

   For example:

   ObjectScript

   ```
   Set settings = ##class(%External.Messaging.SQSSettings).%New()
   Set settings.credentialsFile = "~/.aws/credentials/cred.ini"
   Set settings.region = "us-east-1"
   ```

2. Create the *messaging client object*. To do this, call the **CreateClient()** method of the generic %External.Messaging.Client class, passing the settings object as the first argument. For example:

ObjectScript

```
Set client = ##class(%External.Messaging.Client).CreateClient(settings, .tSC)
If $$$ISERR(tSC) {
      //handle error scenario
}
```

The method returns a status code by reference as the second argument. Your code should check the status before proceeding.

Because the *settings* object is an instance of %External.Messaging.SQSSettings, the returned object (*client*) is an instance of %External.Messaging.SQSClient.

6.2 Amazon SQS Producers

InterSystems IRIS can act as an Amazon SQS publisher by calling API methods to create messages and then send them. If the application needs to create the topics where messages will be sent, see Working with Queues. The following flow uses the client object to interact with Amazon SQS as a publisher:

Set Message Attributes (Optional)

To attach custom metadata to your message using Amazon SQS message attributes, you must create an InterSystems IRIS %ListOfObjects collection of SQS message attribute objects. The Amazon SQS message object you will create in the next section will accepts this list of attributes as an optional property. For each attribute you want to define:

1. Create a new instance of the %External.Messaging.SQSMessageAttribute object.
2. Set the properties of the message attribute object:
   - key, the message attribute key
   - dataType, the data type of the message attribute value (`"String"`, `"Number"`, or `"Binary"`)
   - stringValue or binaryValue, the message attribute value. Set the property appropriate to the data type you have specified.
3. Add each message attribute object to your list of message attribute objects.

Refer to the Amazon SQS message metadata documentation for more information about message attributes. The following example prepares a time stamp attribute:

ObjectScript

```
Set attrList = ##class(%ListOfObjects).%New()

Set key = "timestamp"
Set dataType = "String"
Set value = $zdatetime($horolog)

Set msgAttr1 = ##class(%External.Messaging.SQSMessageAttribute).%New()
Set msgAttr1.key = key
Set msgAttr1.dataType = dataType
Set msg.stringValue = value

Set tSC = attrList.Insert(msgAttr1)
```

Create Message

To prepare a message to be sent, create a new instance of the %External.Messaging.SQSMessage object. Then, define properties for that message object. You must specify the name of the destination queue and the body of the message. If you have defined custom message attributes as described in the previous section, provide the list of message attribute objects as the messageAttributes property. For example:

ObjectScript

```
Set queue = "quick-start-events"
Set body = "MyMessage"

Set msg = ##class(%External.Messaging.SQSMessage).%New()
Set msg.queue = queue
Set msg.body = body
Set msg.messageAttributes = attrList
```

Send Message

After creating a message, you can send it to the topic by executing the **SendMessage()** method for the Amazon SQS client object:

ObjectScript

```
set tSC = client.SendMessage(msg)
if $$$ISERR(tSC) {
      //handle error scenario
}
```

6.3 Amazon SQS Consumers

InterSystems IRIS can act as an Amazon SQS consumer by calling an API method to retrieve messages for a topic. The following flow uses the client object to interact with Amazon SQS as a consumer:

Configure Settings for Message Retrieval (Optional)

The Amazon SQS client can use the **ReceiveMessage()** method to act as an Amazon SQS consumer. This method allows you to specify settings for the message retrieval operation by providing a JSON-formatted string as an optional argument. To do so, create a new instance of the %External.Messaging.SQSReceiveSettings class and set properties as desired. The following properties are available:

- maxNumberOfMessages, an integer specifying the maximum number of messages to return
- waitTimeSeconds, an integer specifying the number of seconds before polling timeout
- visibilityTimeout, an integer specifying the number of seconds during which the messages returned by the method are effectively invisible to other consumers

For example:

ObjectScript

```
Set rset = ##class(%External.Messaging.SQSReceiveSettings).%New()
Set rset.waitTimeSeconds = 5
Set rset.visibilityTimeout = 30
```

Retrieve Messages

To retrieve messages, invoke the **ReceiveMessage()** method inherited by the Amazon SQS client object. This method takes the name of a queue as an argument and returns messages as a %ListOfObjects by reference. If you have specified message retrieval settings as described in the preceding section, provide these settings as a third argument using the **ToJSON()** method. For example:

ObjectScript

```
#dim messages As %ListOfObjects
Set tSC = client.ReceiveMessage(queue, .messages, rset.ToJSON())
```

Delete Messages from the Queue

An Amazon SQS consumer is responsible for deleting messages from a queue as the consumer receives and processes them. To delete a message, invoke the **DeleteMessage()** method for the client object. **DeleteMessage()** requires you to provide the name of the queue as the first argument and the receipt handle for the message as the second argument. The receipt handle is stored in the receiptHandle property for each message object the **ReceiveMessage()** method returns.

ObjectScript

```
For i=1:1:messages.Size {
       Set msg = messages.GetAt(i)
       Write "Message: ", msg.ToJSON(), !
       Set tSC = client.DeleteMessage(queue, msg.receiptHandle)
}
```

6.4 Working with Queues

InterSystems IRIS provides an API that can be used to create and delete Amazon SQS queues.

Specify Queue Settings (Optional)

If you would like to specify settings for your queue, create an %External.Messaging.SQSQueueSettings object and set the properties of that object corresponding to your desired settings. For more information about the configuration options available, refer to the Amazon SQS documentation.

For example, the following code creates a queue settings object which specifies a first-in-first-out queue and delays the delivery of all messages in the queue for five seconds:

ObjectScript

```
Set queueSet = ##class(%External.Messaging.SQSQueueSettings).%New()
Set queueSet.FifoQueue = 1
Set queueSet.DelaySeconds = 5
```

Create a Queue

To create a queue, invoke the **CreateQueue()** method of the client object. **CreateQueue()** requires you to provide a queue name as an argument. If you have created a queue settings object for the queue (as described in the previous section), you may provide this object as an optional second argument.

ObjectScript

```
Set queue = "quick-start-events"
Set tSC = client.CreateQueue(queue, queueSet)
```

As an alterative, you can create the queue with a method that is common to all messaging platforms. When using this alternative, you can provide the contents of your queue settings object as a JSON object using the **ToJSON()** method. See %External.Messaging.Client.CreateQueueOrTopic() for details.

Delete a Queue

An application can delete an Amazon SQS queue by invoking the **DeleteTopic()** method of the client object. This method accepts the queue name as an argument.

ObjectScript

```
Set tSC = client.DeleteQueue(queue)
```

As an alterative, you can delete the queue with a method that is common to all messaging platforms. See %External.Messaging.Client.DeleteQueueOrTopic() for details.

6.5 Close Client

An InterSystems IRIS application that is done communicating with Amazon SQS should close the client with the **Close()** method. For example:

ObjectScript

```
Do:client'="" client.Close()
```